NFS
Overview

• Sharing files is useful
• Network file systems give users seamless integration of a shared file system with the local file system
• Many options:
  – NFS, SMB/CIFS, AFS, etc.
• Security an important consideration
Big picture
(from Sandberg et al.)
Intuition

• Instead of translating VFS requests into hard drive accesses, translate them into remote procedure calls to a server

• Simple, right? I mean, what could possibly go wrong?
Challenges

- Server can crash or be disconnected
- Client can crash or be disconnected
- How to coordinate multiple clients accessing same file?
- Security
- New failure modes for applications
  - Goal: Invent VFS to avoid changing applications; use network file system transparently
Disconnection

• Just as a machine can crash between writes to the hard drive, a client can crash between writes to the server
• The server needs to think about how to recover if a client fails between requests
  – Ex: Imagine a protocol that just sends low-level disk requests to a distributed virtual disk.
  – What happens if the client goes away after marking a block in use, but before doing anything with it?
  – When is it safe to reclaim the block?
  – What if, 3 months later, the client tries to use the block?
Stateful protocols

• A stateful protocol has server state that persists across requests (aka connections)
  – Like the example on previous slide

• Server Challenges:
  – Knowing when a connection has failed (timeout)
  – Tracking state that needs to be cleaned up on a failure

• Client Challenges:
  – If the server thinks we failed (timeout), recreating server state to make progress
Stateless protocol

• The (potentially) simpler alternative:
  – All necessary state is sent with a single request
  – Server implementation much simpler!

• Downside:
  – May introduce more complicated messages
  – And more messages in general

• Intuition: A stateless protocol is more like polling, whereas a stateful protocol is more like interrupts
  – How do you know when something changes on the server?
NFS is stateless

• Every request sends all needed info
  – User credentials (for security checking)
  – File identifier and offset

• Each protocol-level request needs to match VFS-level operation for reliability
  – E.g., write, delete, stat
Challenge 1: Lost request?

• What if I send a request to the NFS server, and nothing happens for a long time?
  – Did the message get lost in the network (UDP)?
  – Did the server die?
  – Don’t want to do things twice, like write data at the end of a file twice

• Idea: make all requests *idempotent* or having the same effect when executed multiple times
  – Ex: write() has an explicit offset, same effect if done 2x
Challenge 2: Inode reuse

- Suppose I open file ‘foo’ and it maps to inode 30
- Suppose another process unlinks file ‘foo’
  - On a local file system, the file handle holds a reference to the inode, preventing reuse
  - NFS is stateless, so the server doesn’t know I have an open handle
    - The file can be deleted and the inode reused
    - My request for inode 30 goes to the wrong file! Uh-oh!
Generation numbers

- Each time an inode in NFS is recycled, its generation number is incremented
- Client requests include an inode + generation number
  - Detect attempts to access an old inode
Security

• Local uid/gid passed as part of the call
  – Uids must match across systems
  – Yellow pages (yp) service; evolved to NIS
  – Replaced with LDAP or Active Directory

• Root squashing: if you access a file as root, you get mapped to a bogus user (nobody)
  – Is this effective security to prevent someone with root on another machine from getting access to my files?
Removal of open files

• Unix allows you to delete an open file, and keep using the file handle; a hassle for NFS

• On the client, check if a file is open before removing it

• If so, rename it instead of deleting it
  – .nfs* files in modern NFS

• When file is closed, then delete the file

• If client crashes, there is a garbage file left which must be manually deleted
Changing Permissions

• On Unix/Linux, once you have a file open, a permission change generally won’t revoke access
  – Permissions cached on file handle, not checked on inode
  – Not necessarily true anymore in Linux
  – NFS checks permissions on every read/write---introduces new failure modes

• Similarly, you can have issues with an open file being deleted by a second client
  – More new failure modes for applications
Time synchronization

- Each CPU’s clock ticks at slightly different rates
- These clocks can drift over time
- Tools like ‘make’ use modification timestamps to tell what changed since the last compile
  - In the event of too much drift between a client and server, make can misbehave (tries not to)
- In practice, most systems sharing an NFS server also run network time protocol (NTP) to same time server
Cached writes

• A local file system sees performance benefits from buffering writes in memory
  – Rather than immediately sending all writes to disk
  – E.g., grouping sequential writes into one request

• Similarly, NFS sees performance benefits from caching writes at the client machine
  – E.g., grouping writes into fewer synchronous requests
Caches and consistency

• Suppose clients A and B have a file in their cache
• A writes to the file
  – Data stays in A’s cache
  – Eventually flushed to the server
• B reads the file
• Does B read the old contents or the new file contents?
Consistency

• Trade-off between performance and consistency
• Performance: buffer everything, write back when convenient
  – Other clients can see old data, or make conflicting updates
• Consistency: Write everything immediately; immediately detect if another client is trying to write same data
  – Much more network traffic, lower performance
  – Common case: accessing an unshared file
Close-to-open consistency

- NFS Model: Flush all writes on a close
- When you open, you get the latest version on the server
  - Copy entire file from server into local cache
- Can definitely have weirdness when two clients touch the same file
- Reasonable compromise between performance and consistency
NFS Evolution

• Basic, working design: NFS v2
• Version 3 (1995):
  – 64-bit file sizes and offsets (large file support)
  – Bundle file attributes with other requests to eliminate more stats
  – Other optimizations
  – Still widely used today

• Attempts to address many of the problems of V3
  – Security (eliminate homogeneous uid assumptions)
  – Performance

• Becomes a stateful protocol

• pNFS – proposed extensions for parallel, distributed file accesses

• Slow adoption
NFS Server Configuration

• Fairly easy: just add entries to /etc/exports

/etc/exports

/filer

130.245.153.4(rw,async,root_squash,subtree_check)

• Export folder /filer

• To IP address 130.245.153.4

• With options:
  – Read + write, asynchronous writes
  – Squash any writes that claim to come from user 0 (root)
  – And add extra checks that all client requests are under /filer
Client Configuration

• Just like any file system, configured in /etc/fstab

• Local FS:
  
  UUID=f874ae7f-2bd5-45dd-8921-400936352440 / ext4 errors=remount-ro 0 1

• NFS:
  
  camilla:/data/filer /filer nfs tcp,vers=3,noatime,nodiratime,noacl,retry=3 0 0
Other Client Configuration Options

• What happens when the server is disconnected?
  – Hard: Hang the **system** until the server comes back up
  – Soft: Return an error

• `noatime`: By default, Unix propagates access times to all root directories **even on a read**
  – Only required by a small number of programs
  – Very expensive, often disabled
Summary

• NFS is still widely used, in part because it is simple and well-understood
  – Even if not as robust as its competitors
• You should understand architecture and key trade-offs
• Basics of NFS protocol from paper