15-440/15-640 Distributed Systems

Remote Procedure Calls

Lecture 07 – Tuesday, Sept 21st, 2021

Carnegie Mellon University
School of Computer Science

FIRST,

Announcements

- **P1 Released last week.**
  - Do you have a partner? If not, please look at the Piazza post.
  - Post on Piazza, talk to course staff if still having trouble finding a partner
  - **Checkpoint** Due: 9/28, **Part A** Due: 10/8, **Part B** Due: 10/14
  - Oct 11th, course drop deadline. Talk with your partner early about any issues
- **P1 Recitation (Wednesday, Sept 22nd)**
- **General Debugging Recitation (Wednesday, Sept 29th)**
- **Pandemic related exception to OH, form fill in OHQueue**

CONTEXT

Building up to today

- **Abstractions for *Communication***
  - EXAMPLE: TCP masks some of the pain of communicating across unreliable IP
- **Abstractions for *Computation***

CONTEXT

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?

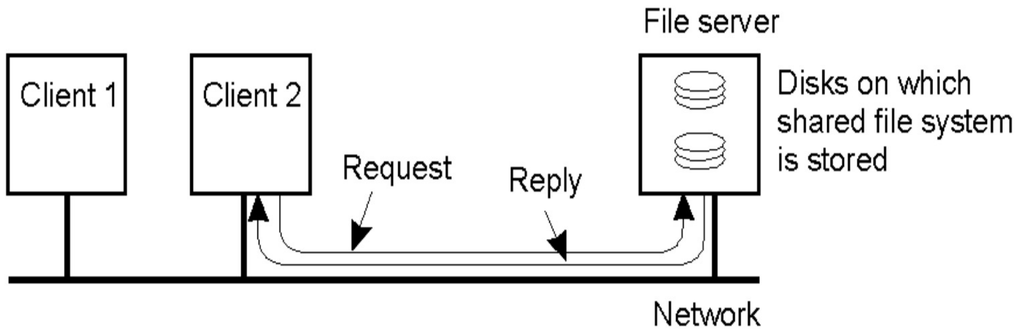
Spoiler: there are many abstractions.

CONTEXT

Client-Server Model

Let's start with the most common system model

(there are others, of course, e.g., peer-to-peer)

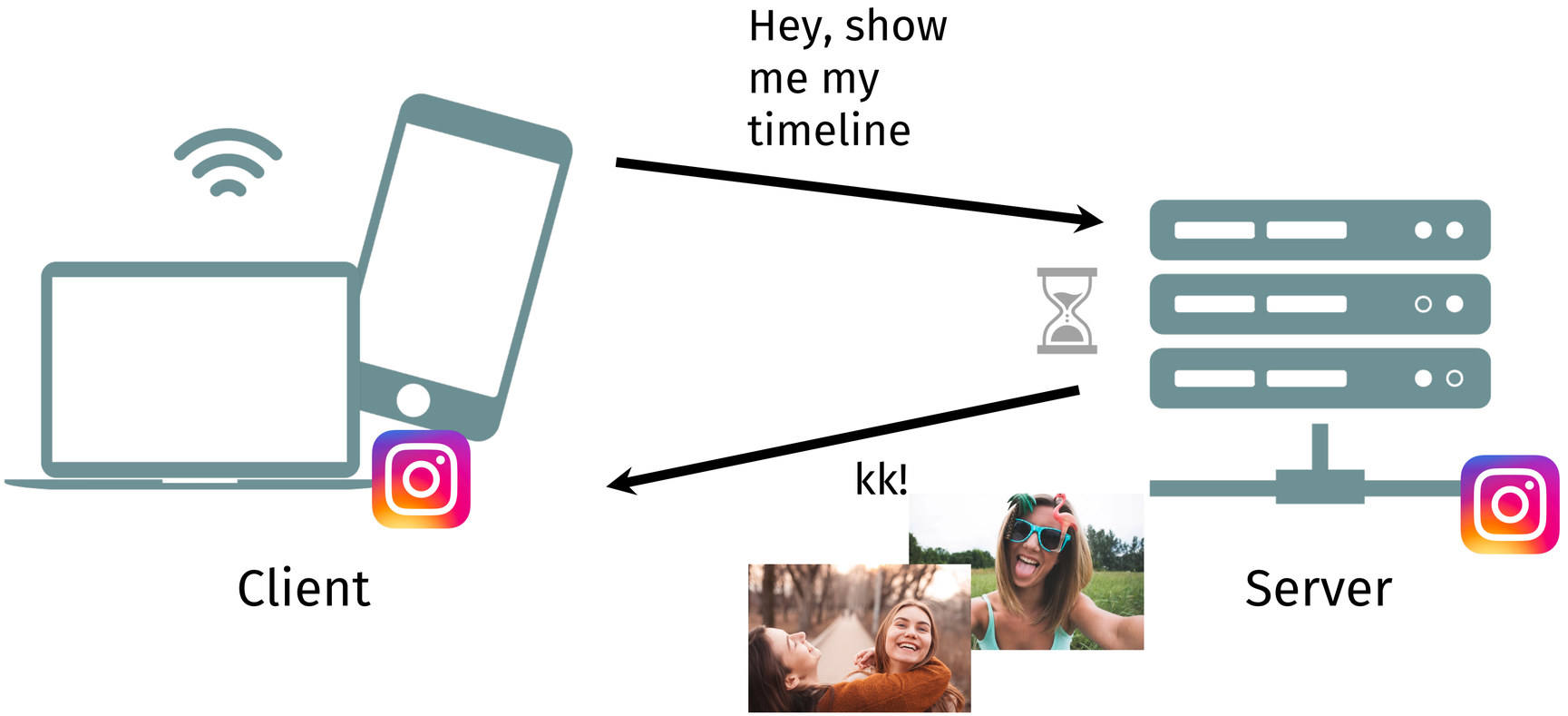


- 1970s: development of Local Area Networks (LANs)
- 1980s: standard deployment involves small number of servers, plus many workstations
  - Servers: always-on, powerful machines
  - Workstations: personal computers
  - Workstations request 'service' from servers over the network, e.g. access to a shared file-system.

CONTEXT

Typical Communication Pattern

Client-server still looks much the same today!



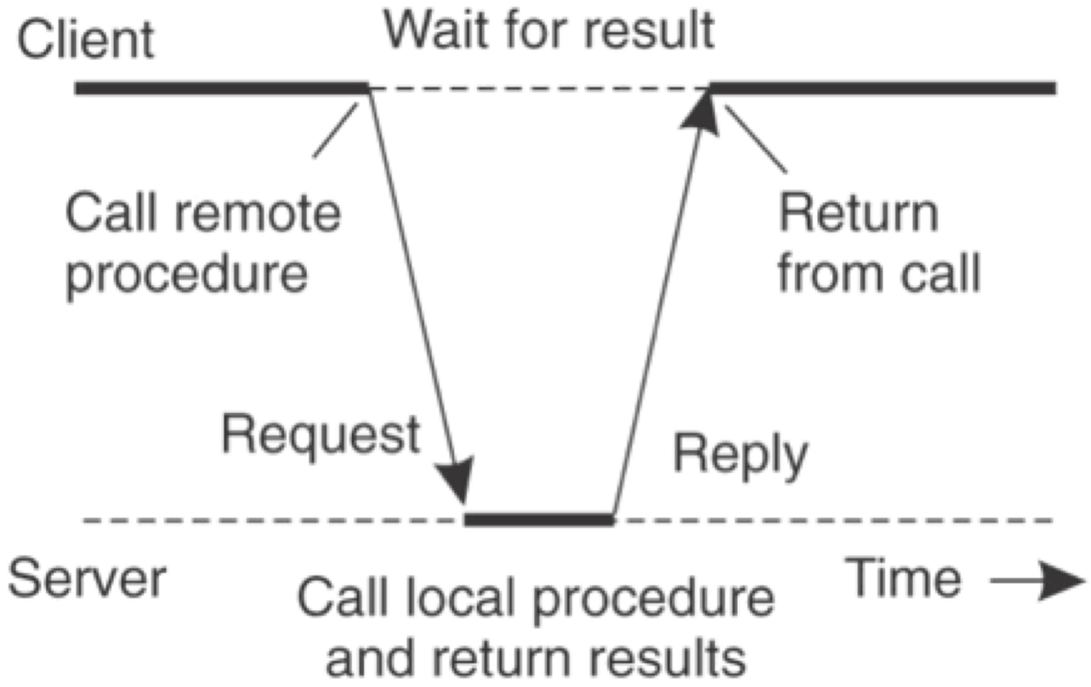
SO...

Why not use an abstraction to communicate with the server that resembles this pattern?

ENTER:

Remote Procedure Calls (RPC)

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



Goals of RPC

- **A nicer abstraction than programming on the network**
  - Programmer simply invokes a procedure…
  - …but it executes on a remote machine (the server)
  - RPC subsystem handles message formats, sending & receiving, handling timeouts, etc
- **Aim is to make distribution (mostly) transparent**
  - Certain failure cases wouldn’t happen locally
  - Distributed and local function call performance different

SOUNDS NICE…

But it’s not always simple!

- **Machines and network can fail**
- **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**

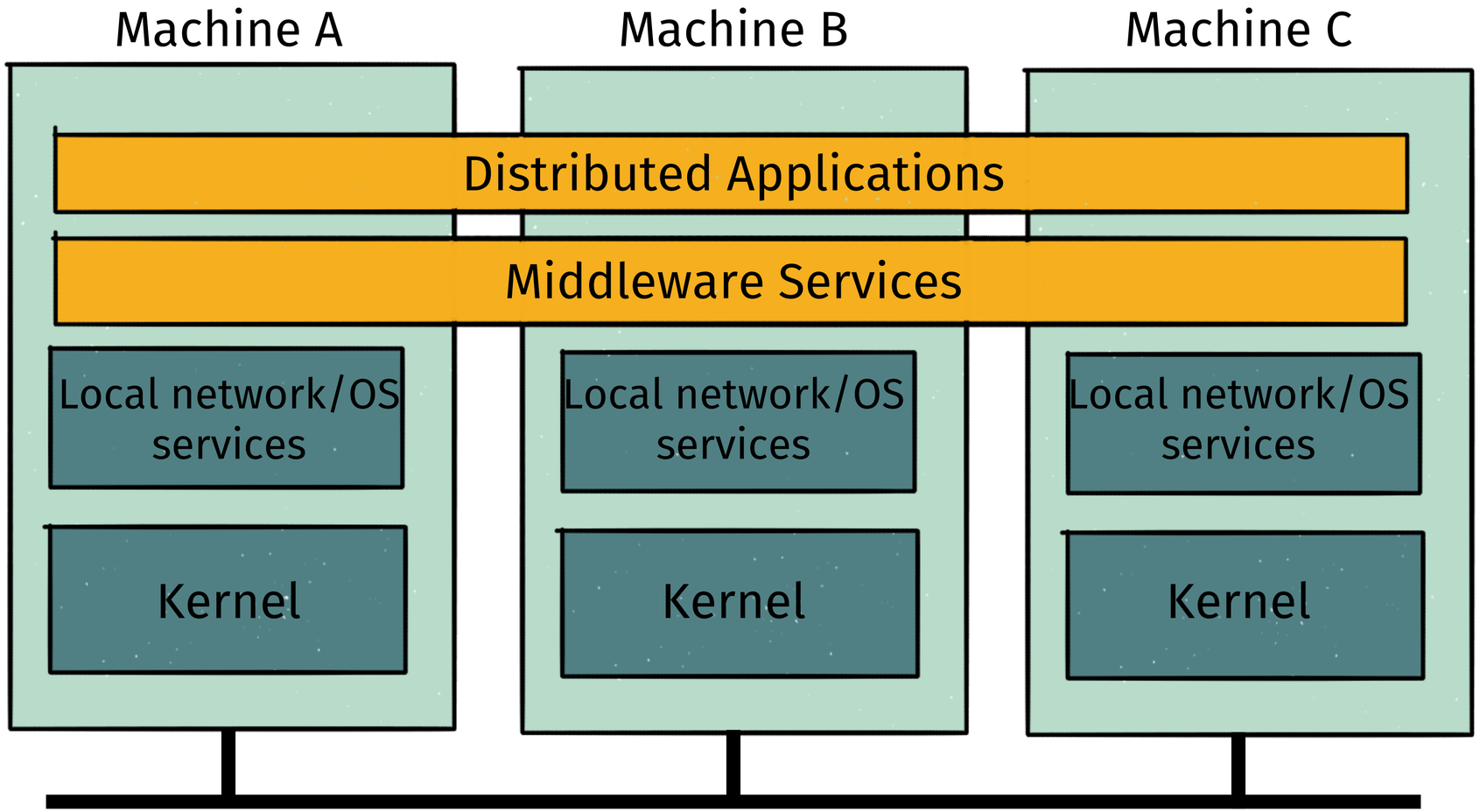
Flavors of Transparency

This is the sort of transparency we are concerned about for RPC!

Transparency	Description
Access	Hide differences in data representation and how a resource is accessed
Location	Hide where a resource is located
Migration	Hide that a resource may move to another location
Relocation	Hide that a resource may be moved to another location while in use
Replication	Hide that a resource may be provided by multiple cooperating systems
Concurrency	Hide that a resource may be simultaneously shared by several competitive users
Failure	Hide the failure and recovery of a resource
Persistence	Hide whether a (software) resource is in memory or on disk

ZOOMING OUT…

Where are we in the stack?



Your app!

E.g., RMI/RPC

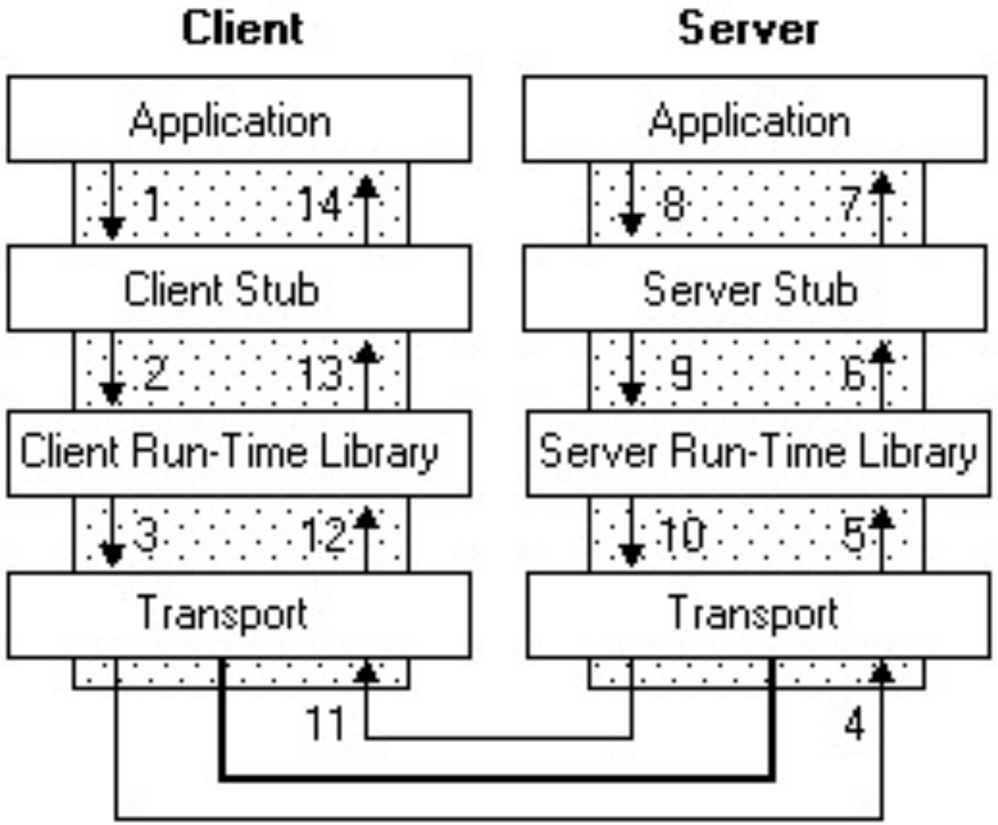
E.g., Java runtime

E.g., Linux, BSD, Windows

E.g., TCP/IP

ZOOMING IN ON RPC

How does RPC work?




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

How to think about RPC in Go

RPC Package in Go

- Provides access to the *exported methods* of an object across a network or other I/O connection.
  - *A server registers an object, making it visible as a service with the name of the type of the object.*
- After registration, exported methods of the object will be accessible remotely.
- A server may register multiple objects (services) of different types but it is an error to register multiple objects of the same type.

Go Example, server side

Basic RPC code:

```
package server
import "errors"
type Args struct {  A, B int    }
type Quotient struct {  Quo, Rem int    }
type Arith int

func (t *Arith) Multiply(args *Args, reply *int) error {
    *reply = args.A * args.B
    return nil
}

func (t *Arith) Divide(args *Args, quo *Quotient) error {
    if args.B == 0 {
        return errors.New("divide by zero")
    }
    quo.Quo = args.A / args.B
    quo.Rem = args.A % args.B
    return nil
}
```

Let's say we have a server with an object of type Arith that it wishes to export.

Wait, what do we mean by "export"?

Go Example, server side

Basic RPC code:

```
package server
import "errors"
type Args struct {  A, B int    }
type Quotient struct {  Quo, Rem int    }
type Arith int

func (t *Arith) Multiply(args *Args, reply *int) error {
    *reply = args.A * args.B
    return nil
}

func (t *Arith) Divide(args *Args, quo *Quotient) error {
    if args.B == 0 {
        return errors.New("divide by zero")
    }
    quo.Quo = args.A / args.B
    quo.Rem = args.A % args.B
    return nil
}
```

The server then calls (for HTTP service):

```
arith := new(Arith)
rpc.Register(arith)
rpc.HandleHTTP()
l, e := net.Listen("tcp", ":1234")
if e != nil {
    log.Fatal("listen error:", e)
}
go http.Serve(l, nil)
```

RPC IN GO

Go Example, client side

Client first dials the server

```
client, err := rpc.DialHTTP("tcp", serverAddress + ":1234")
if err != nil {
    log.Fatal("dialing:", err)
}
```

Go Example, client side

Client first dials the server

```
client, err := rpc.DialHTTP("tcp", serverAddress + ":1234")
if err != nil {
    log.Fatal("dialing:", err)
}
```

Then it can make a remote call:

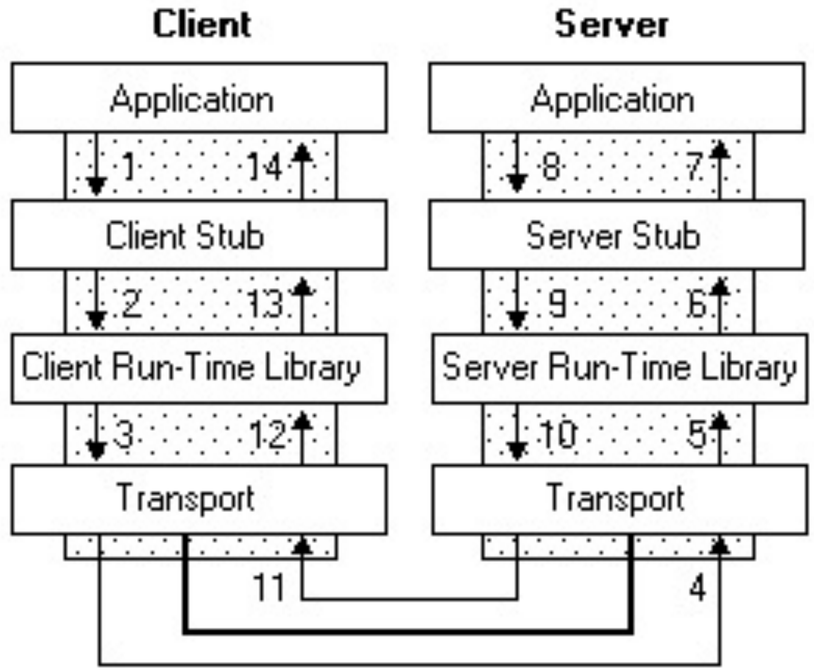
```
// Synchronous call
args := &server.Args{7,8}
var reply int
err = client.Call("Arith.Multiply", args, &reply)
if err != nil {
    log.Fatal("arith error:", err)
}
fmt.Printf("Arith: %d*%d=%d", args.A, args.B, reply)
```

VISUALIZING RPC OPERATION

How RPC works

RPC occurs in the following steps

1. client procedure calls the client stub in the normal way
2. client stub builds a message and calls the local OS
3. client OS sends the message to the remote OS
4. the remote OS gives the message to the server stub
5. server stub unpacks the parameters and calls the server
6. server does the work and returns the result to the stub
7. server stub packs the result in a message and calls its local OS
8. server's OS sends the message to the client's OS
9. client's OS gives the message to the client stub
10. client stub unpacks the result and returns it to the client



WHAT'S THE POINT OF A STUB?

Stubs

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

Writing an RPC by hand

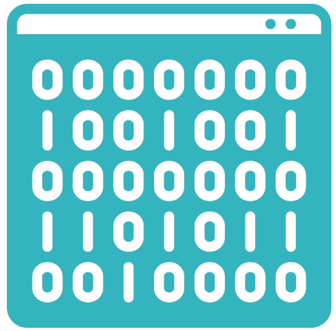
- E.g., if you had to write a, say, password cracker

```
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);
}
```

Marshaling/Unmarshaling

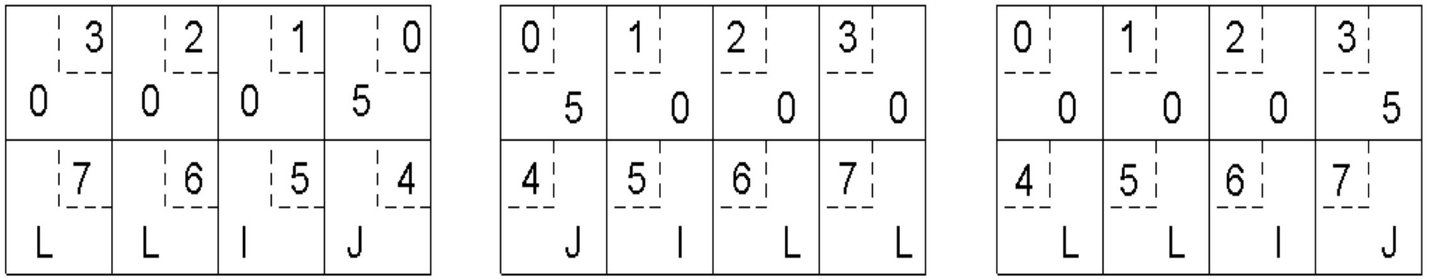
Example:

Serialize a string. First, integer representing length of the string, next, bytes representing the string.



- (From example) htonl() -- “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)

Endianness matters



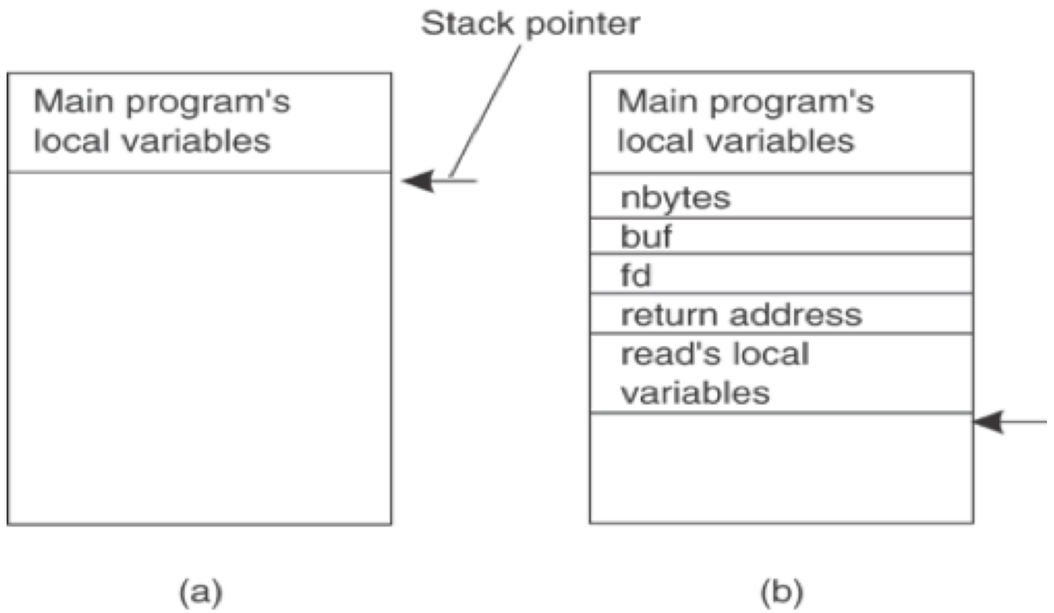
- a)Original message on x86 (Little Endian)
- b)The message after receipt on the SPARC (Big Endian)
- c)The message after being inverted. The little numbers in boxes indicate the address of each byte

Stubs & 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.

Parameter Passing in Local Procedure Calls



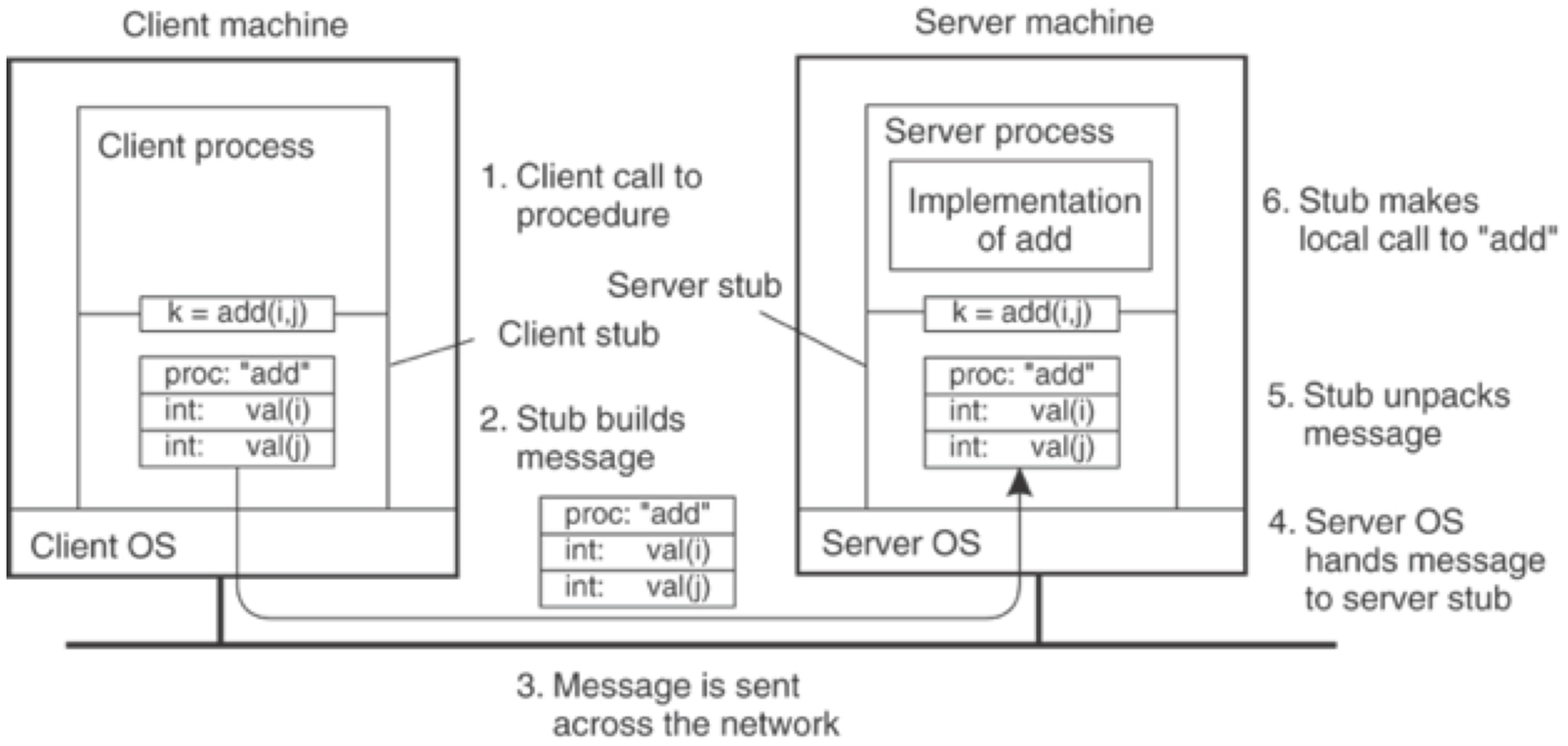
count = read(fd, buf, nbytes)

(a) Parameter passing in a local procedure call: the stack before the call to read

(b) The stack while the called procedure – read(fd, buf, nbytes) - is active.

WHAT ABOUT PARAMETERS?

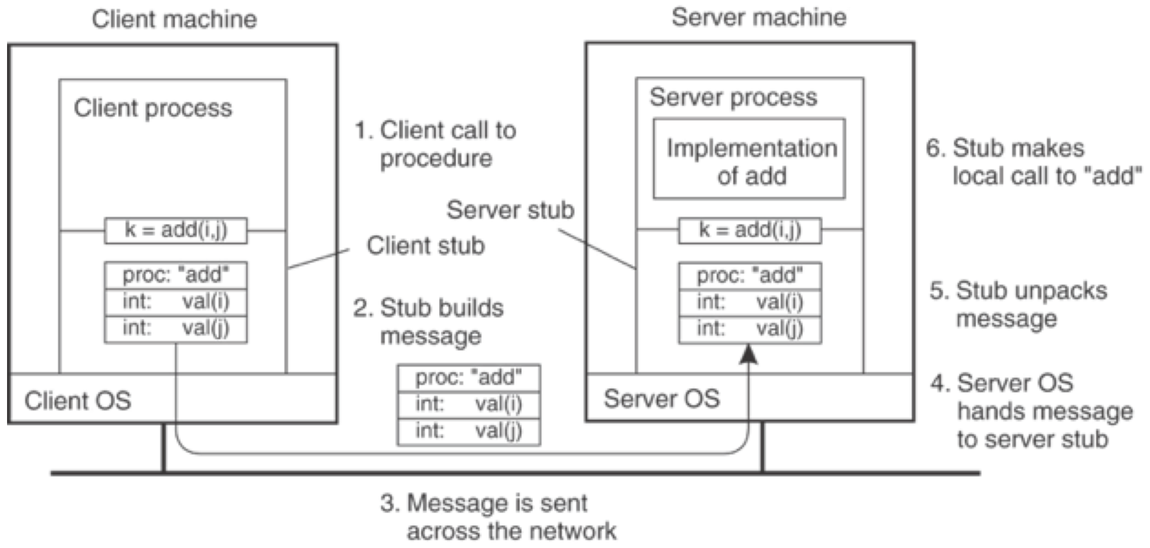
Passing value parameters



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

WHAT ABOUT PARAMETERS?

Passing value 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?

Two styles of RPC implementations

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.

RPCs want to look like LPCs

But they’re fundamentally different!

3 properties of distributed computing that make achieving transparency difficult:

- Memory access
- Partial failures
- Latency

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
- 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
  - one 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**
- Impossible in practice

- **At least once:**
- Only for idempotent operations

- **At most once**
- Zero, don’t know, or once

- **Zero or once**
- Transactional semantics

Exactly Once?

- Sorry – impossible to 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! (as long as idempotent)

- **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 (example in the book)
- If buffer is an input or output to the server stub, can eliminate a copy
  - E.g. if input parameter (e.g. call to write) need not be copied back

•Topic of Research, data center RPCs

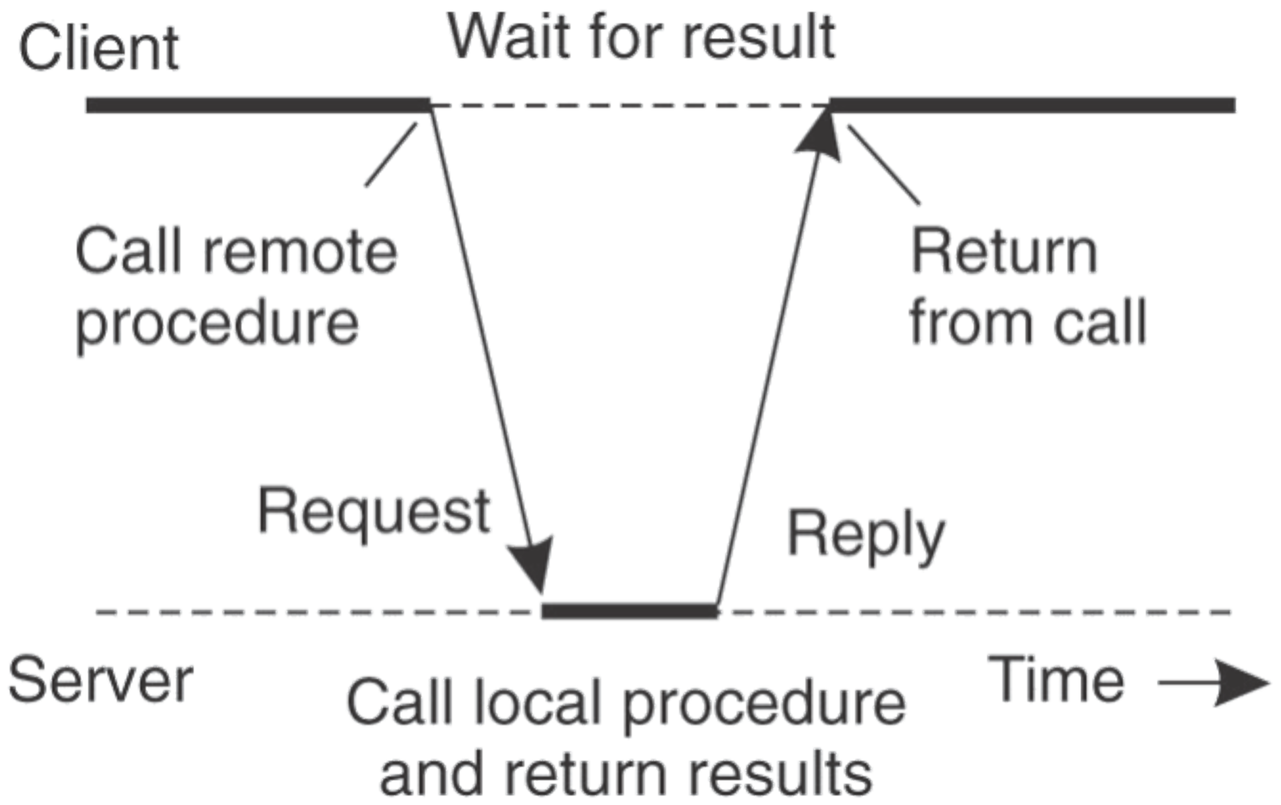
Dealing with Environmental Differences

- If my function does: read(foo, …)
- Can I make it look like it was really a local procedure call??

- Maybe!
  - Distributed filesystem…

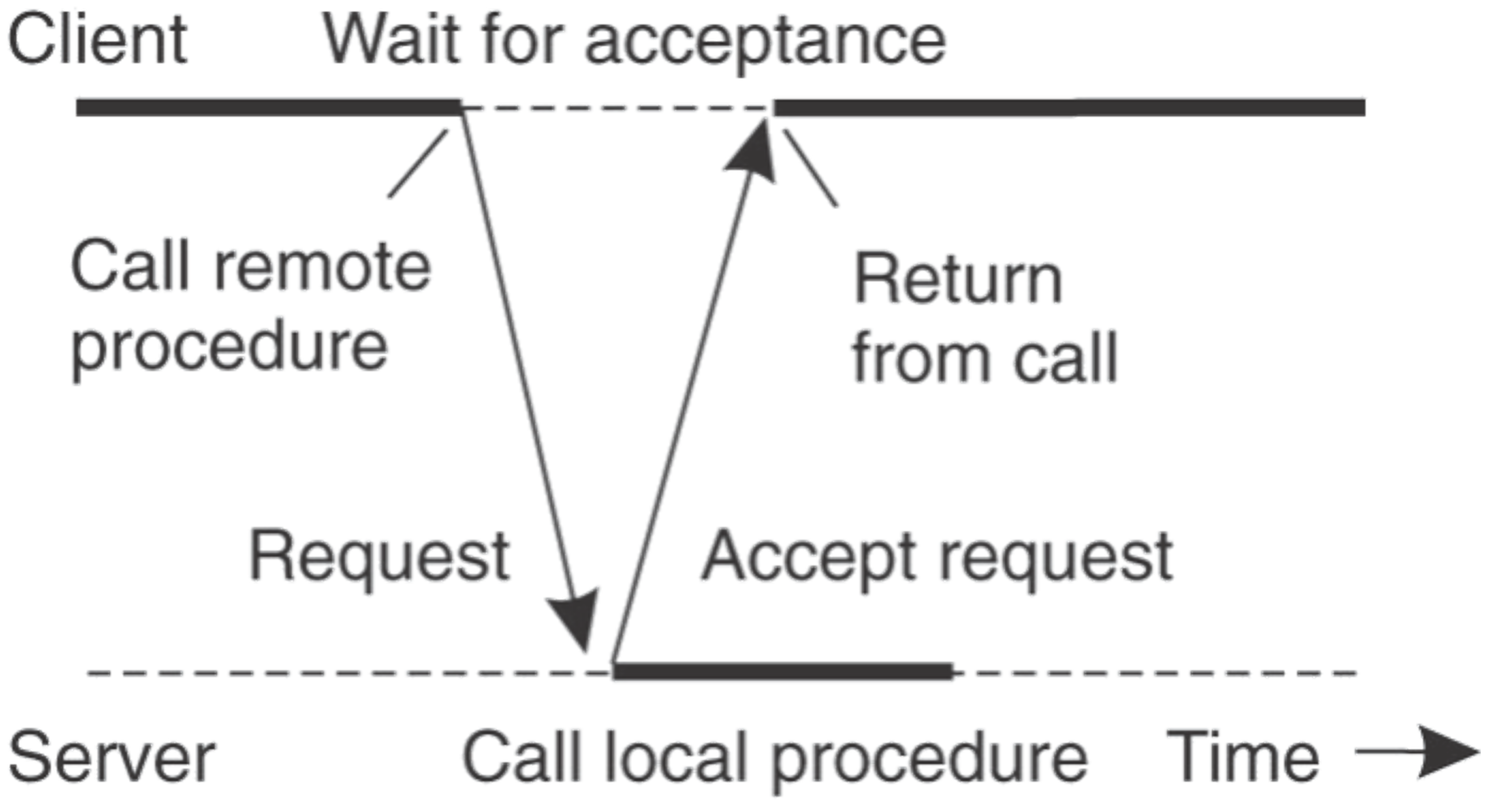
- But what about address space?
  - This is called distributed shared memory
  - People have kind of given up on it - it turns out often better to admit that you are doing things remotely

Asynchronous RPCs (1)



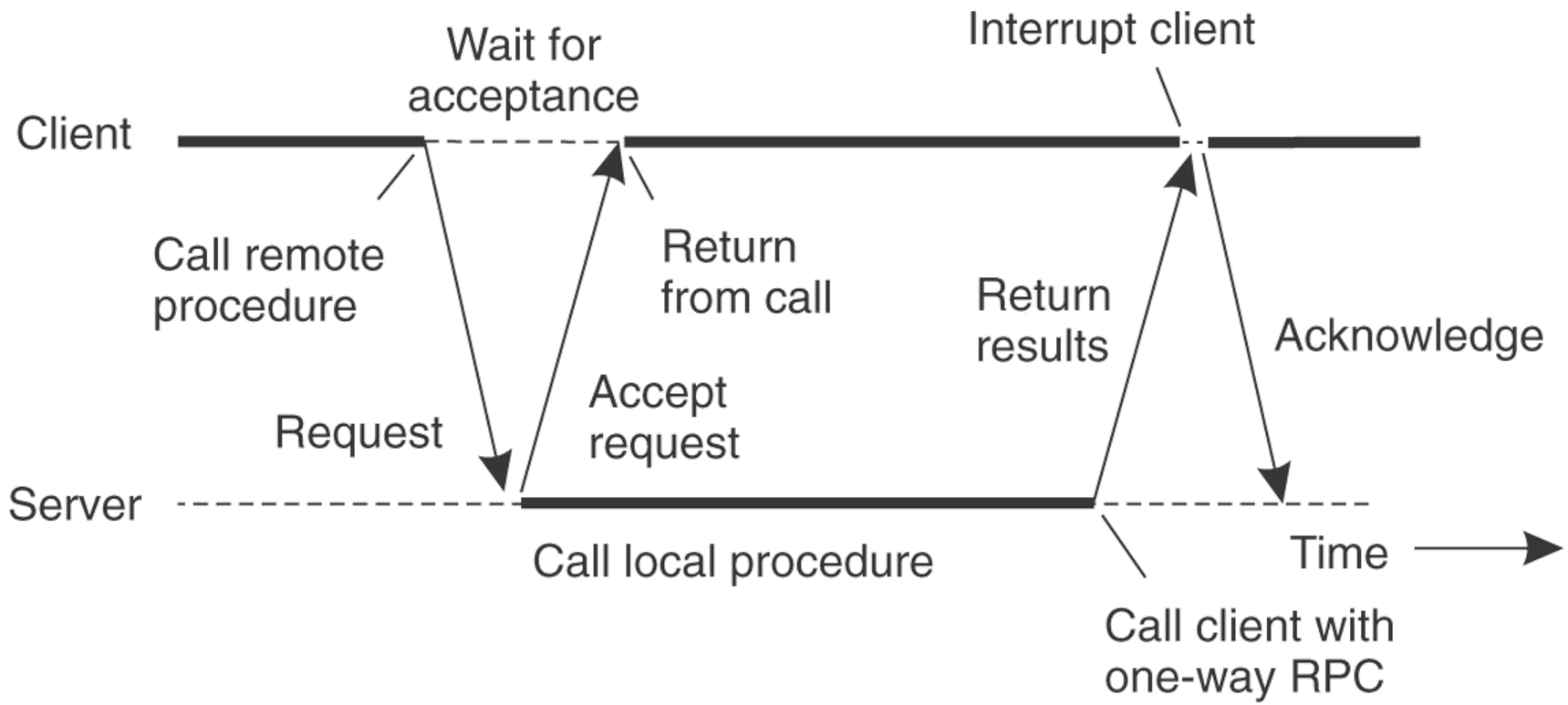
- The interaction between client and server in a traditional synchronous RPC.

Asynchronous RPCs (2)



- The interaction between client and server in an *asynchronous* RPC.

Asynchronous RPCs (3)



- A client-sever interaction using two *asynchronous* RPCs.

RPC IN GO

Go Example, client side

Client first dials the server

```
client, err := rpc.DialHTTP("tcp", serverAddress + ":1234")
if err != nil {
    log.Fatal("dialing:", err)
}
```

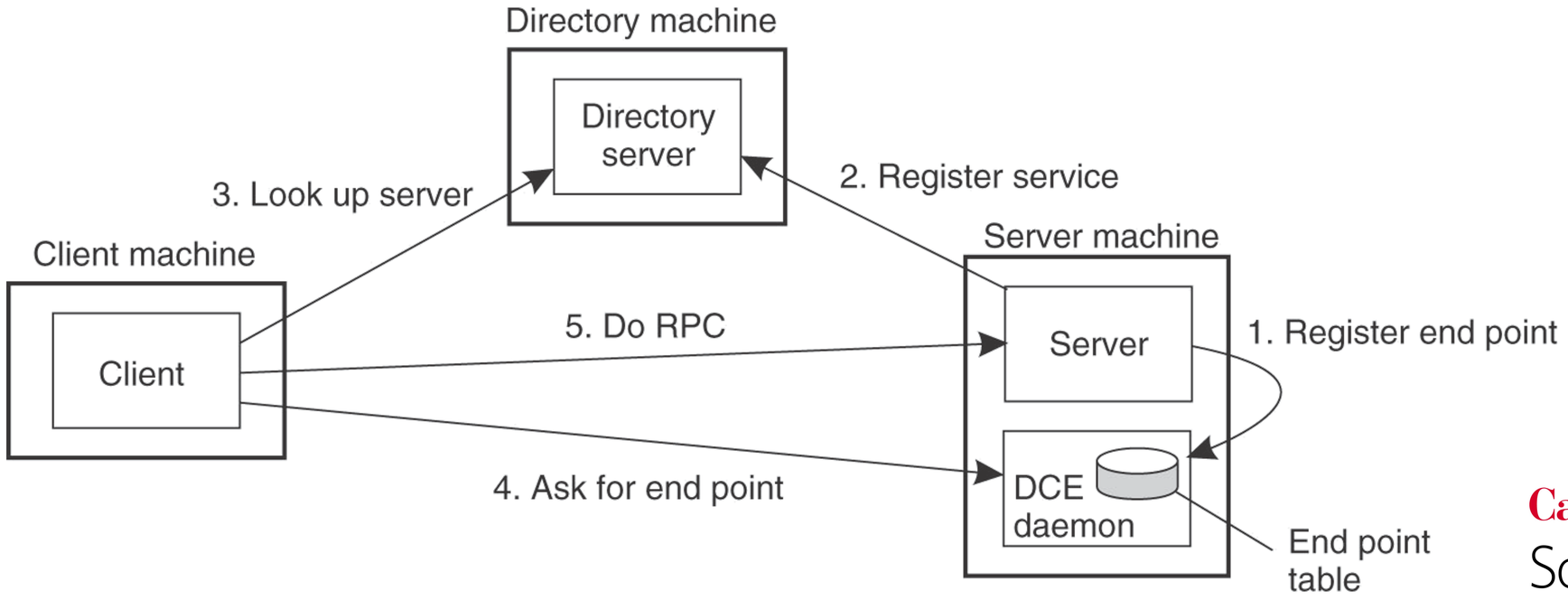
Then it can make a remote call:

```
// 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.
```

Binding a Client to a Server

- Registration of a server enables a client to locate the server and bind to it

  - Step 1: Locate the servers machine
  - Step 2: Locate the server on that machine



Go Example, server side

Basic RPC code:

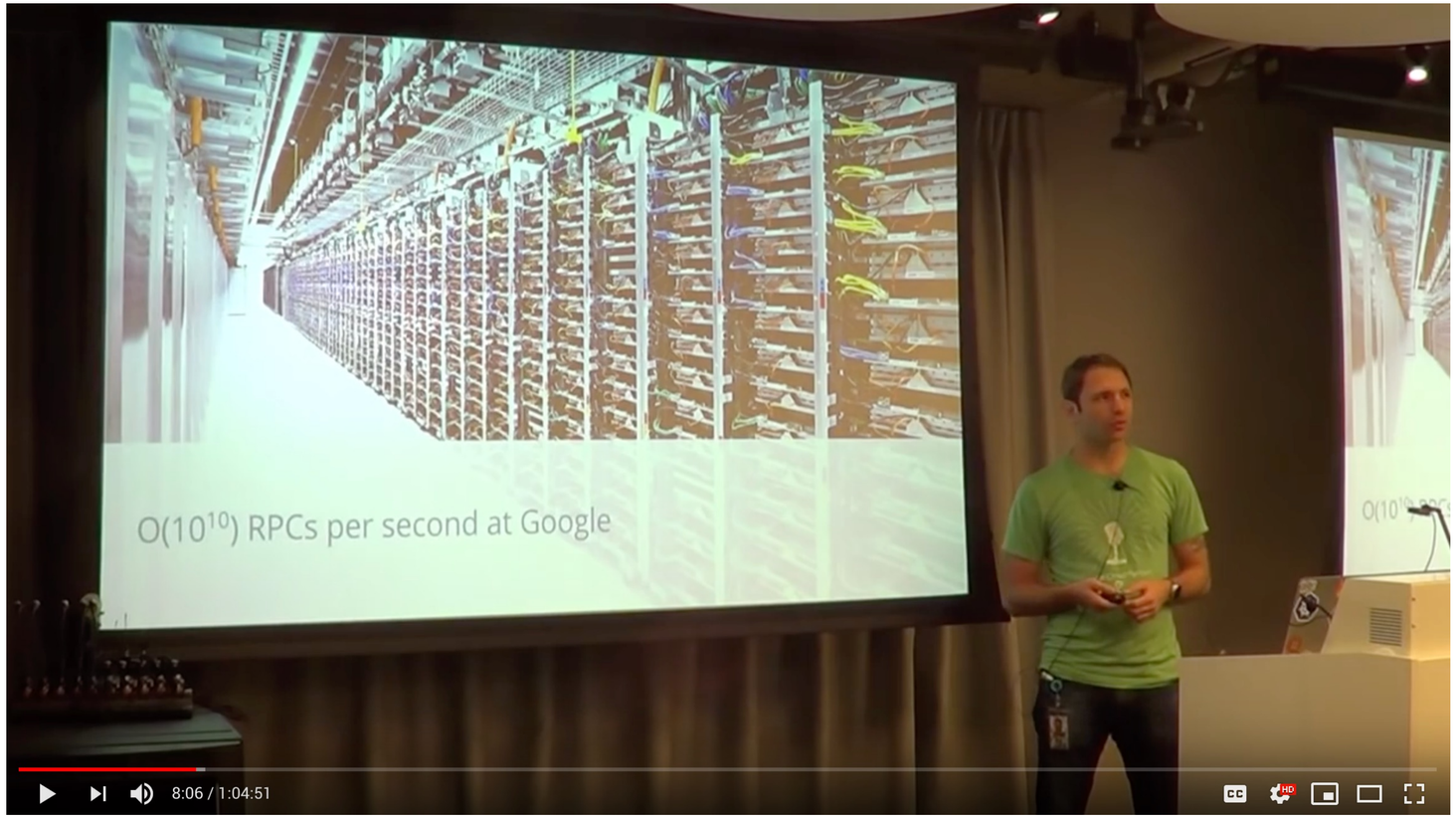
```
package server
import "errors"
type Args struct {  A, B int    }
type Quotient struct {  Quo, Rem int    }
type Arith int

func (t *Arith) Multiply(args *Args, reply *int) error {
    *reply = args.A * args.B
    return nil
}

func (t *Arith) Divide(args *Args, quo *Quotient) error {
    if args.B == 0 {
        return errors.New("divide by zero")
    }
    quo.Quo = args.A / args.B
    quo.Rem = args.A % args.B
    return nil
}
```

The server then calls (for HTTP service):

```
arith := new(Arith)
rpc.Register(arith)
rpc.HandleHTTP()
l, e := net.Listen("tcp", ":1234")
if e != nil {
    log.Fatal("listen error:", e)
}
go http.Serve(l, nil)
```

RPC is widely used

At Google, 10^{10} RPCs per second



https://www.youtube.com/watch?v=xb8u2s7cxzg&t=486s

RPC systems worth knowing about

gRPC, by Google (2015)

Finagle, by Twitter (2011)

Apache Thrift, by Facebook (2007)

Apache Avro (2009)

Takeaways

- **Remote procedure calls**
  - Simple way to pass control and data
  - Elegant transparent way to distribute applications
  - Not the only way…
- **Hard to provide true transparency**
  - Failures
  - Performance
  - Memory access
  - Etc.
- **Application writers have to decide how to deal with partial failures**
  - Consider: E-commerce application vs. game