Remote Procedure Calls

Carnegie Mellon University
15-440 Distributed Systems

Administrivia

- Readings are now listed on the syllabus
- See .announce post for some details
  - The book covers a ton of material pretty quickly; readings are diced up pretty finely to try to hit best info.

Building up to today

- 2x ago: Abstractions for *communication*
  - example: TCP masks some of the pain of communicating across unreliable IP
- Last time: Abstractions for *computation*

Reminder about last time

- Processes: A resource container for execution on a single machine
- Threads: One "thread" of execution through code. Can have multiple threads per process.
  - Impl as userland, kernel; each has diff. benefits

Threads - impl

- Use:
  - Exploit multiple processors
  - Hide long delays
  - Run long ops concurrent with short ones to improve response time (UI events, etc)
- Thread *interface*
  - Creating and managing threads
  - Provide ways to avoid race conditions for updates to shared data

pthreads interface

- threads
  - create
  - join == wait until it's done
- mutex
  condition variables
  coming up in next lecture

On to today...

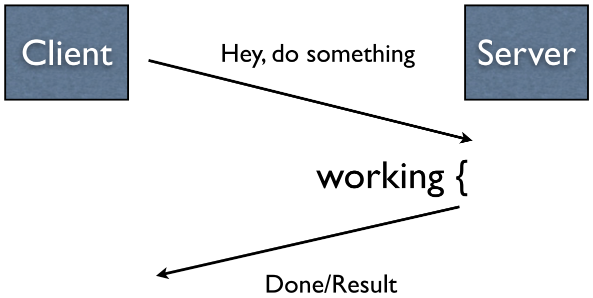
Splitting computation across the network

- We've looked at primitives for computation and for communication.
  Today, we'll put them together
- Key question:

What programming abstractions work well to split work among multiple networked computers?

(caveat: we'll be looking at <u>many</u> possible answers to this question...)

Common communication pattern



Writing it by hand...

- eg, if you had to write a, say, password cracker

```
struct foomsg {
  u_int32_t len;
}

send_foo(char *contents) {
   int msglen = sizeof(struct foomsg) + strlen(contents);
   char buf = malloc(msglen);
   struct foomsg *fm = (struct foomsg *)buf;
   fm->len = htonl(strlen(contents));
   memcpy(buf + sizeof(struct foomsg),
          contents,
          strlen(contents));
   write(outsock, buf, msglen);
}
```

Then wait for response, etc.

RPC

- A type of client/server communication
- Attempts to make remote procedure calls look like local ones

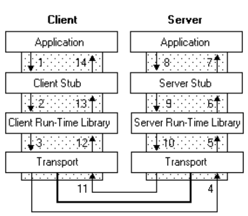


figure from Microsoft MSDN

```
{ ...
  foo()
}
void foo() {
  invoke_remote_foo()
}
```

RPC Goals

- Ease of programming
- Hide complexity (we'll get to next)
- automate a lot of task of implementing
- Familiar model for programmers (just make a function call)

Historical note: Seems obvious in retrospect, but RPC was only invented in the '80s. See Birrell & Nelson, "Implementing Remote Procedure Call" ... or
Bruce Nelson, Ph.D. Thesis, Carnegie Mellon University: Remote Procedure Call., 1981 :)

But it's not always simple

- Calling and called procedures run on different machines, with different address spaces
  - And perhaps different environments .. or operating systems ..
- Must convert to local representation of data
- Machines and network can fail

Marshaling and Unmarshaling

- (From example) hotnl() -- "host to network-byte-order, long".
  - network-byte-order (big-endian) standardized to deal with cross-platform variance
- Note how we arbitrarily decided to send the string by sending its length followed by L bytes of the string? That's marshalling, too.
- Floating point...
- Nested structures? (Design question for the RPC system - do you support them?)
- Complex datastructures? (Some RPC systems let you send lists and maps as first-order objects)

"stubs" and IDLs

- RPC stubs do the work of marshaling and unmarshaling data
- But how do they know how to do it?
- Typically: Write a description of the function signature using an *IDL* -- interface definition language.
  - Lots of these. Some look like C, some look like XML, ... details don't matter much.

SunRPC

- Venerable, widely-used RPC system
- Defines "XDR" ("eXternal Data Representation") -- C-like language for describing functions -- and provides a compiler that creates stubs

```
struct fooargs {
  string msg<255>;
  int baz;
}
```

And describes functions

```
program FOOPROG {
  version VERSION {
    void FOO(fooargs) = 1;
    void BAR(barargs) = 2;
  } = 1;
} = 9999;
```

More requirements

- Provide reliable transmission (or indicate failure)
  - May have a "runtime" that handles this
- Authentication, encryption, etc.
  - Nice when you can add encryption to your system by changing a few lines in your IDL file
    - (it's never really that simple, of course -- identity/key management)

Big challenges

- What happens during communication failures? Programmer code still has to deal with exceptions! (Normally, calling foo() to add 5 + 5 can't fail and doesn't take 10 seconds to return)
- Machine failures?
  - Did server fail before/after processing request?? Impossible to tell, if it's still down...
- It's impossible to hide all of the complexity under an RPC system. But marshaling/unmarshaling support is great!

<break>

RPC Context

- In lab 2, you'll first implement a remote lock server
  - Supports 2 operations: acquire(lock), release(lock). Implemented using RPC.

RPC failures

- Request from cli -> srv lost
- Reply from srv -> cli lost
- Server crashes after receiving request
- Client crashes after sending request

RPC semantics

- At-least-once semantics
  - Keep retrying...
- At-most-once
  - Use a sequence # to ensure idempotency against network retransmissions
  - and remember it at the server

```
At-least-once versus at-most-once?
  let's take an example: acquiring a lock
    if client and server stay up, client receives lock
    if client fails, it may have the lock or not (server needs a plan!)
    if server fails, client may have lock or not
      at-least-once: client keeps trying
      at-most-once:  client will receive an exception
    what does a client do in the case of an exception?
      need to implement some application-specific protocol
        ask server, do i have the lock?
    server needs to have a plan for remembering state across reboots
        e.g., store locks on disk.
    at-least-once (if we never give up)
      clients keep trying.  server may run procedure several times
      server must use application state to handle duplicates
        if requests are not idempotent
    but difficult to make all request idempotent
      e.g., server good store on disk who has lock and req id
        check table for each requst
       even if server fails and reboots, we get correct semantics
  What is right?
    depends where RPC is used.
       simple applications:
         at-most-once is cool (more like procedure calls)
       more sophisticated applications:
         need an application-level plan in both cases
     not clear at-once gives you a leg up
  => Handling machine failures makes RPC different than procedure calls
```

comparison from Kaashoek, 6.842 notes

Implementing at-most-once

- At-least-once: Just keep retrying on client side until you get a response.
  - Server just processes requests as normal, doesn't remember anything. Simple!
- At-most-once: Server might get same request twice...
  - Must re-send *previous* reply and not process request (implies: keep cache of handled requests/responses)
  - Must be able to identify requests
  - Strawman: remember *all* RPC IDs handled. -> Ugh! Requires infinite memory.
  - Real: Keep sliding window of valid RPC IDs, have client number them sequentially.

Exactly-Once?

- Sorry - no can do *in general*.
- Imagine that message triggers an external physical thing (say, a robot fires a nerf dart at the professor)
- The robot could crash immediately before or after firing and lose its state. Don't know which one happened. Can, however, make this window very small.

Implementation Concerns

- As a general library, performance is often a big concern for RPC systems
- Major source of overhead: copies and marshaling/unmarshaling overhead
- Zero-copy tricks:
  - Representation: Send on the wire in native format and indicate that format with a bit/byte beforehand. What does this do? Think about sending uint32 between two little-endian machines
  - Scatter-gather writes (writev() and friends)

Dealing with Environmental Differences

- If my function does: read(foo, ...)
- Can I make it look like it was really a local procedure call??
- Maybe!
  - Distributed filesystem...
- But what about address space?
  - This is called distributed shared memory
  - People have kind of given up on it - it turns out often better to admit that you're doing things remotely

Complex / Pointer Data Structures

- Very few low-level RPC systems support
  - C is messy about things like that -- can't always understand the structure and know where to stop chasing
  - One way was to send pointers and use DSM, but ...
- Java RMI (and many other higher-level languages) allows sending objects as part of an RPC
  - But be careful - don't want to send megabytes of data across network to ask simple question!