Distributed Systems [Fall 2013]

Lec 4: Remote Procedure Calls (RPC)

Slide acks: Dave Andersen, Jinyang Li
(http://www.cs.cmu.edu/~dga/15-440/F10/lectures/05-rpc.pdf,
http://www.news.cs.nyu.edu/~jinyang/fa10/notes/ds-lec2.ppt)

News

- HW 2 has been released
  - Due in two weeks
  - Much longer than HW1!
  - And it's GRADED!
  - So start very early

- Yu will give background now

YFS Lab Series Background

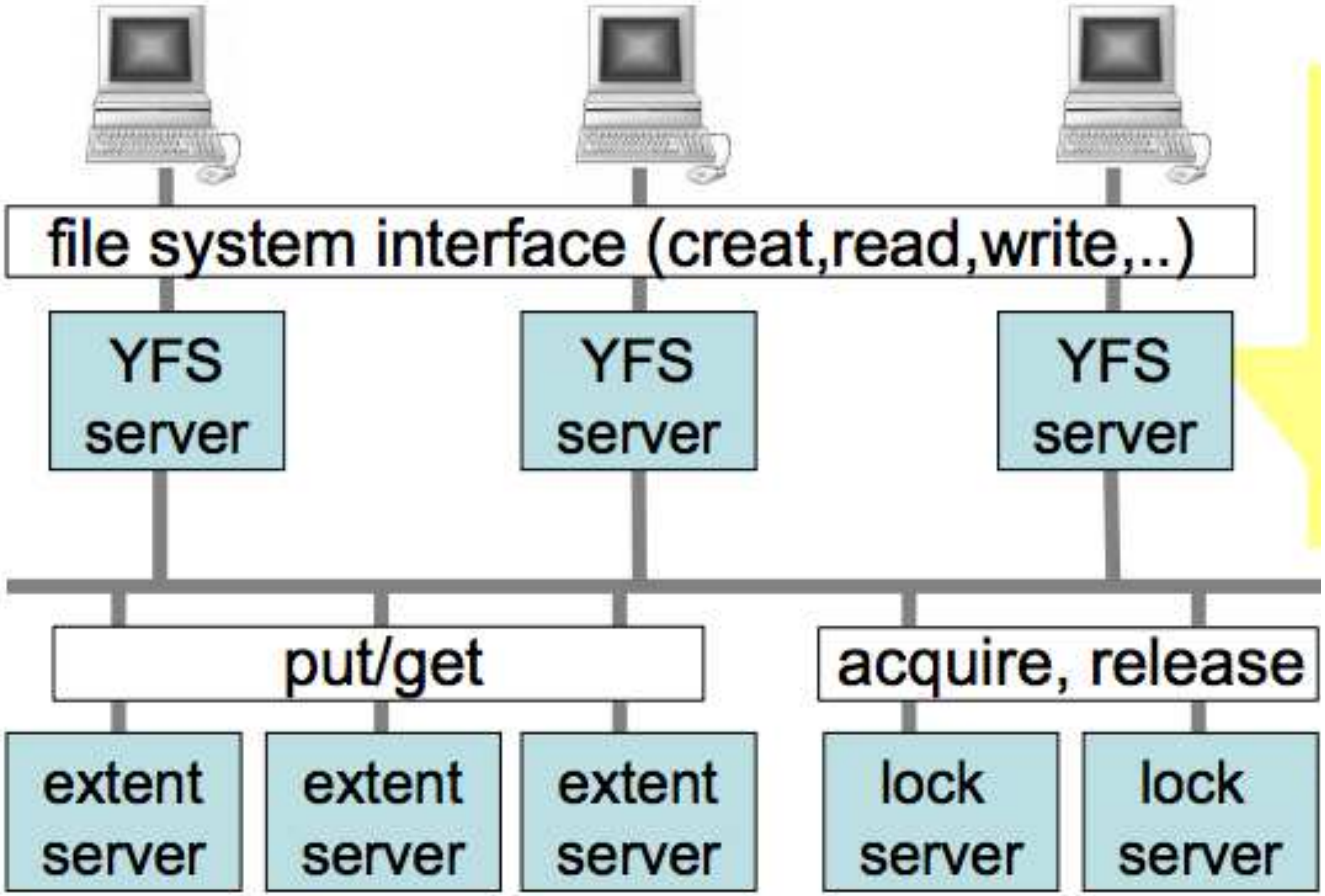
YFS

- Instructional distributed file system developed by MIT after a research distributed file system, called Frangipani
  - Analogous to xv6 for OS courses
  - When we discuss YFS, we really refer to Frangipani (or a simplified version thereof)
  - Thekkath, Chandramohan A., Timothy Mann, and Edward K. Lee. "Frangipani: A scalable distributed file system." ACM SIGOPS Operating Systems Review. Vol. 31. No. 5. ACM, 1997.

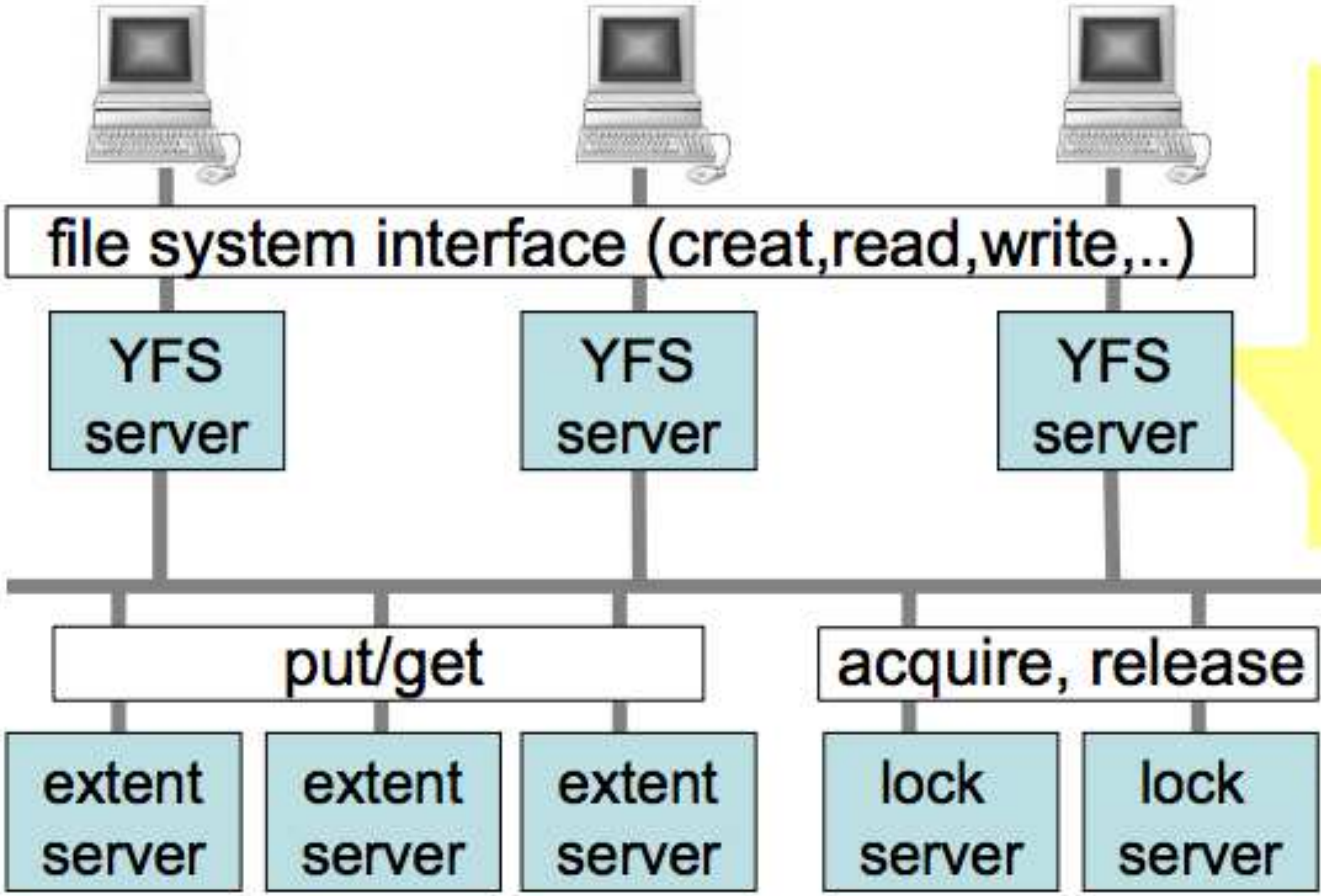
YFS Design Goals

- Aggregate many disks from many servers
- Incrementally scalable
- Tolerates and recovers from node, network, disk failures

Design


file system interface (creat,read,write,..)
YFS server
YFS server
YFS server
put/get
acquire, release
extent server
extent server
extent server
lock server
lock server

Design


file system interface (creat,read,write,..)
YFS server
YFS server
YFS server
put/get
acquire, release
extent server
extent server
extent server
lock server
lock server

Lock Service

- Resolve concurrent read/write issues
  - If one client is writing to a file while another client is reading it, inconsistency appears.
- Lock server grants/releases locks to client upon request
- Becomes scalable with more instances
- All the actual data are provided by the extent server

Some Specific Questions I

- How to make a file system in user space?
  - We need to build a real file system that can be mounted. How?
  - Use FUSE library (http://fuse.sourceforge.net/)
  - We will provide you with skeleton code
  - You are responsible for filling in the actual function implementations.

Some Specific Questions II

- How do nodes communicate with each other?
  - Use the in-house RPC library
  - No more low-level socket programming!
  - It has drawbacks though, next homework will ask you to fix it.
- How do I store the actual data?
  - The data does not need to be persistent.
  - Store them in memory is OK.

Lab Schedule

- Lab 1: C++ warm up (Passed)
- Lab 2: Lock server and reliable RPC
- Lab 3: File server
- Lab 4: Cache Locks
- Lab 5: Cache extent server
- Lab 6: Paxoes
- Lab 7: Replicated lock server

Lab 2: Lock Server and Reliable RPC

- Centralized lock server
- Lock service consists of:
  - Lock server: grant a lock to clients, one at a time
  - Lock client: talk to server to acquire/release locks
- Correctness:
  - At most one lock is granted to any client
- Additional requirement:
  - acquire() at client does not return until lock is granted

Lab 2 Steps

- Step 1: Checkout the skeleton code from ds-git
- Step 2: Implement server lock and client lock
  - Test it using locker_tester
- Step 3: Implement RPC at-most-once semantics
  - Use a sliding window to store the sent RPC ids.

Today's Lecture

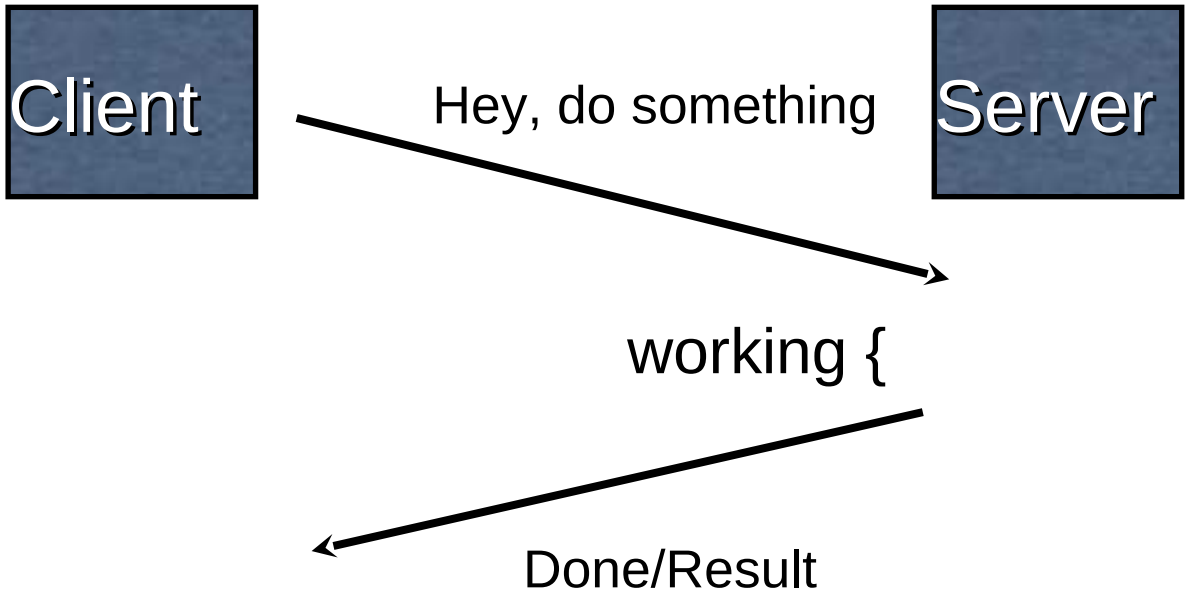
Last Time (Reminder/Quiz)

- Processes: A resource container for execution on a single machine
- Threads: One “thread” of execution through code. Can have multiple threads per process.
- Why processes? Why threads? Why either?
- Local communication
  - Inter-process communication
  - Thread synchronization

Today: Distributed Communication

- Socket communication
- Remote Procedure Calls (RPCs)
- RPC challenges

Common Communication Pattern



Communication Mechanisms

- Many possibilities and protocols for communicating in a distributed system
  - Sockets (mostly in HW1)
  - RPC (today)
  - Distributed shared memory (possibly later classes)
  - Map/Reduce, Dryad (later classes)
  - MPI (on your own)

Socket Communication

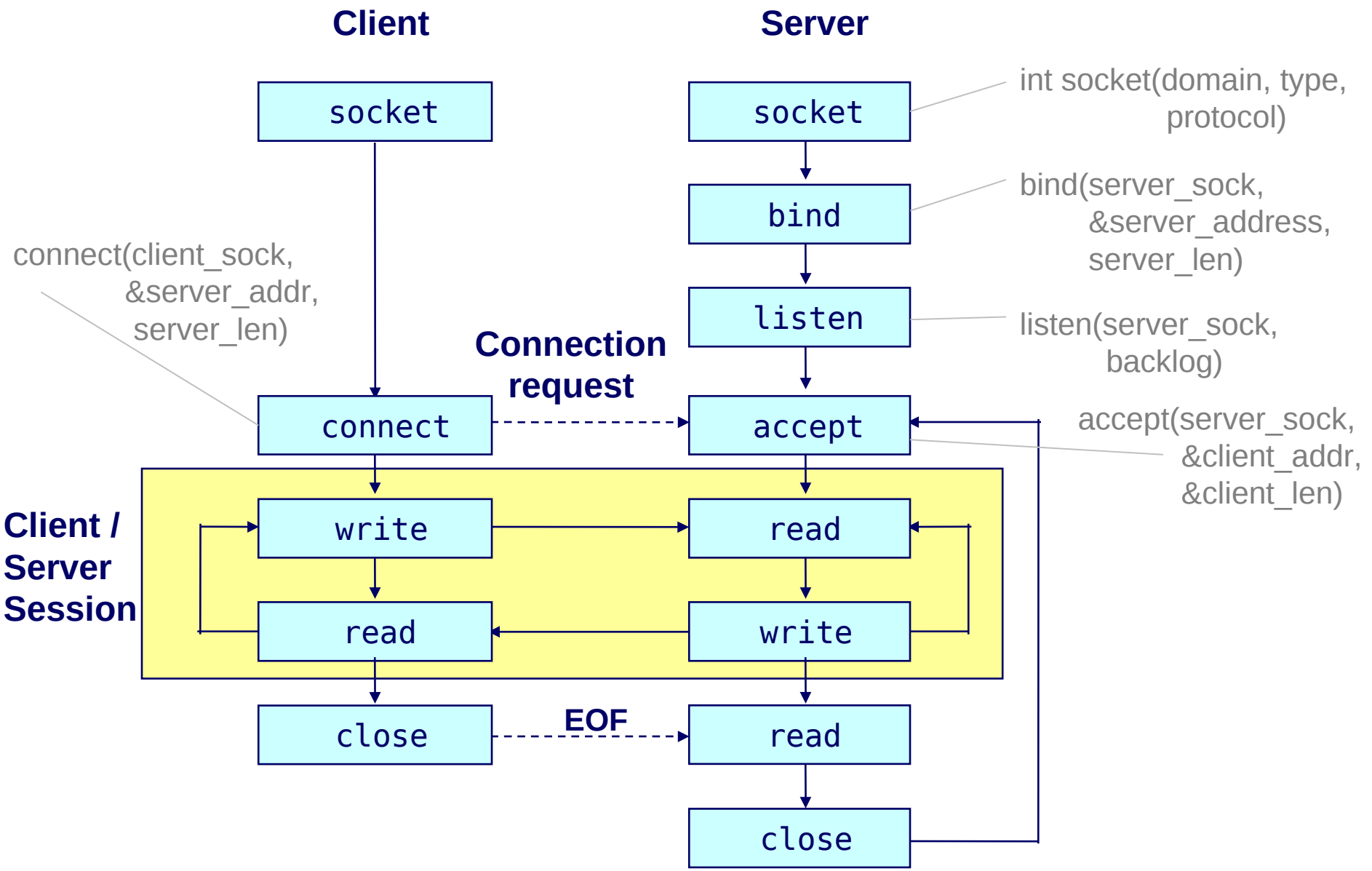
- You your own protocol on top of a transmission protocol (e.g., TCP or UDP)

- Quiz from your networking course ☺:
  - What's TCP?
  - What's UDP?
  - When is it best to use TCP vs. UDP?

Socket Communication

- TCP (Transmission Control Protocol)
  - Protocol built upon the IP networking protocol, which supports sequenced, reliable, two-way transmission over a connection (or session, stream) between two sockets
- UDP (User Datagram Protocol)
  - Also protocol built on top of IP. Supports best-effort, transmission of single datagrams
- Use:
  - TCP when you need reliability, but when performance of setting up connection is not a huge (e.g., file transmission, a lock service)
  - UDP when it's OK to lose, re-order, or duplicate messages, but you want low latency (e.g., online games, messaging, games)

Socket API Overview



Complexities of Using the Socket API

- Lots of boiler-plate when using a raw socket API
- Lots of bugs/inefficiencies if you're not careful
  - E.g.: retransmissions, multi-threading, …
- Plus, you have to invent the data transmission protocol
  - Can be complex
  - Hard to maintain
  - May not interact well with others' protocols

  …

```
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);
}
```

RPC

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

```
Client:
 { ...
    resp = foo("hello");
 }

 Server:
 int foo(char* arg) {
     …
 }
```

RPC Goals

- Ease of programming
  - Familiar model for programmers (just make a function call)
- Hide complexity (or some of it – we'll see later)
- Automate a lot of task of implementing
- Standardize some low-level data packaging protocols across components

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 :)

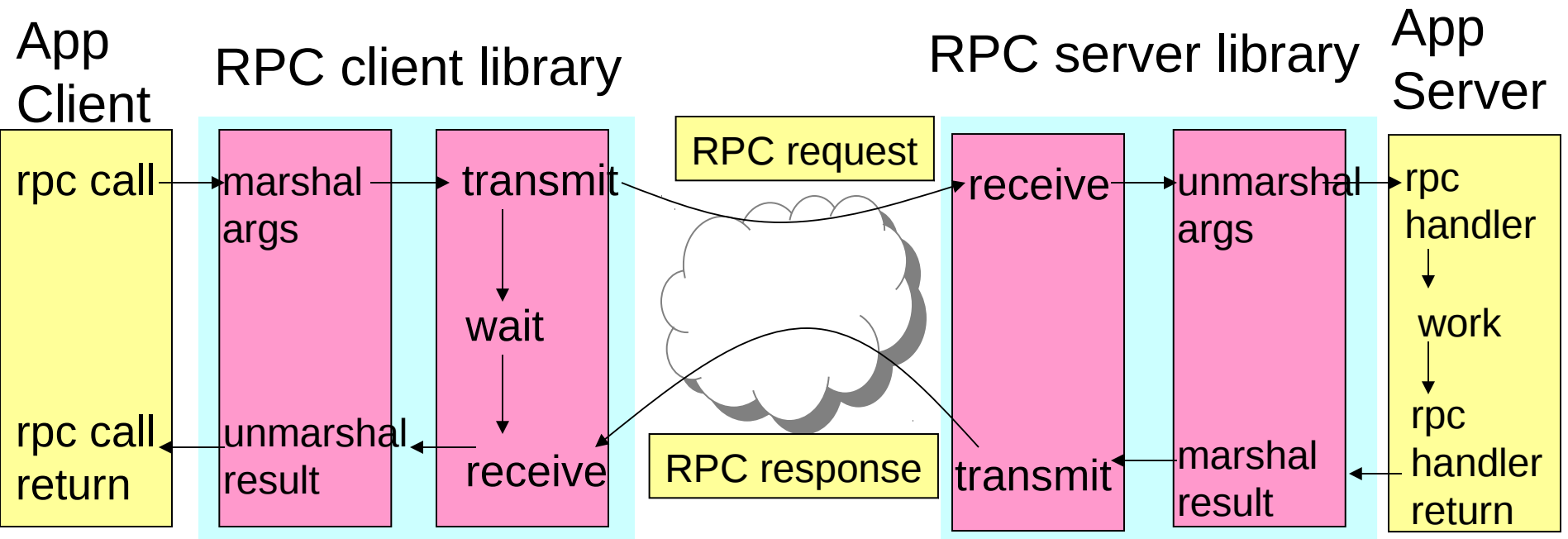
RPC Architecture Overview

- Servers export their local procedure APIs
- On client, RPC library generates RPC requests over network to server
- On server, called procedure executes, result is returned in RPC response to client
- Back on client, RPC library reconstructs the response and returns it to the caller

```
Client:
{ ...
   resp = foo("hello");
}

Server:
int foo(char* arg) {
    …
}
```

RPC Architecture




```
Client:
 { ...
    resp = foo("hello");
 }
```

```
Server:
int foo(char* arg) {
     …
}
```

Why Marshaling?

- Calling and called procedures run on different machines, with different address spaces
  - Therefore, pointers are meaningless
  - Plus, perhaps different environments, different operating systems, different machine organizations, …
  - E.g.: the endian problem:
    - If I send a request to transfer $1 from my little-endian machine, the server might try to transfer $16M if it's a big-endian machine
- Must convert to local representation of data
- That's what marshaling does

Marshaling and Unmarshaling

- hotnl() -- “host to network-byte-order, long”
  - network-byte-order (big-endian) standardized to deal with cross-platform variance

Marshaling and Unmarshaling

- htonl() -- “host to network-byte-order, long”
  - network-byte-order (big-endian) standardized to deal with cross-platform variance
- In our foomsg example, remember how we arbitrarily decided to send the string by sending its length followed by “len” bytes of the string? That’s marshaling, too.

```
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);
}
```

Marshaling and Unmarshaling

- htonl() -- “host to network-byte-order, long”
  - network-byte-order (big-endian) standardized to deal with cross-platform variance
- In our foomsg example, remember how we arbitrarily decided to send the string by sending its length followed by “len” bytes of the string? That’s marshaling, too.
- Other things to marshal:
  - Floating point
  - Nested structures
  - Complex data structures? (Some RPC systems let you send lists and maps as first-order objects)

“Stubs” and IDLs

- RPC stubs are automatically generated codes that appear to implement the desired functions, but actually do just marshalling/unmarshalling and then call the RPC library for request transmission

- How does this stub generation work?

- Typically: Write a description of the function signature using an *IDL* -- interface definition language
  - Lots of these. Some look like C, some like XML
  - Example: SunRPC (now), next time we’ll look at other IDLs (e.g., Google’s protocol buffers)

SunRPC

- Venerable, widely-used RPC system
- Defines “XDR” (“eXternal Data Representation”) -- C-like language for describing structures and 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!

Key challenges of RPC

- RPC semantics in the face of
  - Communication failures
    - delayed and lost messages
    - connection resets
    - expected packets never arrive
  - Machine failures
    - Server or client failures
    - Did server fail before or after processing the request?
  - Might be impossible to tell communication failures from machine failures

RPC failures

- Request from cli -> srv lost
- Reply from srv -> cli lost
- Server crashes after receiving request
  - Before it has completed it or
  - After it has completed it
- Client crashes after sending request
  - He won’t know whether the server executed the 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

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

Next Time

- A bunch of RPC library examples
- With code! ☺