6.033 notes, Appendix 4-B, Case study of the Network File System (NFS)

The original paper on NFS:
  Design and Implementation of the Sun Network File System
  Sandberg, Goldberg, Kleiman, Walsh, Lyon
  Usenix 1995

NFS is a neat system:
  NFS was very successful, and still is
  You can view much net fs research as fixing problems with NFS
  You'll use NFS in labs

Why would anyone want a network file system?
  Why not store your files on the local disk?

What's the architecture?
  Server w/ disk
  LAN
  Client hosts
  apps, system calls, RPCs

What RPCs would be generated by reading file.txt, e.g.:
  fd = open("file.txt", 0)
  read(fd, buf, 8192)
  close(fd)

What's in a file handle?
  i-number (basically a disk address)
  generation number
  file system ID

What's the point of the generation number?

Why not embed file name in file handle?

How does client know what file handle to send?
  Client got it from previous LOOKUP or CREATE
  Returned handle stored in appropriate vnode
  A file descriptor refers to a vnode

Where does the client get the very first file handle?

Why not slice the system at the system call interface?
  I.e. directly package up system calls in RPCs?
  UNIX semantics were defined in terms of files
    not just file names
    the files themselves have identities, i-number in the disk file system
  These refer to the same object even after a rename:
    File descriptor
    Home directory
    Cache contents
  So vnodes are there to remember file-handles

What RPCs would be generated by creating a file, e.g.:
  fd = creat("d/f", 0666);
  write(fd, "foo", 3);

```
  close(fd);

If a server crashes and reboots, will client requests still work?
  Will client's file handles still make sense?
  File handle == disk address of i-node.

What if the server crashes just after client sends it an RPC?

What if the server crashes just after replying to a WRITE RPC?

So what has to happen on the server during a WRITE?
  I.e. what does it do before it replies to the RPC?
  Data safe on disk.
  Inode with new block # and new length safe on disk.
  Indirect block safe on disk.
  Three writes, three seeks, 45 milliseconds.
  22 writes per second. 180 kb/sec.

How could we do better than 180 kb/sec?
  Write whole file sequentially at a few MB/sec.
  Then update inode &c at end.
  Why doesn't NFS do this?

NFS v3 unstable WRITE and COMMIT help solve this performance problem.
  server doesn't write to disk on WRITE, just caches, waits to batch
many writes
  server returns a "verifier" that changes on reboot
  client leaves written data dirty in its file cache
    remembers verifier
  in client close():
    make sure all WRITEs sent and replied to
    send COMMIT for file handle
    wait for reply
    if reply verifier != any cached block verifier, re-send all WRITEs
and COMMIT
    else free file cache blocks
  why in close()? for cache consistency among clients, in case
    server crashes after close() and reveals old data to other clients

What caches do typical NFS implementations have?
  And why exactly is each cache helpful?
  Server caches disk blocks, and maybe others.
  Client caches file content blocks, clean and dirty.
  Client caches file attributes.
  Client caches name-> fh mappings.
  Client caches directory contents.

You will need to think a little about the client caches for your labs.
  They suppress RPCs you might expect to receive at the server.
  They may become stale and cause client to see things different from
server and other clients.

What if client A has something cached, and client B changes it?

Examples where we might care about cache consistency?
  Two windows open, different clients,
    emacs -> make
```

```
    make -> run the program
  Or distributed app (cvs?) with its own locks.

Examples where we might not care about consistency?
  I just use one client workstation.
  Different users don't interact / share files.

Straw man consistency protocol:
  Every read() asks server if file has changed; if not, used cache
copy.
  Is that sufficient to make each read see latest write?
  What's the effect on performance?
  Do we need that much consistency?

Compromise: close-to-open consistency
  this is what most NFS clients do
  promise: if client A writes a file, then close()s it,
    then client B open()s the file, and reads it,
    client B's reads will reflect client A's writes.
  the point: clients only need to contact server during open() and
close()
    not every read and write

close-to-open consistency fixes the emacs/make example
  but the user has to wait until emacs says it's done writing!
  and cvs has to wait until close() returns before releasing lock

How NFS implements close-to-open consistency:
  taken from FreeBSD source; NFS spec doesn't say.
  client keeps file mtime and size for each cached file block
  close() starts WRITEs all file's dirty blocks
  close() waits for all of server's replies to those WRITEs
  open() always sends GETATTR to check file's mtime and size, caches
fattr
  read() uses cached blocks only if mtime/length have not changed
  client checks cached directory contents w/ GETATTR and ctime

name-to-filehandle cache may not be checked for consistency on each
LOOKUP
  you may get a stale file handle error if file was deleted
  or the wrong answer if file was renamed, and a new file created w/
same name

What prevents random people from sending NFS messages to my NFS server?
  Or from forging NFS replies to my client?

Would it be reasonable to use NFS in MIT Server?
  Security -- untrusted users with root on workstations
  Scalability -- how many clients can a server support?
    Writes &c always go through to server.
    Even for private files that will soon be deleted.
  Can you run it on a large complex network?
    How is it affected by latency? Packet loss? Bottlenecks?
```