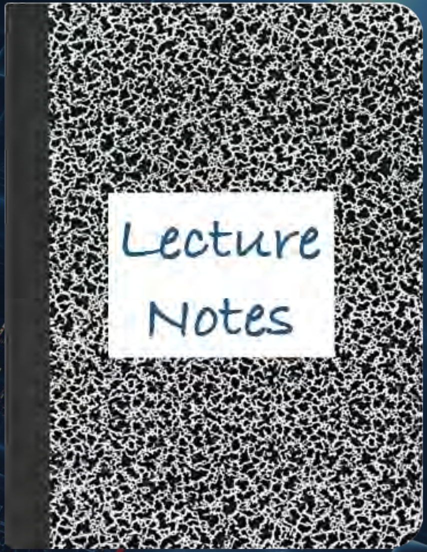
CS 417 – DISTRIBUTED SYSTEMS

Week 2: Part 1

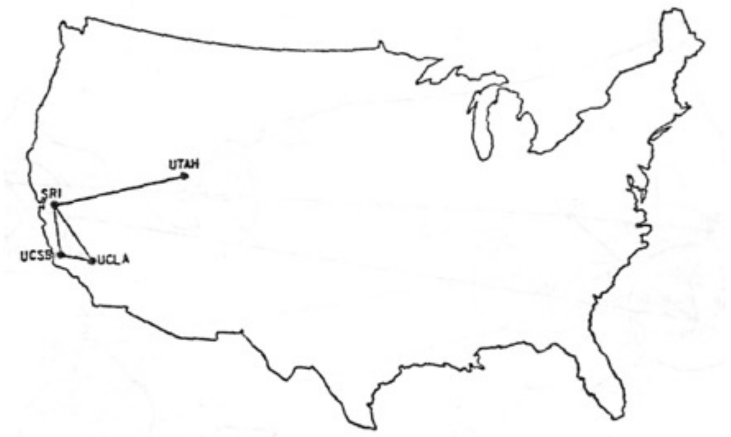
Point-to-point communication: Remote Procedure Calls



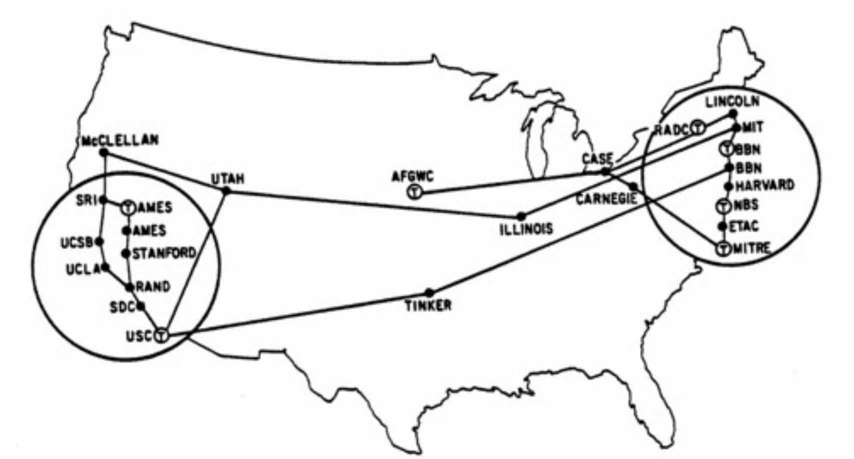
Paul Krzyzanowski

IP Communication

The Internet



ARPANET – December 1969

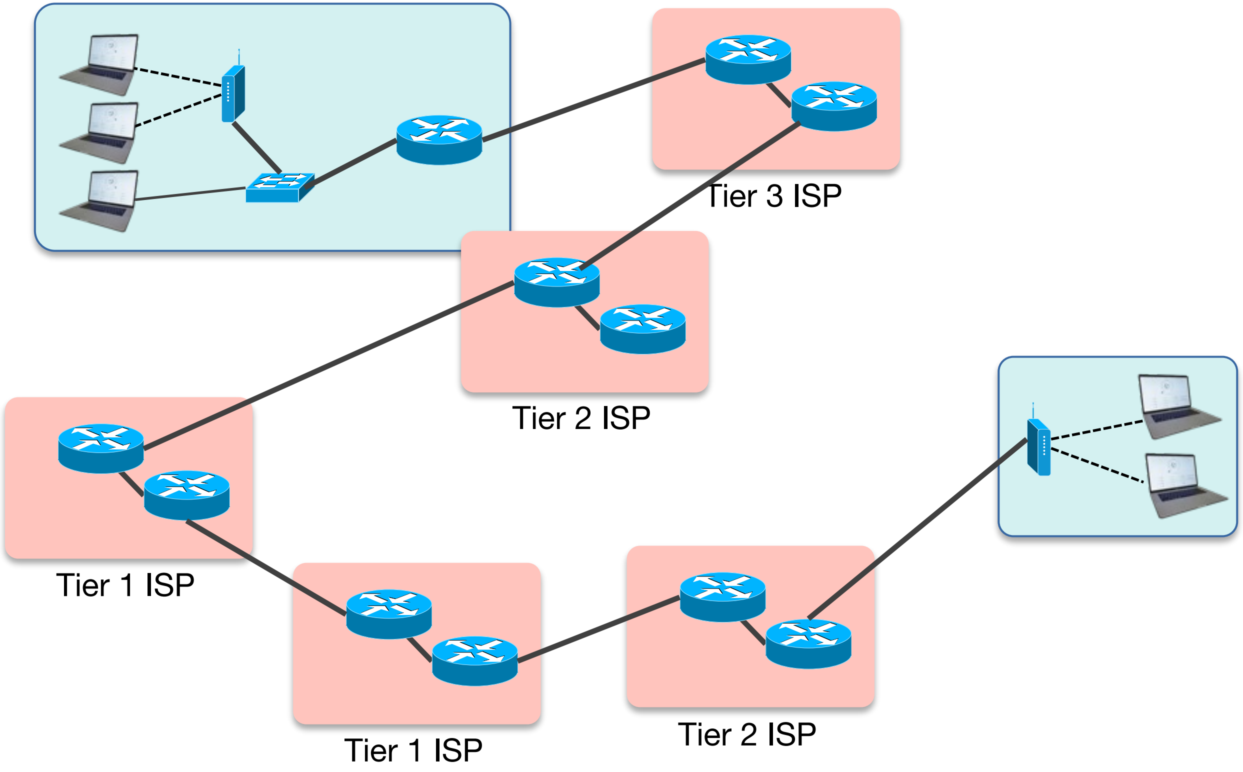


ARPANET - 1972

The Internet: Key Design Principles

- Support interconnection of networks
  - No changes needed to the underlying physical network
  - IP is a logical network
- Assume unreliable communication
  - If a packet does not get to the destination, software on the receiver will have to detect it and the sender will have to retransmit it
- Routers connect networks
  - Store & forward delivery
- No global (centralized) control of the network

Routers tie LANs together into one Internet



A packet may pass through many networks – within and between ISPs

Internet Protocol

A set of protocols designed to handle the interconnection of many local and wide-area networks that together comprise the Internet

IPv4 & IPv6: network layer

- Other IP-based protocols include TCP, UDP, RSVP, ICMP, etc.
- Relies on routing from one physical network to another
- IP is connectionless

  No state needs to be saved at each router
- Survivable design: support multiple paths for data

… but packet delivery is not guaranteed!

IP addressing

- Each network endpoint has a unique IP address
  - No relation to an ethernet address
  - IPv4: 32-bit address `www.rutgers.edu = 128.6.46.88`
  - IPv6: 128-bit address `www.google.com = 2607:f8b0:4004:811::2004`
- Data is broken into packets
  - Each IP packet contains
    - source & destination IP addresses
    - Header checksum
    - Data

IP gives us machine-to-machine communication

Communication over IP

Transport Layer (4)	TCP, UDP
Network Layer (3)	Internet Protocol: IPv4, IPv6
Data Link Layer (2)	Ethernet, Wi-Fi, DOCSIS, ATM, Frame Relay, …

- TCP: Reliable, in-order <u>byte</u> stream
- UDP: Unreliable, <u>message</u> stream (order not guaranteed)

Transport Layer

- We want to communicate between applications
- The transport layer gives us logical "channels" for communication
  - Processes can write to and receive from these channels
- Two transport layer protocols in IP are TCP & UDP
  - A port number identifies a unique channel on each computer
    - 16-bit number (range 0…65535)

IP transport layer protocols

IP gives us two transport-layer protocols for communication

TCP: Transmission Control Protocol

- Connection-oriented service – operating system keeps state
- Full-duplex connection: both sides can send messages over the same link
- Reliable data transfer: the protocol handles retransmission
- In-order data transfer: the protocol keeps track of sequence numbers
- Flow control: receiver stops sender from sending too much data
- Congestion control: “plays nice” on the network – reduce transmission rate
- 20-byte header

Byte stream interface

UDP: User Datagram Protocol

- Connectionless service: lightweight transport layer over IP
- Data may be lost
- Data may arrive out of sequence
- Checksum for corrupt data: operating system drops bad packets
- 8-byte header

Message stream interface

TCP Upsides & Downsides

- Upsides – huge!
  - In-order, reliable byte streams
  - Congestion control (plays nice in sharing the network), flow control (avoids queue overflow)
- Downsides
  - Storing & managing state in the operating system
    - Sequence numbers, Buffering out-of-order data, Acknowledgments
    - Significant kernel memory use when lots of connections
  - Congestion control
    - Slows down transmission but doesn't always accurately reflect network congestion (based on packet loss)
  - Recovery
    - All state is lost if a system goes down – connections will need to be re-established
  - Increased latency
    - Session setup
    - Data may not be immediately transmitted or presented to the receiving app
      - Nagle's algorithm: delay sending to see if more bytes need to be sent to avoid sending lots of small packets

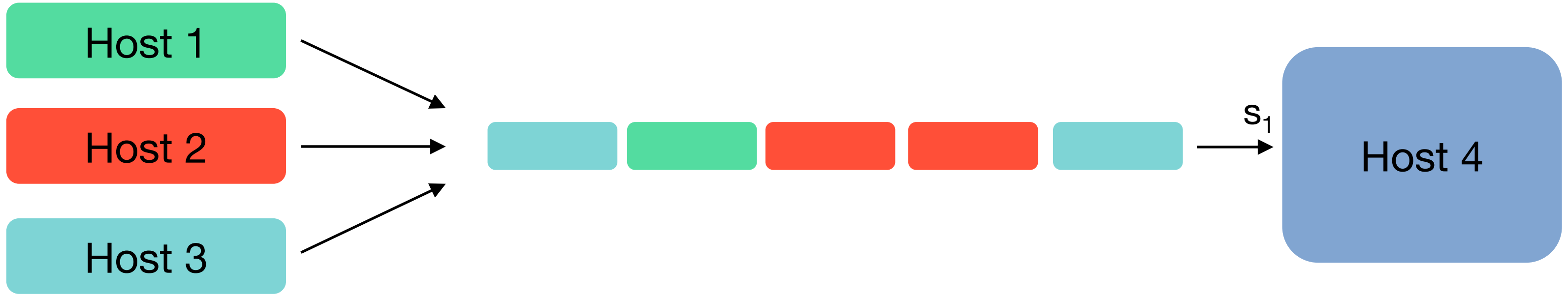
UDP Upsides & Downsides

- Upsides
  - Fewer kernel resources
  - No connection setup overhead – useful data can be sent with 1st packet
  - Received data immediately sent & delivered to the application
    - No delay in sending messages
  - No state recovery – traffic can be easily redirected to a standby system

- Downsides
  - Delivery & message order not guaranteed
    - Usually perfect on local area networks; less reliable on wide area networks

Identifying Sessions: UDP

All traffic goes to a socket that reads from a host address & port

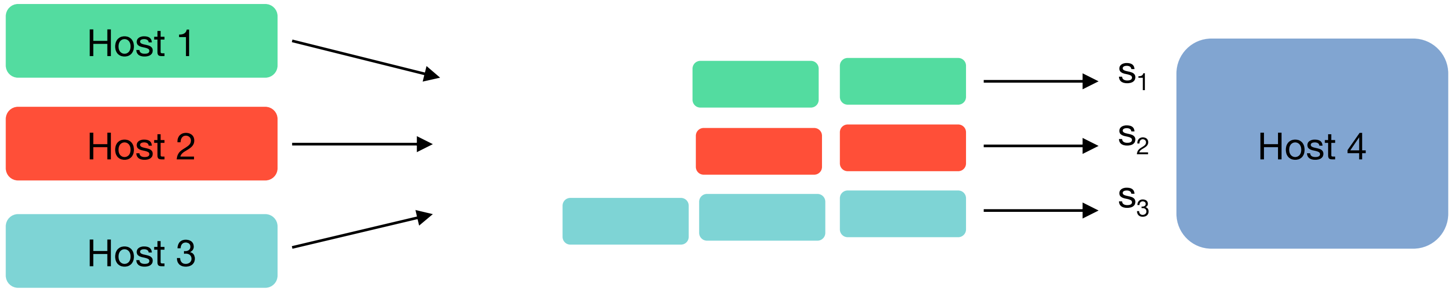


A server creates a socket to receive messages on a specific port number.
Packets sent from different processes and/or systems
all arrive on the same socket on the server

Identifying Sessions: TCP

Unique channels identified by

- { Remote host, Remote port, Local host, Local port }
- One socket for **listening** for new connections on a *local host, port*
- Separate communication socket for each “connection”



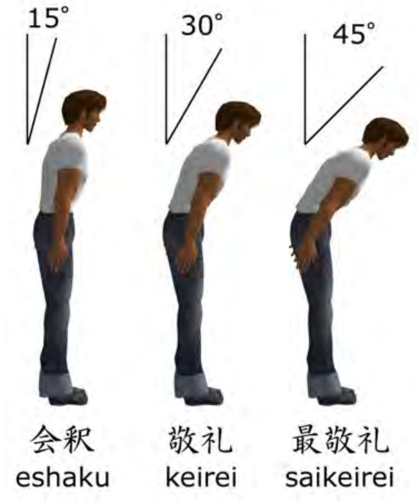
A server creates a socket to *listen for connections* on a specific port number.
Each connection results in a new socket at the server

Protocols

- Set of rules (& customs) for communicating
- Exist at different levels

  Humans:
  - Body language
  - Voice frequency, phonemes, language
  - Phrases & responses

  Computers:
  - Exist at each layer of the network stack
  - Meaning of bytes
  - Sequence of request & response messages



Parlez-vous français?

¿Hablas español?

Loquerisne Latine?

Facio, ita!

Software interaction model

- Socket API: all we get from the OS to access the network
- Socket = distinct end-to-end communication channels

read/write interface

- Line-oriented, text-based protocols common
  - Not efficient but easy to debug & use

Sample SMTP Interaction

SMTP = Simple Mail Transfer Protocol

```
$ telnet porthos.rutgers.edu 25
Trying 128.6.25.90...
Connected to porthos.rutgers.edu.
Escape character is '^]'.
220 porthos.cs.rutgers.edu ESMTP Postfix (Ubuntu)
HELO poopybrain.com
250 porthos.cs.rutgers.edu
MAIL FROM: <paul@poopybrain.com>
250 2.1.0 Ok
RCPT TO: <pxk@cs.rutgers.edu>
250 2.1.5 Ok
DATA
354 End data with <CR><LF>.<CR><LF>
From: Paul Krzyzanowski <myname@somewhere.edu>
Subject: test message
Date: Mon, 30 Sep 2023 17:00:16 -0500
To: Whomever <testuser@pk.org>

Hi,
This is a test
.
250 2.0.0 Ok: queued as 82D315F7C5
quit
221 2.0.0 Bye
Connection closed by foreign host.
```

This is the message body.
Headers may define the structure of the message but are ignored for delivery.

Sample HTTP Interaction

HTTP = Hypertext Transfer Protocol

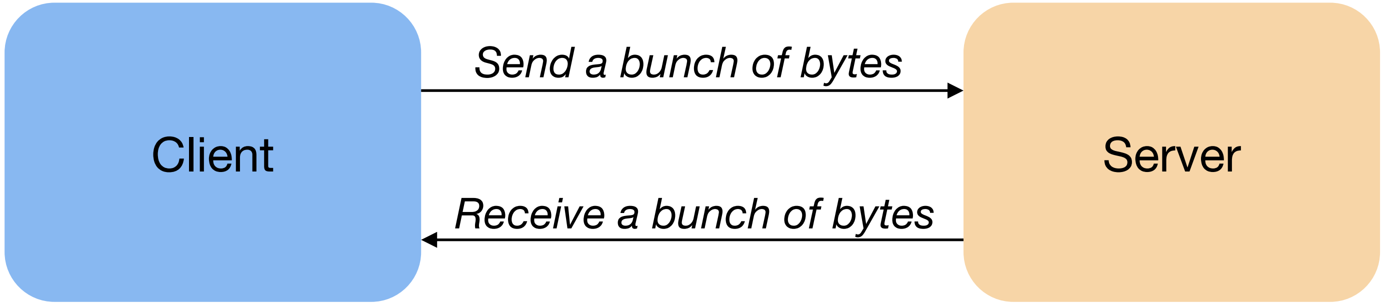
```
$ telnet www.google.com 80
Trying 172.217.12.196...
Connected to www.google.com.
Escape character is '^]'.
GET /index.html HTTP/1.1
HOST: www.google.com
Accept: image/gif, image/jpeg, */*
Accept-Language: en-us
User-Agent: Mozilla/4.0

HTTP/1.1 200 OK
Date: Sun, 29 Jan 2023 22:58:25 GMT
Expires: -1
Cache-Control: private, max-age=0
Content-Type: text/html; charset=ISO-8859-1
...
Transfer-Encoding: chunked

5584
<!doctype html><html itemscope=""
itemtype="http://schema.org/WebPage"
lang="en"><head>
     …
     …
```

First part of the response – HTTP headers

Second part of the response – HTTP content

Problems with the sockets API

The sockets interface forces a read/write mechanism



Programming is often easier with a functional interface

To make distributed computing look more like centralized computing, I/O (read/write) is not the way to go

Remote Procedure Calls (RPC)

RPC

1984: Birrell & Nelson

- Mechanism to call procedures on other machines

Remote Procedure Call

Implementing RPC

No architectural support for remote procedure calls

Simulate it with tools we have (local procedure calls)

Simulation makes RPC a
language-level construct

The compiler creates code to send messages to invoke remote functions

instead of an
operating system construct

The OS gives us sockets

Implementing RPC

The trick:

Create stub functions
to make it appear to the user that the call is local

On the client

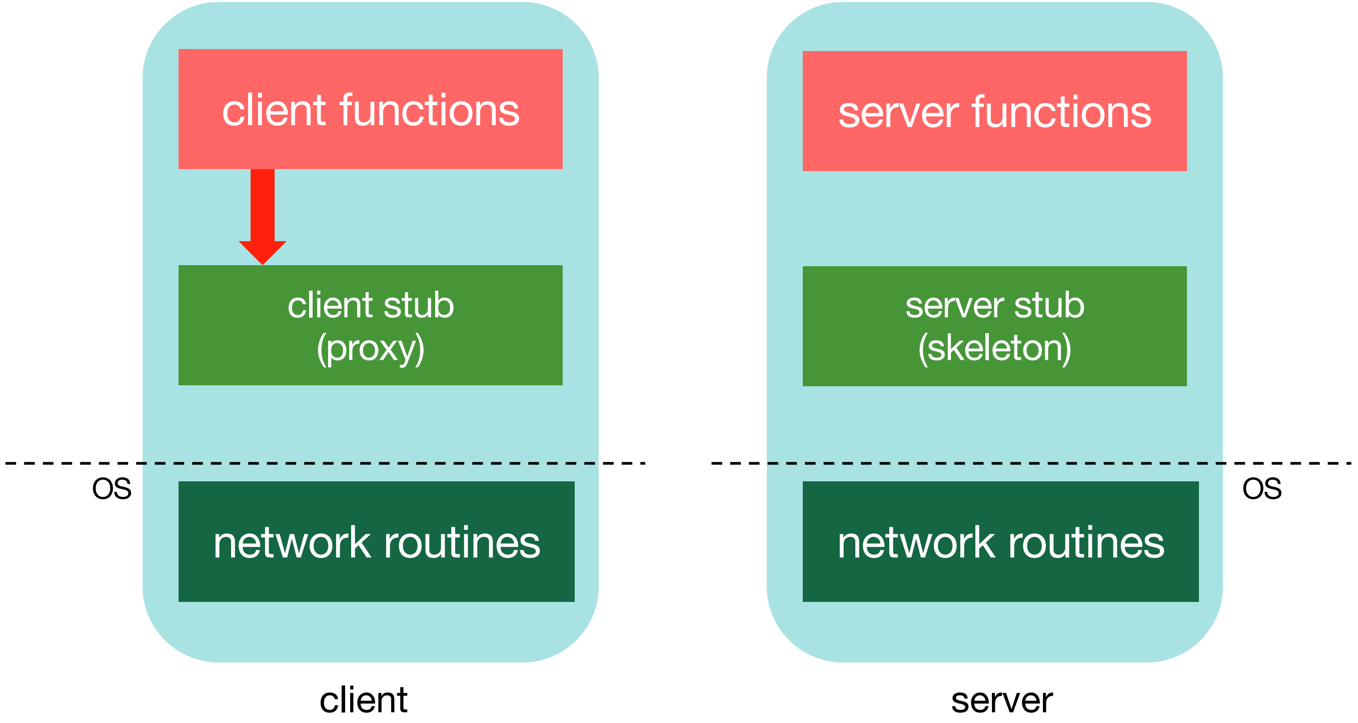
The stub function (proxy) has the function’s interface
Packages parameters and calls the server

On the server

The stub function (skeleton) receives the request and calls the local function

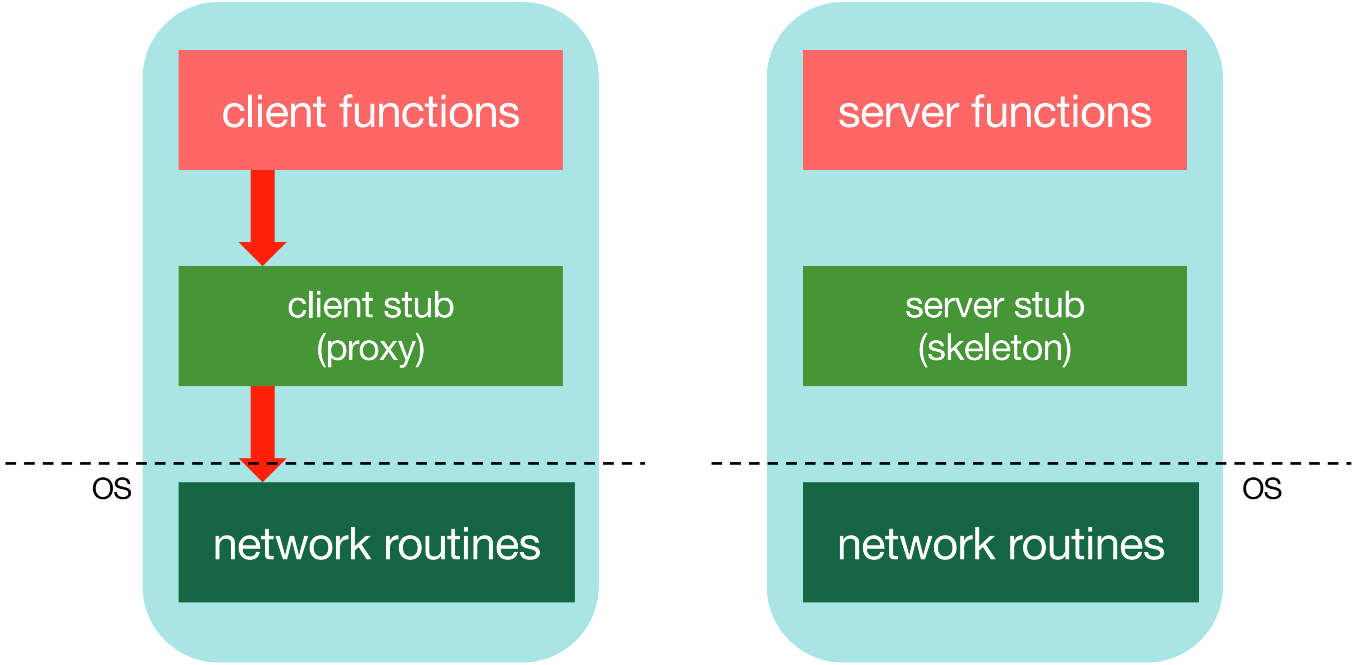
Stub functions

1. Client calls stub (params on stack)



Stub functions

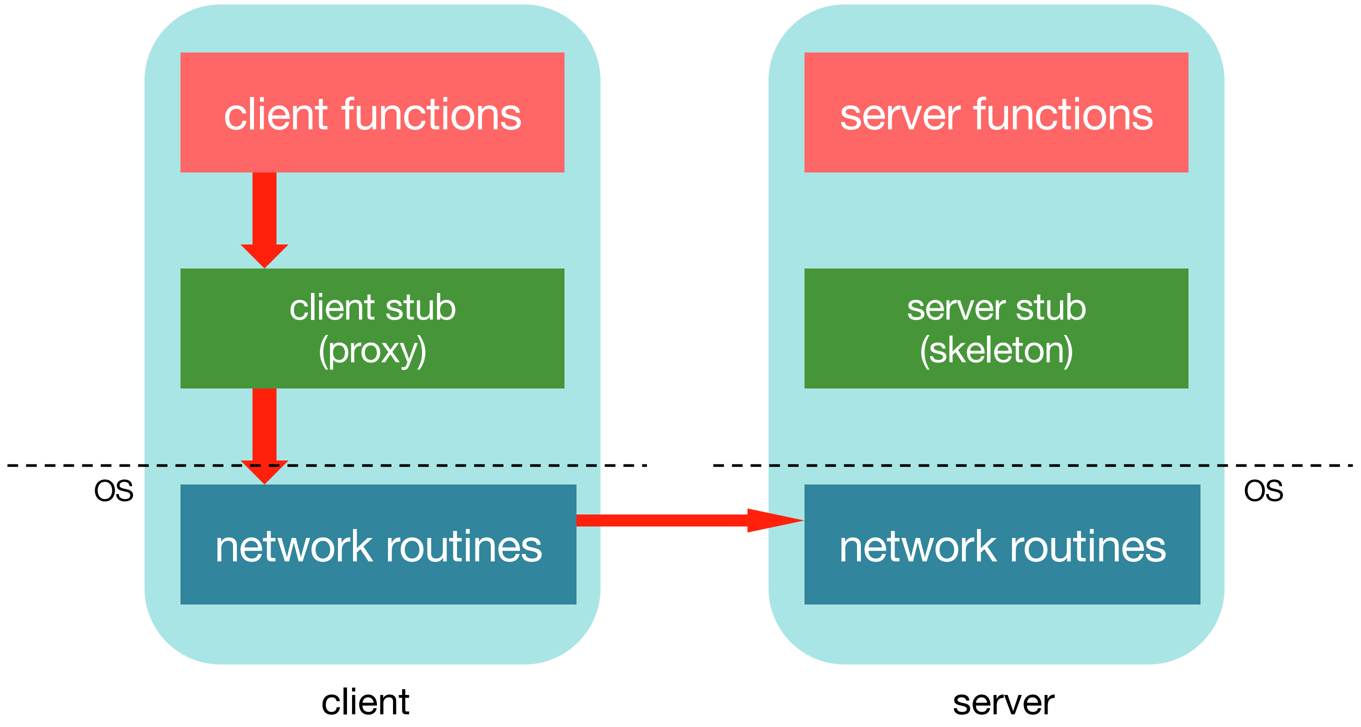
2. Stub marshals params to network message



Marshaling *= put parameters in a form suitable for transmission over a network (serialized) along with information about the function (function/method identifier, object ID, version, …)*

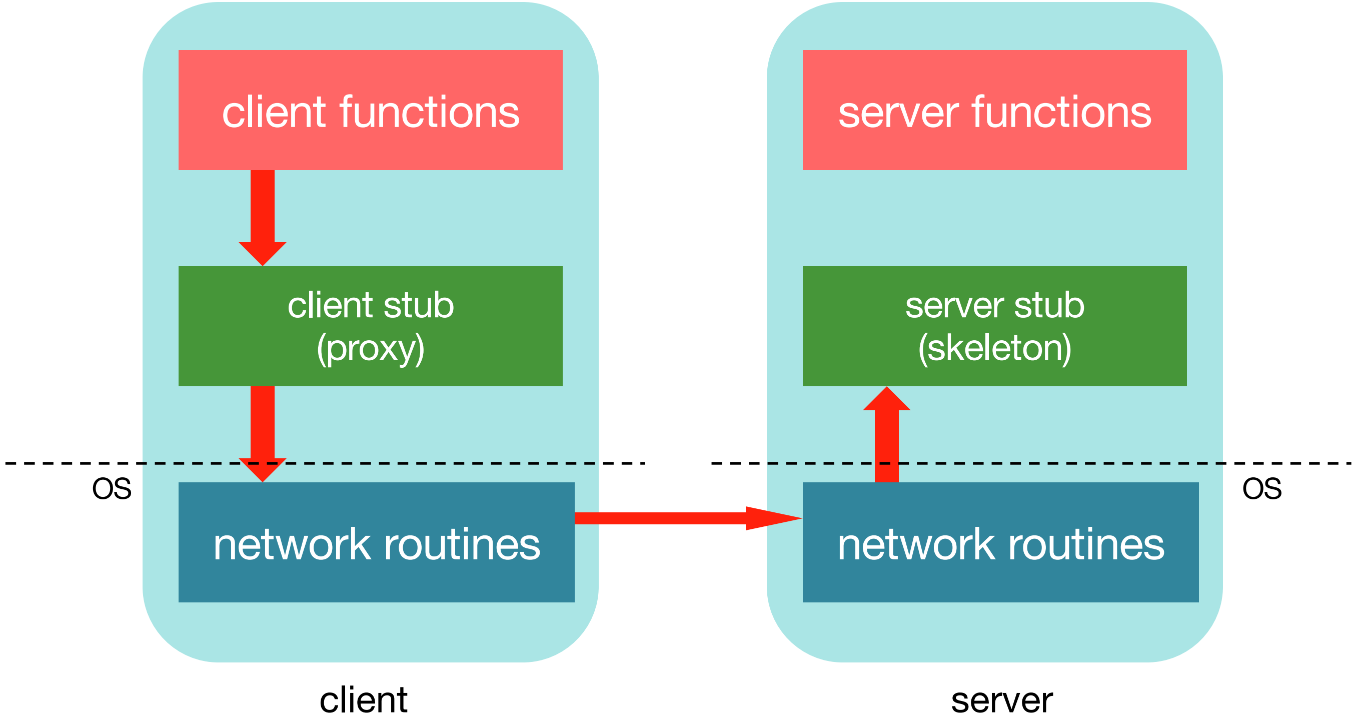
Stub functions

3. Network message sent to server



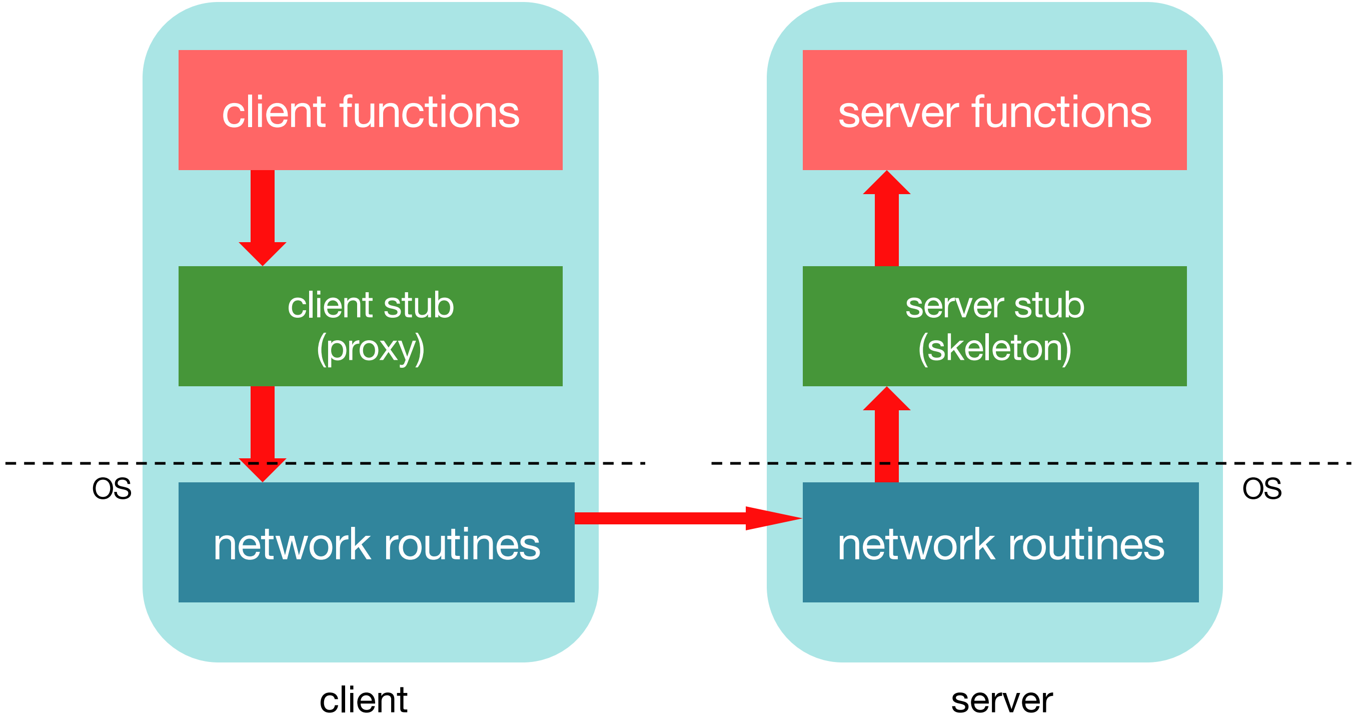
Stub functions

4. Receive message: send it to server stub



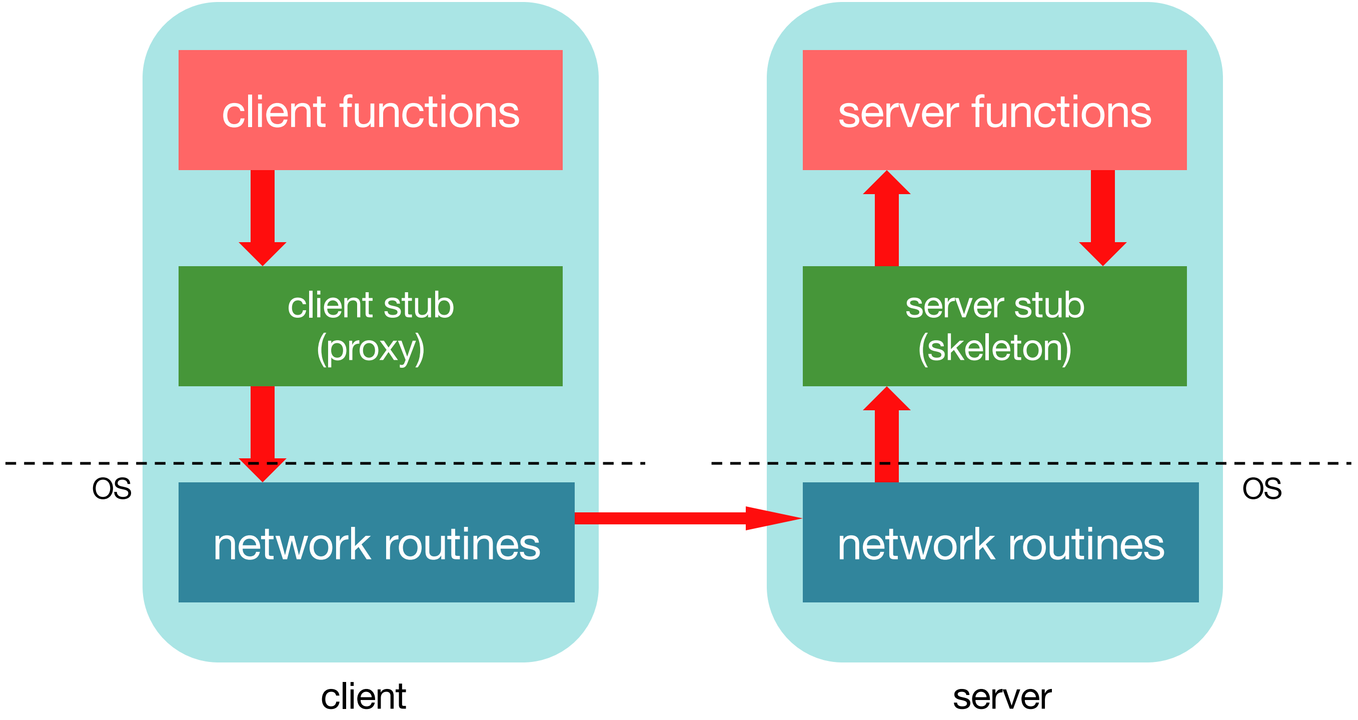
Stub functions

5. Unmarshal parameters, call server function



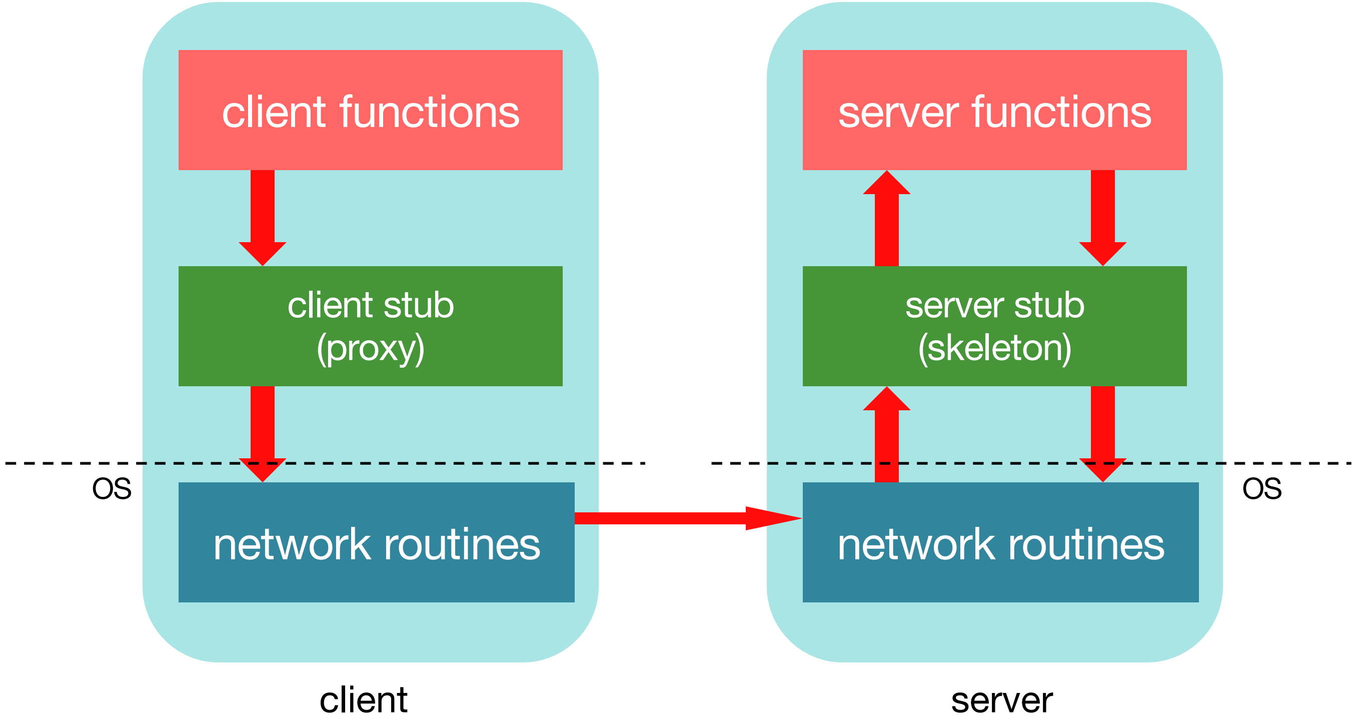
Stub functions

6. Return from server function



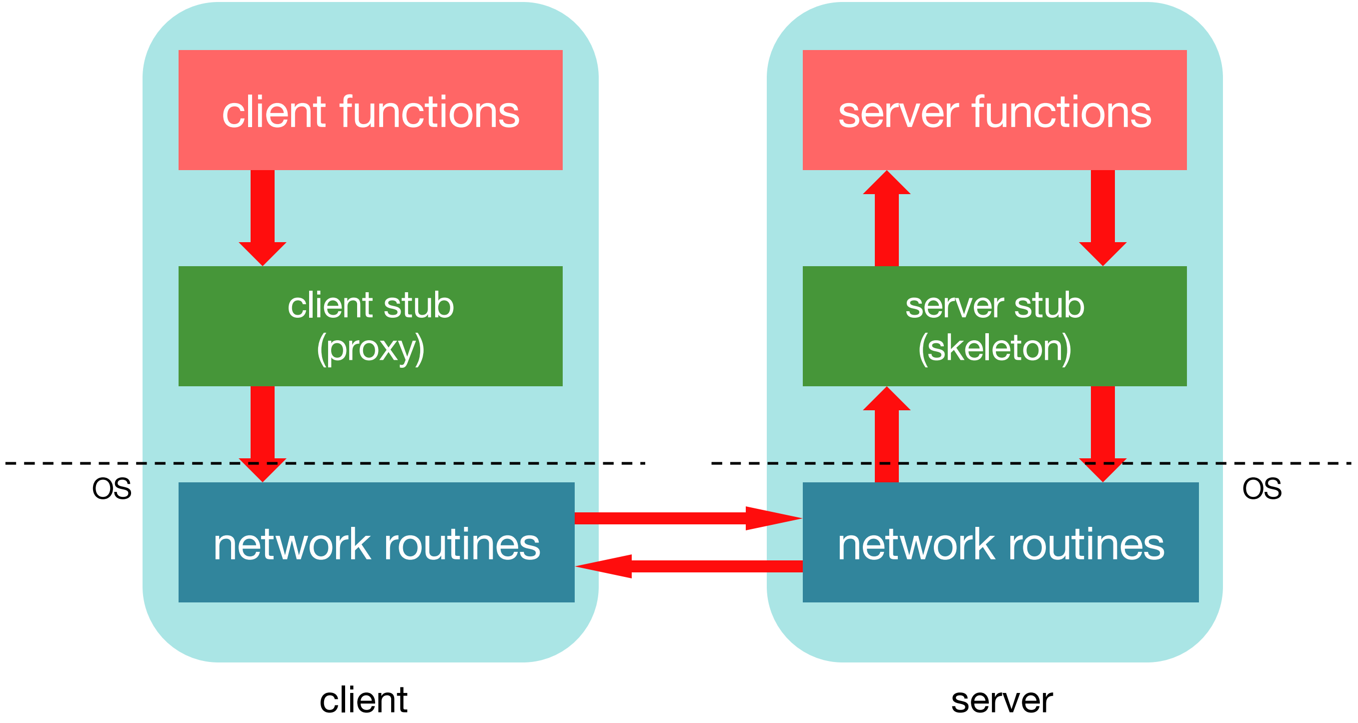
Stub functions

7. Marshal return value and send message



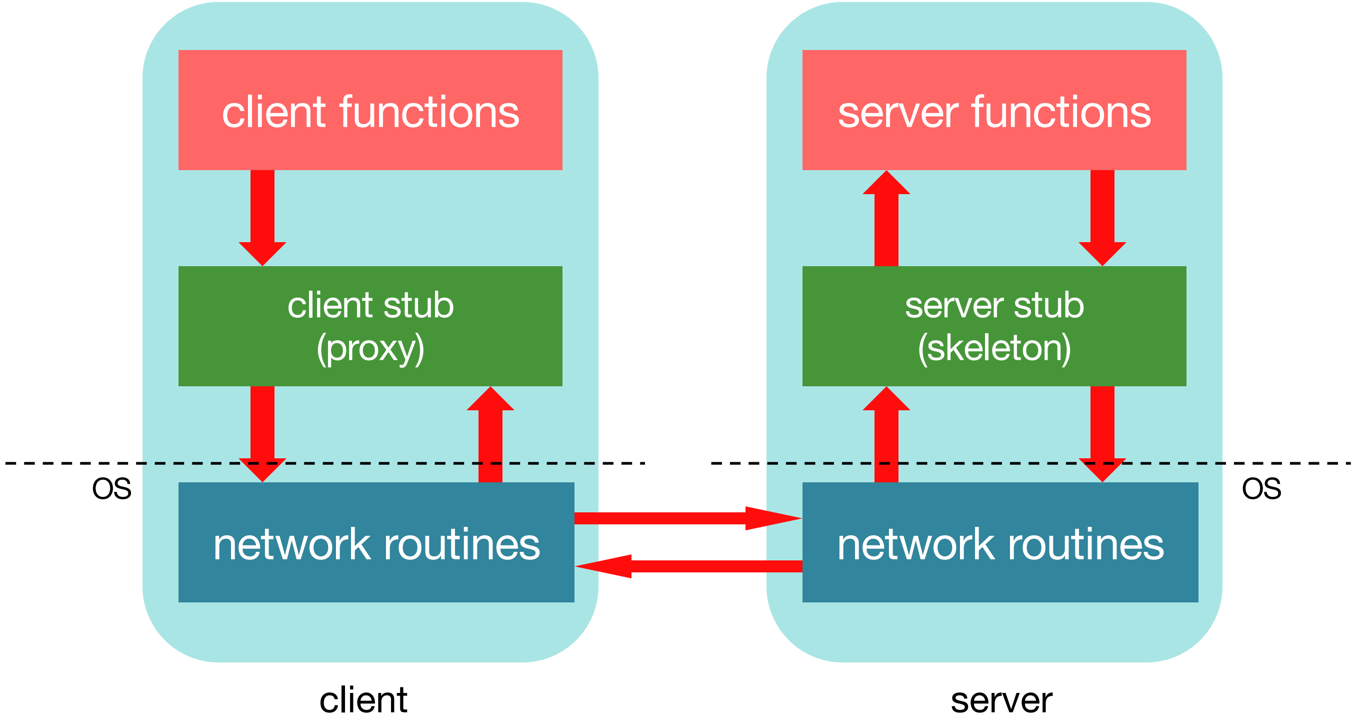
Stub functions

8. Transfer message over network



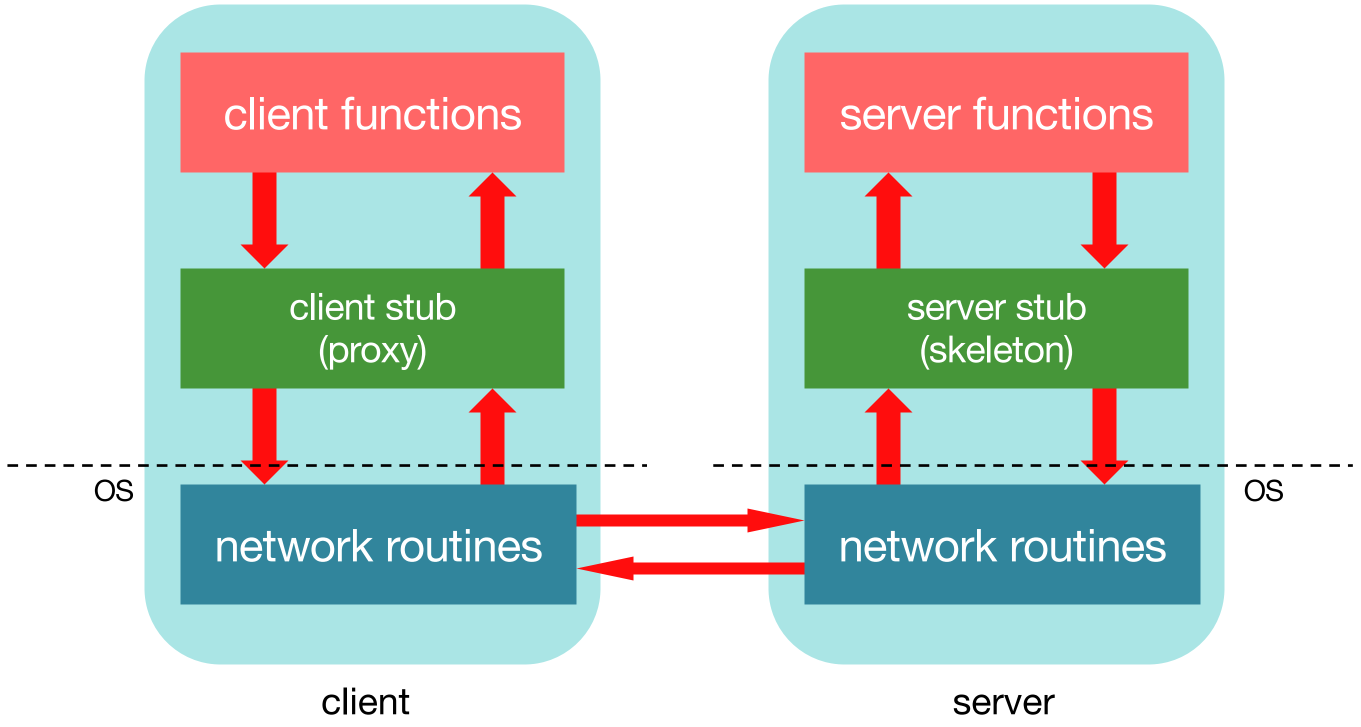
Stub functions

9. Receive message: client stub is receiver



Stub functions

10. Unmarshal return value(s), return to client code



A client proxy looks like the remote function

- Client proxy (stub) has the same interface as the remote function
- Looks & feels like the remote function to the programmer
  - But its function is to
    - Marshal parameters
    - Send the message
    - Wait for a response from the server
    - Unmarshal the response & return the appropriate data
    - Generate exceptions if problems arise

RPC Benefits

- RPC gives us a procedure call interface
- Writing applications is simplified
  - RPC hides all network code into stub functions
  - Application programmers don't have to worry about details
    - Sockets, port numbers, byte ordering

Implementation challenges

RPC Challenges

- Parameter passing
  - *Pass by value* or *pass by reference*?
  - All data must be sent in a pointerless representation
- Service binding
  - How do we register & locate the server endpoint?
  - Central database listing all services and their corresponding host & port #?
  - Or a database of services running on each server?
- Transport protocol
  - TCP? UDP? Either? HTTP/HTTPS over TCP?
- Error handling
  - Opportunities for failure

Semantics of Remote Procedure Calls

- Local procedure call: executed ***exactly once*** each time it's invoked
- Most RPC systems will offer either
  - ***at least once*** semantics (client might retry)
  - or ***at most once*** semantics (client will not retry)
- Decide which to use based on the application
  - **idempotent** functions: may be called any number of times without harm
  - **non-idempotent** functions: those with side-effects
- Ideally – design your application to be idempotent … and stateless
  - Then you don't worry about retries
  - Not always easy!
  - That makes it easy to enable other servers to handle the request

More Challenges

Performance

- RPC is slower … a lot slower than a local procedure call (why?)

Security

- messages may be visible over network – do we need to hide them?
- Authenticate client?
- Authenticate server?

Programming with RPC

Language support

- Many programming languages have no language-level concept of remote procedure calls (C, C++, Java <J2SE 5.0, …)
  - These compilers will not automatically generate client and server stubs
- Some languages have support (e.g., reflection) that enables RPC packages (Java, Python, Haskell, Go, Erlang)
  - But we may need to support heterogeneous environments (e.g., a Java client communicating with a Python service)

Common solution

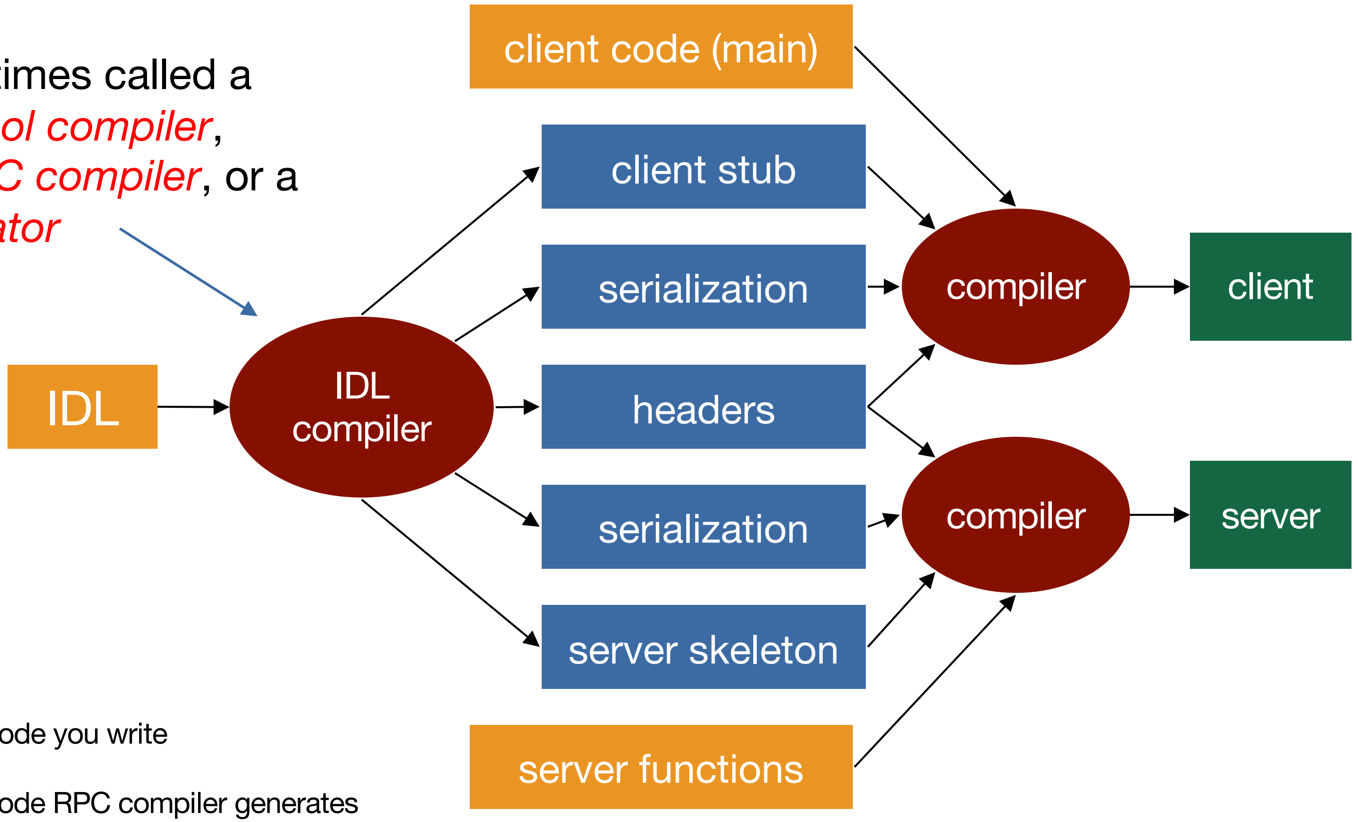
- **Interface Definition Language** (**IDL**): describes remote procedures
- A separate **compiler** generates client & server stubs

Interface Definition Language (IDL)

- Allow programmer to specify remote procedure interfaces (*names, parameters, return values*)
- IDL compiler can use this to generate client and server stubs
  - Marshaling code
  - Unmarshaling code
  - Network transport routines
  - Conform to defined interface
- An IDL looks similar to function prototypes

RPC compiler

Sometimes called a *protocol compiler*, an *RPC compiler*, or a *generator*



🟧 Code you write

🟦 Code RPC compiler generates

Writing the program

- Client code has to be modified
  - Initialize RPC-related options
    - Identify transport type
    - Locate server/service
  - Handle failure of remote procedure calls

- Server functions
  - Generally, need little or no modification
  - Need a container that runs those functions
    - Either the user writes a server that registers the functions and starts a listener or the RPC complier creates one

The End