RPC

1. Group presentation
2. Notes from reviews:
    a.
3. Notes to discuss:
    a. Interface definition: can you just use header files?
        i. Separate language or integrate into source?
        ii. Stub compiler or normal compiler?
    b. Complex arguments: pointer-based structures?
        i. Can marshall by following ptr?
        ii. How know about C arrays?
    c. Soft state / Stateless import
    d. Optimizing common case – round trips, short messages, short execution
    e. Focus on low latency
    f. Synchronous calls
    g. Considerations:
        i. Few packets
        ii. Little memory
        iii. Simple implementation – e.g. ack strategy. Never send spontaneously; only when have data to send or asked for one
    h. Error handling –
4. Notes from Creator:
    a. RPC: Andrew Birrell
        i. Were lucky to be 5 years ahead of everybody else in having a LAN of person computers
            1. Most people were using time sharing on a minicomputer
            2. Predates IBM PC, MacIntosh
            3. PCs were apple – 1 MHz 8 bit processors
        ii. Got so solve lots of interesting problems
        iii. Design still holds up
5. Debate:
    a. What is RPC?
        i. Synchronous calls over a network?
        ii. Concealing network interaction to make remote operations look local
        iii. Sending/receiving complex data structures, invoking a routine on the other side
    b. Problems:
        i. Hiding the network – it is different, programmers need to know

        ii. Too low level: method-level interactions rather than semantic/business level
        iii. Hard to extend - add new parameters, new functions

6. Context
    a. Xerox Parc
    b. Birth of local area networks, distributed computing
    c. Used with Mesa; lightweight processes in shared memory
        i. Creating a thread (calleda process)30x slower than a procedure call
    d. What kinds of things were being remoted?
        i. Deliver email message, receive email message
        ii. Lookup name/address of something
        iii. Generally not for parallelism (e.g. offloading computation), but for sharing state (e.g. shared data across many workstations).
    e. They were building RPC for their own use; not trying to solve all potential problems in distributed communications.
7. Problem
    a. QUESTON: What problem were they solving?
        i. Distributed programming
            1. QUESTION: why important? improve performance by distributed code to different machines
        ii. Hard to write distributed programs using messages
            1. Like writing in ASM

```
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);
}
```

            2.
            3. Everybody sets their own timeouts, retry mechanisms
            4. Example: Amazon; everybody did linear backoff
                a. Under overload, whole network collapsed
        iii. How do you make an efficient high-level communication mechanism?
            1. Similar to using compiler instead of ASM, or scripting language instead of C
        iv. Target environment: local area network, closely-coupled computation, generally reliable
8. Goal:
    a. QUESTION: What was goal for this work?

            i. Find the right paradigm for distributed computing
            ii. Fine-tune the semantics
                1. Make it as powerful as possible so don't need to layer mechanism above it
            iii. Implementation choices for efficiency
    b. NOTE: want to let programmers reason about performance (unlike shared memory)
9. Rejected ideas
    a. Remote fork – launch remote program that returns values
        i. Still has problems of data & argument passing
    b. Distributed shared memory
        i. Difficult to make fast
        ii. Hard to program – memory classes not exposed in language
10. QUESTION: Why RPC?
11.
    a. Review procedure call:
        i. save current state on stack (e.g caller-save registers)
        ii. push arguments on stack (scalar values or pointers to shared memory)
        iii. transfer control to destination procedure
        iv. Destination procedure allocates local space for temporary variables
        v. Destination executes code
        vi. Destination returns value through a register
        vii. Destination returns control by restoring old program counter
        viii. Caller resumes control, looks at return value or modifications to input parameters
    b. Note: data transfer happens through passing scalar values/pointers on stack, and passing data structures by reference through memory
    c. Note: control transfer happens by suspending calling thread before call & resuming afterwards. In the middle, assuming a single-threaded system, calling thread doesn't see intermediate changes to values because it is suspended, so it can't tell difference between call-by-reference and call-by-value-result (send values by copy, receive results and copy back)

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

d.

e. USE FOR REMOTE COMMUNICATION:
   i. Clean, simple semantics
   ii. Well understood to programmers
   iii. Commonly used already for structuring programs
   iv. QUESTION: Why only synchronous communication?
      1. Is the common case
      2. Can use fork/join for asynchronous communication

12. Big picture

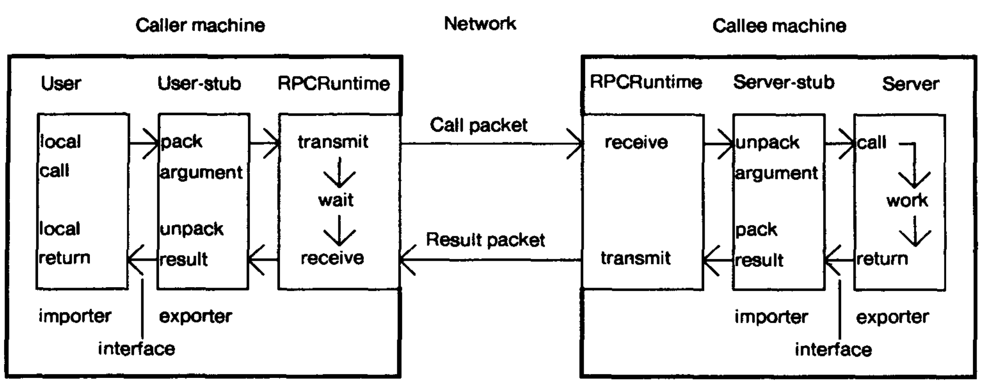

Fig. 1. The components of the system, and their interactions for a simple call.

13.

a. Show how RPC works
   i. Client, client stub, runtime, server stub, server
   ii. Name server
   iii. IDL compiler – Lupine

14. Questions to solve

a. What should failures semantics be?
b. How do you handle pointer-based data structures?
   i. Don't allow
   ii. Marshall automatically
c. NEED programming language integration to make it look local
d. How do you identify the target of a call?
e. What protocols should be used? Where in the stack should you sit (e.g. Ethernet, ip, udp, tcp)

15. Principles

a. Make RPC as much like procedure call as possible

i. No time-outs
    1. Question: Why?
    2. Answer: how do you set timeouts? How do you specify them? What do you do on a timeout?
    3. Answer: most people put the call in a loop and try it again. Generally, not the right thing; most people choose the wrong value for a timeout (from experience, 6 seconds is way too long)

ii. Return communication failures as exceptional conditions
    1. QUESTION: What does this mean for RPC packages in C?
        a. New error parameter?
        b. Return a pointer to the return value or NULL?
    2. QUESTION: how does this impact programming?
        a. New failure modes
        b. Depends on whether programmers already handle exceptions
    3. QUESTION: What should a program do on failure?

iii. No asynchronous RPC
    1. Question: Why?
        a. A: not RPC
        b. A: can achieve by forking a thread
        c. A: allows multiple outstanding calls per client process; complicates protocol
        d. A: not simple; not even a solved problem within a single process

iv. NOTE: google RPC
    1. Allows streamed RPC – send a sequence of requests with one response, or vice versa.
    2. Basically means you don't need all the data when you make the call, can keep generating it over time

16. Stubs
    a. Automatically generated
        i. QUESTION: From where?
            1. Source code?
            2. Interface definition?
    b. Look like normal procedure to client; hides distribution
    c. Runtime can hide architectural differences
        i. Convert between endian-ness
        ii. Convert between pointer sizes
        iii. HOW?

            1. Option 1: send in sender format, convert on receiver if necessary (and indicate in packet)
            2. Option 2: convert to a canonical format for wire
    d. Better: an Interface Compiler
        i. Specify the functions in your interface
        ii. Specify the types in your interface in sufficient detail to send them
        iii. E.g. C: char * x; Is it a pointer to single character? A null-terminate string? A counted array?
        iv. In this system, use existing Mesa interfaces that do this job. Typically, have to write separately from C, but can re-use C header files for types.
        v. Can generate stubs in multiple languages (sometimes)
    e. How do you return errors?
        i. What if the server fails while processing a call, or the network gets unplugged?
        ii. QUESTION: can you return an error?
            1. Answer: not; not all calls return an error
            2. Answer: no; error returns from a function already defined by application
            3. SO?
                a. In Mesa; throw an exception
                b. In C:
                    i. Throw an exception (if you have fancy C)
                    ii. Change interface to take an additional out parameter
                        1. For return value
                        2. For RPC error code

17. Binding
    a. QUESTION: What is binding?
        i. How do you do binding in a local program?
            1. C: function pointer assignment, link time
            2. C++: inheritance – run time/compile time
    b. How do you specify someone to talk to?
        i. Naming:
            1. type (interface name)
                a. What service is provided? Email, http, ssh?
            2. instance (host name / service name for replicated services)
                a. Specific or one of a set of identical services
            3. Names
                a. Groups: a list of individual names
                    i. Good for a set of replicas
                        1. E.g. I want some mail server

            b. Individuals: specific host address/port number
                i. E.g. I want a specific printer
    ii. **QUESTION: What do you want from naming?**
        1. Security: should be able to say who is part of a group
            a. I can't set up mail server, but I can set up a game server
        2. Human readable: so can type in?
c. How do you find someone that meets that specification
    i. Contact a name service:
        1. Grapevine
            a. Entry for each type
                i. Lists instances of the type
            b. Entry for each instance
                i. Addressing information for host
            c. **QUETSION: What about DNS?**
                i. DNS for mail services
                ii. LDAP in Windows
        2. QUESTION: is it reasonable to have such a database?
            a. Context: LAN
d. How do you announce that you provide a service?
    i. **ExportInterface** registers information with grapevine automatically when server starts up
    ii. RPC runtime maintains a table mapping interface name to dispatch procedure & 32 bit instance/incarnation identifier (changes after reboot)
    iii. QUESTION: Does time have to be synchronized across machines?
        1. ANSWER: No, time is used locally as a per-machine unique ID across reboots. Read once at reboot; then increment counter and assume that by next reboot, time will be > counter value before reboot.
    iv. QUESTION: How handle reboots
e. What do you do to initiate a conversation?
    i. **ImportInterface** asks grapevine for addressing information (or uses provided name/address)
        1. When several available, client runtime gets all
        2. Client tries them in useful order to establish service is running
    ii. Runtime on client deos RPC to server to receive binding association (unique identifier/incarnation number)
        1. NOTE: verify during binding, not during call

2. Gets index into per-server table
   a. Fast lookup for that server + a check (so if table changes later, will detect)

f. ISSUES:
   i. Binding does not create state on server → scalable
   ii. Bindings broken when server crashes → automatically informs client
   iii. Access controls
      1. Who should be able to export an interface?
         a. What about dept. imap?
         b. on grapevine limits who can register an interface
      2. QUESTION: Should it limit who can import?
         a. Can learn of imports other ways, e.g. port scanning
   iv. Early vs late:
      1. Early = before you need it; could be embedding an Ethernet / ip address / dns name in code
      2. Late == as late as possible; could be as late as on every packet (e.g. broadcast)
      3. Which has better reliability implications?

g. DESIGN CONCERNS:
   i. Priniple: Soft state: state on server can be discarded; is just an optimization
   ii. Minimal memory consumption -> allows to scale to more clients

18. Protocol Implementation
   a. QUESTION: What are goals:
      i. Minimize latency of calls
      ii. Minimize state needed on server for handling many clients – throughput
      iii. Provide useful semantics:
         1. Exactly once
            a. QUESTION: How?
               - 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.
            b.
         2. At **least** once: call may execute more than once (e.g. must be **idempotent**).

a. Example: set back account value to 100
b. Not example: add $10 to bank account
c. How? Just keep retrying until succeeds

3. At most once: call executes no more than once
    a. On success, exactly one execution
    b. On exception, zero or one execution
        i. QUESTION: Why? Impossibility result
    c. No timeouts
        i. QUESTION: Good? Bad? What is user experience?
    d. **How implement?**
        i. Server has to remember previous requests and not re-execute, just resesnd reply

iv. QUESTION: what is the right choice?
    1. At most once allows non-idempotent operations
    2. At most once is responsive, because you can return an error to application quickly (after first failure), and let application retry.
    3. At most once is like a normal procedure call.
        a. Don’t know where it failed…

b. Solution:
    i. Principle: Optimize for common case:
        1. Request & reply happen in a single packet
        2. Reply takes less than a roundtrip of computation
    ii. **Piggyback** ack’s on subsequent packet
    iii. Leverage protocol properties
        1. Only one outstanding request per client on an interface → no sliding window
        2. Not need to establish connection; server just remembers highest # request from client to detect duplicates
        3. **Sender** of data packet resends until ACKd, by next call or explicit (if call takes longer)
    iv. Handle complex case simply
        1. Multiple-packet request/reply **explicitly ACK** every non-terminating packet before sending next packet
            a. Only last packet must be buffered on either side
            b. Use other protocols for bulk transfer
    v. Detect failure: no ACK in response
        1. Client re-sends request periodically to ensure server alive

            a. Server detects as duplicate and ignores
            b. Network notifies sender if server isn't running or not listening on port (e.g. failed)
        2. QUESTION: How deliver?
            a. Cannot just return an error code (that comes from the procedure)
            b. Raise exception instead
        3.
    c. QUESTION: Why not use TCP/IP?
        i. A: didn't really exist yet, not in wide use
        ii. A: requires 3 packets to set up a connection, more packets to send/receive data; stream approach doesn't match RPC request/reply that well.
        iii.
    d. Avoid expensive process creation for handling requests
        i. Server uses separate process / concurrent request (no threads)
            1. Processes really are threads (sharing an address space)
        ii. Creates pool of processes to avoid expensive creation cost on call
        iii. **Hints** to client what process to request to use same process for all requests in a conversation
            1. **QUESTION:** What are the implications? Each call independent? No state across calls? Servers must share shared dynamic state across processes?

19. **Evaluation**
    a. **QUESTION:** what should be evaluated?
        i. Complexity of using system
        ii. Amount of code to solve a problem
        iii. Fault tolerance
        iv. Latency
        v. Scalability / throughput / simultaneous clients
    b. **QUESTION**: what is evaluated?
        i. Performance of calls relative to procedure call and messaging latency
        ii. What about compared to bare message passing?

20. **Repeated themes in the design of RPC**
    a. layer of indirection
        i. used to insert remote into a procedure call
        ii. used in naming to indirect from a group to an individual
            1. allows locality or performance-based server selection

b. Early binding:
    i. Make binding before making RPC
        1. Can detect errors
        2. Can select correct one
        3. Can amortize cost of binding
c. Late binding: through names, group names
    i. Can change which server you talk to
    ii. Can change which instance of a replicated service
d. Piggy backing
    i. Re-use existing message to send another one;
        1. ACK on reply message
e. Stateless server
    i. No per-client state in RPC runtime on server
    ii. Allows server to crash & recover without worrying about clients
    iii. Clients have to detect failure
    iv. Better scalability, more complicated clients
    v.
f. Soft state
    i. Server can discard connection state after an idle period; can be reconstructed on next call
g. Caching
    i. Idle server processes
    ii. Put PID in packet to help speed dispatch if process is waiting. Allows locality of using the same server process repeatedly.

21. Commentary
a. RPC useful technique for loosely coupled distributed systems
b. Performance can be made quite high with optimized runtimes (see next week)
c. Failure semantics cause problems; callers often not prepared to deal well with failure
    i. QUESTION: What should you do on failure? Retry ? How many times? How long should you wait?
d. Makes it almost as easy to build a system of proceses as one of a single process
e. Basis for distributed object systems like DCOM and RMI and XML-RPC
f. Problems
    i. Procedure call level may be too low; message formats for internet protocols may encourage better separation between code and protocol
    ii. Encourages synchronous round trips; hard to batch requests that can be overlapped

iii. Difficult to revise interfaces; is handled but leads to ugly code on server
iv. Generally language specific