```
6.824 2006 Lecture 5: RPC Transparency

Goal: transparency
  Preserve original client code
  Preserve original server code
  RPC glues client/server together w/o changing behavior
  Programmer doesn't have to think about network

How well does this work?

Let's use NFS as a case study of RPC transparency

What is the non-distributed version of NFS?
  I.e. what are we letting RPC split up?
  apps, syscalls, kernel file system, local disk

Where's the module boundary that NFS RPCs cut at:
  In kernel, just below syscall interface.
  "vnode" layer.
  NFS client code essentially a set of stubs for system calls.
    Package up arguments, send them to the server.

Does NFS preserve client function call API? At syntactic level?
  Yes: no change to arguments / return values of system calls.

Does NFS use server-side implementation w/o changes?
  Fairly close.
  NFS provides in-kernel threads.
  They act much like processes making system calls.
  But NFS server required file system implementation changes:
    File handles instead of file descriptors.
    Generation numbers in on-disk i-nodes.
    User-id carried as arguments instead of implicit in process owner.
    Flag arguments indicating synchronous updates.

Does NFS preserve semantics of file system operations?
  It is not enough to preserve just the API.
  System calls must *mean* the same thing.
    Otherwise existing programs may compile and run but not be correct.

New semantics: server failure
  Before, open() only failed if file didn't exist.
  Now it (and all others) can fail if server has died.
    Apps have to know to retry or fail gracefully.
  *Or* open() could hang forever, which was never the case before.
    Apps have to know to set their own timeouts if they don't want to
hang.
  This is fundamental, not an NFS quirk.

New semantics: close() might fail if server disk out of space.
  Side effect of async write RPCs in client, for efficiency.
  Client only waits in close().
  close() never returns an error for local file system.
  So apps have to check close() for out-of-space, as well as write().
  This is caused by NFS trying to hide latency by batching.
    They could have made write() synchronous (and much slower).
```

```
New semantics: error returns from successful operations
  I call rename("a", "b") on an NFS file.
  Suppose server performs rename, crashes before sending reply.
  So NFS client re-sends rename().
  But now "a" doesn't exist, so I get an error.
  This never used to happen.
  Side-effect of NFS's statelessness.
    NFS server could remember all operations it has performed.
    But hard to fix: hard to keep that state consistent across crashes.
    Update state first? Or perform operation first?

New semantics: deletion of open files
  I open a file for reading.
  Some other client deletes it while I have it open.
  Old behavior: my reads still work.
  NFS behavior: my reads fail.
  Side-effect of NFS's statelessness.
    They could have fixed this; server could track opens.
    AFS keeps the state required to do this.

Preserving semantics leads to bad performance:
  If you write() local file, UNIX can delay write to disk.
    To collect multiple adjacent blocks and write with single seek.
  What if machine crashes?
    Then you lose both app and the dirty buffers in memory.
  Suppose we did the same with NFS server:
    WRITE request just write buffer in server's memory, then return.
  What if server crashed and rebooted?
    App would *not* have crashed, in fact would not notice.
    But its written data would mysteriously disappear.
  To solve this, NFS server does synchronous writes.
    Does not reply to write RPC until data is on disk.
    So if write returns and server crashes, data safe on disk.
  Takes three seeks: data, indirect block, i-node.
    i.e. 45 milliseconds.
    So 22 writes per second, 180 kb/sec.
  That's about 10% of max disk throughput.
  This is just NFS v2 lameness; AFS and NFS v3 fix it.

Improving performance leads to different consistency semantics:
  Suppose clients cache disk blocks when they read them.
  But writes always go through to the server.
  This is not enough:
    Suppose I'm using two workstations (w/ two X windows).
    Write out editor buffer on one workstation.
    Type make on the other workstation.
    Does make/compiler see my changes?
  Ask server "has the file changed" before every read()?
    Almost as slow as just reading from the server.
  NFS solution:
    Ask server "has the file changed" at each open().
    But don't ask for individual read()s after open().
    So if you change file while I have it open, I don't see your
changes.
  This is OK for editor/make, but not always what you want:
    make > out  (on compute server)
    tail -f out (on my workstation)
  Side-effect of NFS's statelessness.
    Server could remember who has read-cached blocks.
    Send them invalidation messages.

Security is totally different
  On local system: UNIX enforces read/write protections
    Can't read my files w/o my password
  How does NFS server know who the user is?
  What prevents random people from sending NFS requests to my NFS
server?
  Or from forging NFS replies to my client?
  Does it help for the server to look at the src IP address?
  Why aren't NFS servers ridiculously vulnerable?
    Hard to guess correct file handles.
  This is fixable:
    SFS, AFS, even some NFS variants do it.
    Require clients to authenticate themselves cryptographically.
    Very hard to reconcile with statelessness.

However, it turns out none of these issues prevent NFS from being
useful.
  People fix their programs to handle new semantics.
  Or install firewalls for security.
  And get most advantages of transparent client/server.

Multi-module example
  NFS very simple: only one server, only one data type (file handle).
  What if symmetric interaction, lots of data types?
  suppose program starts w/ three modules in same address space
  example modules:
    web front end; customer DB; order DB
  class connection { int fd; int state; char *buf; }
  easy to pass object references among all three
    e.g. pointer to current connection
  what if you split all three off into separate servers
    how do you pass a "class connection"?
      Stub generator would just send the structure elements?
    what if processing for connection goes through order DB,
      then customer DB, then back to front end to send reply
    front end only knows *contents* of passed connection object
      *real* connection may have changed...
    so we actually wanted to pass object references, not object
contents
    NFS solves this with file handles
      But got no support from the RPC package

Areas of RPC non-transparency
  1. Partial failure, network failure
  2. Latency
  3. Efficiency/semantics tradeoff
  4. Security. You can rarely deal with it transparently.
  5. Pointers. Write-sharing. Portable object references.
  6. Concurrency (if multiple clients)
  Solution 1: expose RPC to application
  Solution 2: work harder on transparent RPC

Conclusion
```

```
  Automatic marshaling has been a big success
  Mimicing procedure call interface is not that useful
  Attempt at full transparency has been mostly a failure
  But you can push this pretty hard: Network Objects, Java RMI
```