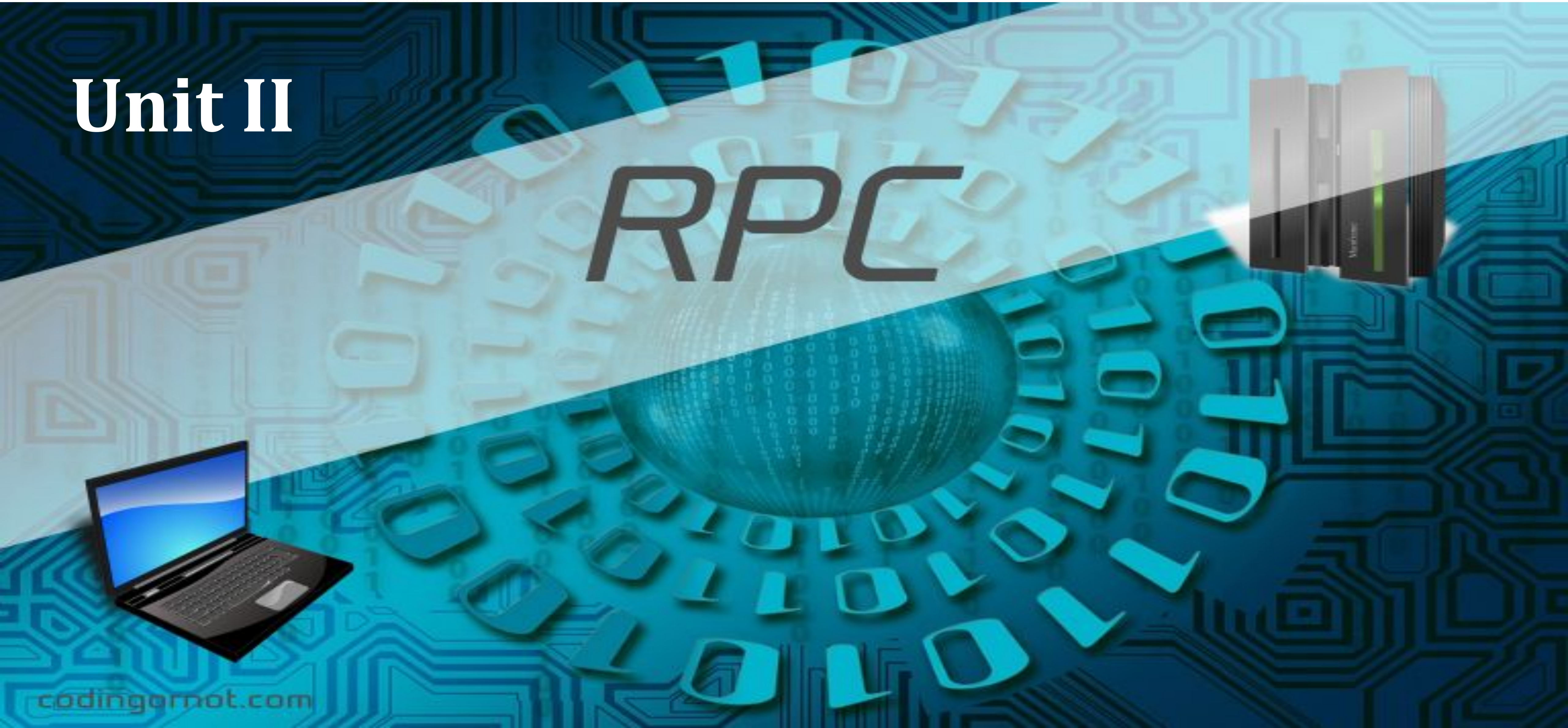
18CSC302J- Computer Networks

Unit II



Syllabus - Unit II

- Byte ordering
- Byte ordering conversion functions
- System calls
- Sockets
- System calls used with Sockets
- Iterative and concurrent server
- Socket Interface
- Structure and Functions of Socket
- **Remote Procedure Call**
- **RPC Model, Features**
- TCP Client Server Program
- Input, Output Processing Module
- UDP Client Server Program
- UDP Control block table & Module
- UDP Input & Output Module
- SCTP Sockets
- SCTP Services and Features, Packet Format
- SCTP Client/Server

Remote Procedure Call (RPC)

Problems in sockets?

- Sockets are a fundamental part of client-server networking.
- They provide a relatively easy mechanism for a program to establish a connection to another program, either on a remote or local machine and send messages back and forth (using read and write system calls).
- This interface, however, forces us to design our distributed applications using a read/write (I/O) interface which is not how we generally design non-distributed applications.
- In designing centralized applications, the **procedure call** is usually the standard interface model.
- If we want to make distributed computing look like centralized computing, I/O-based communications is not the way to accomplish this.

Solution

- In 1984, Birrell and Nelson devised a mechanism to allow programs to call procedures on other machines.
- A process on machine A can call a procedure on machine B.
- When it does so, the process on A is suspended and execution continues on B.
- When B returns, the return value is passed to A and A continues execution.
- This mechanism is called the **Remote Procedure Call (RPC)**.
- For a programmer, it looks like a normal procedure call is taking place.
- i.e., a RPC is different from a local one in the underlying implementation

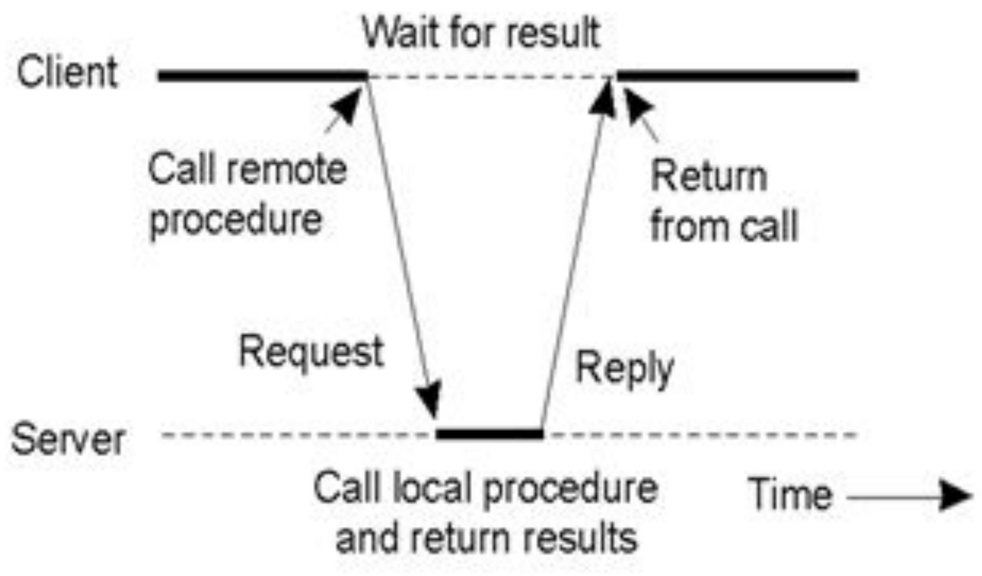
Programming-based example

```
// 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
```

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

RPC

- **Remote Procedure Call (RPC)** is a **protocol** that one program can use to request a service from a program located in another computer on a network without having to understand the network's details.
- RPC is used to call other processes on the remote systems like a local system.
- A procedure call is also sometimes known as a **function call** or a **subroutine call**.
- RPC uses the **client-server model**



Timing diagram - RPC

The requesting program is a **client**, and the service-providing program is the **server**.

RPC

- A remote procedure call is an inter-process communication technique that is used for client server based applications.
- RPC allows programs to call the procedure which is located on the other machines.
- Message passing is not visible to the programmer , so it is called as **RPC**
- RPC enables a procedure call that does not reside in the address space of the calling process.
- In RPC, the caller and the callee has disjoint address space, hence there is no access to data and variables in the callers environment.
- RPC performs a message passing scheme for information exchange between the caller and the callee process.

RPC Goals

- Ease of programming
- Hide complexity
- Automates task of implementing distributed computation
- Familiar model for programmers (just make a function call)

RPC Features

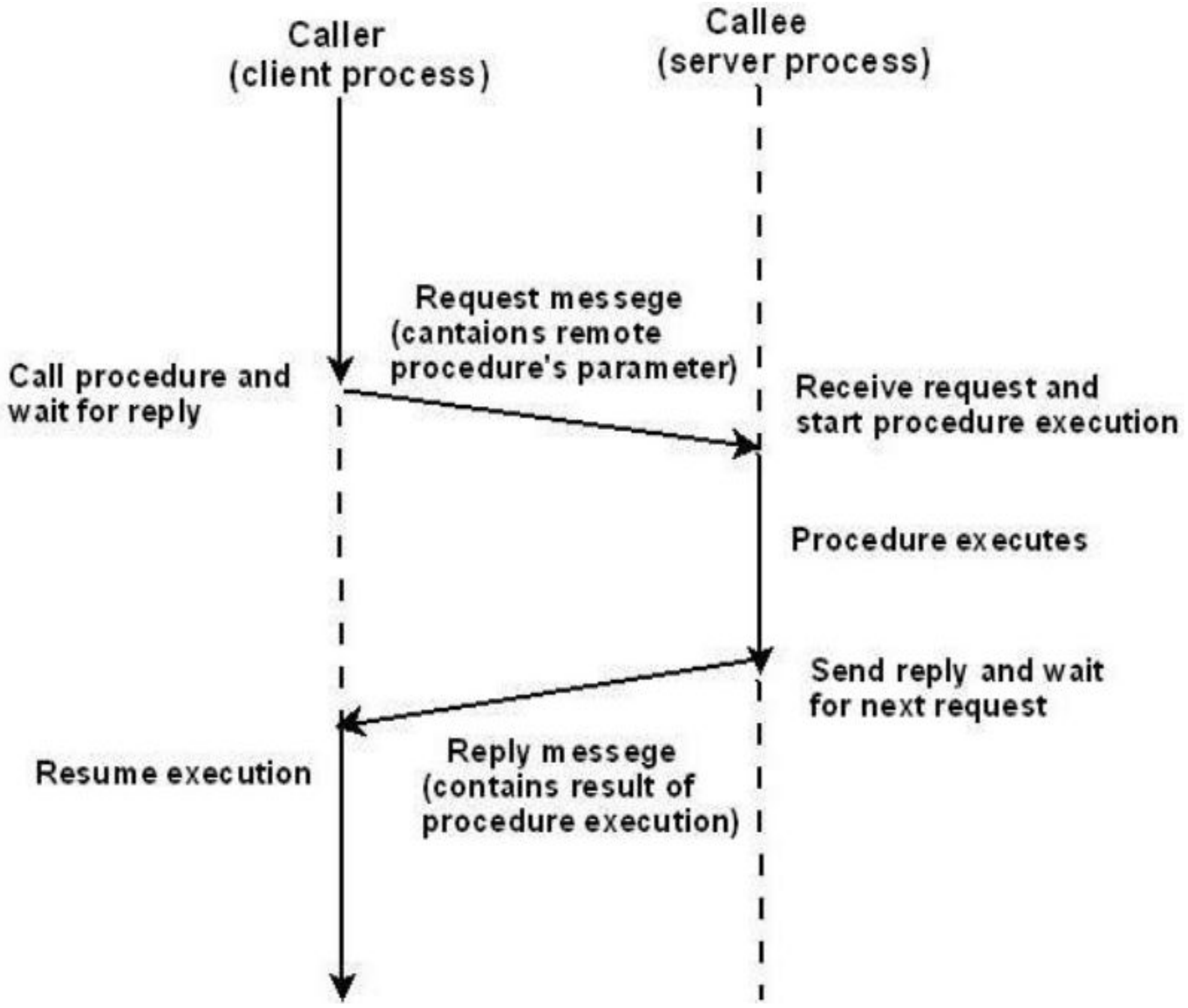
- A client has a request message that the RPC translates and sends to the server.
- This request may be a procedure or a function call to a remote server.
- When the server receives the request, it sends the required response back to the client.
- The client is blocked while the server is processing the call and only resumed execution after the server is finished

RPC

- 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
- 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

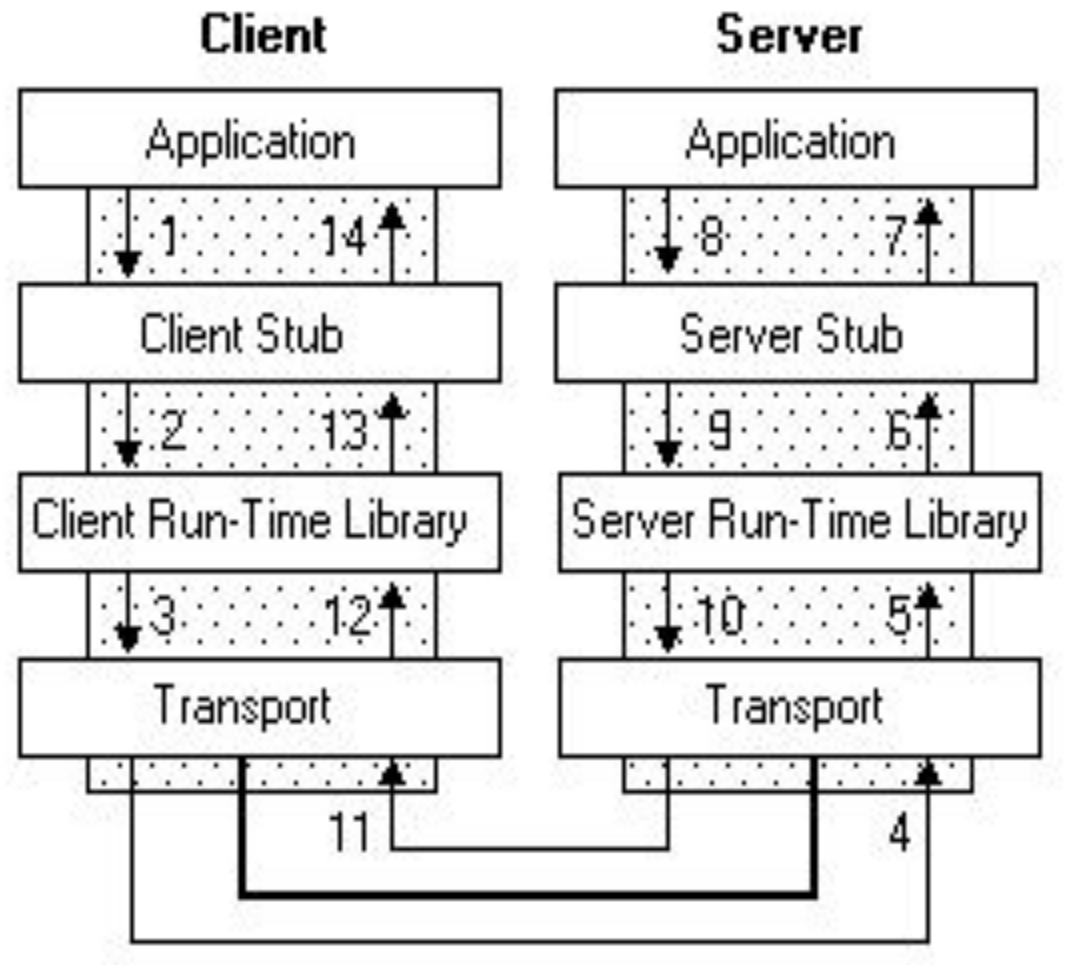
RPC Model

- A client has a request message that the RPC translates and sends to the server.
- This request may be a procedure or a function call to a remote server.
- When the server receives the request, it sends the required response back to the client.
- The client is blocked while the server is processing the call and only resumed execution after the server is finished



RPC

- Important elements in RPC
  - Client (Application/Process)
  - Client stub (stub-Piece of code used for converting parameters)
  - RPC runtime (Communication package)
  - Server stub
  - Server



RPC Architecture

1. Client

- It is a user process which initiates a RPC
- The client makes a perfectly normal call that invokes a corresponding procedure in the client stub

2. Client stub

- On receipt of a request it packs a requirement into a message and asks to RPC runtime to send
- On receipt of a result it unpacks the result and passes it to a client

3. RPC runtime

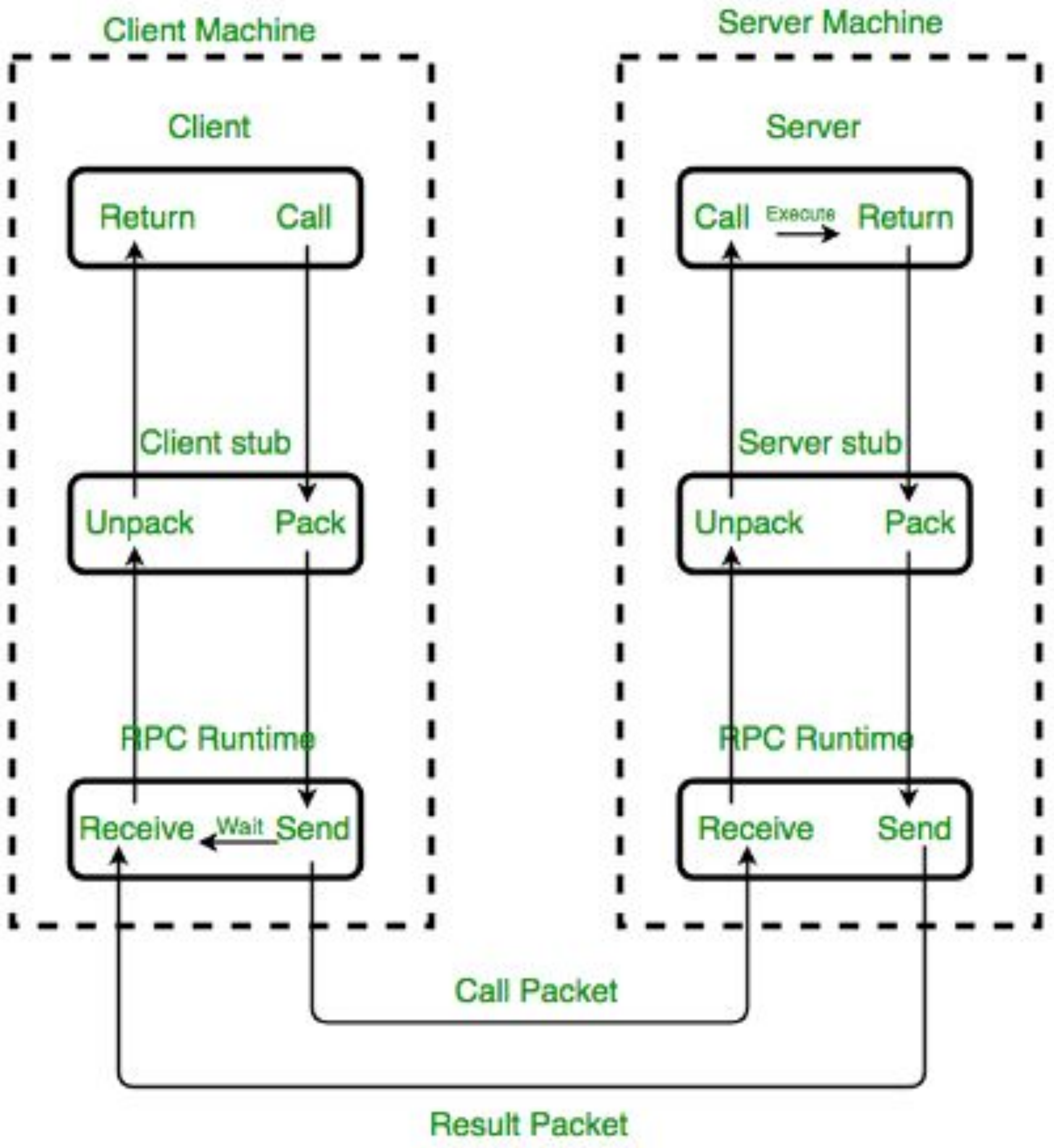
- It handles transmission of messages between client and server

4. Server stub

- It unpacks a call request and make a perfectly normal call to invoke the appropriate procedure in server.
- On receipt of a result a procedure execution it packs the result and asks RPC runtime to send

5. Server

- It execute an appropriate procedure and returns the results from a server stub.



Implementation of RPC mechanism

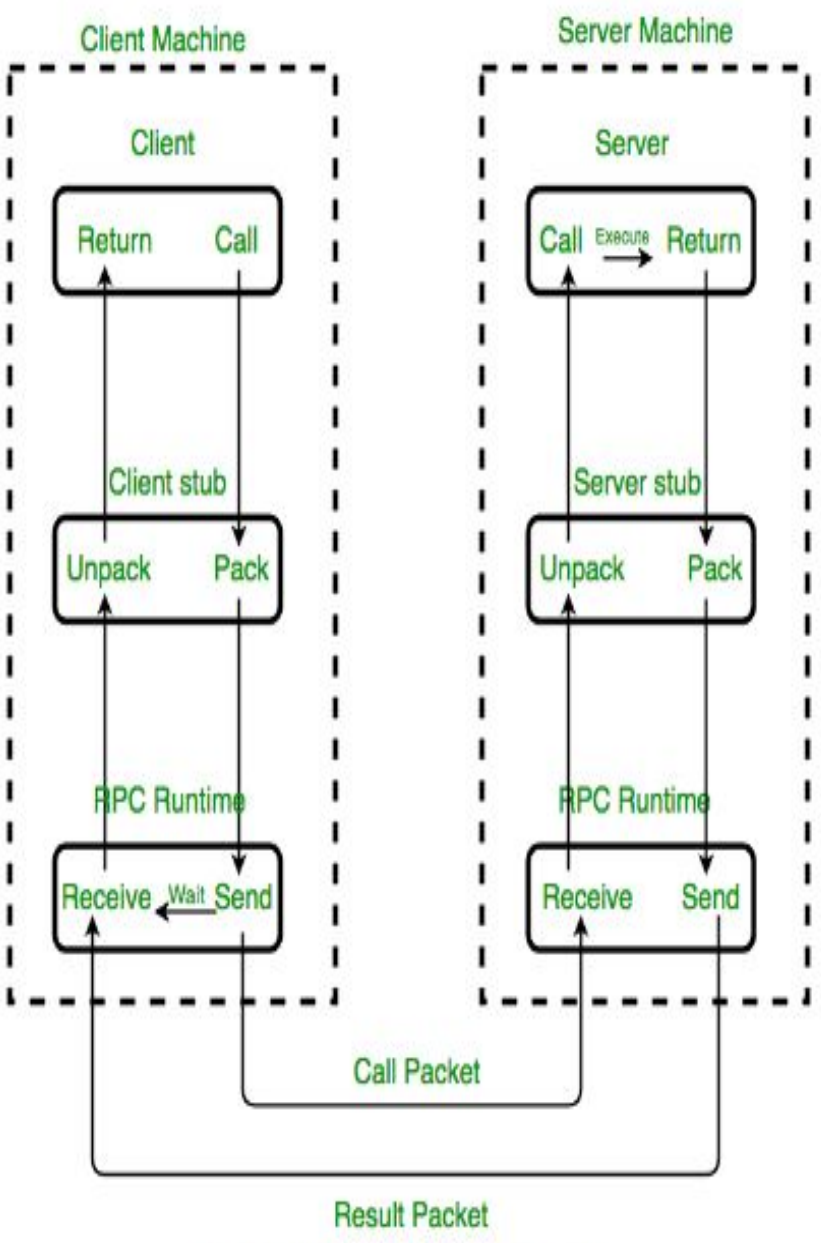
Steps

- 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 OS.
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

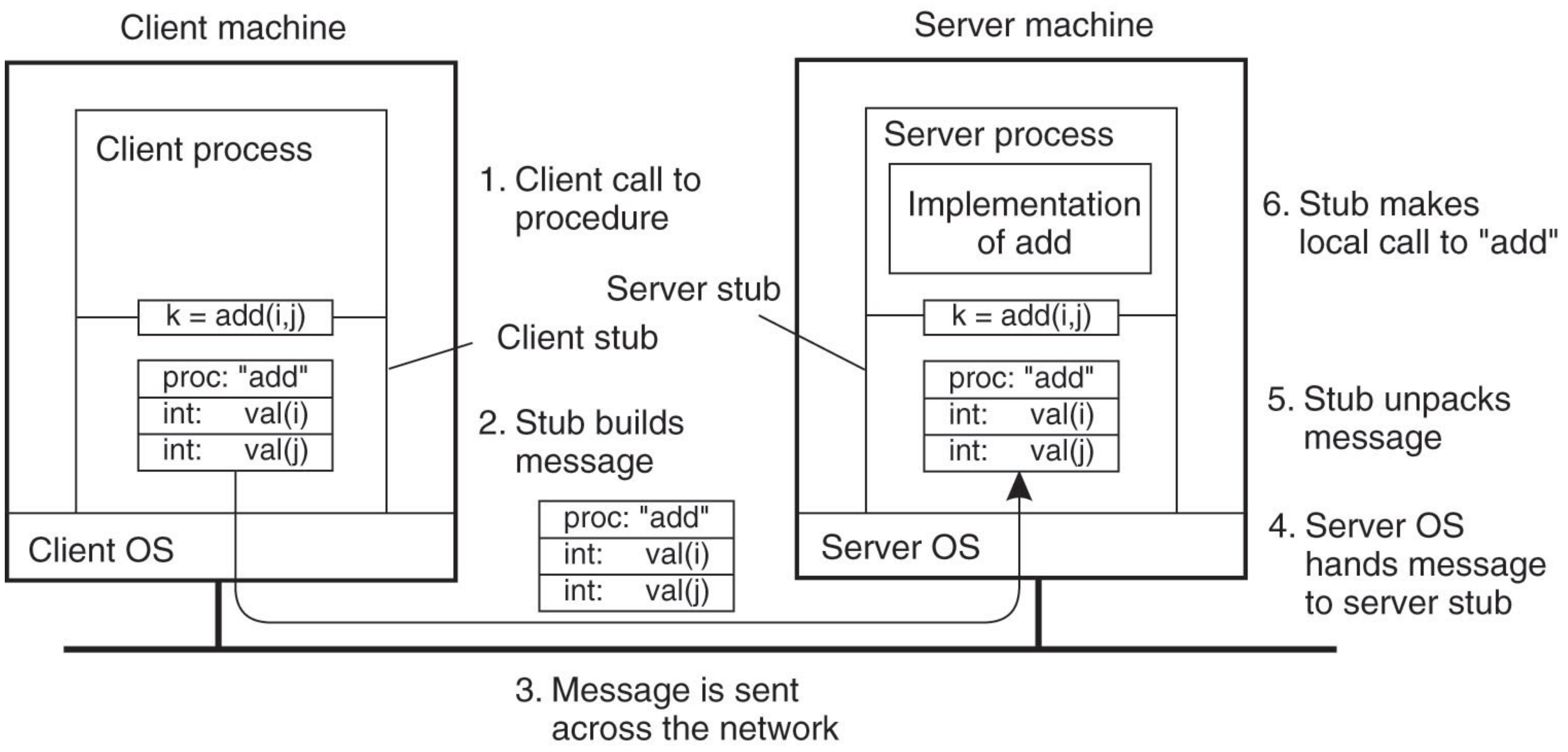
- A remote procedure call occurs in the following steps:

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.
0. The stub unpacks the result and returns to the client.



Implementation of RPC mechanism

Example :The steps involved in a doing a remote computation through RPC



Stub-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”.

- 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...

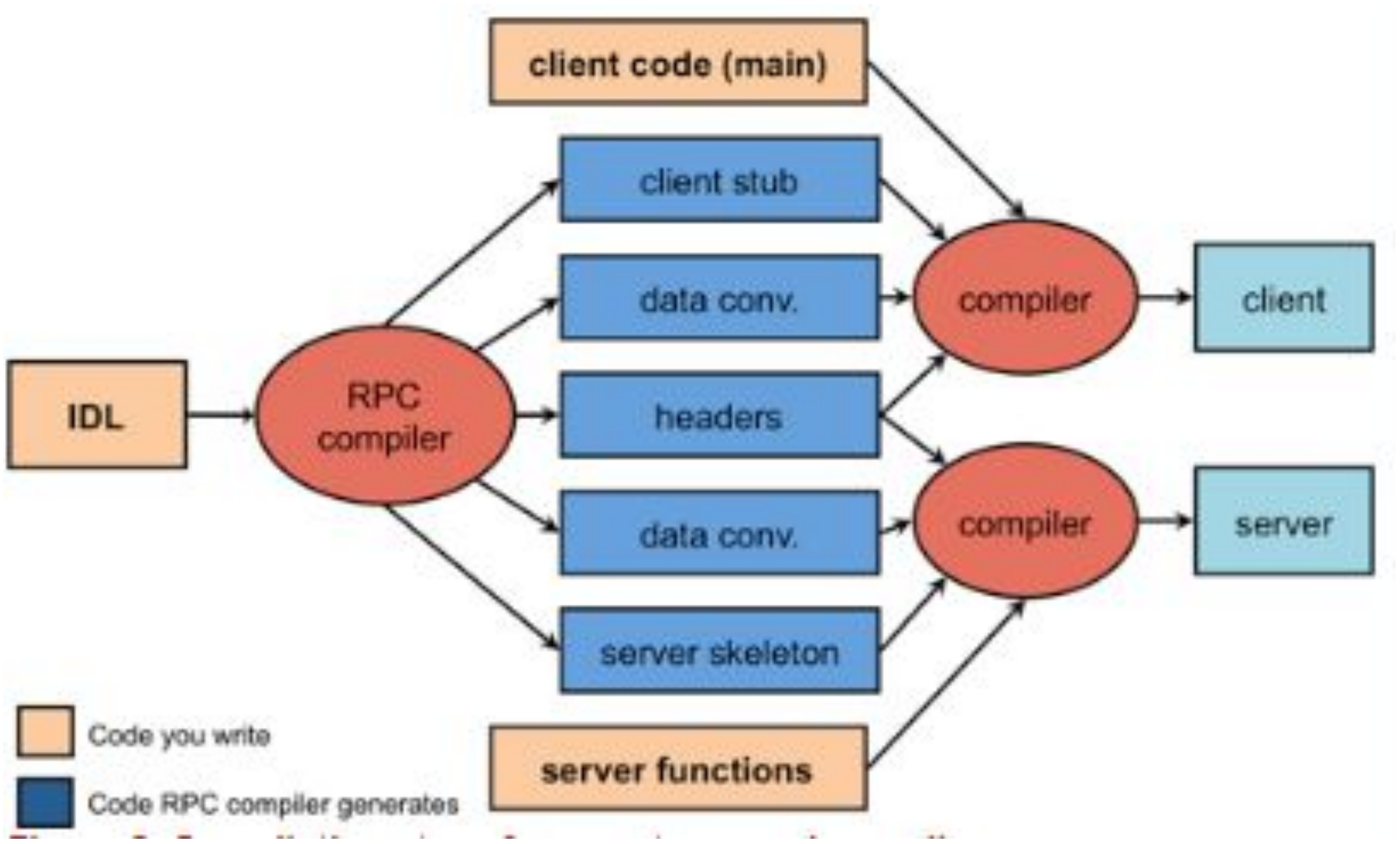
Nested structures? (Design question for the RPC system - do you support them?)

Complex data structures? (Some RPC systems let you send lists and maps as first-order objects)

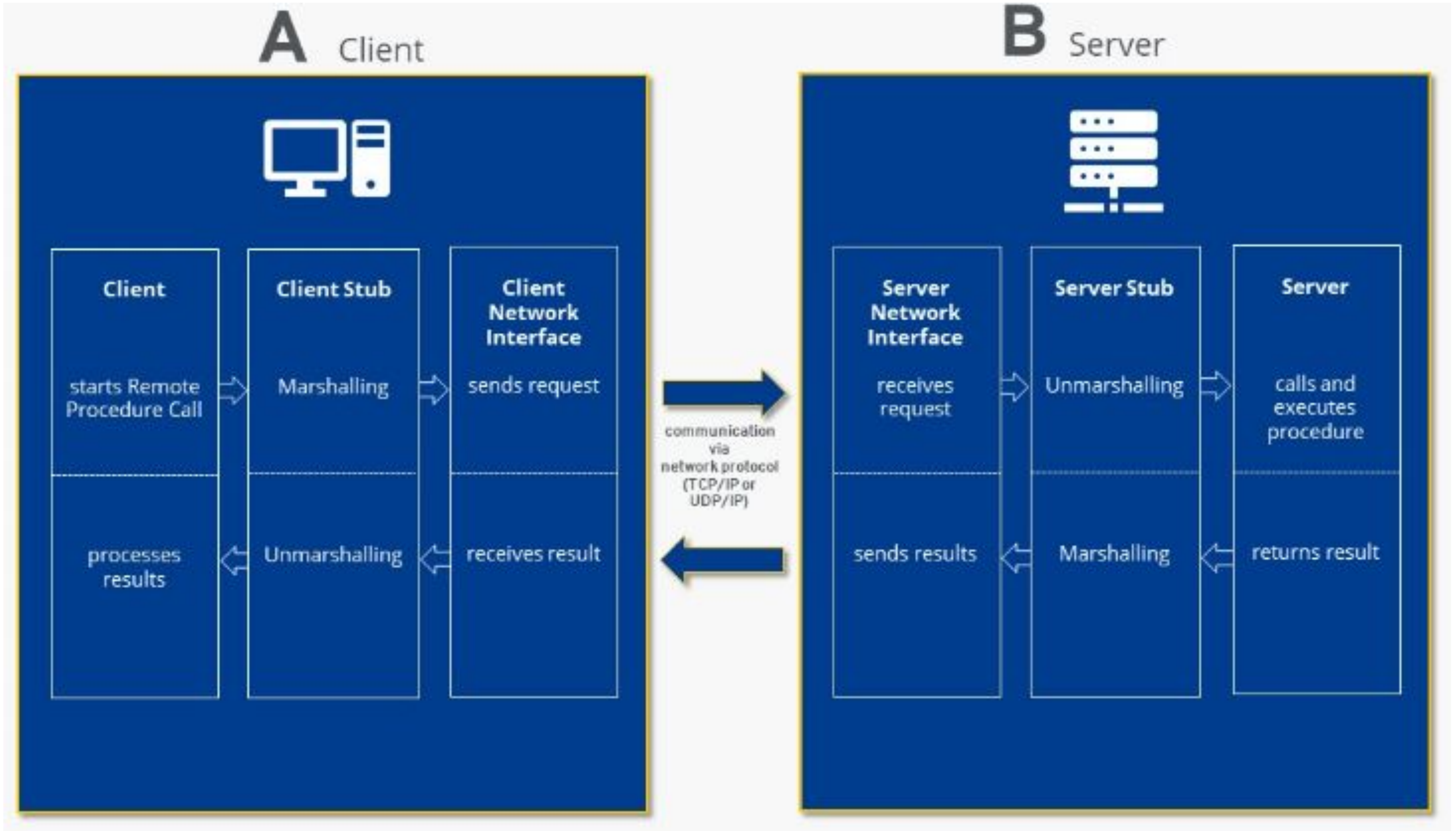
“stubs” and IDLs

- 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.

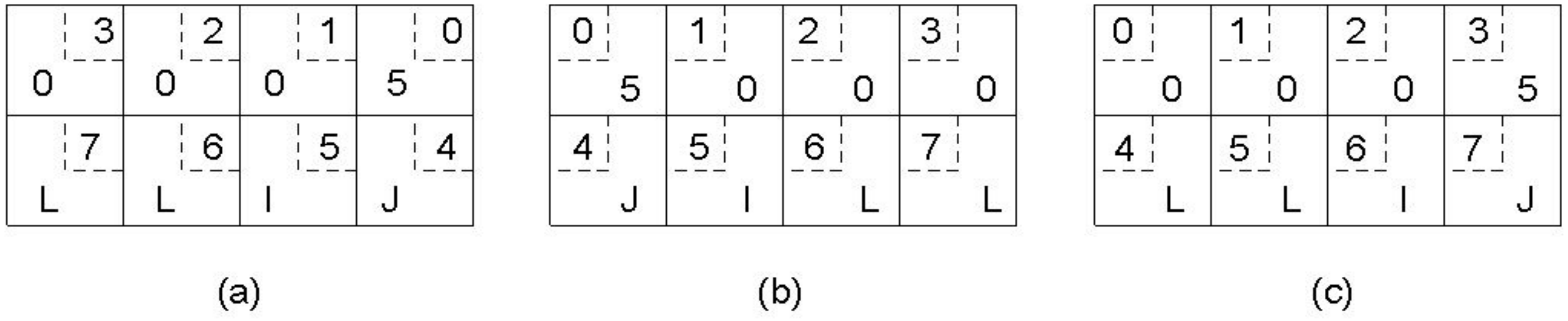
Example-IDL


client code (main)
client stub
data conv.
headers
data conv.
server skeleton
server functions
IDL
RPC compiler
compiler
compiler
client
server
Code you write
Code RPC compiler generates

Example-RPC


A Client
B Server
Client
starts Remote Procedure Call
processes results
Client Stub
Marshalling
Unmarshalling
Client Network Interface
sends request
receives result
communication via network protocol (TCP/IP or UDP/IP)
Server Network Interface
receives request
sends results
Server Stub
Unmarshalling
Marshalling
Server
calls and executes procedure
returns result

Passing Value Parameters (2)



a) Original message on x86

b) The message after receipt on the SPARC

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

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 failures

- Request from client server lost
- Reply from srever client 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

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.

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.

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

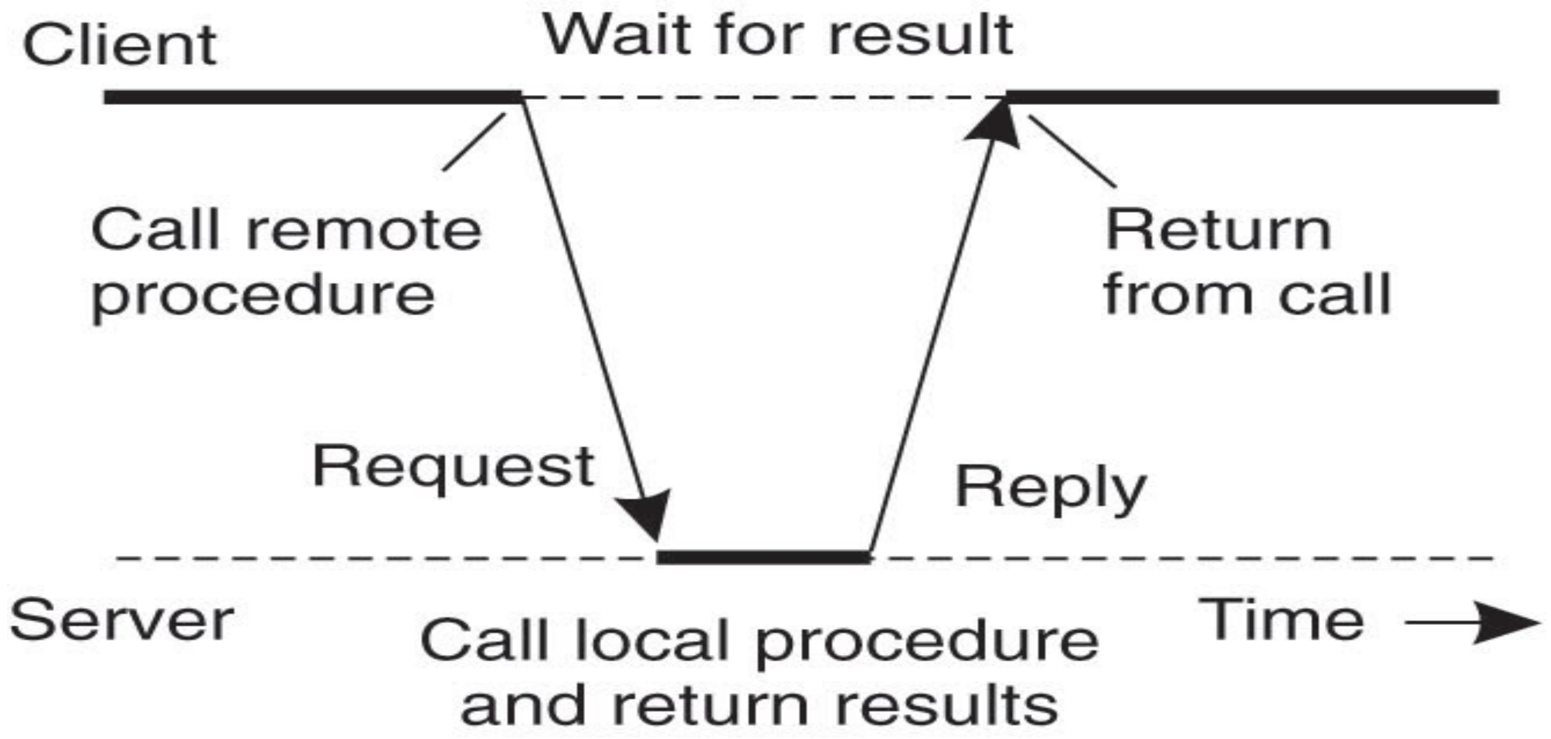
- Expose RPC properties to client, since you cannot hide them

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

Important Lessons

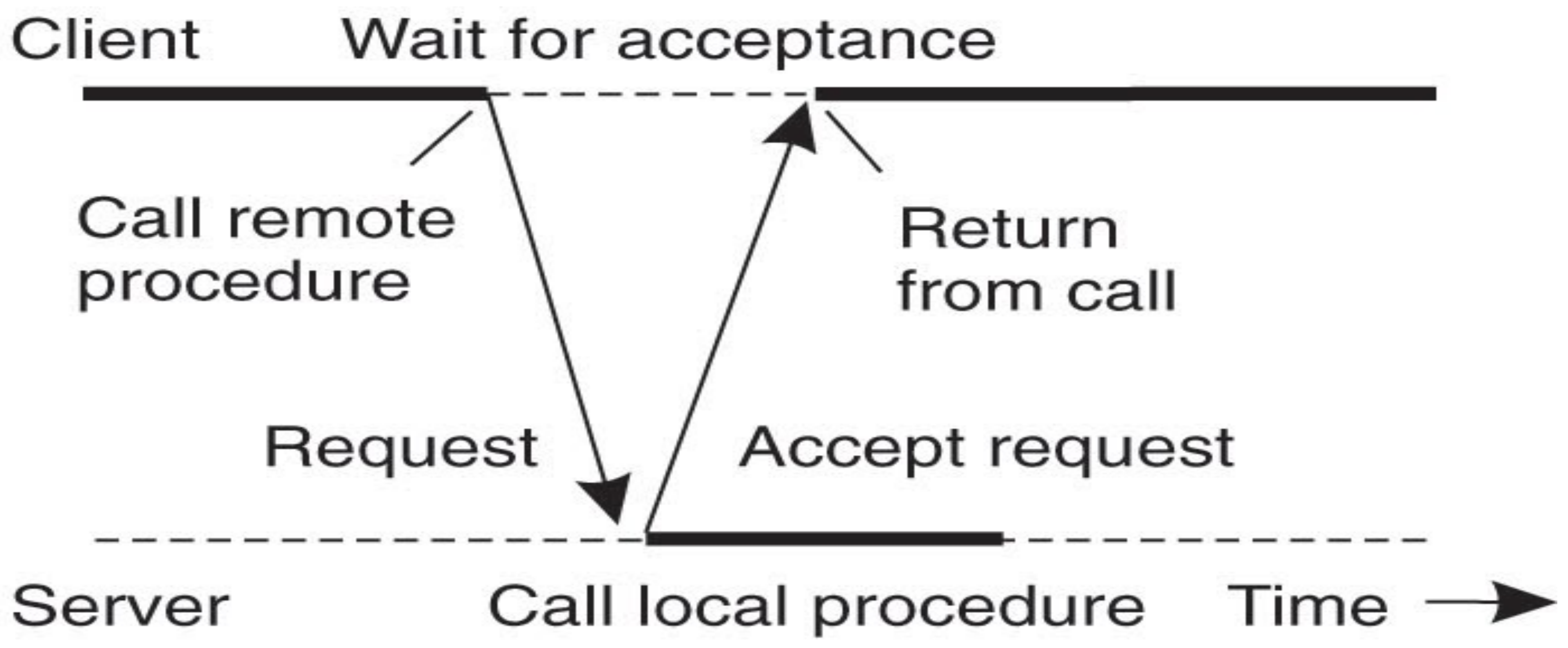
- Procedure calls
  - Simple way to pass control and data
  - Elegant transparent way to distribute application
  - Not only way…
- Hard to provide true transparency
  - Failures
  - Performance
  - Memory access
  - Etc.
- How to deal with hard problem give up and let programmer deal with it
  - “Worse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

Asynchronous RPC (1)



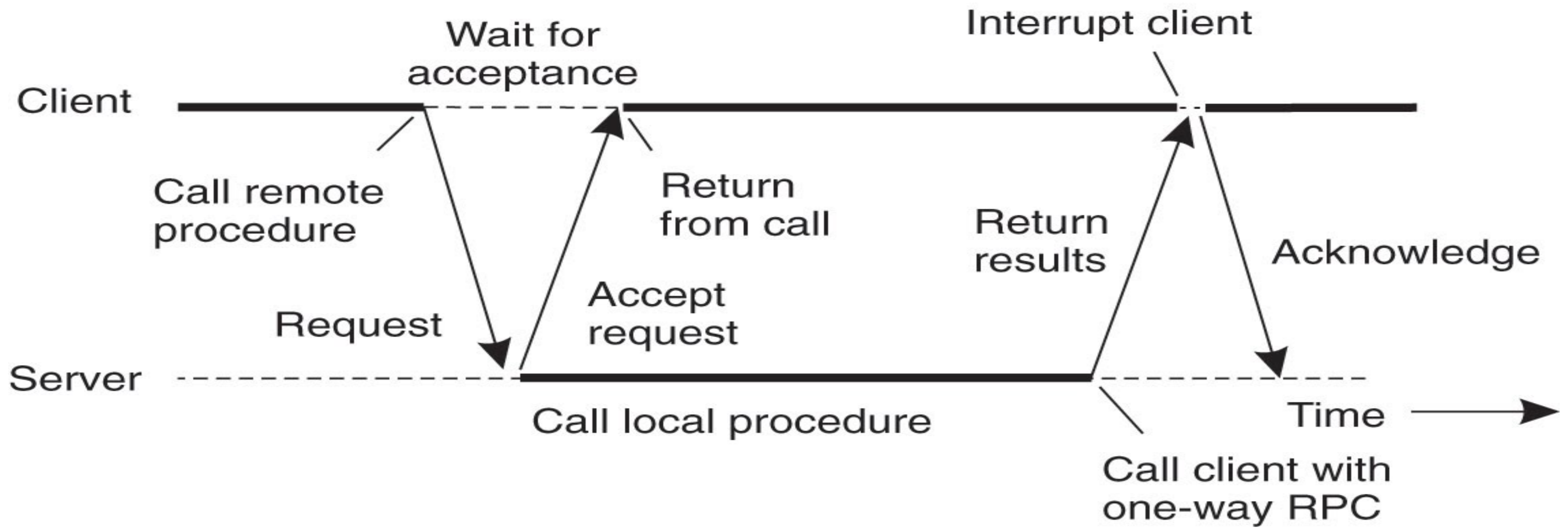
- The interaction between client and server in a traditional RPC.

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 that of 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

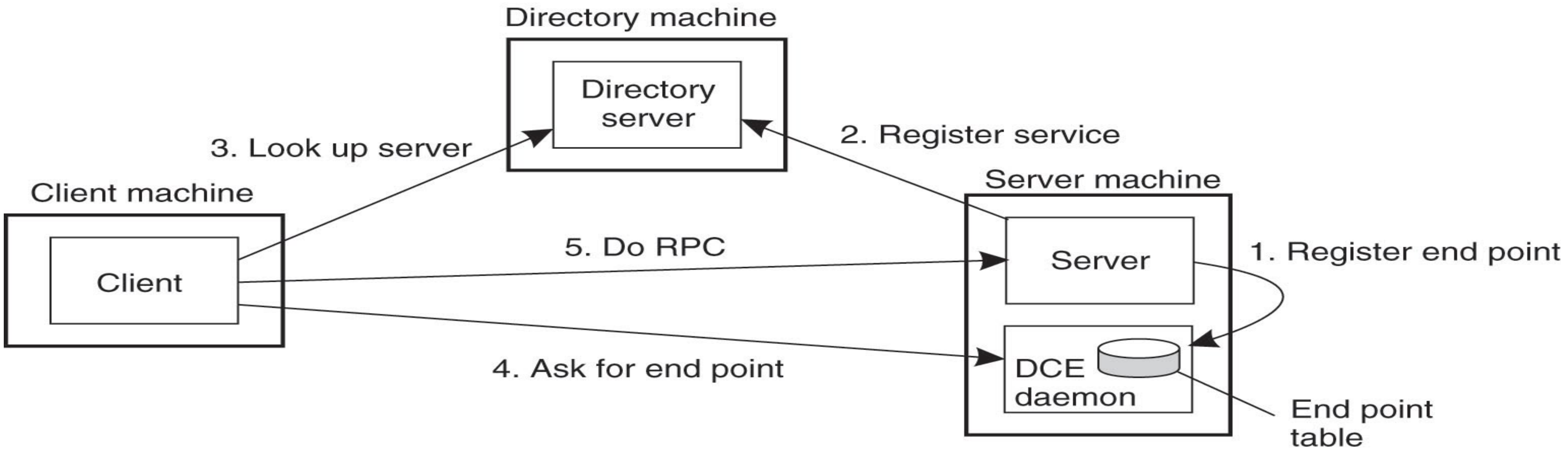
- Request⍰Server⍰Response: Classic synchronous RPC

- 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: JavaScript Object Notation (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
}
```

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":

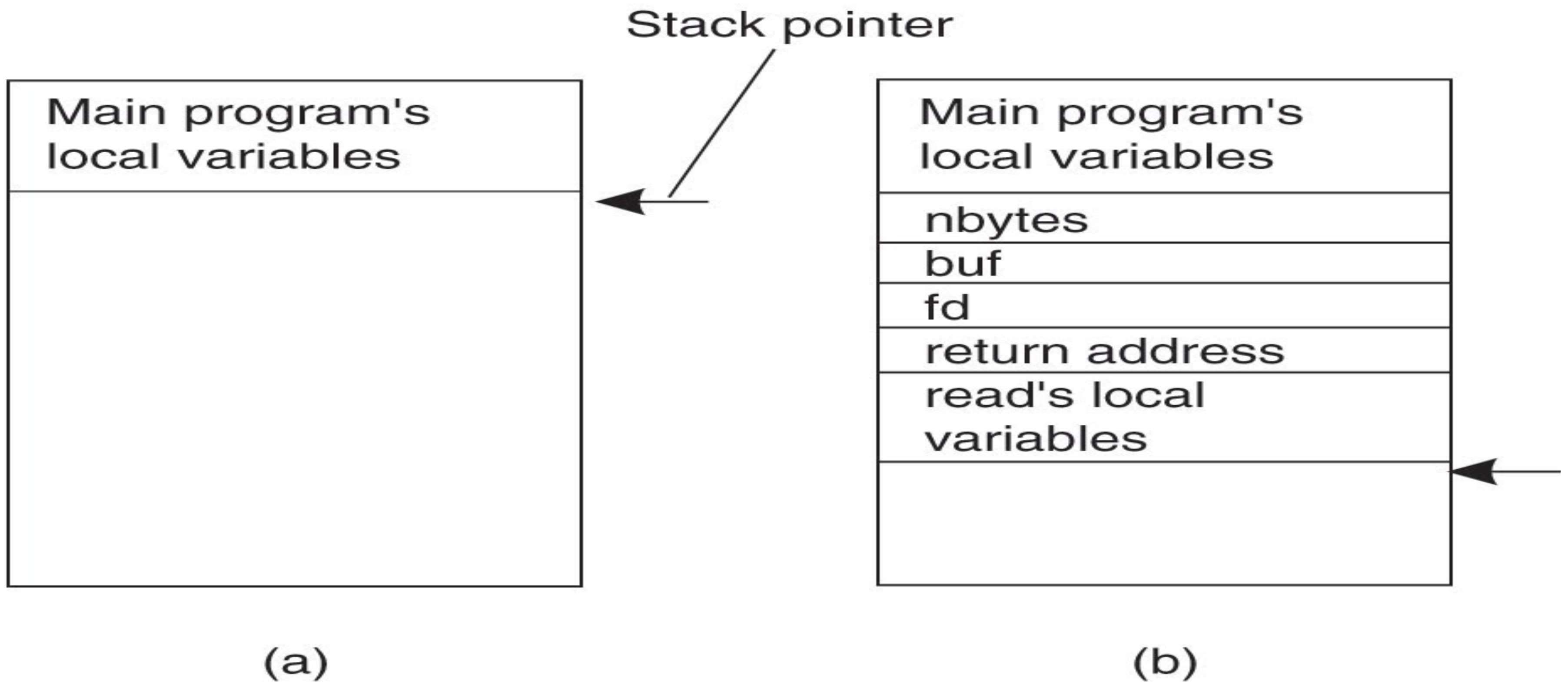
```
{
 "Head": {
   "Val": "pig",
   "Next": {
     "Val": "cat",
     "Next": {
       "Val": "dog",
       "Next": null
     }  } }  }
```

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.

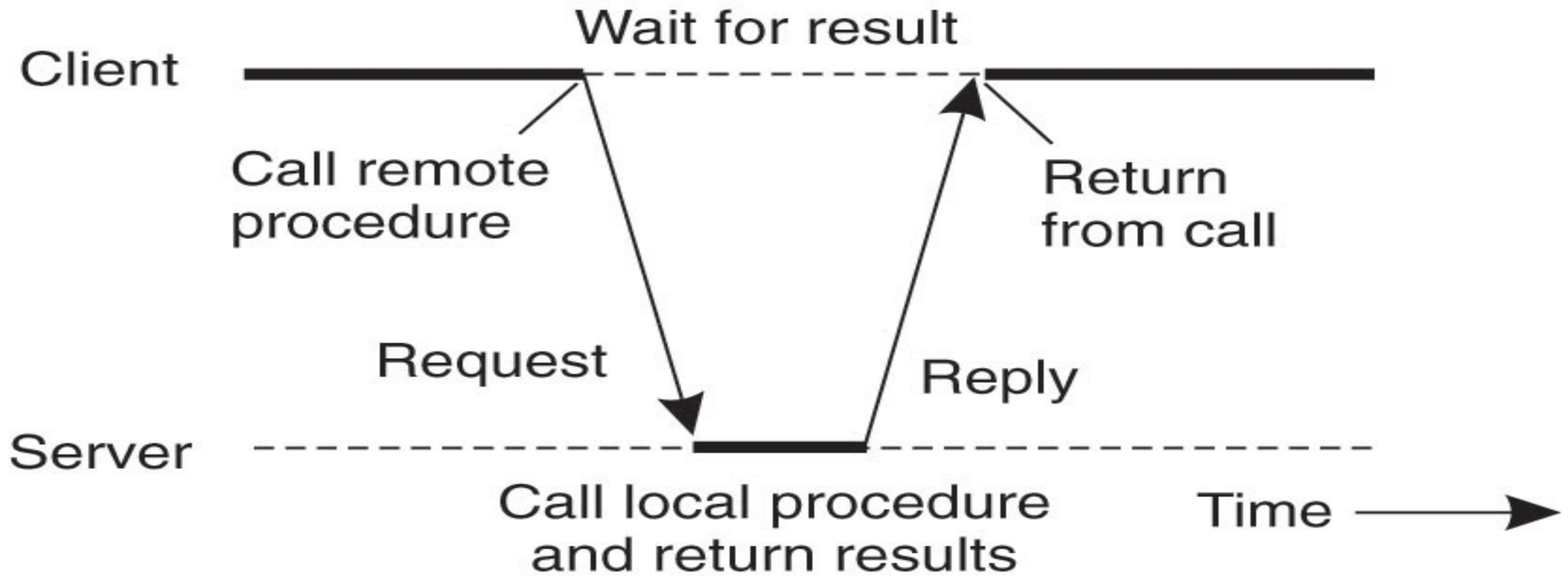
- Deep integration.
    - Data formatting done based on type declarations
    - (Almost) all public methods of object are registered.

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

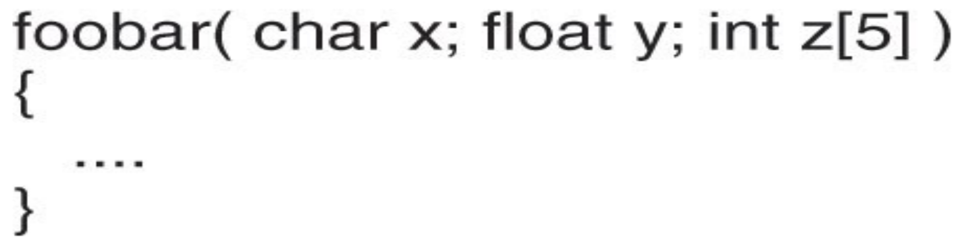
```
struct fooargs {
  string msg<255>;
  int baz;
}
```

And describes functions

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

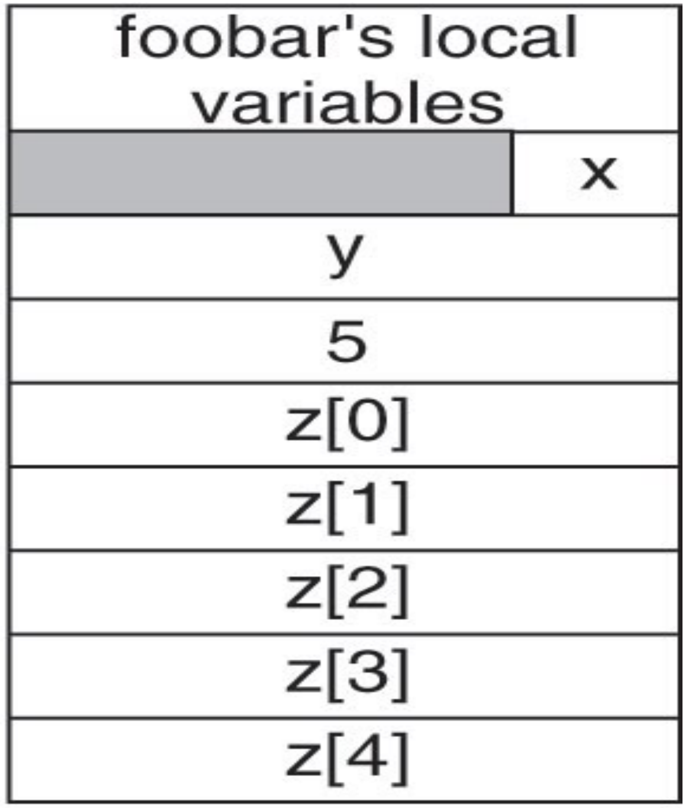
Parameter Specification and Stub Generation

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



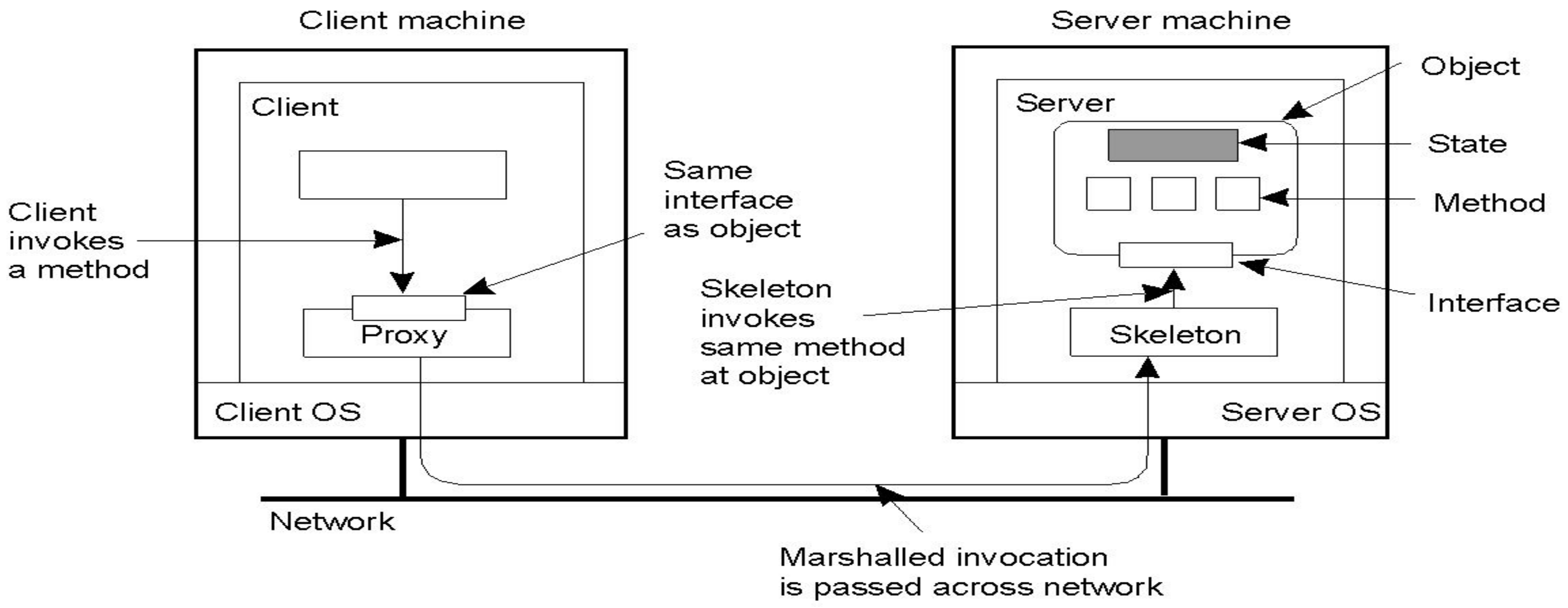
```
foobar( char x; float y; int z[5] )
{
   ....
}
```

(a)



(b)

Distributed Objects



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

Thank
You