# 416 Distributed Systems

RPC Day 2

Jan 15, 2016

## Last class

- Finish networks review
  - Fate sharing
  - End-to-end principle
  - UDP versus TCP; blocking sockets
  - IP thin waist, smart end-hosts, dumb (stateless) network
- Start RPC (remote procedure calls)
  - What is an RPC, goals/benefits of RPC
  - Three transparencies of RPC
  - Instant distributed system recipe via LPC -> RPC?

## Remote procedure call

- A remote procedure call makes a call to a remote service look like a local call
  - RPC makes transparent whether server is local or remote
  - RPC allows applications to become distributed transparently
  - RPC makes architecture of remote machine transparent

Emphasis on transparency

What are some problems with this transparency?

# RPC: 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

## Two styles of RPC implementation

- Shallow integration.  Must use lots of library calls to set things up:
  - How to format data
  - Registering which functions are available and how they are invoked.
- Deep integration.
  - Data formatting done based on type declarations
  - (Almost) all public methods of object are registered.
- Go is the latter.

## Stubs: obtaining transparency

- Compiler generates from API stubs for a procedure on the client and server
- Client stub
  - **Marshals** arguments into machine-independent format
  - Sends request to server
  - Waits for response
  - **Unmarshals** result and returns to caller
- Server stub
  - **Unmarshals** arguments and builds stack frame
  - Calls procedure
  - Server stub **marshals** results and sends reply

# Marshaling and Unmarshaling

- (From example) hotnl() -- “host to network-byte-order, long” (in C)
  - 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 marshaling, too.
- Floating point...
- Nested structures? (Design question for the RPC system - do you support them?)
- Complex data structures? (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.

## Remote Procedure Calls (1)

- A remote procedure call occurs in the following steps:

1. The client procedure calls the client stub in the normal way.
2. The client stub builds a message and calls the local operating system.
3. The client's OS sends the message to the remote OS.
4. The remote OS gives the message to the server stub.
5. The server stub unpacks the parameters and calls the server.

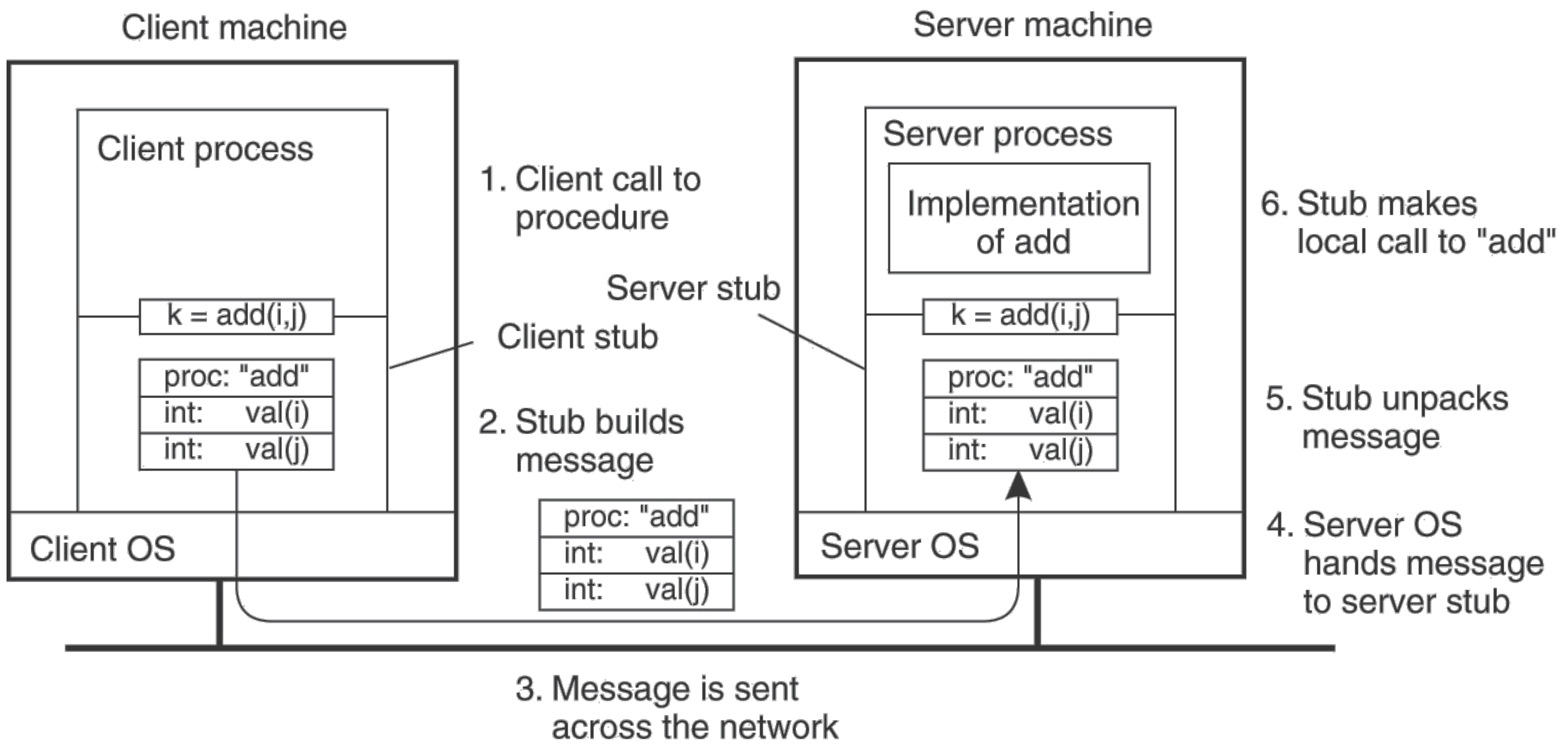
Continued …

# Remote Procedure Calls (2)

- A remote procedure call occurs in the following steps (continued):

6. The server does the work and returns the result to the stub.
7. The server stub packs it in a message and calls its local OS.
8. The server's OS sends the message to the client's OS.
9. The client's OS gives the message to the client stub.
10. The stub unpacks the result and returns to the client.

# Passing Value Parameters (1)



- The steps involved in a doing a remote computation through RPC.

## Passing Reference Parameters

- Replace with pass by copy/restore
- Need to know size of data to copy
  - Difficult in some programming languages

- Solves the problem only partially
  - What about data structures containing pointers?
  - Access to memory in general?

# RPC land

- RPC overview
- RPC challenges
- RPC other stuff

## RPC vs. LPC

- 4 properties of distributed computing that make achieving transparency difficult:
  - Partial failures
  - Latency
  - Memory access

## RPC failures

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

## Partial failures

- In local computing:
  - if machine fails, application fails
- In distributed computing:
  - if a machine fails, part of application fails
  - cannot tell the difference between a machine failure and network failure
- How to make partial failures transparent to client?

## Strawman solution

- Make remote behavior identical to local behavior:
  - Every partial failure results in complete failure
    - You abort and reboot the whole system
  - You wait patiently until system is repaired
- Problems with this solution:
  - Many catastrophic failures
  - Clients block for long periods
    - System might not be able to recover

## Real solution: break transparency

- Possible semantics for RPC:
  - Exactly-once (what local procedure calls provide)
    - Impossible in practice
  - At least once:
    - Only for idempotent operations
  - At most once
    - Zero, don’t know, or once
  - Zero or once
    - Transactional semantics (databases!)

## 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.

# Real solution: break transparency

- **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.

## 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 reads/writes (readv/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

## Summary:
## expose remoteness to client

- Expose RPC properties to client, since you cannot hide them

- Application writers have to decide how to deal with partial failures
  - Consider: E-commerce application vs. game

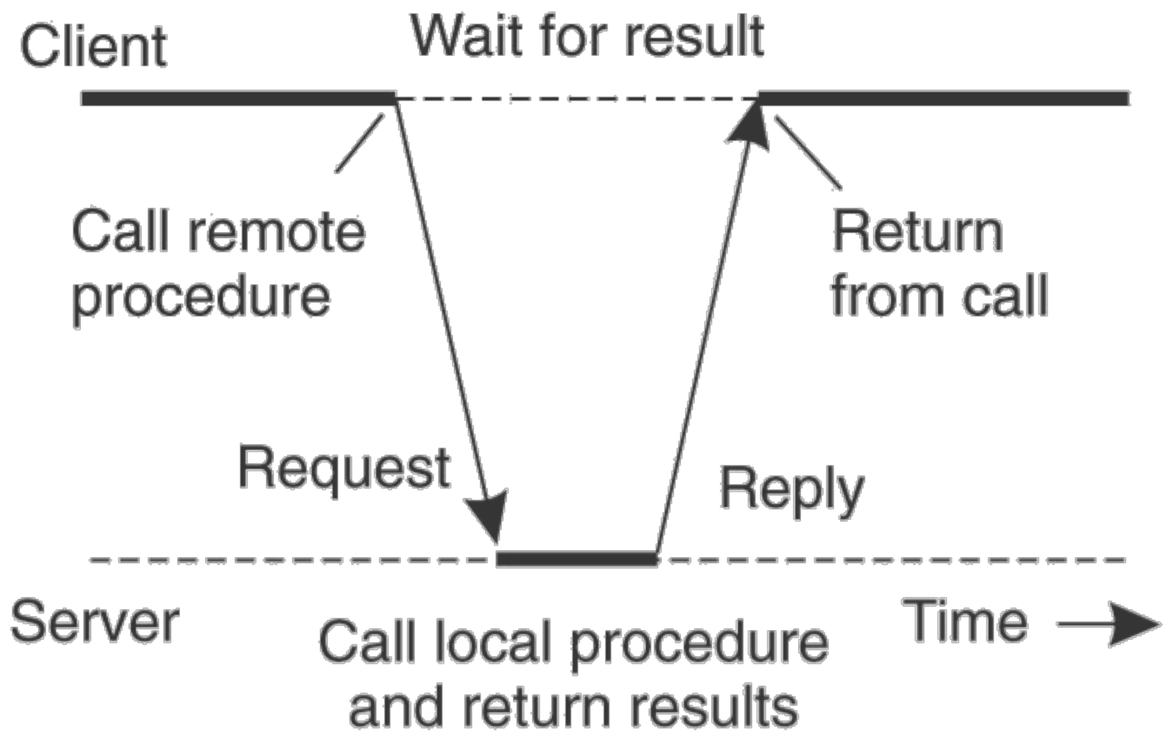
## Important Lessons

- Procedure calls
  - Simple way to pass control and data
  - Elegant/transparent way to distribute application
  - Not only way…
- Hard to provide true transparency
  - Failures
  - Performance
  - Memory access
  - Etc.
- How to deal with hard problem → give up and let programmer deal with them
  - "Worse is better"

## RPC land

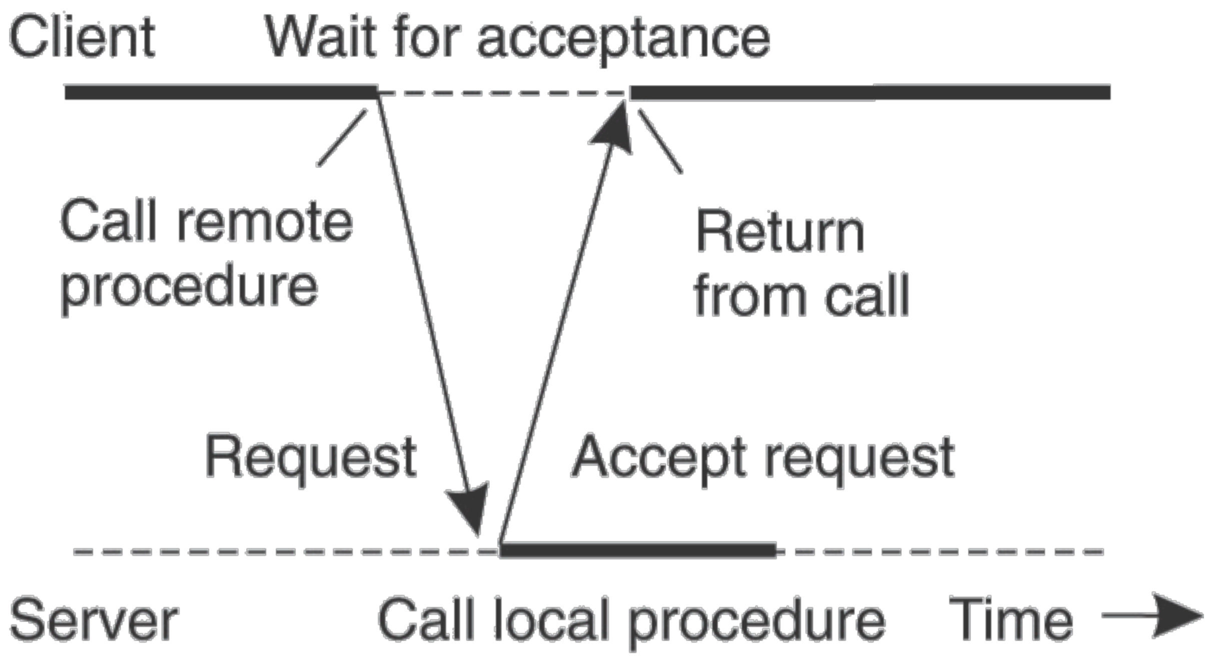
- RPC overview
- RPC challenges
- RPC other stuff

## Asynchronous RPC (1)



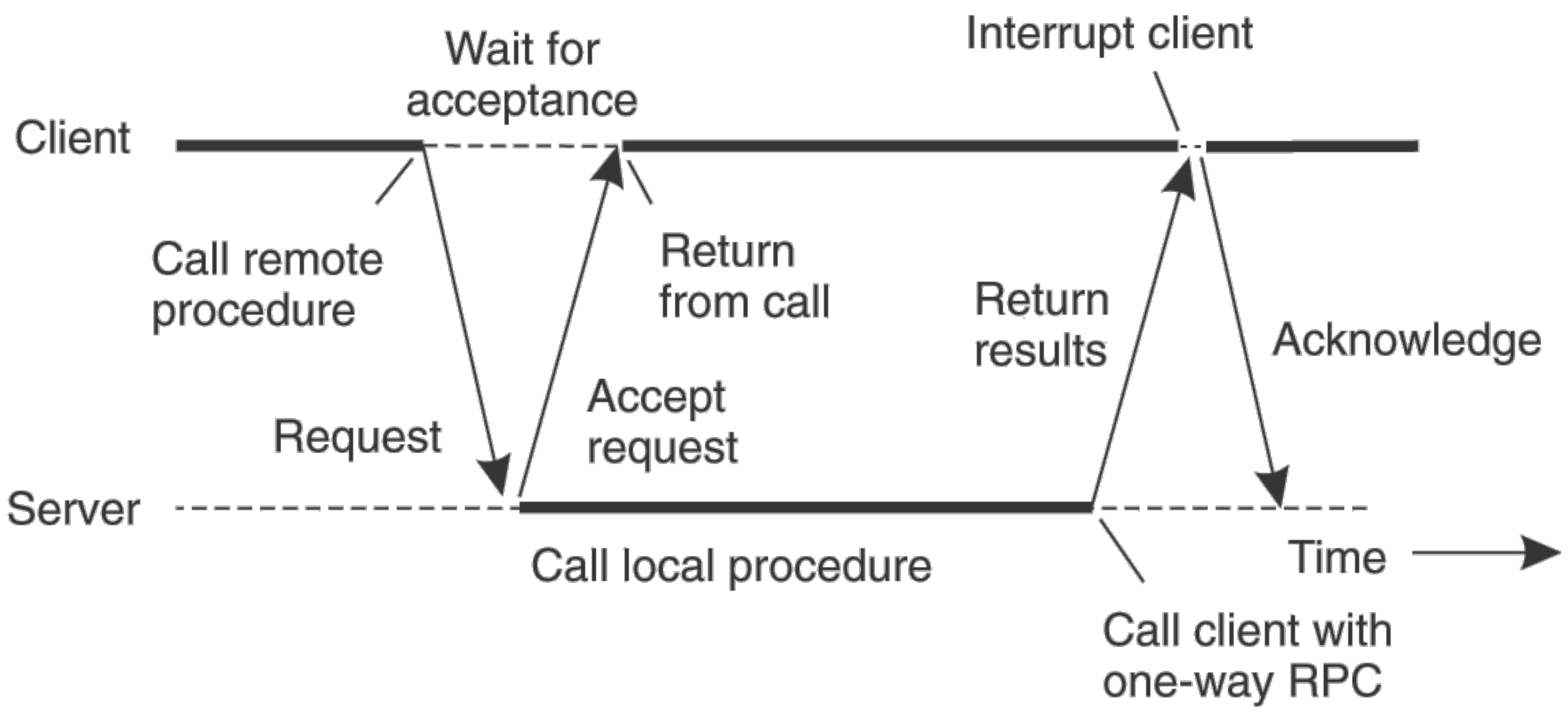
- The interaction between client and server in a traditional RPC.

# Asynchronous RPC (2)



- The interaction using asynchronous RPC.

# Asynchronous RPC (3)



- A client and server interacting through two asynchronous RPCs.

## Go Example

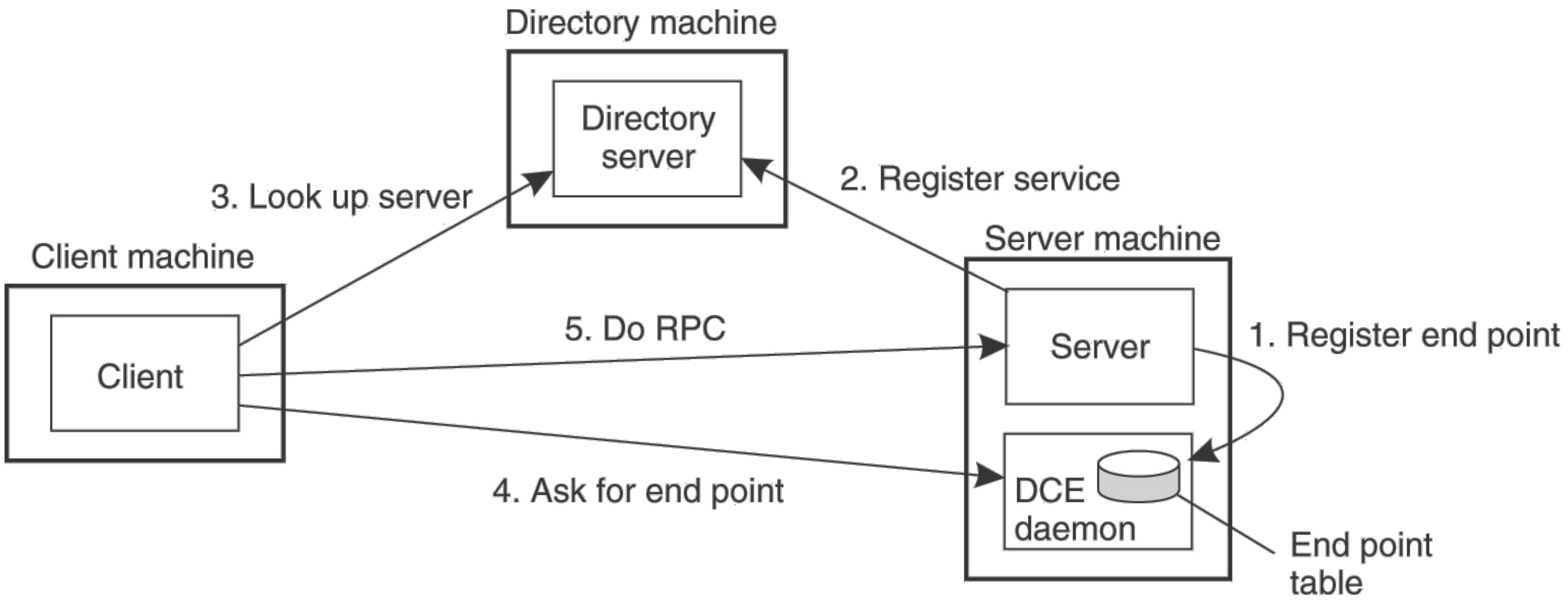
```
// Asynchronous call
quotient := new(Quotient)
divCall := client.Go("Arith.Divide", args, quotient, nil)
replyCall := <-divCall.Done    // will be equal to divCall
// check errors, print, etc.
```

## Using RPC

- Request→Server→Response: Classic synchronous RPC
- Worker-->Server.
    - Synch RPC, but no return value.
    - "I'm a worker and I'm listening for you on host XXX, port YYY."
- Server-->Worker.
    - Synch RPC? No that would be a bad idea. Better be Asynch.
    - Otherwise, it would have to block while worker does its work, which misses the whole point of having many workers.

## Binding a Client to a Server

- Registration of a server makes it possible for a client to locate the server and bind to it
- Server location is done in two steps:
  - Locate the server's machine.
  - Locate the server on that machine.



## Other RPC systems

- ONC RPC (a.k.a. Sun RPC). Fairly basic. Includes encoding standard XDR + language for describing data formats.
- Java RMI (remote method invocation). Very elaborate. Tries to make it look like can perform arbitrary methods on remote objects.
- Thrift. Developed at Facebook. Now part of Apache Open Source. Supports multiple data encodings & transport mechanisms. Works across multiple languages.
- Avro. Also Apache standard. Created as part of Hadoop project. Uses JSON. Not as elaborate as Thrift.