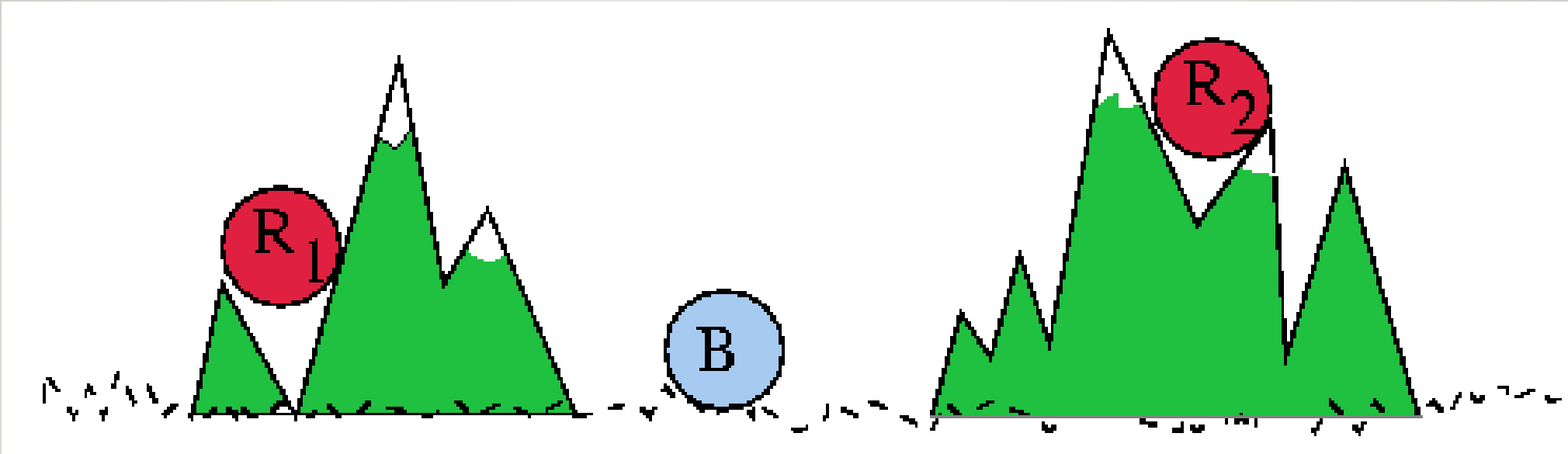
14-736 DISTRIBUTED SYSTEMS

RPC AND RMI (KESDEN, SPRING 2018)

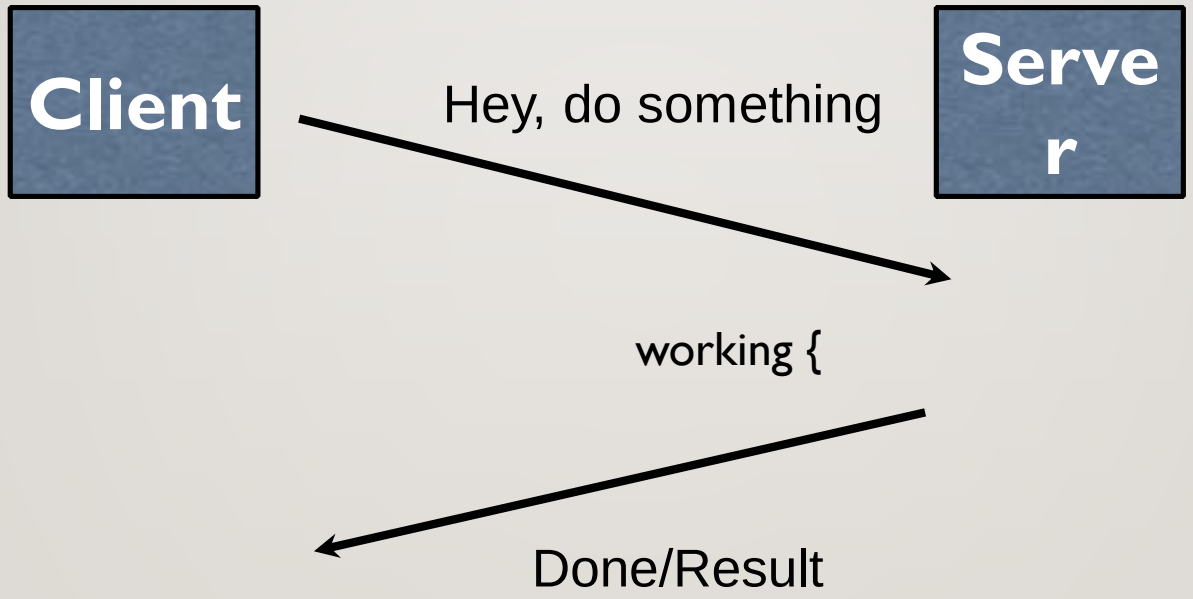
MANY THANKS TO PROFS. SRINI AND AGARWAL

…MOST OF THESE SLIDES ARE FROM THEIR FALL 2016 15-440/640 LECTURE 6 SLIDE DECK

A QUICK STORY:
TWO ARMIES (AND ONE MESSENGER)



COMMON COMMUNICATION PATTERN



WRITING IT 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);
}
```

Then wait for response, etc.

TODAY'S LECTURE

- RPC overview
- RPC challenges
- RPC other stuff

RPC – REMOTE PROCEDURE CALL: REPLACE COMMUNICATION VIA THE STACK WITH THE NETWORK

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

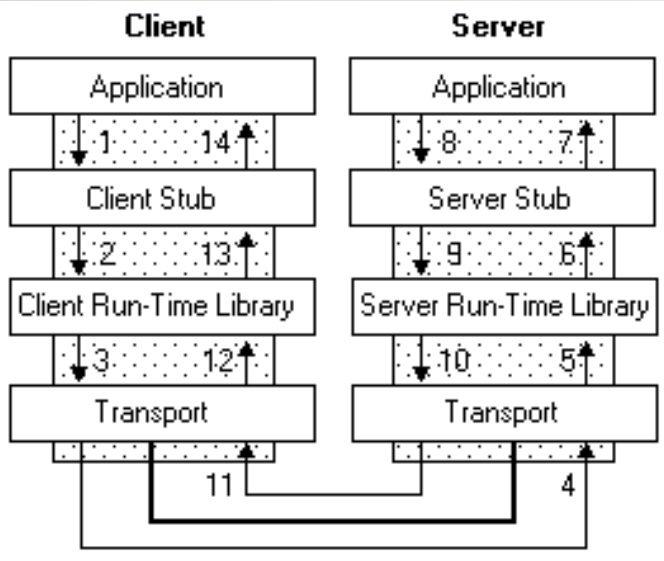


figure from Microsoft MSDN

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

8 GO EXAMPLE

- 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)
```

10 SERVER SIDE

- Basic RPC code:

```
package server
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 GOALS

- Ease of programming
- Hide complexity
- Automates task of implementing distributed computation
- 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 ;)

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

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

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

WRITING IT BY HAND - (AGAIN…)

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

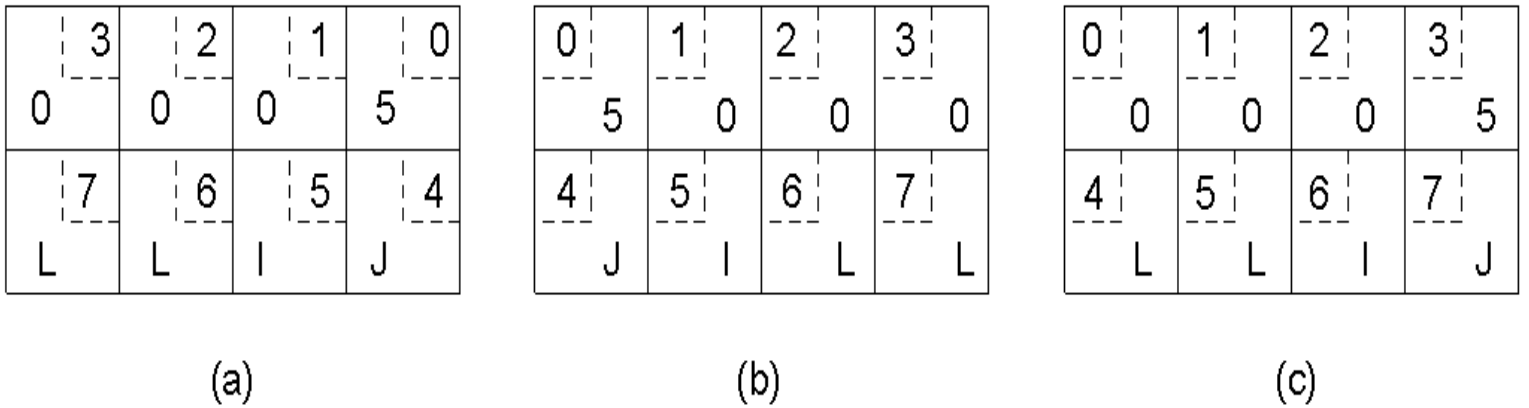
```
struct foomsg {
  u_int32_t len;
}

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);
}
```

Then wait for response, etc.

MARSHALING AND UNMARSHALING

- (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)

ENDIAN



a) Original message on x86 (Little Endian)

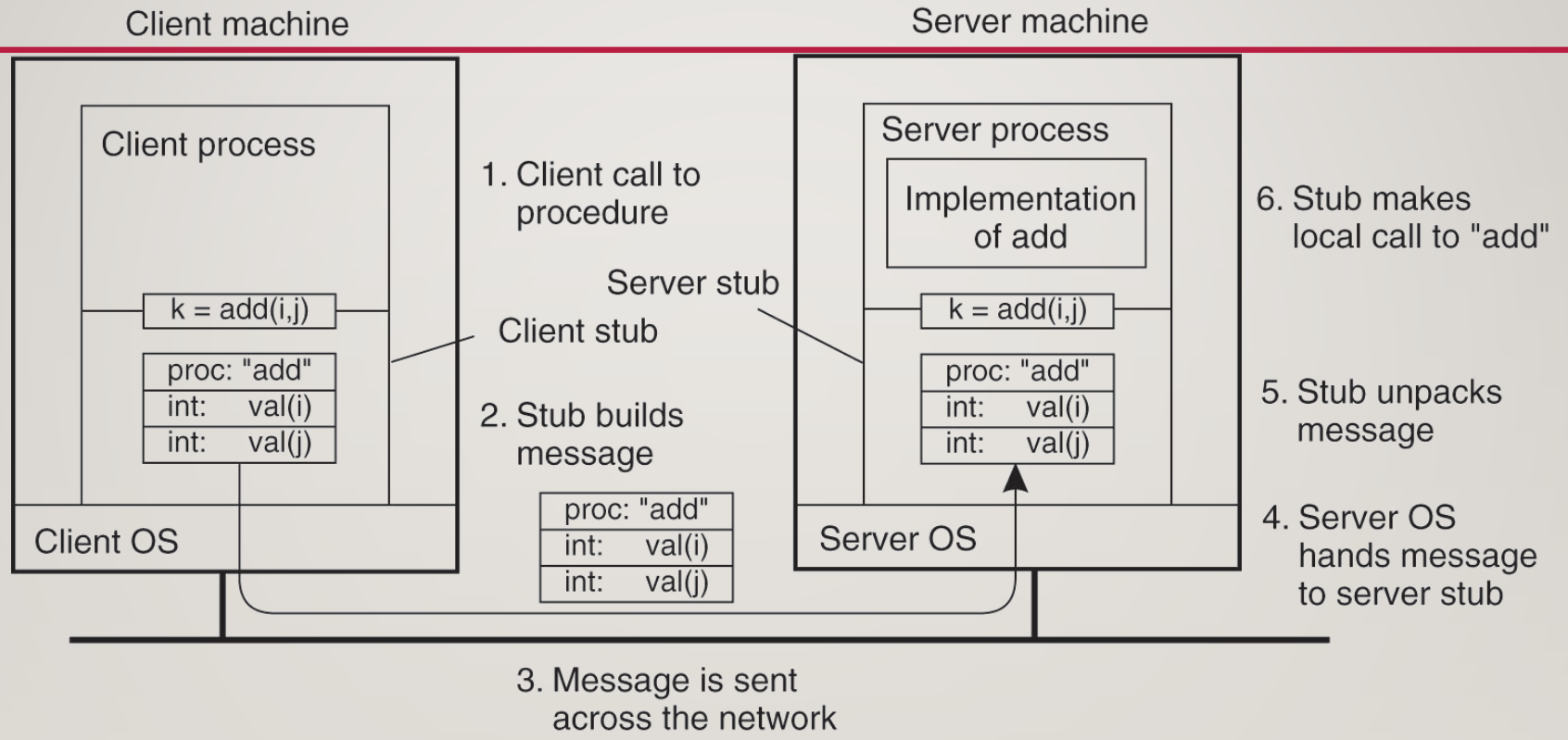
b) The message after receipt on the SPARC (Big Endian)

c) The message after being inverted. The little numbers in boxes indicate the address of each byte

"STUBS" 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.

PASSING VALUE PARAMETERS (1)



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

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 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?

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.

TODAY'S LECTURE

- RPC overview
- RPC challenges
- RPC other stuff

RPC VS. LPC

- 3 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
  - 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 - 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 writes (writev() and friends)

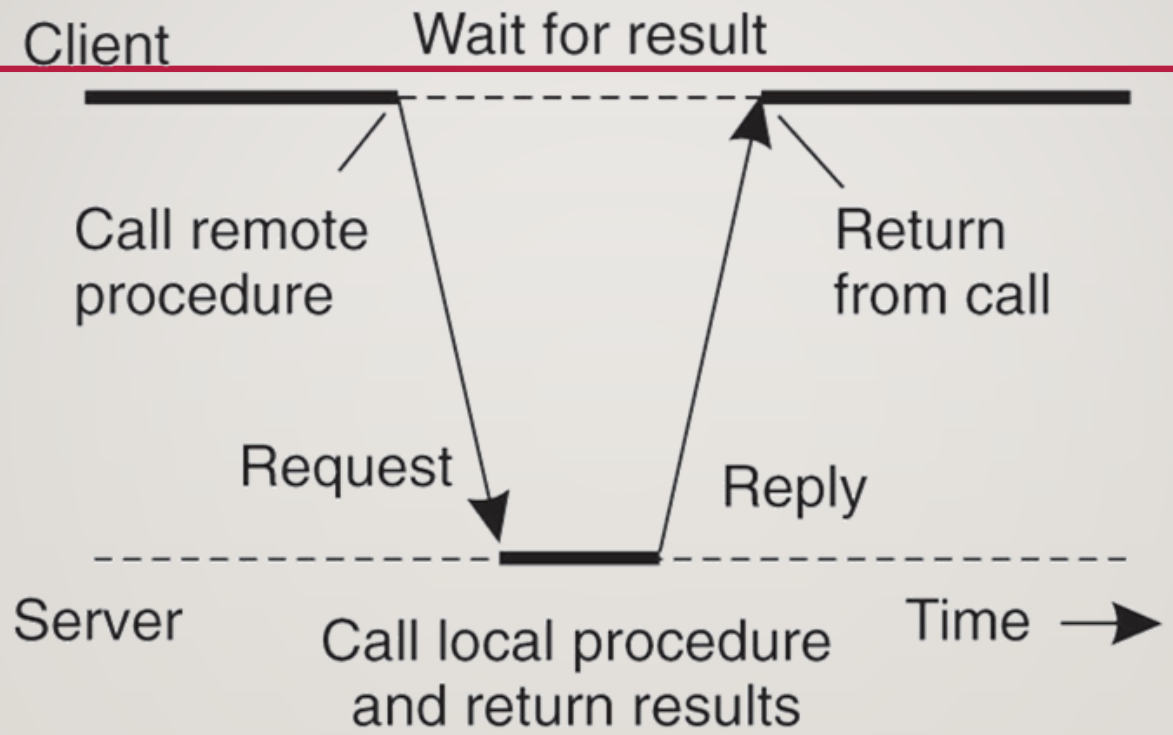
DEALING WITH ENVIRONMENTAL DIFFERENCES

- 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

TODAY'S LECTURE

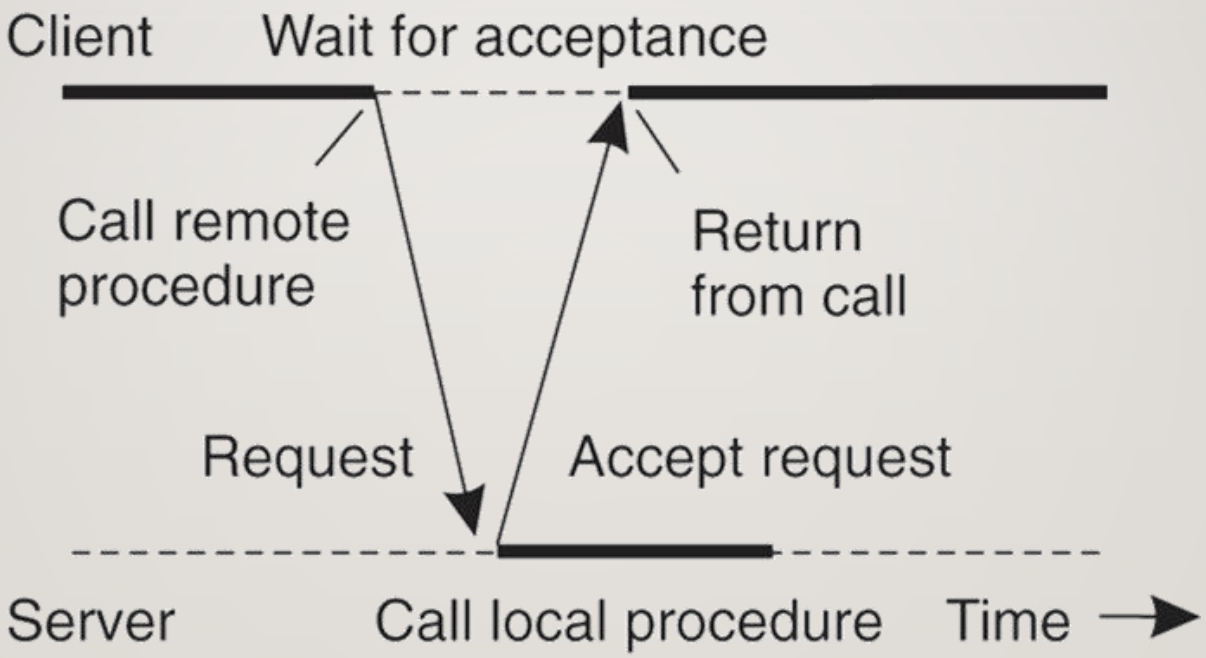
- RPC overview
- RPC challenges
- RPC other stuff

ASYNCHRONOUS RPC (1)



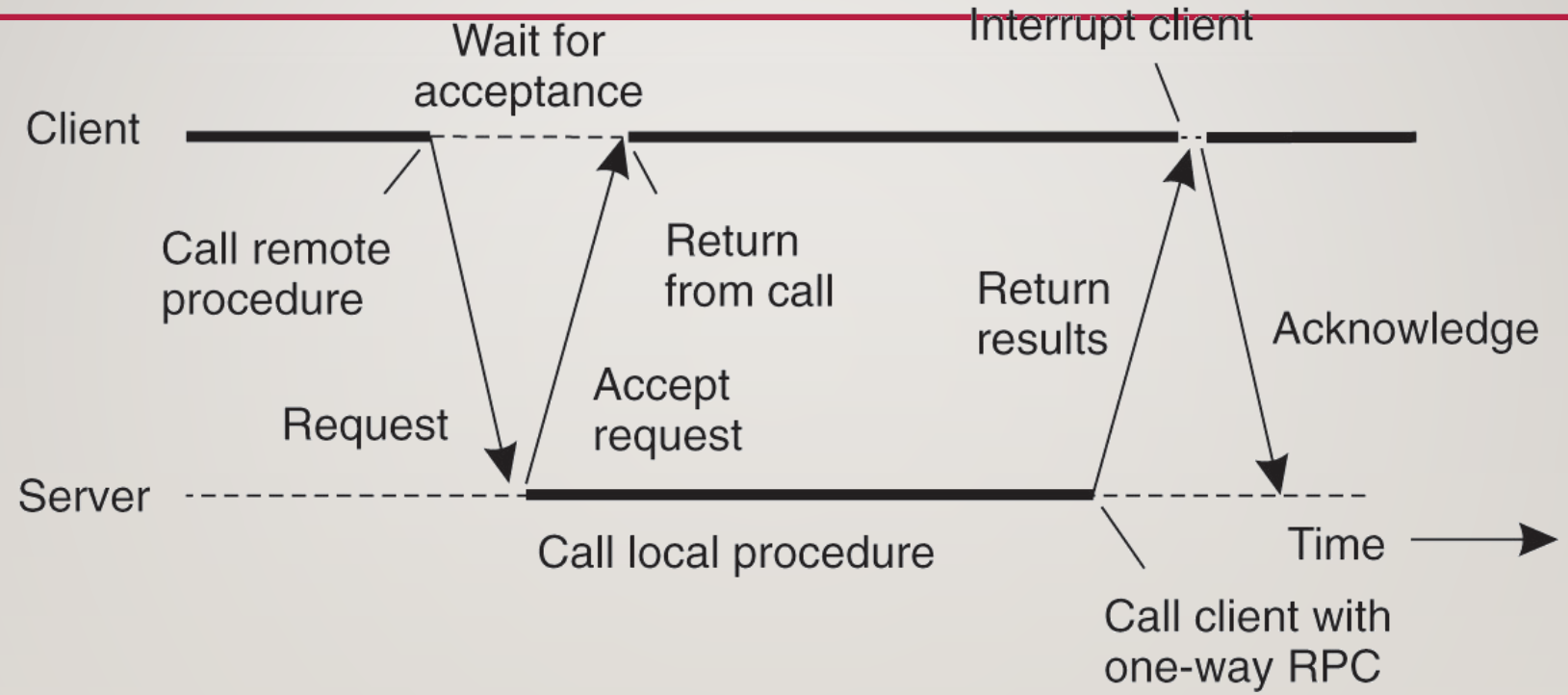
- The interaction between client and server in a traditional RPC.

ASYNCHRONOUS RPC (2)



- The interaction using asynchronous RPC.

ASYNCHRONOUS RPC (3)



- A client and server interacting through two asynchronous RPCs.

GO EXAMPLE

- 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.
```

USING RPC

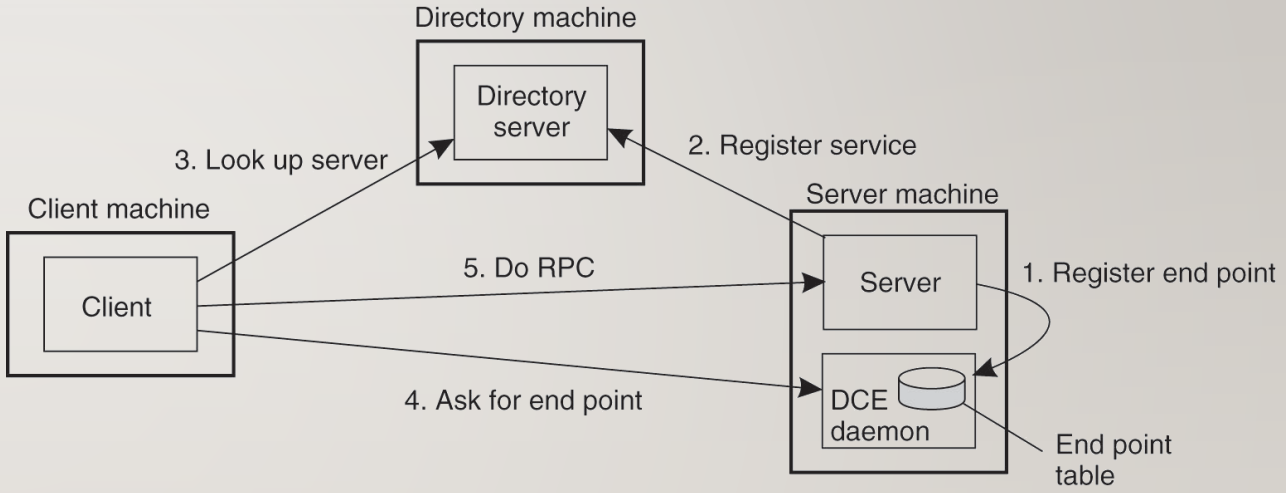
- How about a distributed bitcoin miner.
- Three classes of agents:
  1. Request client. Submits cracking request to server. Waits until server responds.
  2. Worker. Initially a client. Sends join request to server. Now it should reverse role & become a server. Then it can receive requests from main server to attempt cracking over limited range.
  3. Server. Orchestrates whole thing. Maintains collection of workers. When receive request from client, split into smaller jobs over limited ranges. Farm these out to workers. When finds bitcoin, or exhausts complete range, respond to request client.

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.



42 SERVER SIDE

- Basic RPC code:

```
package server
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)
```

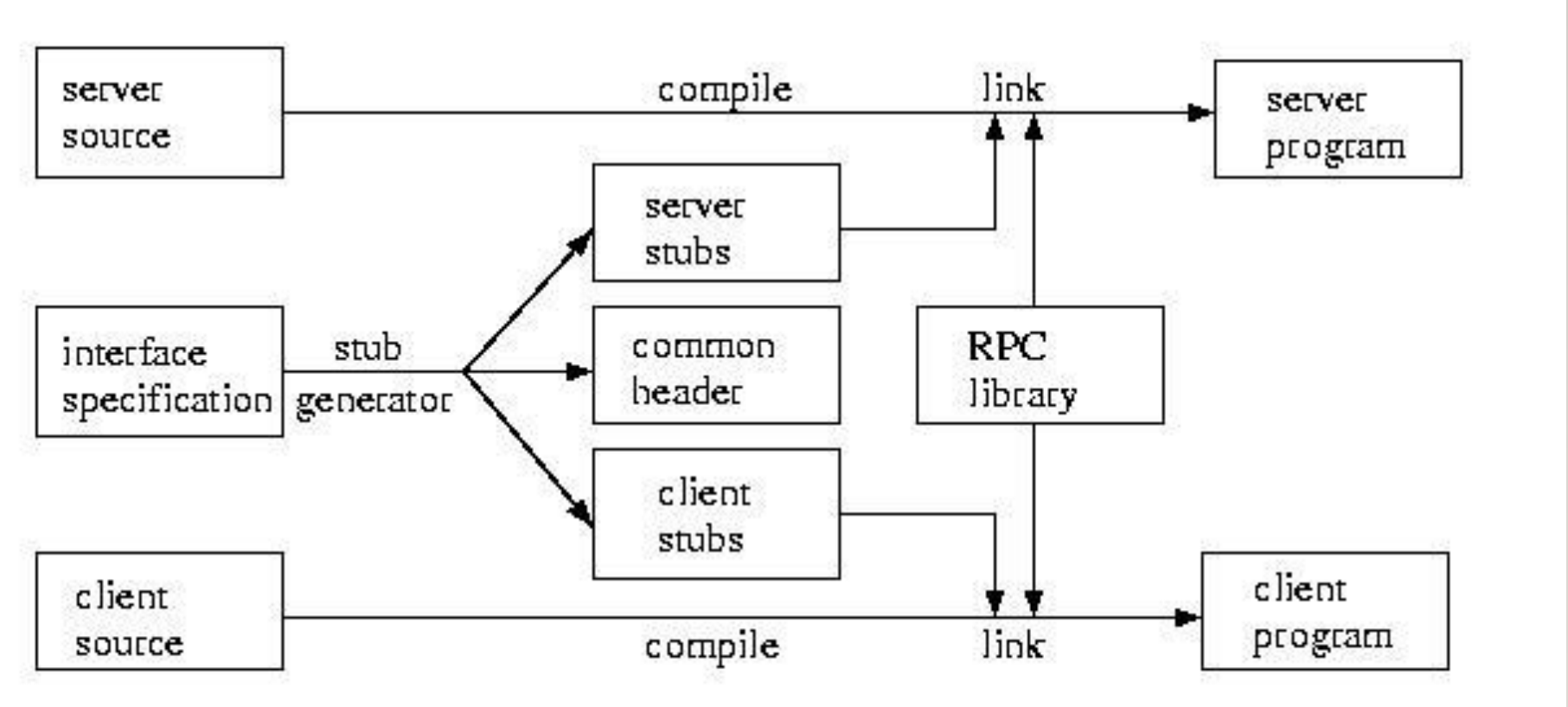
SUMMARY: EXPOSE REMOTENESS TO CLIENT

- 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 it
  - "Worse is better"

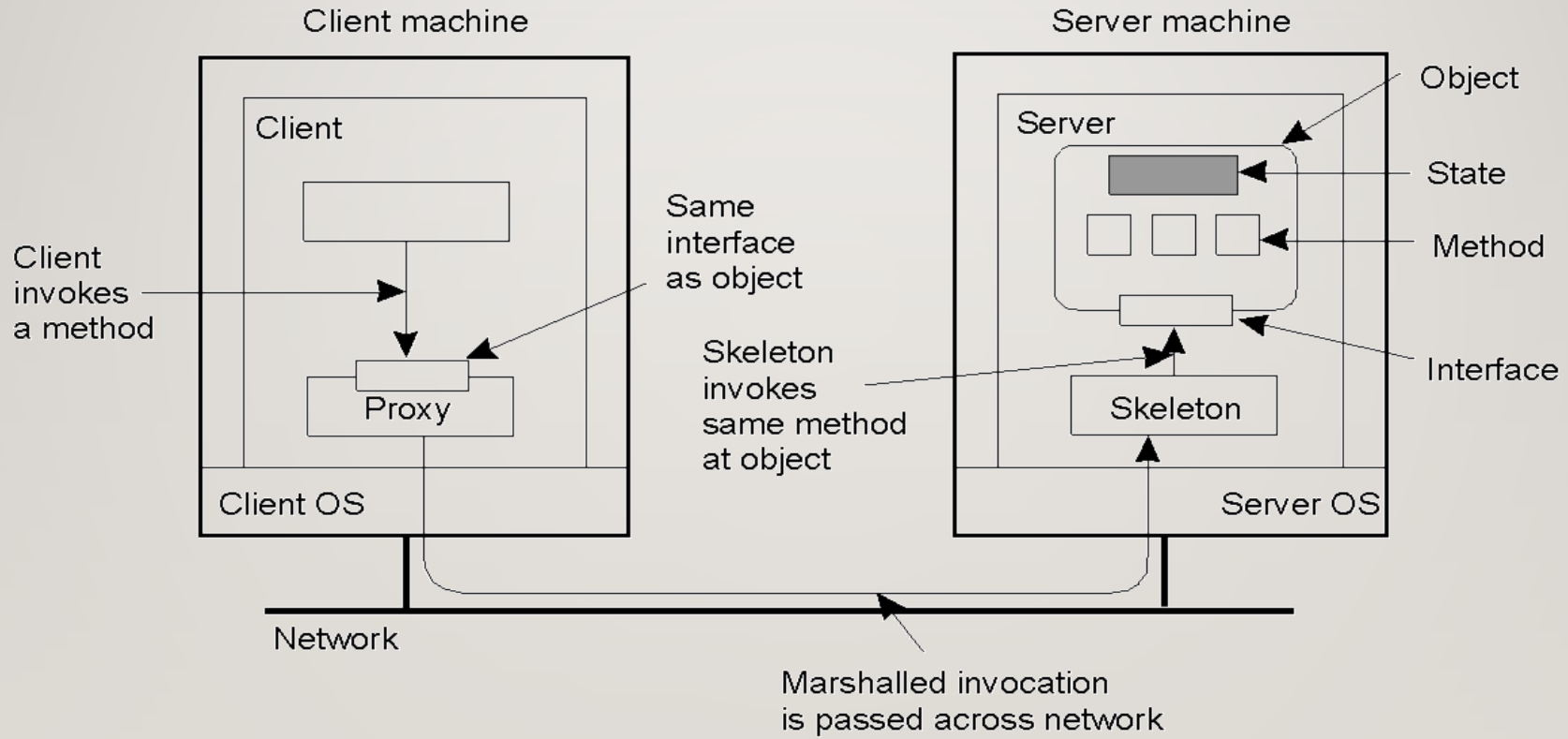
SUN/ONC RPC DIAGRAM



DISTRIBUTED OBJECTS

- Maintain an understanding of the identities of stateful objects
- Invoke methods upon these remotely accessible objects
- Obtain and pass around references to these remotely accessible objects
- Note: Objects versus Classes
  - There can be many instances, some remotely accessible, some note

DISTRIBUTED OBJECTS



- Common organization of a remote object with client-side proxy.

JAVA IS MY FAVORITE RMI EXAMPLE: SIMPLE SOLUTIONS, EASY USE

- Interfaces provide common reference type for proxy and remote instances
- Serializable vs Remote interface
  - Remote: Send Remote-Object-Reference (ROR) and localize to proxy reference
  - Does not implement Remote: Needs to be Serializable (and not Remote):
    - Send copy and deserialize
  - Neither: Error
- Registry: Trade name for ROR

JAVA RMI

- Original version:
  - "rmic" generated proxy and skeleton classes from the base .class file
- Step to simplicity:
  - The skeletons were really formulaic. They all had the same code. All they did was invoke a local method
    - Replace with a dispatcher that parses the incoming invocation and dispatches it locally
  - Proxies are formulaic. Interfaces provide all the methods, arguments, etc. ROR provides server information
    - Automatically generate them dynamically.
    - No more need for rmic. All dynamic.

JAVA RMI EXAMPLE: INTERFACE

HelloInterface.java:

```
interface HelloInterface extends Remote {
  public String sayHello(String name)
        throws RemoteException;
}
```

JAVA RMI EXAMPLE: SERVER

Hello.java

```
class Hello extends UnicastRemoteObject implements HelloInterface {
  private static final String serverName = "hello";

  public Hello() throws RemoteException { }

  public String sayHello(String name) throws RemoteException {
    return "Hello World! Hello " + name;
   }

  public static void main (String []args) {
    try {
      Hello server = new Hello();
      Naming.rebind (serverName, server);
      System.out.println ("Hello Server ready");
    } catch (Exception e) {
        e.printStackTrace();
    }
  }
}
```

JAVA RMI EXAMPLE: CLIENT

HelloClient.java

```
class HelloClient {

  static void main (String []args) {
    try {
      String HelloServerURL = args[1];
      System.setSecurityManager (new RMISecurityManager());

      HelloInterface hello = (HelloInterface) Naming.lookup(HelloServerURL);

      String theGreeting = hello.sayHello (args[0]); System.out.println (theGreeting);
    } catch (Exception e) {
      e.printStackTrace();
    }
  }
}
```