

Remote Procedure Call (RPC) and Transparency

Brad Karp
UCL Computer Science



CS GZ03 / M030
15th October, 2010

Transparency in Distributed Systems

- Programmers accustomed to writing code for a single box
- Transparency: retain “feel” of writing for one box, when writing code that runs distributedly
- Goals:
 - Preserve original, unmodified client code
 - Preserve original, unmodified server code
 - RPC should glue together client and server without changing behavior of either
 - Programmer shouldn’t have to think about network

Transparency in Distributed Systems

**How achievable is true transparency?
We will use NFS as a case study.
But first, an introduction to RPC itself.**

- Goals:
 - Preserve original, unmodified client code
 - Preserve original, unmodified server code
 - RPC should glue together client and server without changing behavior of either
 - Programmer shouldn't have to think about network

Remote Procedure Call: Central Idea

- Within a single program, running on a single box, well-known notion of **procedure call** (aka function call):
 - Caller pushes arguments onto stack
 - Jumps to address of callee function
 - Callee reads arguments from stack
 - Callee executes, puts return value in register
 - Callee returns to next instruction in caller
- **RPC aim: let distributed programming look no different from local procedure calls**

RPC Abstraction

- Library makes an API available to locally running applications
- Let servers **export** their local APIs to be accessible over the network, as well
- On client, procedure call generates request over network to server
- On server, called procedure executes, result returned in response to client

RPC Implementation Details

- Data types may be different sizes on different machines (e.g., 32-bit vs. 64-bit integers)
- Little-endian vs. big-endian machines
 - Big-endian: 0x11223344 is 0x11, 0x22, 0x33, 0x44
 - Little-endian is 0x44, 0x33, 0x22, 0x11
- Need mechanism to pass procedure parameters and return values in **machine-independent fashion**
- Solution: **Interface Description Language (IDL)**

Interface Description Languages

- Compile interface description, produces:
 - Types in native language (e.g., Java, C, C++)
 - Code to **marshal** native data types into machine-neutral byte streams for network (and vice-versa)
 - **Stub** routines on client to forward local procedure calls as requests to server
- For Sun RPC, IDL is XDR (eXternal Data Representation)

Example: Sun RPC and XDR

- Define API for procedure calls between client and server in XDR file, e.g., `proto.x`
- Compile: `rpcgen proto.x`, producing
 - `proto.h`: RPC procedure **prototypes**, argument and return value **data structure definitions**
 - `proto_clnt.c`: per-procedure client stub code to **send RPC request to remote server**
 - `proto_svc.c`: server stub code to **dispatch** RPC request to specified procedure
 - `proto_xdr.c`: argument and result **marshaling/unmarshaling** routines, host-network/network-host **byte order conversions**

Example: Sun RPC and XDR

- Define API for procedure calls between client and server in XDR file, e.g., `proto.x`
- Compile: `rpcgen proto.x`, producing
 - `proto.h`: RPC procedure **prototypes**, argument and return value **data structure definitions**

Let's consider a simple example...

- `proto_svc.c`: server stub code to **dispatch** RPC request to specified procedure
- `proto_xdr.c`: argument and result **marshaling/unmarshaling** routines, host-network/network-host **byte order conversions**

Sun RPC and XDR: Programming Caveats

- Server routine return values **must always be pointers** (e.g., `int *`, not `int`)
 - should declare return value `static` in server routine
- Arguments to server-side procedures are **pointers to temporary storage**
 - to store arguments beyond procedure end, must copy **data**, not merely pointers
 - in these cases, typically **allocate memory** for copy of argument using `malloc()`
- If new to C, useful background in Mark Handley's "C for Java programmers" tutorial:
 - <http://moodle.ucl.ac.uk/file.php/11513/c-intro.pdf>
 - § 2.9 – 2.13 describe memory allocation

Sun RPC and XDR: Programming Caveats

- Server routine return values **must always be pointers** (e.g., `int *`, not `int`)
 - should declare return value `static` in server routine
- Arguments to server-side procedures are **pointers to temporary storage**
 - to store arguments beyond procedure end, must copy

Now, back to our NFS case study...

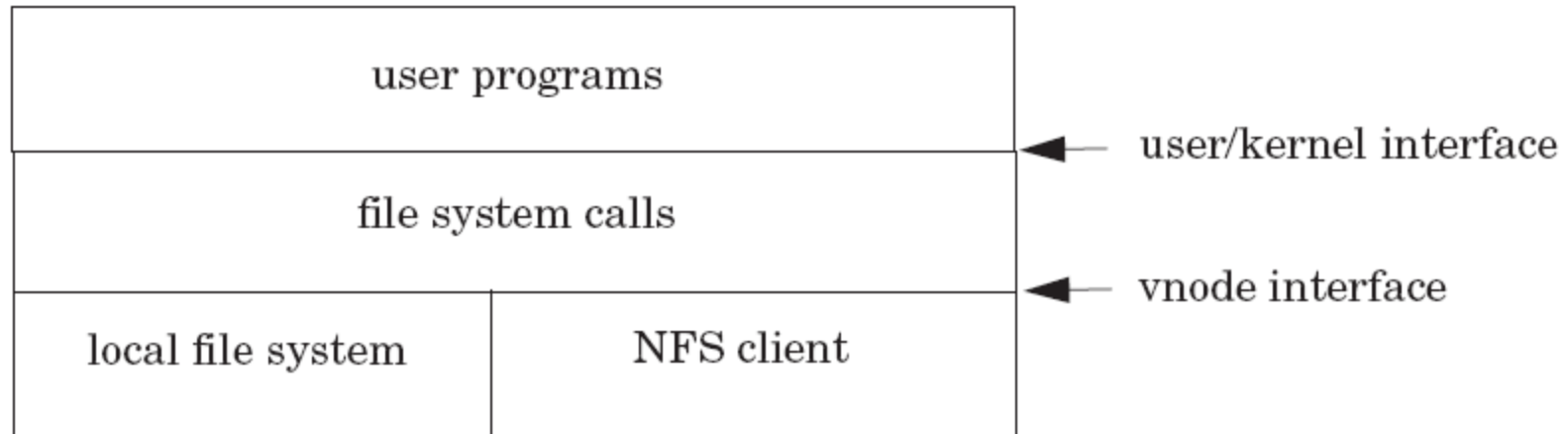
- If new to C, useful background in Mark Handley's "C for Java programmers" tutorial:
 - <http://moodle.ucl.ac.uk/file.php/11513/c-intro.pdf>
 - § 2.9 – 2.13 describe memory allocation

“Non-Distributed” NFS

- Applications
- Syscalls
- Kernel filesystem implementation
- Local disk

- RPC must “split up” the above
- **Where does NFS make the split?**

NFS Structure on Client



- NFS splits client at vnode interface, below syscall implementation
- Client-side NFS code essentially **stubs** for system calls:
 - Package up arguments, send them to server

NFS and Syntactic Transparency

- Does NFS preserve the syntax of the client function call API (as seen by applications)?
 - Yes!
 - Arguments and return values of system calls not changed in form or meaning

NFS and Server-Side Transparency

- Does NFS require changes to pre-existing filesystem code on server?
 - Some, but not much.
 - NFS adds **in-kernel threads** (to block on I/O, much like user-level processes do)
 - Server filesystem implementation changes:
 - **File handles over wire, not file descriptors**
 - **Generation numbers added to on-disk i-nodes**
 - **User IDs carried as arguments, rather than implicit in process owner**
 - **Support for synchronous updates (e.g., for WRITE)**

NFS and File System Semantics

- You don't get transparency merely by preserving the same API
- System calls must **mean** the same thing!
- **If they don't, pre-existing code may compile and run, but yield incorrect results!**
- **Does NFS preserve the UNIX filesystem's semantics?**
- **No! Let us count the ways...**

NFS's New Semantics: Server Failure

- On one box, `open()` only fails if file doesn't exist
- Now `open()` and all other syscalls can fail if server has died!
 - Apps must know how to retry or fail gracefully
- **Or** `open()` could hang forever—never the case before!
 - Apps must know how to set own timeouts if don't want to hang
- This is **not** a quirk of NFS—it's fundamental!

NFS's New Semantics: close() Might Fail

- Suppose server out of disk space
- But client WRITES asynchronously, only on close(), for performance
- Client waits in close() for WRITES to finish
- close() never returns error for local fs!
 - Apps must check not only write(), but also close(), for disk full!
- Reason: NFS batches WRITES
 - If WRITES were synchronous, close() couldn't fill disk, but performance would be awful

NFS's New Semantics: Errors Returned for Successful Operations

- Suppose you call `rename("a", "b")` on file in NFS-mounted fs
- Suppose server completes RENAME, crashes before replying
- NFS client resends RENAME
- "a" doesn't exist; error returned!
- **Never happens on local fs...**
- Side effect of statelessness of NFS server:
 - Server could remember all ops it's completed, but that's hard
 - Must keep that state consistent and persistent across crashes (i.e., on disk)!
 - Update the state first, or perform the operation first?

NFS's New Semantics: Deletion of Open Files

- Client A open()s file for reading
- Client B deletes it while A has it open
- Local UNIX fs: A's subsequent reads work
- NFS: A's subsequent reads fail
- Side effect of statelessness of NFS server:
 - Could have fixed this—server could track open ()s
 - AFS tracks state required to solve this problem

Semantics vs. Performance

- Insight: **preserving semantics produces poor performance**
- e.g., for write() to local fs, UNIX can delay actual write to disk
 - Gather writes to multiple adjacent blocks, and so write them with one disk seek
 - If box crashes, you lose **both** the running app and its dirty buffers in memory
- Can we delay WRITES in this way on NFS server?

NFS Server and WRITE Semantics

- Suppose WRITE RPC stores client data in buffer in memory, returns success to client
- Now server crashes and reboots
 - App doesn't crash—in fact, doesn't notice!
 - And **written data mysteriously disappear!**
- Solution: NFS server does **synchronous WRITES**
 - Doesn't reply to WRITE RPC until data on disk
 - If write() returns on client, even if server crashes, data safe on disk
 - Per previous lecture: 3 seeks, 45 ms, 22 WRITES/s, 180 KB/s max throughput!
 - < 10% of max disk throughput
- **NFS v3 and AFS fix this problem (more complex)**

Semantics vs. Performance (2)

- Insight: **improving performance changes consistency semantics!**
- Suppose clients cache disk blocks when they read them
- But writes always go through to server
- **Not enough to get consistency!**
 - Write editor buffer on one box, make on other
 - **Do make/compiler see changes?**
- Ask server “has file changed?” at every read()?
 - **Almost as slow as just reading from server...**

NFS: Semantics vs. Performance

- NFS' solution: close-to-open consistency
 - Ask server “has file changed?” at each open()
 - Don't ask on each read() after open()
 - If B changes file while A has it open, A doesn't see changes
- OK for emacs/make, but **not always what you want:**
 - `make > make.log` (on server)
 - `tail -f make.log` (on my desktop)
- Side effect of statelessness of NFS server
 - Server could track who has cached blocks on reads
 - Send “invalidate” messages to clients on changes

Security Radically Different

- Local system: UNIX enforces read/write protections per-user
 - Can't read my files without my password
- How does NFS server authenticate user?
- Easy to send requests to NFS server, and to forge NFS replies to client
- Does it help for server to look at source IP address?
- **So why aren't NFS servers ridiculously vulnerable?**
 - Hard to guess correct file handles!

Security Radically Different

- Local system: UNIX enforces read/write protections per-user
 - Can't read my files without my password
- How does NFS server authenticate user?
- Easy to send requests to NFS server, and to forge NFS replies to client
- Does it help for server to look at source IP address?

Fixable: SFS, AFS, some NFS versions use cryptography to authenticate client

Very hard to reconcile with statelessness!

NFS Still Very Useful

- People **fix programs to handle new semantics**
 - Must mean NFS useful enough to motivate them to do so!
- People **install firewalls for security**
- NFS still gives **many advantages of transparent client/server**

Multi-Module Distributed Systems

- NFS in fact rather simple:
 - One server, one data type (file handle)
- What if **symmetric interaction, many data types?**
- Say you build system with three modules in one address space:
 - **Web front end, customer DB, order DB**
- Represent user connections with object:

```
class connection {  
    int fd; int state; char *buf; }
```
- **Easy to pass object references among three modules (e.g., pointer to current connection)**

Multi-Module Distributed Systems

- NFS in fact rather simple:
 - One server, one data type (file handle)

What if we split system into three separate servers?

What if we split system into three separate servers?

- Web front end, customer DB, order DB
- Represent user connections with object:

```
class connection {  
    int fd; int state; char *buf; }
```
- Easy to pass object references among three modules (e.g., pointer to current connection)

Multi-Module Systems: Challenges

- How do you pass `class connection` between servers?
 - Could RPC stub just send object's elements?
- What if processing flow for connection goes: order DB -> customer DB -> front end to send reply?
- Front end only knows **contents** of passed connection object; underlying connection may have changed!
- Wanted to pass object references, not object contents
- NFS solution: file handles
 - No support from RPC to help with this!

RPC: Failure Happens

- New failure modes not seen in simple, same-host procedure calls:
 - Remote server failure
 - Communication (network) failure
- RPCs can return “failure” instead of results
- Possible failure outcomes:
 - Procedure didn't execute
 - Procedure executed once
 - Procedure executed multiple times
 - Procedure partially executed
- Generally, “at most once” semantics preferred

Achieving At-Most-Once Semantics

- Risk: Request message lost
 - Client must retransmit requests when no reply received
- Risk: Reply message lost
 - Client may retransmit previously executed request
 - OK when operations **idempotent**; some aren't, though (e.g., **"charge customer"**)
 - Server can keep **"replay cache"** to reply to repeated requests without re-executing them

Summary: RPC Non-Transparency

- Partial failure, network failure
- Latency
- Efficiency/semantics tradeoff
- Security—rarely transparent!
- Pointers: write-sharing, portable object references
- Concurrency (if multiple clients)
- Solutions:
 - Expose “remoteness” of RPC to application, or
 - Work harder to achieve transparent RPC

Conclusions

- Of RPC's goals, automatic marshaling most successful
- Mimicking procedure call interface in practice not so useful
- Attempt at full transparency mostly a failure!
 - (You can try hard: consider Java RMI)
- Next time: implicit communication through distributed shared memory!