15-440 Distributed Systems

RPC

Building up to today

- Few lectures ago: Abstractions for *communication*
    - example: TCP masks some of the pain of communicating across unreliable IP

- Last time: Abstractions for *computation*

Splitting computation across the network

- We’ ve looked at primitives for computation and for communication.

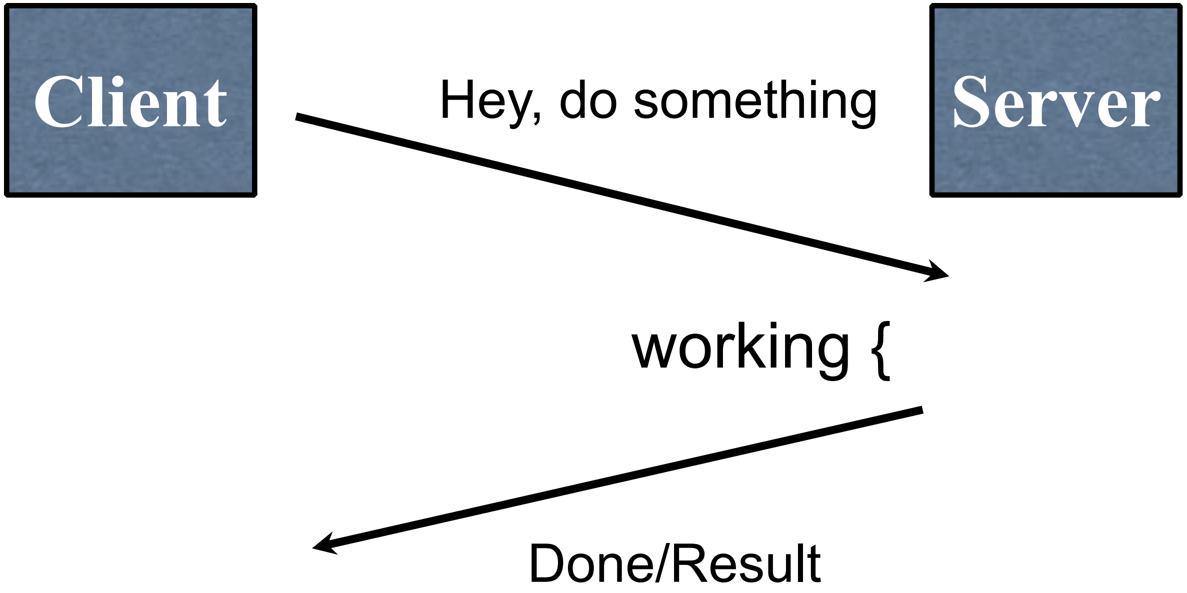
- Today, we’ ll put them together

Key question:

What programming abstractions work well to split work among multiple networked computers?

(caveat: we’ ll be looking at many possible answers to this question...)

Common communication pattern



Client

Hey, do something

Server

working {

Done/Result

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

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

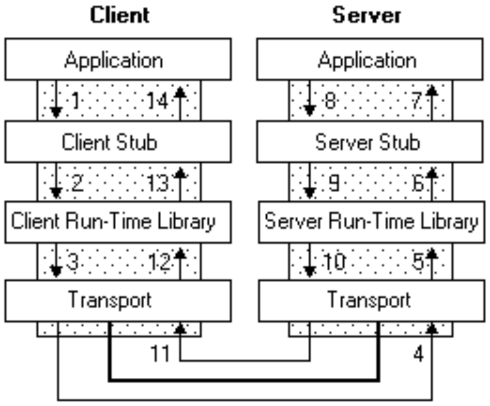


figure from Microsoft MSDN

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

Go Example

- Need some setup in advance of this but…

```
// 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)
```

Full 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)
```

Server Side

```
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 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

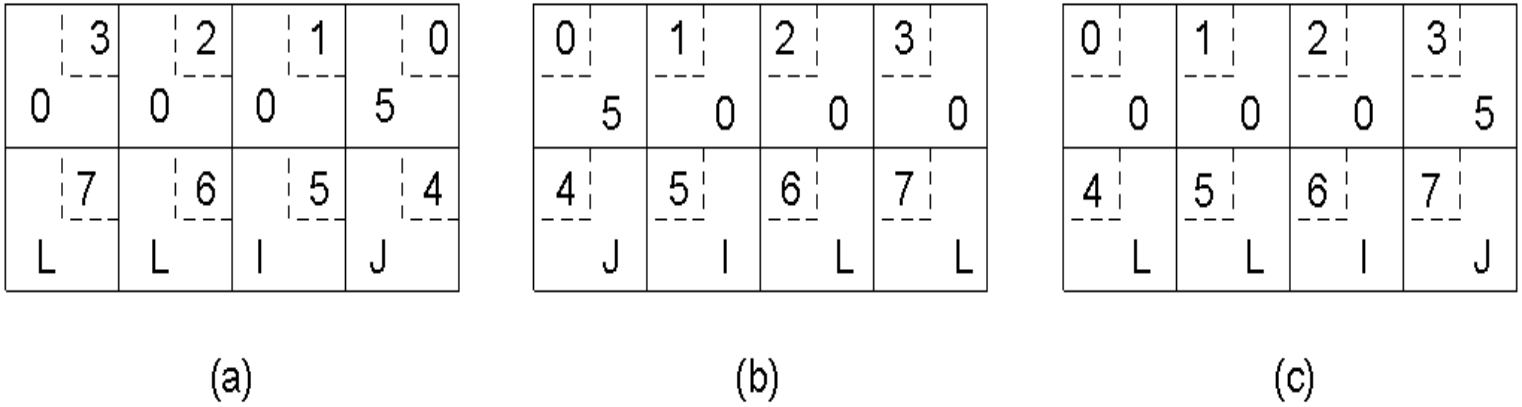
```
struct foomsg {
  u_int32_t len;
}

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)

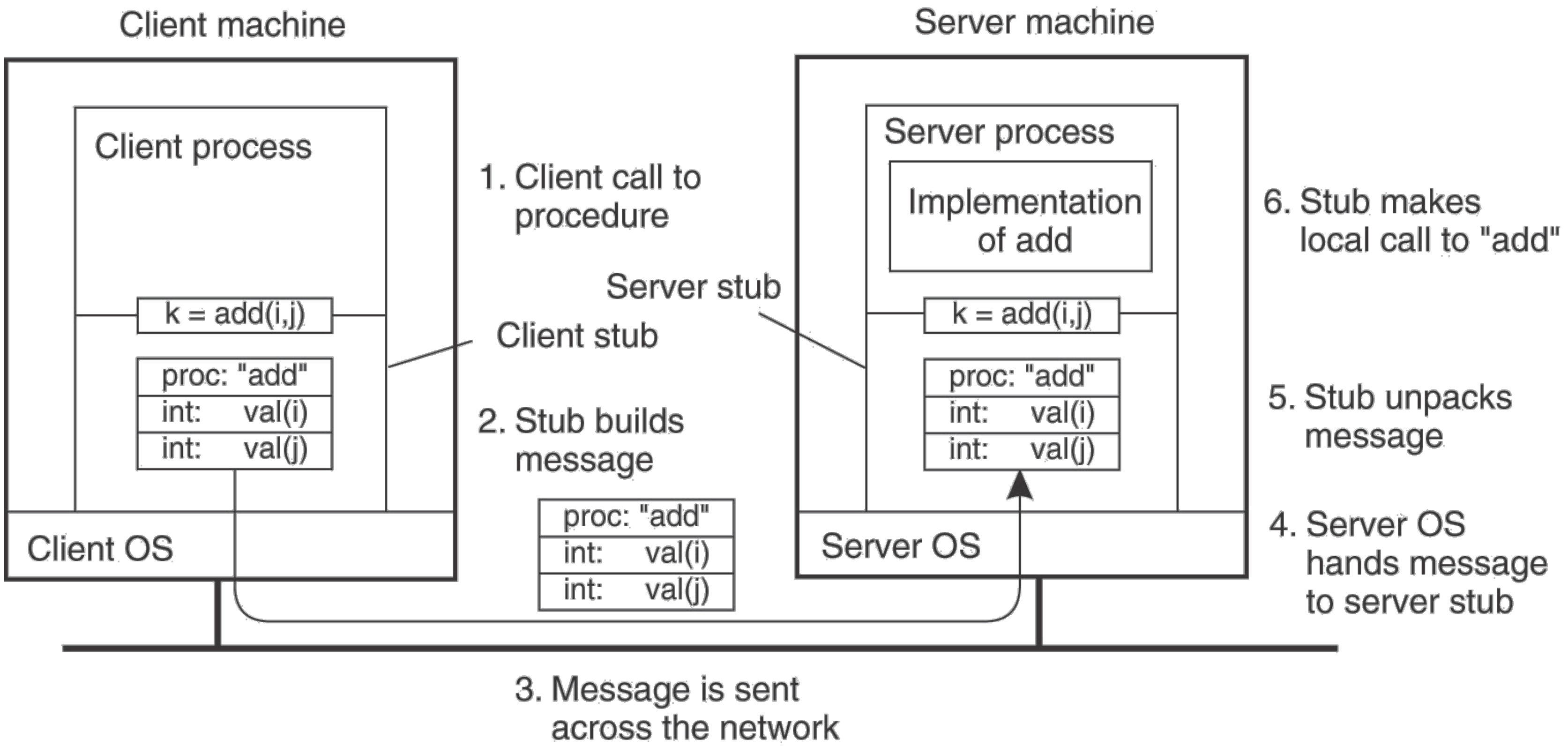
b) The message after receipt on the SPARC (Big Endian)

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 (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 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

- 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
  - 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

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…

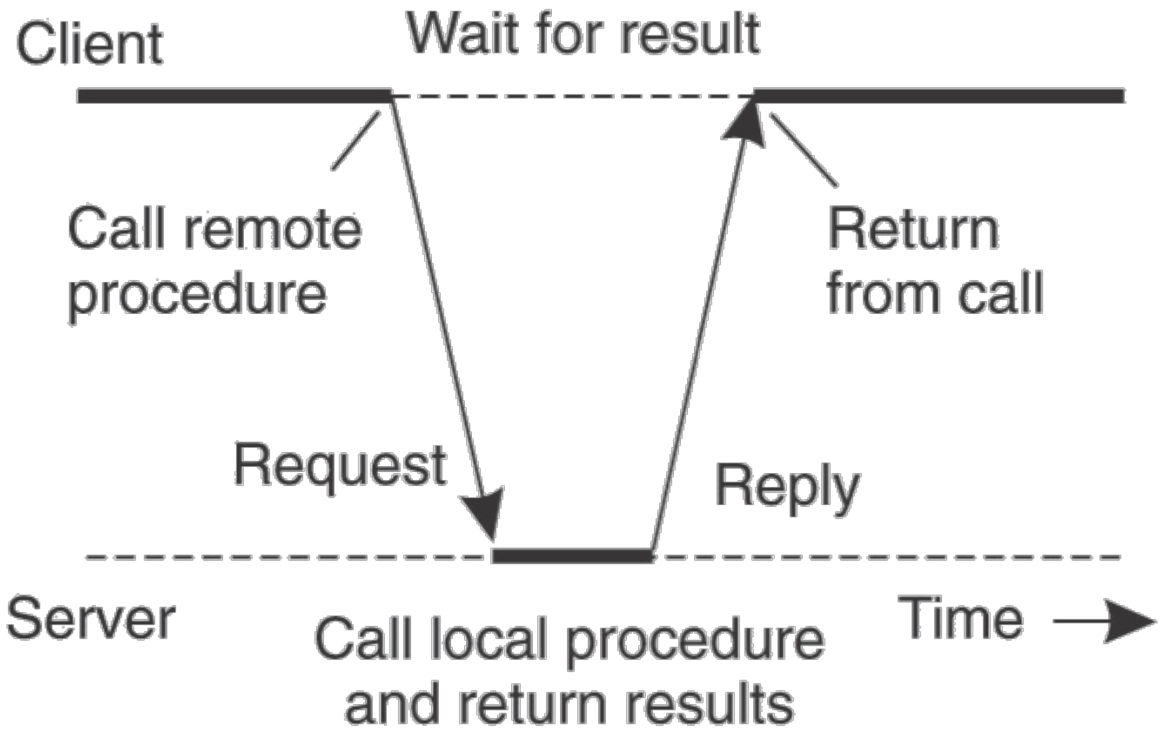
- Hard to provide true transparency
  - Failures
  - Performance
  - Memory access
  - Etc.

- How to deal with hard problem → give up and let programmer deal with it
  - “Worse is better”

Today's Lecture

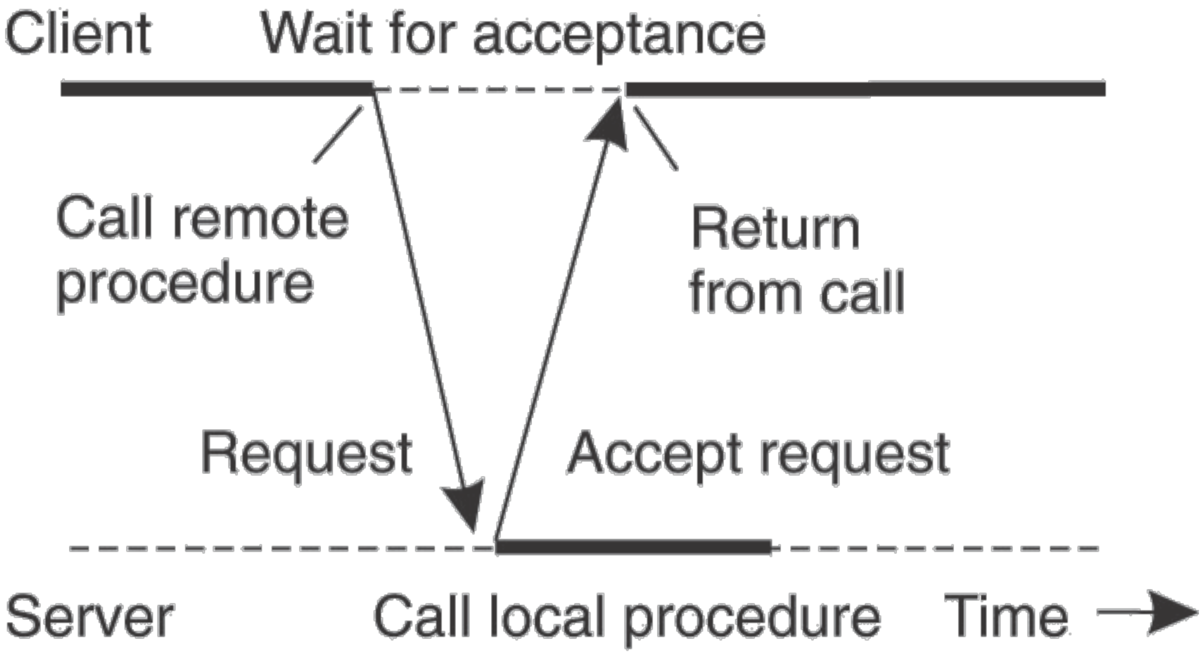
- RPC overview
- RPC challenges
- RPC other stuff

Asynchronous RPC (1)



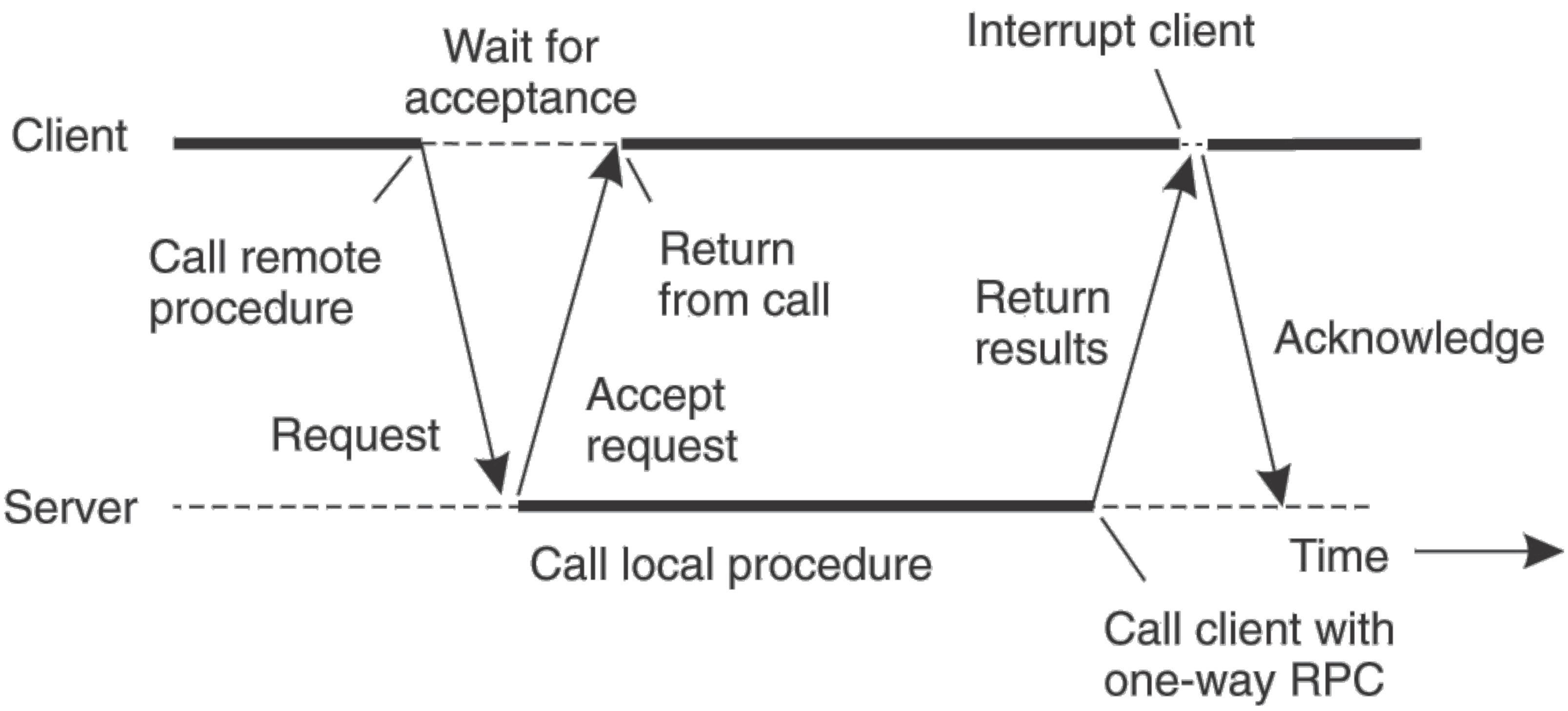
- The interaction between client and server in a traditional RPC.

Asynchronous RPC (2)



- The interaction using asynchronous RPC.

Asynchronous RPC (3)



- A client and server interacting through two asynchronous RPCs.

Go Example

```
// 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. Similar to past Project 1, but designed to use RPC

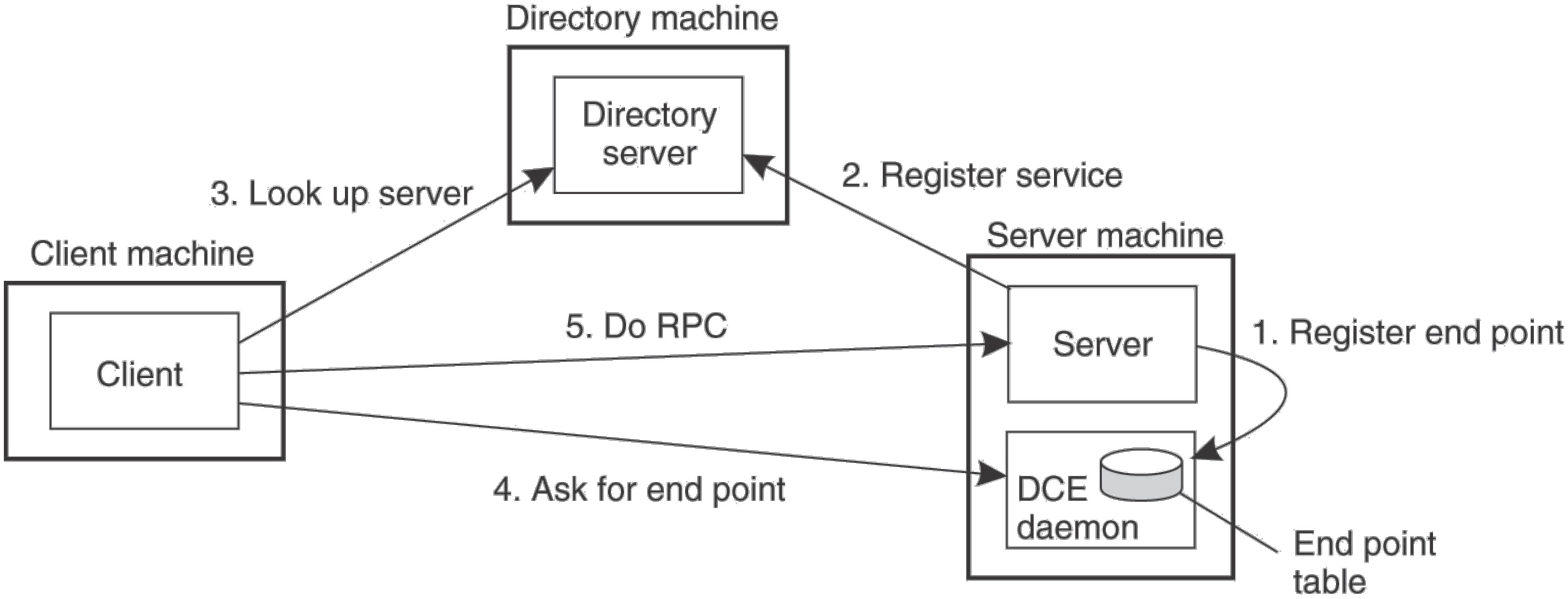
- Three classes of agents:
  1. Request client. Submits cracking request to server. Waits until server responds.
  2. Worker. Initially a client. Sends join request to server. Now it should reverse role & become a server. Then it can receive requests from main server to attempt cracking over limited 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.



Example Marshaling Format: JSON

- Data structures declared as:

```
// Linked list element
type BufEle struct {
    Val interface{}
    Next *BufEle
}

type Buf struct {
    Head *BufEle         // Oldest element
    tail *BufEle         // Most recently inserted element
    cnt int              // Number of elements in list
}
```

- (Note that only upper case names get marshaled.)

Example Marshaling Format: JSON

- Add method to bufi:

```
func (bp *Buf) String() string {
    b, e := json.MarshalIndent(*bp, "", "  ")
    if e != nil {
          return e.Error()
    }
    return string(b)
}
```

- Empty buffer

```
{
  "Head": null
}
```

Example Marshaling Format: JSON

- After inserting "pig", "cat", "dog":

```
{
  "Head": {
    "Val": "pig",
    "Next": {
      "Val": "cat",
      "Next": {
        "Val": "dog",
        "Next": null
      }
    }
  }
}
```

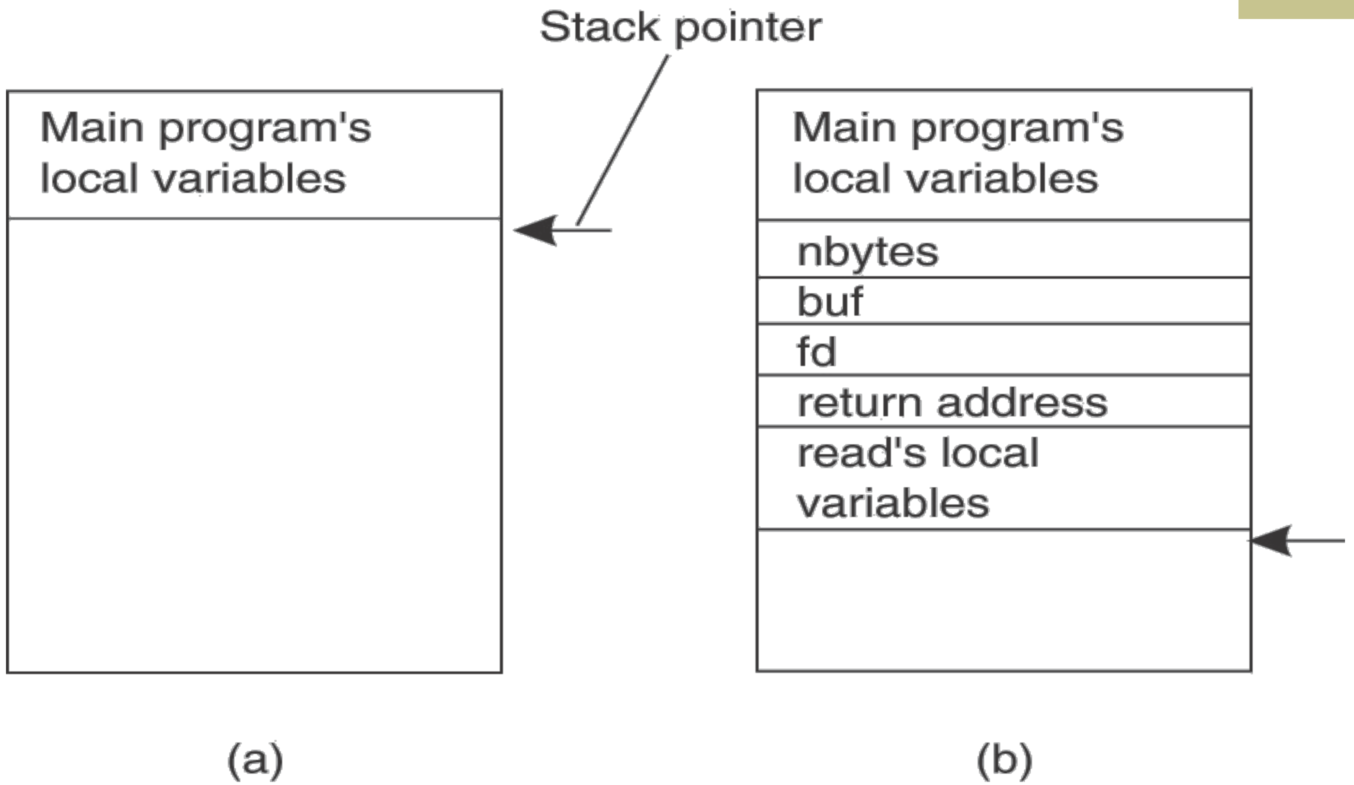
Other RPC systems

- 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.

```
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
  => Handling machine failures makes RPC different than procedure calls
```

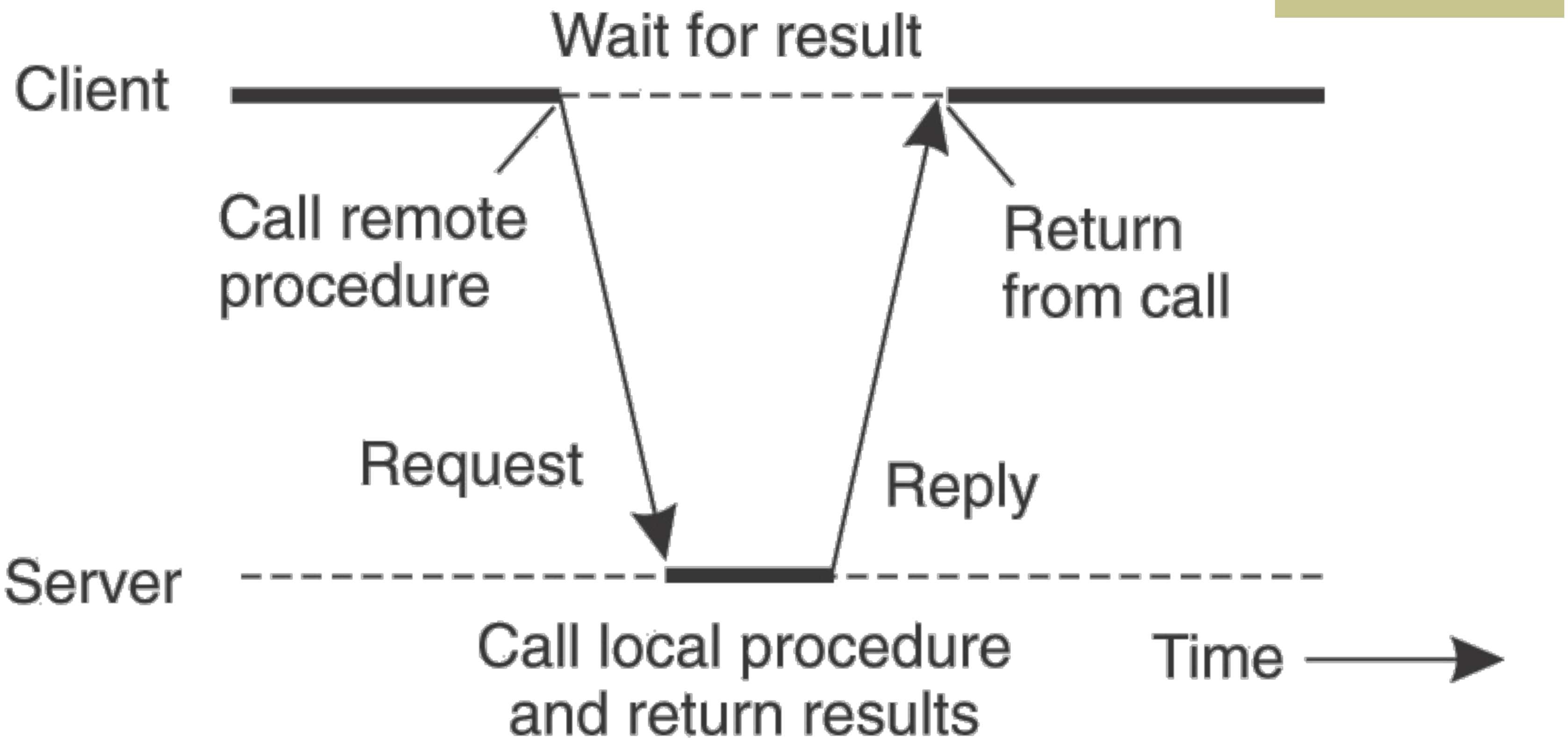
comparison from Kaashoek, 6.842 notes

Conventional Procedure Call



- (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.

Client and Server Stubs



- Principle of RPC between a client and server program.

Client-server architecture

- Client sends a request, server replies w. a response
  - Interaction fits many applications
  - Naturally extends to distributed computing
- Why do people like client/server architecture?
  - Provides fault isolation between modules
  - Scalable performance (multiple servers)
  - Central server:
    - Easy to manage
    - Easy to program

Developing with RPC

1. Define APIs between modules
   - Split application based on function, ease of development, and ease of maintenance
   - Don’t worry whether modules run locally or remotely
2. Decide what runs locally and remotely
   - Decision may even be at run-time
3. Make APIs bullet proof
   - Deal with partial failures

SunRPC

- Venerable, widely-used RPC system
- Defines “XDR” (“eXternal Data Representation”) -- C-like language for describing functions -- and provides a compiler that creates stubs

```
struct fooargs {
  string msg<255>;
  int baz;
}
```

And describes functions

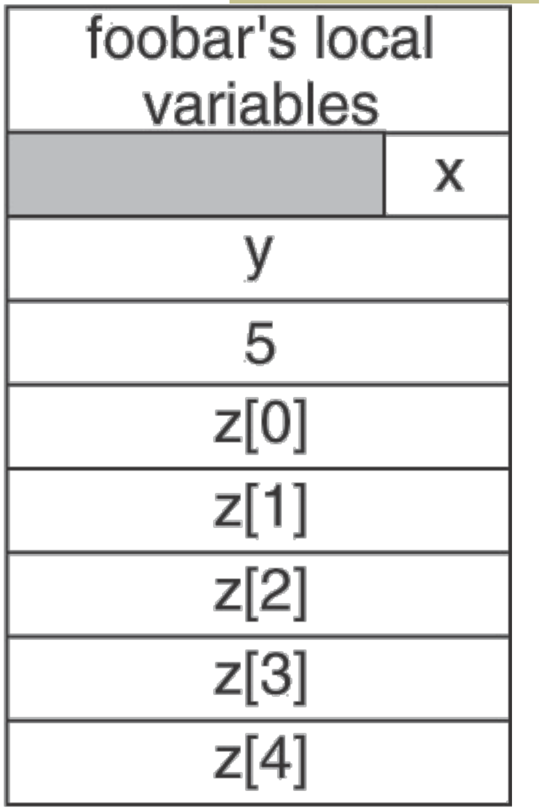
```
program FOOPROG {
  version VERSION {
    void FOO(fooargs) = 1;
    void BAR(barargs) = 2;
  } = 1;
} = 9999;
```

Parameter Specification and Stub Generation

- (a) A procedure
- (b) The corresponding message.

```
foobar( char x; float y; int z[5] )
{
   ....
}
```

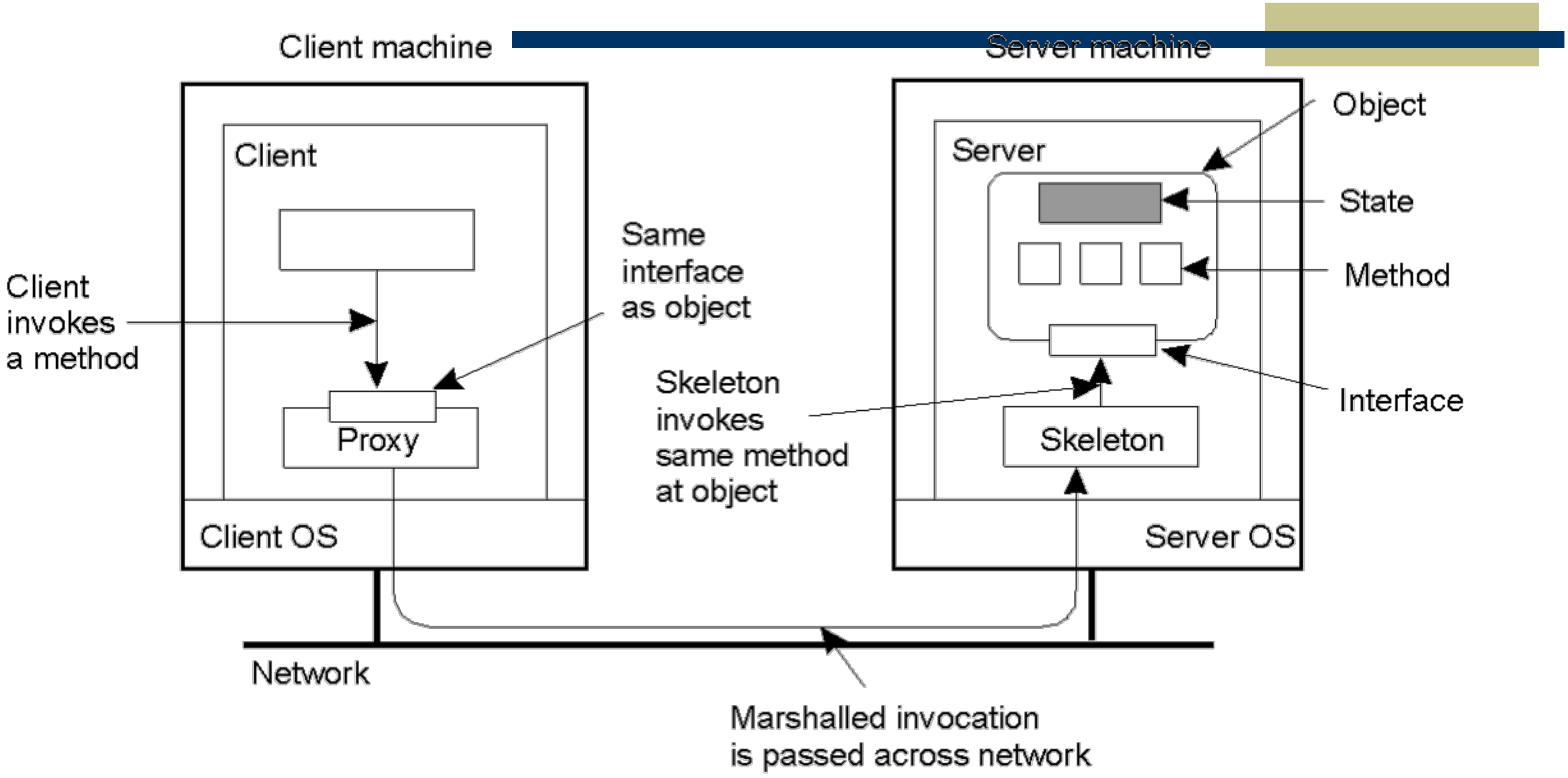
(a)



foobar's local variables	
▨	x
y	
5	
z[0]	
z[1]	
z[2]	
z[3]	
z[4]	

(b)

Distributed Objects



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