NFS

```
We are going from threads to distributed systems and back again
  Both topics are squarely in scope at OS conferences
  Similar kinds of reasoning apply to parallelism & distributed systems
    E.g., Already saw happens before applied to race detection
  Different "gotchas" in each case
    Parallel systems: weak memory consistency can be very unintuitive
    Distributed systems: node & network failures complicate reasoning
  If you like distributed systems, consider CS244b next Fall...

How did Unix systems access network storage before NFS? ND - network disk
  Looks like a disk device on client (e.g., /dev/nd0)
  IO requests go over net to server, which has file storing disk image
  Note concept still exists in the form of NBD under linux

What were goals of NFS? (p. 119)

1. Machine and OS independence
   Required adding a new system call, getdirentries (p. 124).  Why?
     Previously, readdir used read(2) to look at raw directory contents
     getdirentries abstracts directory formats across machines

2. Crash recovery
   When is a local file system allowed to lose data in a crash?
     Data written recently (<30 sec ago), and no call to fsync
     Okay because processes writing files are killed in crash, too
   When should NFS be allowed to lose data?
     Client crash?  Same as local file system
     Server crash?  Never, because processes on client not killed

3. Transparent access
   "Programs should not be able to tell whether a file is remote or local."
   What would be alternative?
     Hack libc so open("/nfs/server/file") doesn't do real open syscall
     Would break applications--e.g., inheriting file descriptors

4. UNIX semantics maintained on client
   Some examples of tricky UNIX semantics?
     Most permission checks happen when file initially opened
     Can delete an open file and still use it

5. Reasonable performance
   No worse than ND, or 80% as good as local disk

Which of above goals met by ND?  All but #1
Why isn't ND good enough?
  NFS also wants *sharing* of file system resources
  Can't share a read-write file system stored on ND
  Ordinary file systems don't expect disk to change out from under them
  Might cross-allocate blocks/inodes, create duplicate file names, etc.

What does NFS look like to administrator?
  Unix namespace comprised of *mount points*
    A root file system mounted on "/" (the root directory) at boot time
    Other file system mounted on other directories
    Created with mount command:  mount /dev/sda3 /usr
  Server admin can *export* some file systems over NFS in /etc/exports
  Client admin can mount remote FS "server:/dir" instead of device
    E.g.: mount my-server:/disk/u1 /home/u1
    Now /home/u1 on client is same FS as /disk/u1 on server!
  Two kinds of NFS mount:  hard and soft (p. 124)--what's the difference?
    Hard mount hangs forever if server crashes
    With soft mount, syscalls eventually return error if no server reply
```

```
  Note, more recently NFS also offers intr vs. nointr
    Kernel assumed disk requests fast, so not FS syscalls interruptible
    So couldn't Ctrl-C process if accidentally accessed unavailable server
    Took many years to fix this by adding intr option

Until NFS, only one kind of file system in kernel.  How to abstract?
  Two new abstractions:  VFS, and vnodes (p. 123)
    Ersatz C++ abstract classes, implemented with C function pointers
  Have one VFS (virtual file system) struct per mount point
    unmount(), root(), statfs(), sync()
  Have one vnode for each active inode
    lookup, open, close, create, rdwr, inactive, ...
    Also contains pointer to its VFS struct

Rewrote namei (p.124) [routine mapping (dir vnode, path) -> vnode] Why?
  Only the client knows which directories are mount points
  Hence, cannot have server translate more than one directory at a time
  So namei must walk file system one vnode loopkup() at a time

VFS+vnode routines for NFS make RPCs to server--see pp. 120-121

What happens if I say "cat dir/file" on an NFS file system?
  lookup (fh1, "dir") -> fh2
  lookup (fh2, "file") -> fh3
  read (fh3, [offset] 0, [len] 8192)
  read (fh3, [offset] 8192, [len] 8192)
  ...
Now what if I "cd dir; cat file" while contents cached?
  (fh2, "file") -> fh3 might already be in name cache
  stat(fh3) -> attr
  Compare mtime/ctime in attr to cached version.  Only read if no match.

p. 120:  "NFS uses a stateless protocol"--what does this mean?
  Obviously a file system cannot be stateless
  But some protocols (e.g., CIFS, 9p) keep client state on server
    E.g., each file open, close sent to server
    Server keeps equivalent of file descriptor table for each client
    Remote descriptors maybe invalidated by server reboot/network outage
      May be hard for client to rebuild state if files renamed
    Or... client crashes and server stuck keeping useless descriptors
  NFS requests are self-contained; no per-client context on server

What happens if server crashes?
  Client keeps retransmitting requests until server answers
  Once server reboots, will eventually get client's request
  Because "stateless", server can execute request with no prior context
What happens if client->server request dropped by network?
  Client keeps retransmitting (over UDP), so no issue
What happens if server->client reply dropped by network?
  Client keeps retransmitting (over UDP), sends second copy of request
  What does server do upon receiving second copy of a request?
    Each RPC starts with 32-bit XID
    Server keeps map of XID->reply message in a *replay cache*
    Reply to duplicate requests from replay cache, rather than reexecuting
    But the cache is state... protocol not stateless after all!
What happens if server->client dropped and server reboots?
  Reboot wipes server replay cache, so server reexecutes request
  Saving grace: protocol makes operations as *idempotent* as possible
    Means executing twice has same effect as executing once
    Example: "x = 1" is idempotent; "++x" is not
  Of calls on pp. 120-121, which are *not* idempotent?
    Not idempotent: remove, rename, link, mkdir, rmdir
    create is idempotent (makes exclusive create unreliable)
    Spec suggests symlink idempotent (maybe no error if content matches)
```

```
What if I say "echo message > file" on NFS file system?
  create (fh1, "file") -> fh2
  write (fh2, 0 [offset], "message") -> 2
What if server crashes during write?
  Okay, client will keep retransmitting until it gets reply
What if server crashes after replying to write?
  Client will stop retransmitting (since it gets write reply)
  Means server cannot reply to write until data safely on disk

Why did authors need to work on IP fragmentation code?
  Maximum Ethernet packet size is 1500 bytes [no jumbo frames in 1984]
  Each request/response constitutes one UDP packet
  What if you break large sequential writes into ~1400-byte requests?
    Smaller than FS block size.
    May have to read surrounding block
    Will likely lock buffer when first write request comes in
      Because writes synchronous, have to send to disk
    Almost certainly pay full disk rotation before servicing next write
  Solution: up to 9000-byte UDP packets (8192 data plus some header)
    Must be broken into multiple IP fragments to send over Ethernet

Even back-to-back 8KiB synchronous writes likely to be slow
  For good performance, want more concurrent write requests at server
What do they do?  Added block I/O daemon on client (p. 125)
  Not really good support for "asynchronous RPC" in kernel
  Instead fork off 4-16 block I/O daemons (biod)
  On write system call:
    If biod is available, hand it request return from syscall immediately
    biod will keep retransmitting until it gets a reply
  On fsync (and later close)
    Block waiting for all write requests to go through

Server hack:  nfsd daemon makes system call that never returns--why?
  Want to handle multiple incoming requests concurrently
  Again, async IO not so easy, and want to integrate with scheduler
    Easiest way to integrate with scheduler to be a process
    But NFS implemented in kernel, so just do it all in a big syscall

What happens when you run "mount server:/dir /mnt"?
  User-level mount program executes MOUNTPROC_MNT("/dir") to server
  User-level mountd program on server returns 32-byte NFS file handle
  mount program makes mount(2) syscall with server IP, fhandle, "/mnt"
  NFS mount handler allocates VFS, hangs it on /mnt's vnode (in root fs)

What's in an NFS file handle?
  Whatever the server wants--it's opaque to the client
  Must uniquely identify file, so:
    - File system ID (new field in superblock)
    - File system major/minor device number (usually)
    - Inode number
    - Generation number (new field in inode)

What's the point of a generation number?
  Generation number changes each time inode recycled
  Say server deletes a file while client has it open
    If client uses old handle, generation number wrong, gets ESTALE
    Not great, but better than reading/writing to unrelated file!
  Also makes file handles hard to guess (access control at mount time)

What is the security model?  Who enforces what permissions?
  Assume numeric user/group IDs are same on client and server
  On server, mountd restricts permitted clients (/etc/exports)
  If client sends valid file handle, NFS server assumes it is authorized
  Client tags each client RPC with local user/group IDs
    Server enforces access control based on claimed credentials
```

```
    But maps root to -2
Does this pose problems for UNIX semantics (goal #4)?
  UNIX does permission only at file open.  How does NFS handle:
  - fd = creat("lockfile", 0444); write(fd, ...); close(fd);
    Server allows write if owner of file matches, even if 0444 perms
  - fd = open(...); setuid(pw->pw_uid) /* drop privs */; read(fd, ...);"
    Client sends credentials from when file originally opened (p. 127)

How is evaluation?

Close-to-open consistency
```