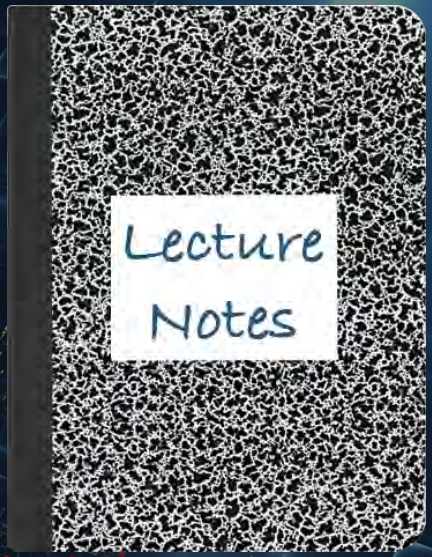
CS 417 – DISTRIBUTED SYSTEMS

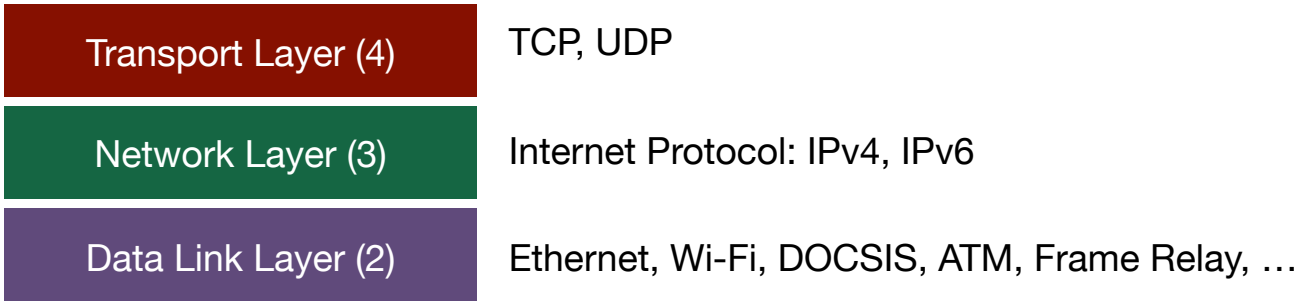
Week 2: Part 1

Point-to-point communication: Remote Procedure Calls



Paul Krzyzanowski

Network Communications



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

TCP Upsides & Downsides

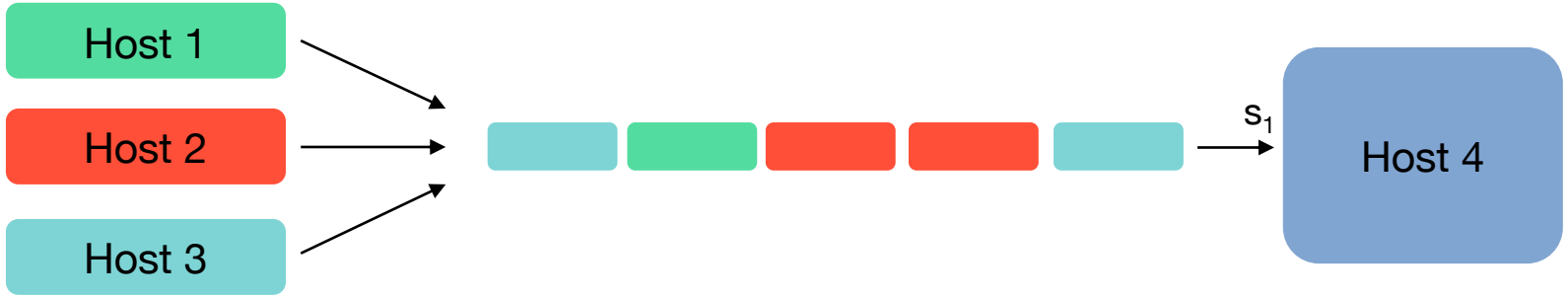
- Upsides – huge!
  - In-order, reliable byte streams
  - Congestion control, flow control (avoids queue overflow)
- Downsides
  - Storing & managing state
    - Sequence numbers, Buffering out-of-order data, Acknowledgments
    - Significant kernel memory use when lots of connections
  - Congestion control slows down transmission but doesn't always reflect network congestion
  - 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
      - TCP uses Nagle's algorithm to avoid sending lots of small packets

UDP Upsides & Downsides

- Downsides
  - Delivery & message order not guaranteed
- Upsides
  - Fewer kernel resources
  - No setup overhead
  - 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

Identifying Sessions:UDP

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

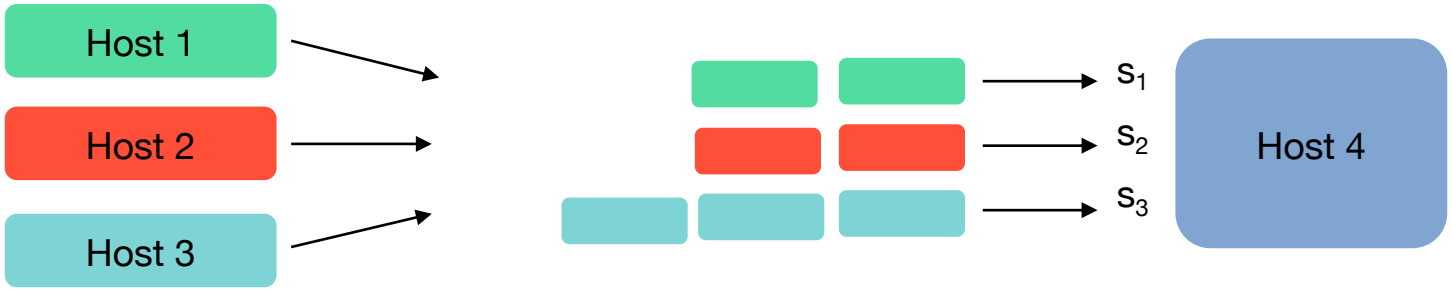


Packets sent from different processes and/or systems
all arrive on the same socket on the server

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”



Each connection results in a new socket at the server

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

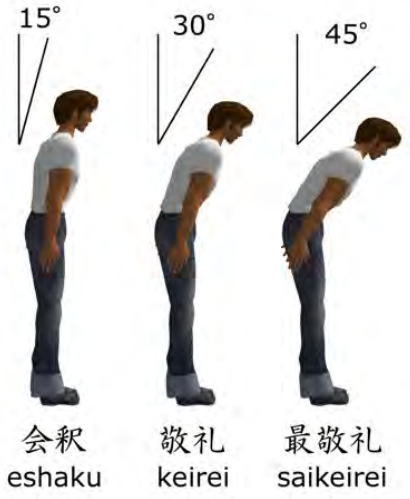
- Line-oriented, text-based protocols common
  - Not efficient but easy to debug & use

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!

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, 18 Sep 2022 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

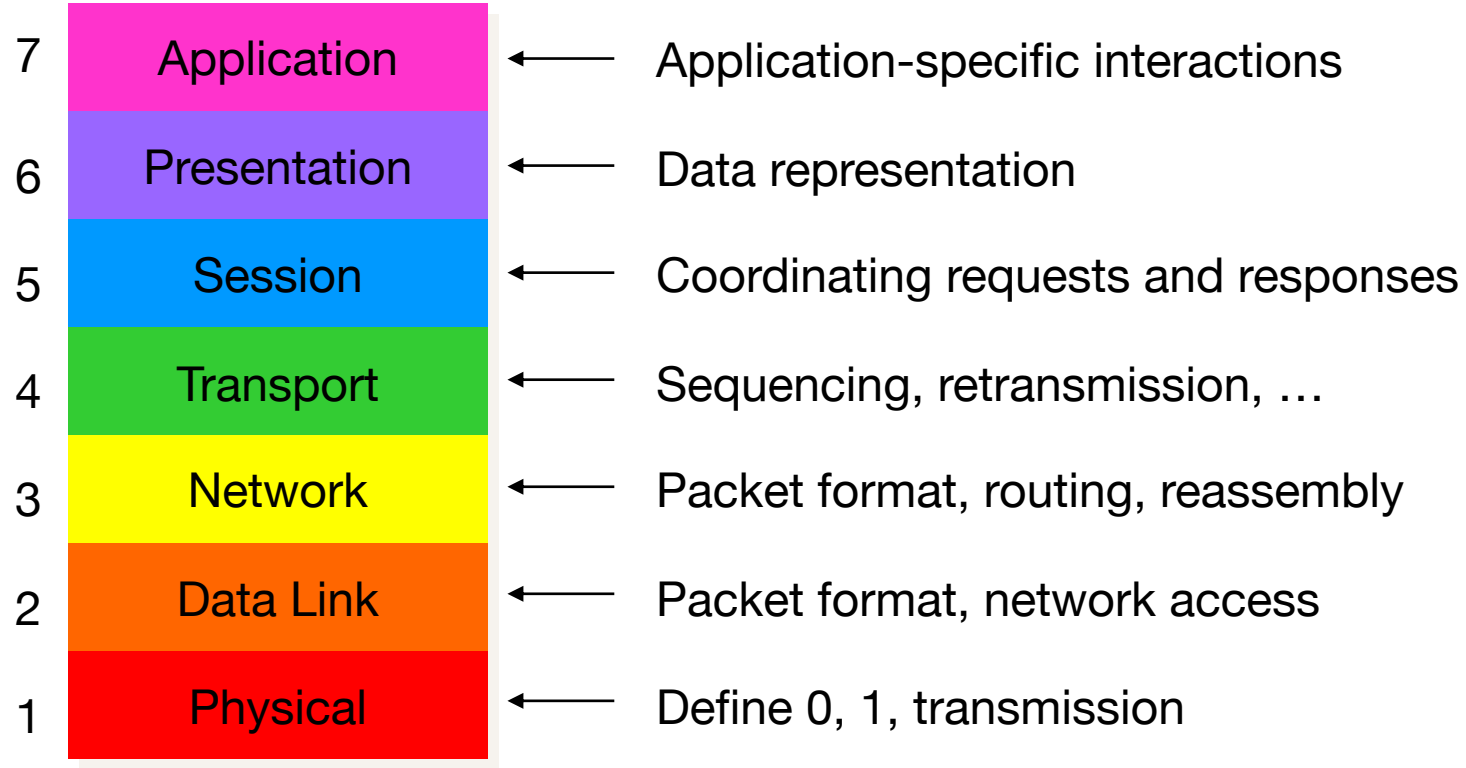
HTTP = Hypertext Transfer Protocol

```
$ 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, 18 Sep 2022 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

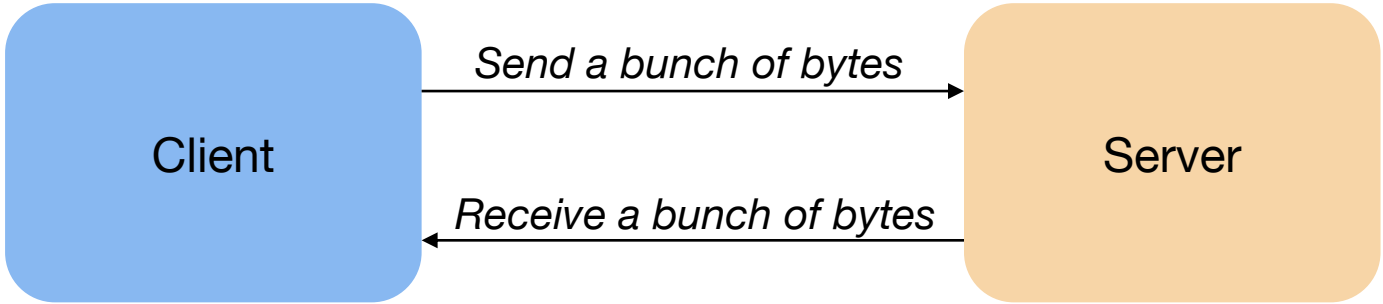
Second part of the response – HTTP content

Network Protocols



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

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

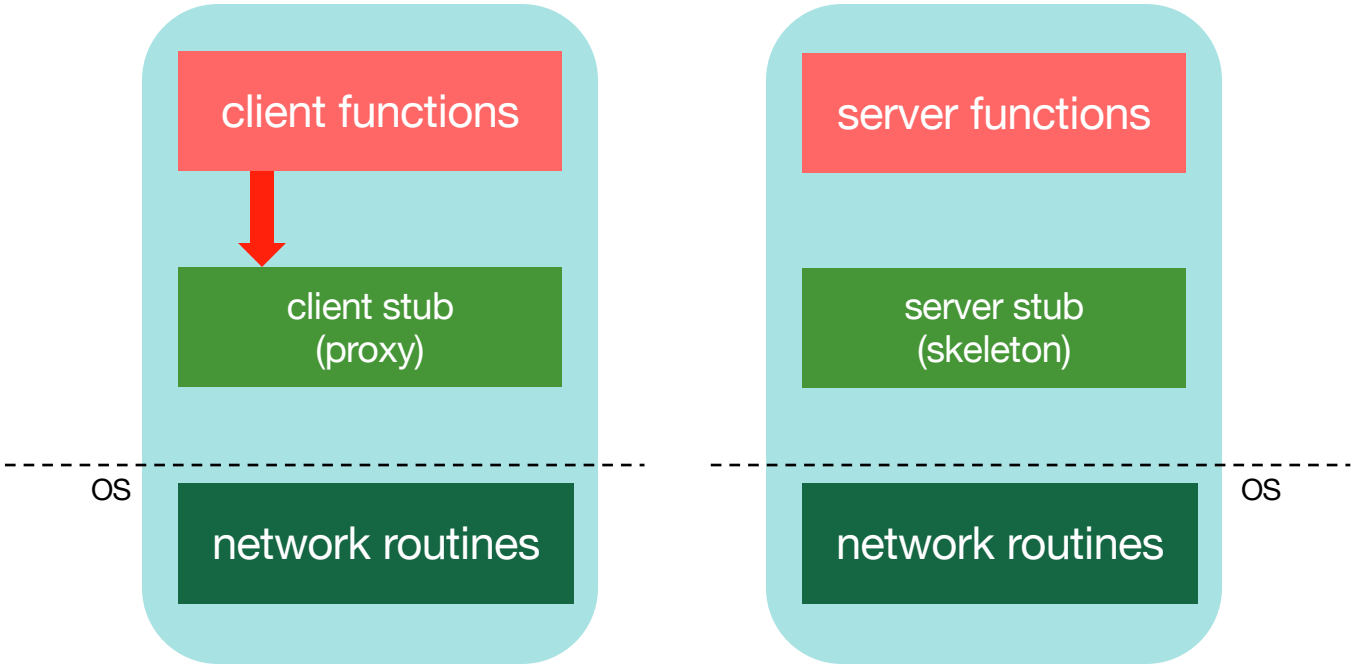
The stub function (proxy) has the function's interface
Packages parameters and calls the server

On the server

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

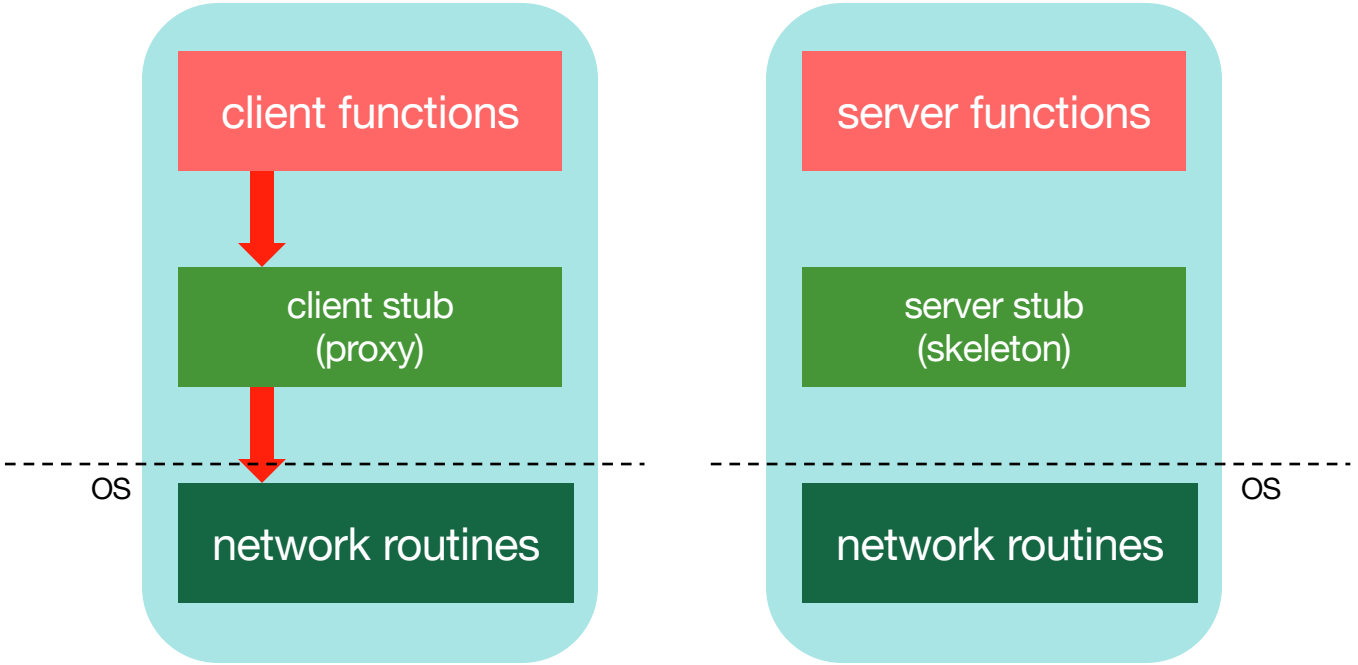
Stub functions

1. Client calls stub (params on stack)



Stub functions

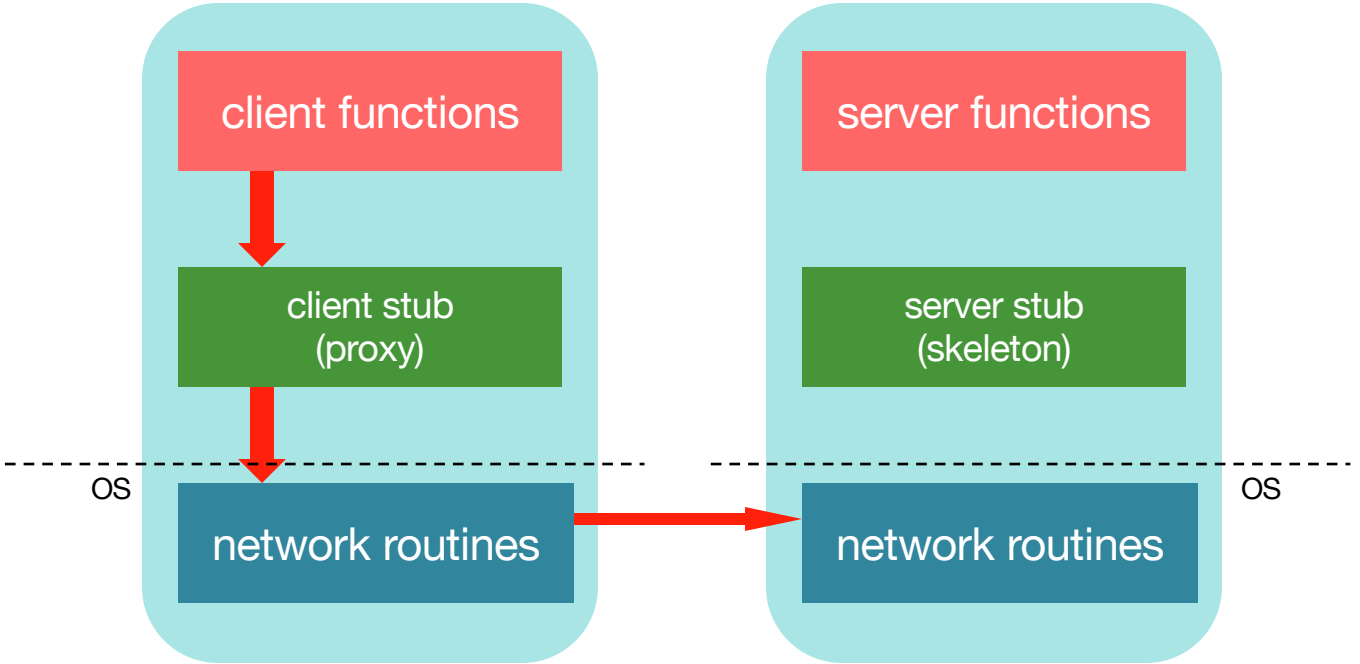
2. Stub marshals params to network message



Marshalling = put parameters in a form suitable for transmission over a network (serialized)

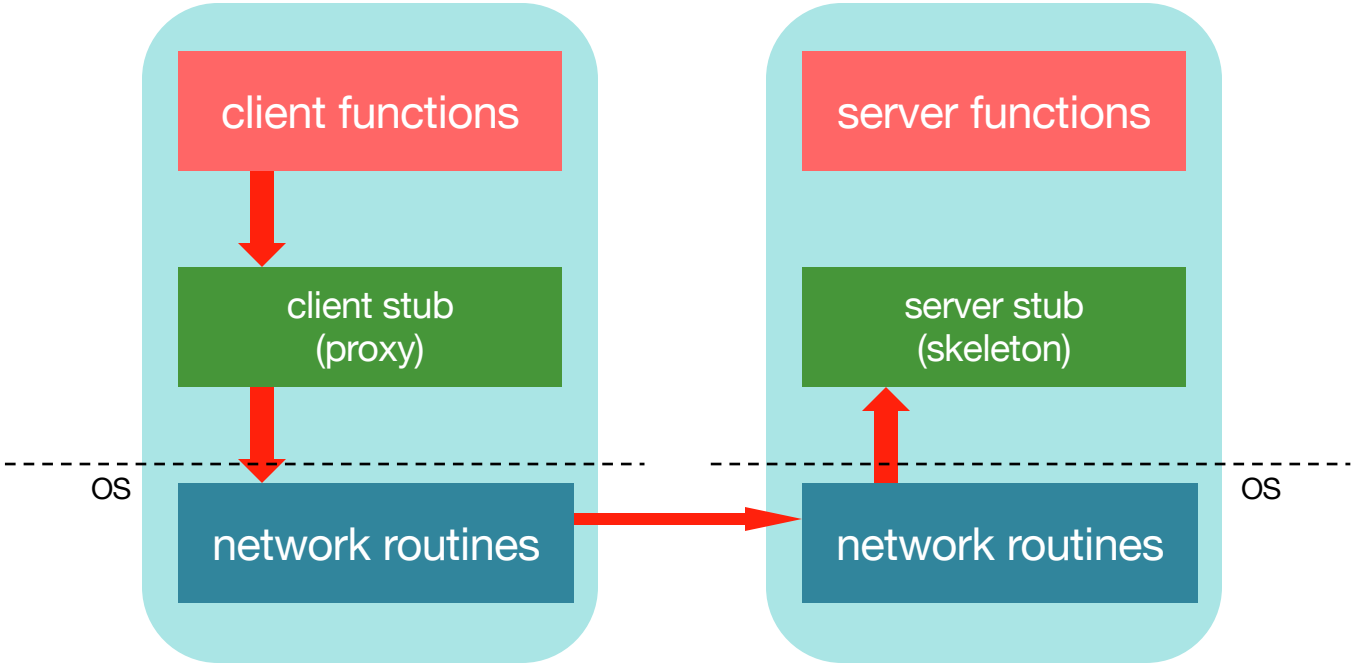
Stub functions

3. Network message sent to server



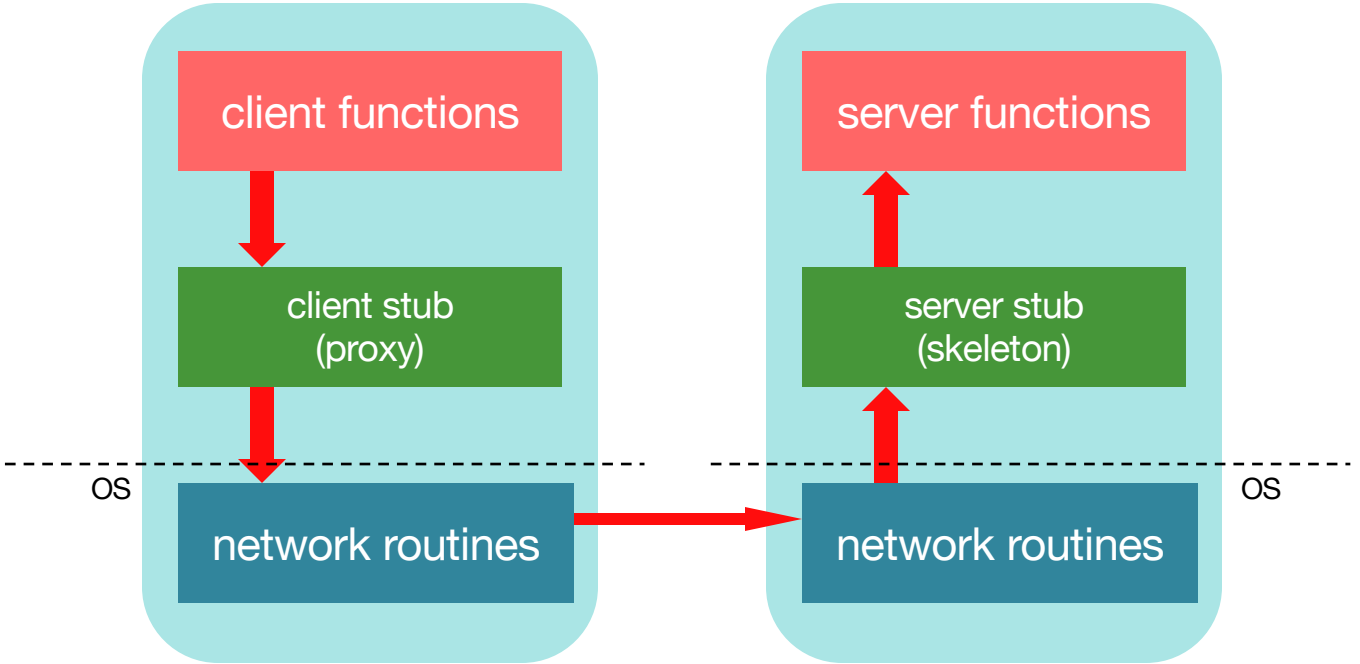
Stub functions

4. Receive message: send it to server stub



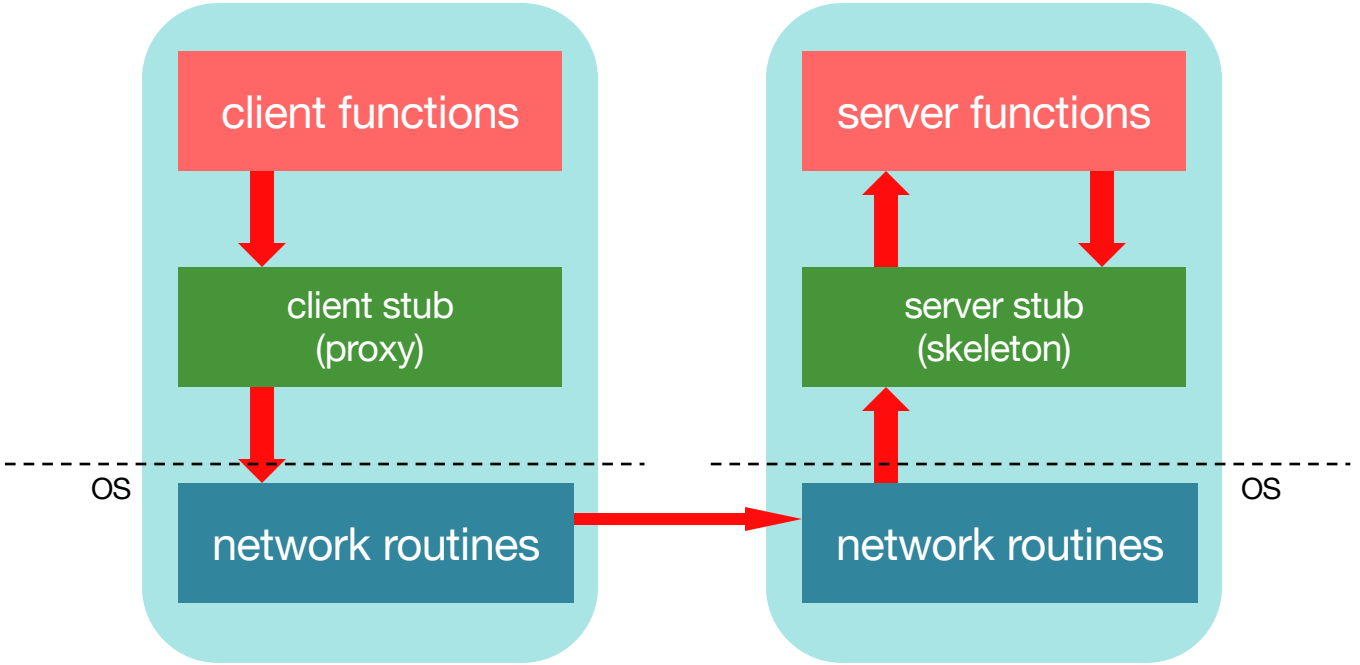
Stub functions

5. Unmarshal parameters, call server function



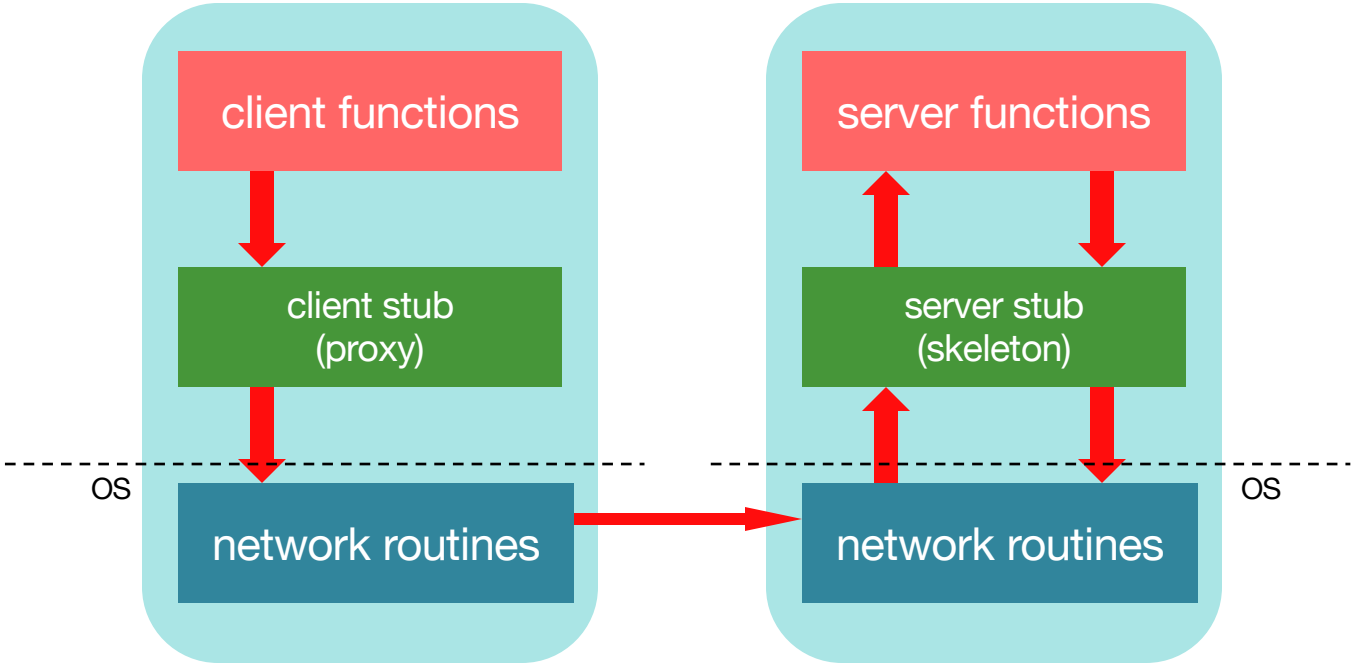
Stub functions

6. Return from server function



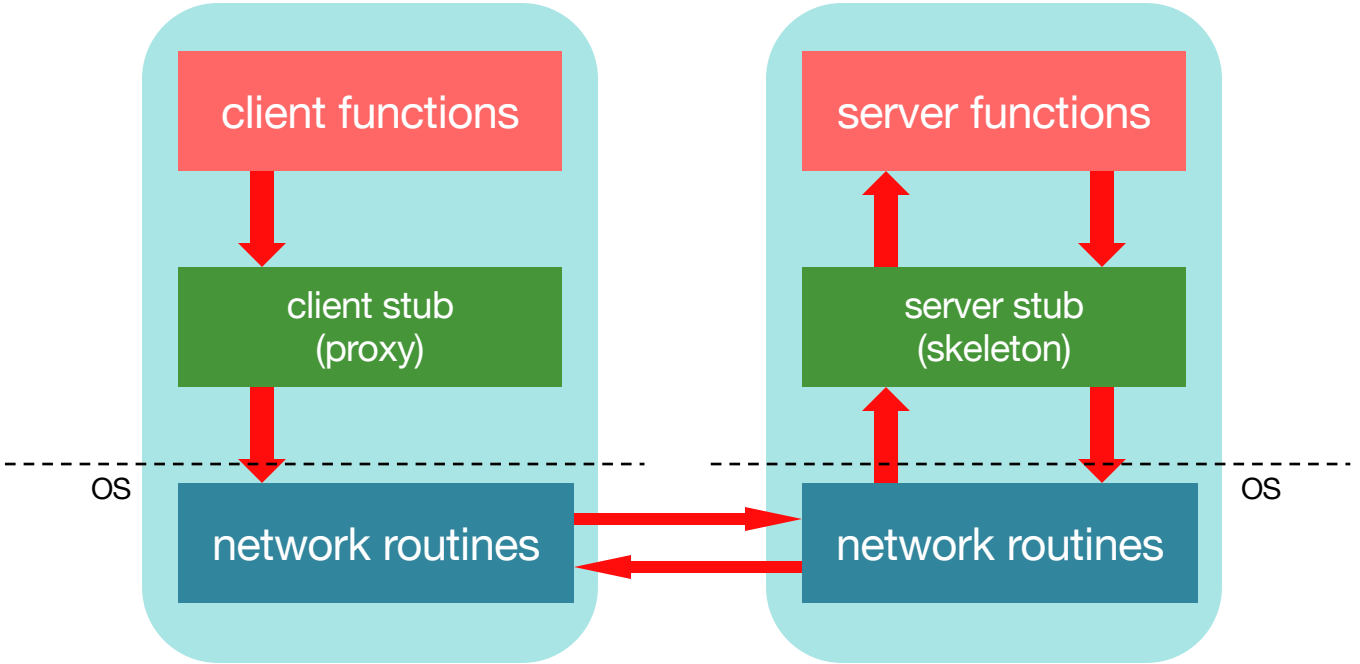
Stub functions

7. Marshal return value and send message



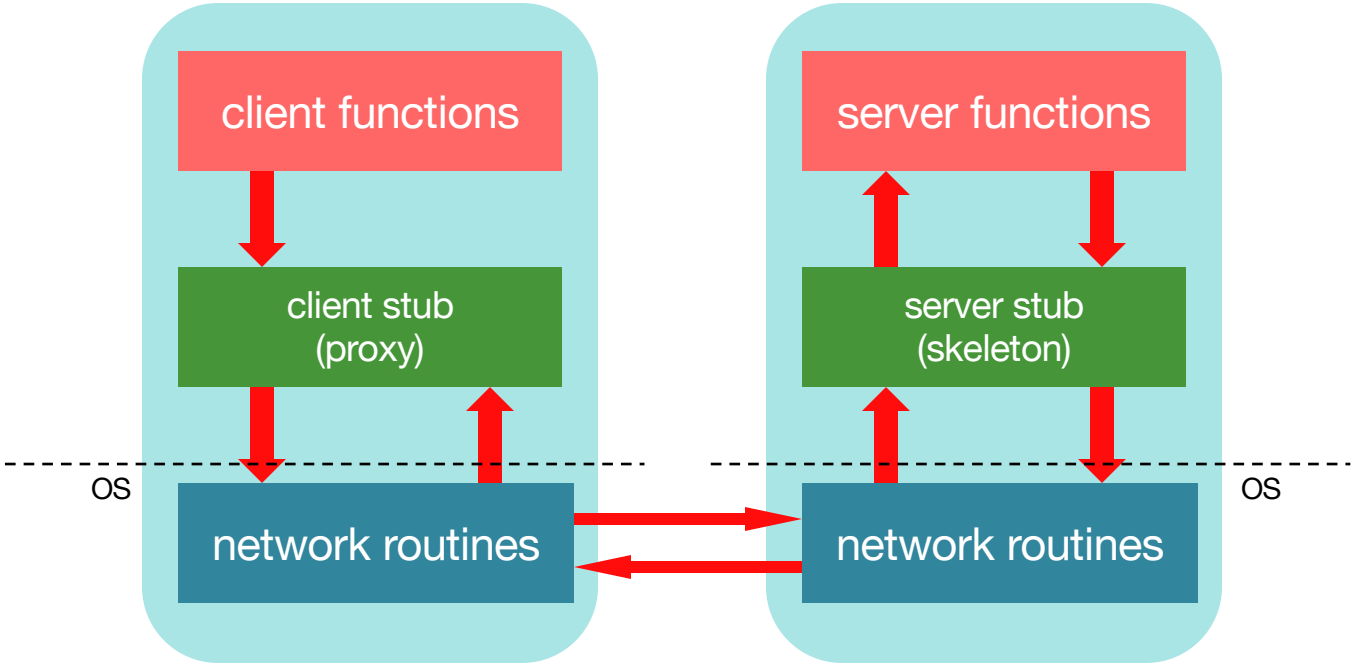
Stub functions

8. Transfer message over network



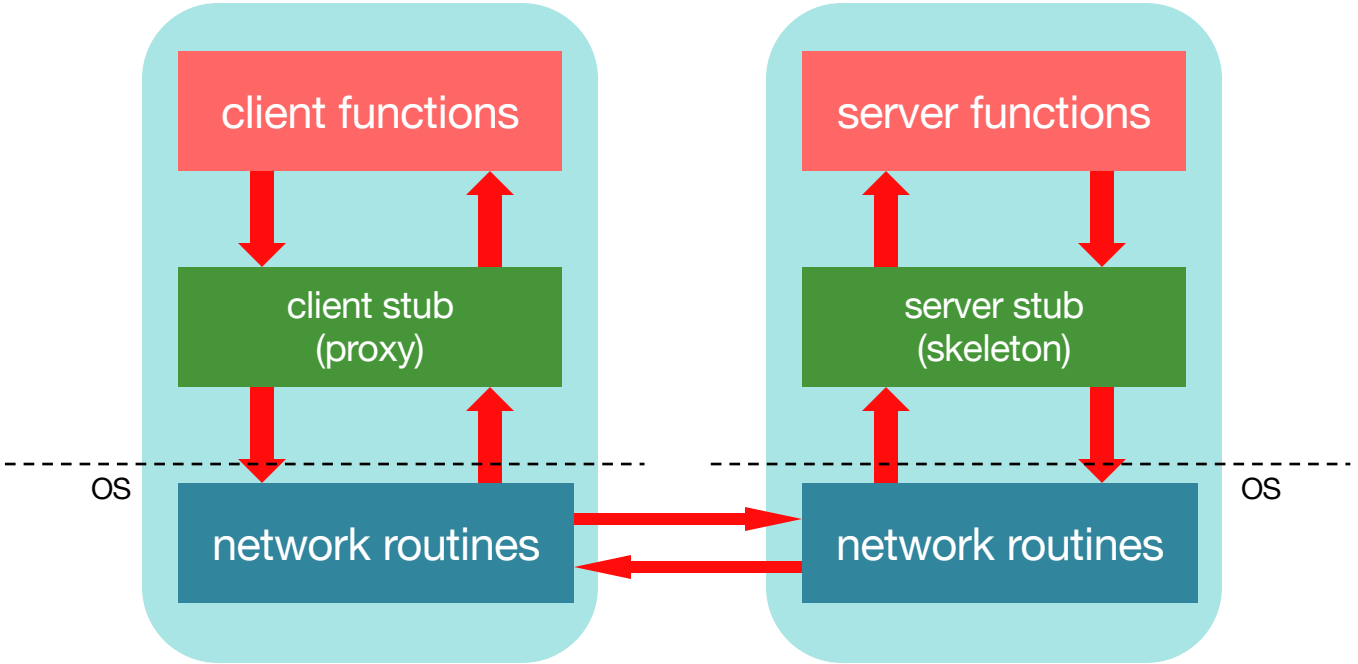
Stub functions

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

RPC has challenges

RPC Issues

- Parameter passing
  - *Pass by value* or *pass by reference*?
  - All data must be sent in a pointerless representation
- Service binding. How do we 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? Both?
- When things go wrong
  - Opportunities for failure

When things go wrong

- Semantics of remote procedure calls
  - Local procedure call: ***exactly once***
- Most RPC systems will offer either
  - ***at least once*** semantics
  - or ***at most once*** semantics
- Decide which to use based on the application
  - **idempotent** functions: may be run any number of times without harm
  - **non-idempotent** functions: those with side-effects
- Ideally – design your application to be idempotent … and stateless
  - Not always easy!
  - Store transaction IDs, previous return data, etc.

More issues

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 that enables RPC
  (Java, Python, Haskell, Go, Erlang)
  - But we may need to deal with heterogeneous environments
    (e.g., Java communicating with a Python service)

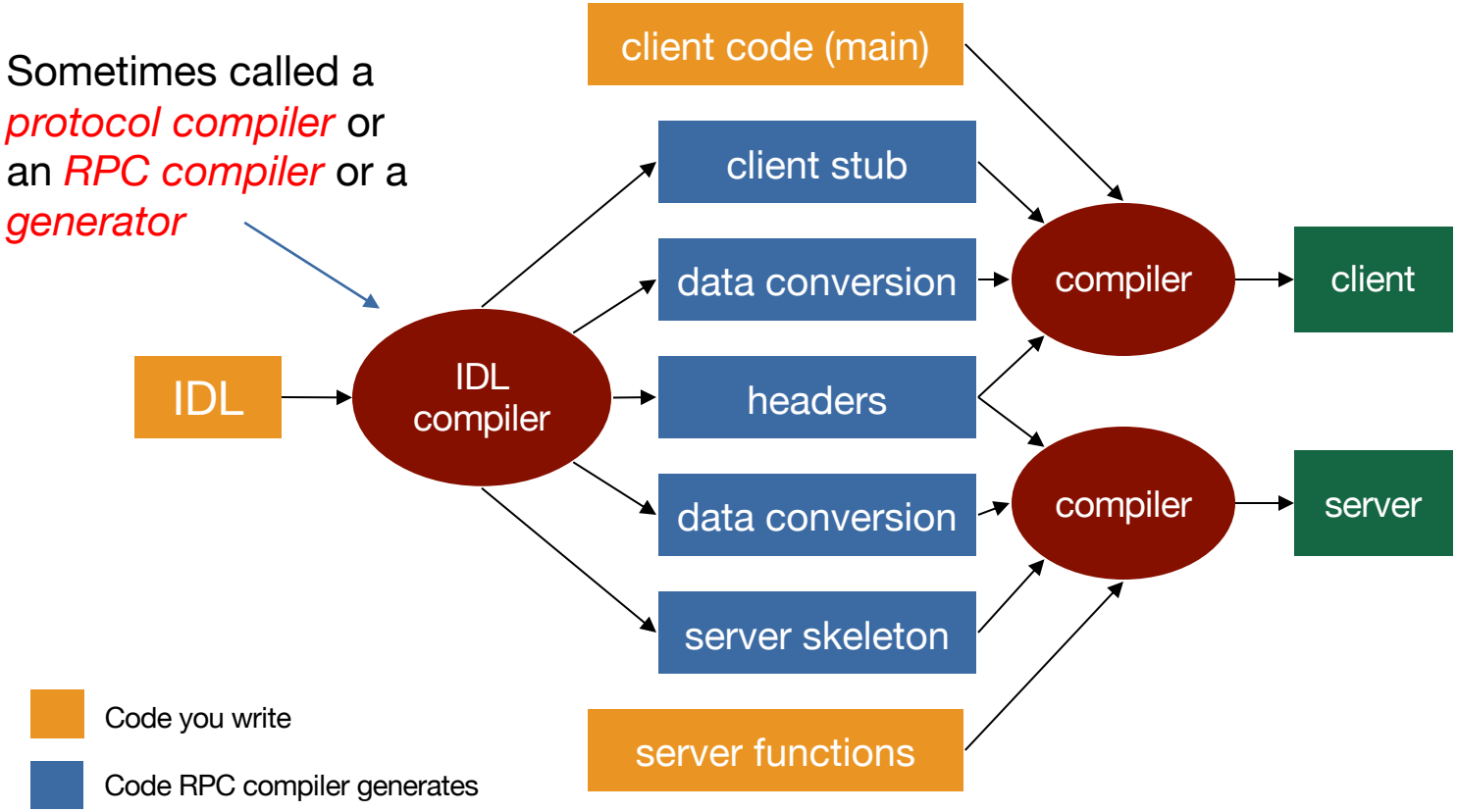
Common solution

- Interface Definition Language (IDL): describes remote procedures
- A separate compiler generates client & server stubs (pre-compiler)
  - There are other terms: gRPC calls this a ***generator***;

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



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

The End