RPC

15-712 Fall 2007

David Andersen

Making RPC Real

- Nelson84: First solid implementation of RPC
- Idea came years before:
  - Nelson's 1981 CMU thesis, "Remote Procedure Call"
    - *The* treatment of design options for RPC.
  - And earlier (e.g., Liskov79, etc.)
- Major contribution: Making it real
  - Failure semantics
  - Dealing with pointers
  - Language issues
  - Binding (finding the server)
  - Actual wire protocols
  - Integrity & Security
    - Hard to avoid tangents when building real system

Context

- At that point, Xerox PARC was *huge* force in experimental CS
  - 1979: "Alto: A Personal Computer"
  - The mouse & GUI... (sometimes they didn't capitalize too well on their ideas...)
- And remember the hardware
  - Ran on a Dorado, a "very powerful" successor to the Alto
    - about the speed of a 386.
    - about 1000x slower than today's machines
    - 80MB hard disk, 3Mbit/sec ethernet, 56Kbit/sec internet

RPC Basics

- Ease of (programmer) use:
  - Local and remote programming with same interface abstraction
- Flow:
  - Caller blocks, arguments are marshalled & sent over net
  - Callee unmarshalls & executes; results marshalled & returned
  - Caller unmarshalls and continues
- Code looks just like local code!

Why RPC?

- Familiar, simple semantics
  - Easier to program => better programs
- Does some of the "grunt-work"
  - Bad: Constantly writing marshalling & unmarshalling code
  - etc.
- Efficiency?
  - Maybe. But marshalling overhead can be high.
    - "Admits efficient impl" - Yes, but came years later via optimizing IDL compilers
- Generality
  - Mostly: No pointer support, etc. -- data structures must be simple
  - Partitioning local/remote separate from code modularity

Alternatives

- Messages
  - Different control mechanism for remote side
- Remote fork
  - Large granularity!
  - What data do you reply with? Entire contents of memory? Imprecise -> hard to be efficient
- Distributed shared memory
  - Needs HW for efficiency
    - Very long-running research (into late 90s)
    - very hard! Simple interface, but hides a _ton_ of details; has weird unintentional sharing semantics (page granularity); very hard to make efficient. RPC and message passing mostly won, except RDMA and CC-NUMA.

RPC == Messages, really

- "On the Duality of Operating Systems Structures" (H.C. Lauer, R. M. Needham; Proc. 2nd International Symposium on Operating Systems, Oct 1978)
- *Functionally*, RPC is the same as messaging (and it's implemented as messages under the hood)
  - Difference: Human productivity and familiarity of interface
  - RPC middleware is more powerful & pervasive
    - Client/server infrastructures mainly RPC (commercial)
      - Sunrpc -> NFS, etc. CORBA. MS RPC.
    - HPC programming mostly message passing (faster, p2p, more flexible communication models -- pass the message in a ring, etc.)

Making it easy: Stubs

- Describe interface in IDL (Interface Definition Language)
  - Think C header files as a decent example
- Compiler automatically turns IDL into "stub"
  - Stub has same function signature as original call
    - But does a the RPC magic under the hood
  - marshal arguments
  - call RPC runtime, do whatever binding/resolution/etc.
  - send call, wait, unmarshal, return arguments

Flow in an RPC system

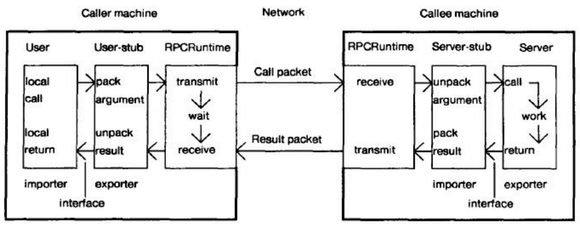

Fig. 1. The components of the system, and their interactions for a simple call.

Binding (Rendezvous)

- How does client find appropriate server?
  - Touches on fundamental issues of naming & indirection!
- Cedar used a registry, Grapevine
  - Originally written for email handling. :)
  - Server publishes interface: type, instance
  - Client names service (& maybe instance) -> network addr
    - Permits load balancing and nearest-server selection (anycast)
    - Cool stuff, now common, e.g., LDAP, ActiveDir
  - Simpler schemes work too: DNS, portmap, IANA
  - Still a source of complexity & insecurity

Binding: Time

- Communication:
  - Step 1: Look up remote receiver
  - Step 2: Communicate with returned address
    - Ensures consistent communication
  - This model repeated at the process addressing level, again to enable efficient but consistent communication
  - *could* embed address directly (but why?)
- B&N skipped one form: late binding of every req
  - Less efficient (must have resolv info in every req)
  - Potentially useful in some scenarios (e.g., sensor query)

Marshalling

- Must represent data "on the wire"
  - Good: Processor/arch dependence (big/little endian)
    - Sometimes ASCII vs. EBCDIC, though less common
    - Sometimes number representation (XML does some)
    - Can get arbitrarily crazy, but only xml does. :)
    - Sometimes called "presentation layer" in networks
    - ex: Sun XDR (external data representation)
  - Tradeoff: always canonical? optimize for instances?

Marshaling Data Structs.

- No shared memory! How to deal with pointers?
  - Simulate it: RPC for all server dereferences? (ugh, slow)
  - Shallow copy the structure?
    - Fragile - tricky for programmers
  - Deep copy the structure?
    - Slow, potentially incorrect if dynamically written struct
  - Disallow?
    - Very common.
    - Makes RPC less transparent, but common compromise
- Forces programmers to plan more about local/ remote and data representation

Communication

- Most communication: 1 packet, 1 response
- Reliability: RPC-specific. (Tricky question)
- Bigger packets? "Stop-and-wait ARQ"
  - Send a packet. Wait for ACK or timeout.
  - Repeat.
- Incredibly inefficient for bulk data transfer on wide-area.
  - Not b/c of extra packets as B&N suggest
  - But because it requires *many* round trips
  - Need real congestion control, e.g., TCP, for efficient bulk data transfer
  - But doesn't matter for most small req/resp uses of RPC!
- Remember their assumptions: single local network, 1 switch
  - Today's environment has changed. Wide-area & campus-wide client-server much more common

Semantics

- B&N chose to emulate *very closely* function calls
  - Explicitly decided against timeout support
  - Defined an RPC to block the client during call
  - This is actually unfortunate
    - Complex, robust systems need more control over remote component timeouts, etc.
    - Ex: Re-captcha system issues synchronous javascript load
      - Forced synchrony prevents easy impl. of fast failover.
- Distributed systems are *not* local. Must still deal with failures, timeouts, delays, etc.
- RPC doesn't make this easier. Fundamentally tough!

Server failures

- Communication is connectionless, but
- Explicit failures if server crashes and restarts
  - So clients can learn what happened
- Good idea?
- Idempotent operations via repeat/reply cache
  - ID on each request
  - "At most once" semantics.
  - (Any stronger guarantees *very* hard to do with losses)
  - Pretty easy to program to.

Server Model

- Pre-forked pool of server processes
  - Why? Saves process creation overhead for reqs
  - Permits consistent client/process communication if wanted
- This technique re-emerged in Apache web server
- Digression: server models
  - Fork (and pre-forked as optimization)
  - Threaded
  - Events
  - Long-running debate. Nearly religious.

Other optimizations

- Bypassing the lower layers
  - Long-standing design debate: cross-layer optimization vs. modularity
    - Can be *very* fast by "cheating"
    - But ties you to specific hardware! (ugh!)
    - Make very sure you need that speed...
- Better generalization: RDMA type approaches
  - Principled mechanisms for skipping layer computations
  - Requires (somewhat complex) HW support

Evaluation

- Microbenchmarks for call-reply latency
  - Null RPC, N args, words, N=1, 2, 4, 10, 40, 100
    - Set of things evaluated kind of standard
    - Modern analysis would have been a bit more statistically sophisticated. (no max? doesn't that matter a lot? :)
      - fairness: graphing tools got a lot better since '84.
- Minimal real eval: no stats, no app. benchmarks
- Not compared to much
  - 10x to 100x slower than local procedure call
- "This is what we did; it is possible"
  - FULLY implemented and in use by PARC!