

# Operating Systems Design

## 15. Networking: Remote Procedure Calls

Paul Krzyzanowski

[pxk@cs.rutgers.edu](mailto:pxk@cs.rutgers.edu)

# Problems with the sockets API

---

The **sockets** interface forces a read/write mechanism

Programming is easier with a functional interface

# RPC

---

1984: Birrell & Nelson

- Mechanism to call procedures on other machines

## ***Remote Procedure Call***

Goal: it should appear to the programmer that  
a normal call is taking place

# Regular procedure calls

---

You write:

```
x = f(a, "test", 5);
```

The compiler parses this and generates code to:

- a. Push the value 5 on the stack
- b. Push the address of the string "test" on the stack
- c. Push the current value of a on the stack
- d. Generate a call to the function f

In compiling *f*, the compiler generates code to:

- a. Push registers that will be clobbered on the stack to save the values
- b. Adjust the stack to make room for local and temporary variables
- c. Before a return, unadjust the stack, put the return data in a register, and issue a return instruction

# Implementing RPC

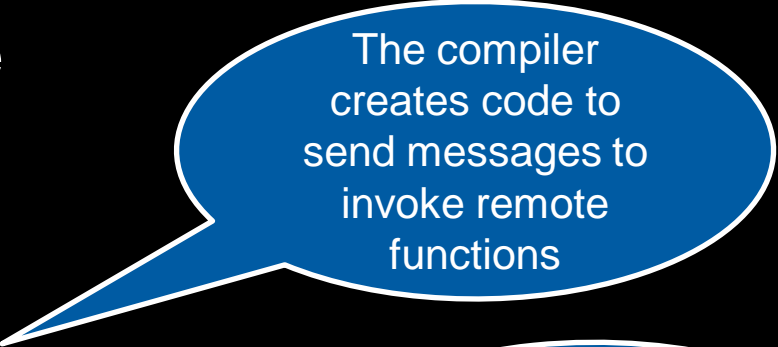
---

No architectural support for remote procedure calls

*Simulate it* with tools we have  
(local procedure calls)

Simulation makes RPC a  
**language-level construct**

instead of an  
**operating system construct**



The compiler  
creates code to  
send messages to  
invoke remote  
functions



The OS gives us  
sockets

# Implementing RPC

---

The trick:

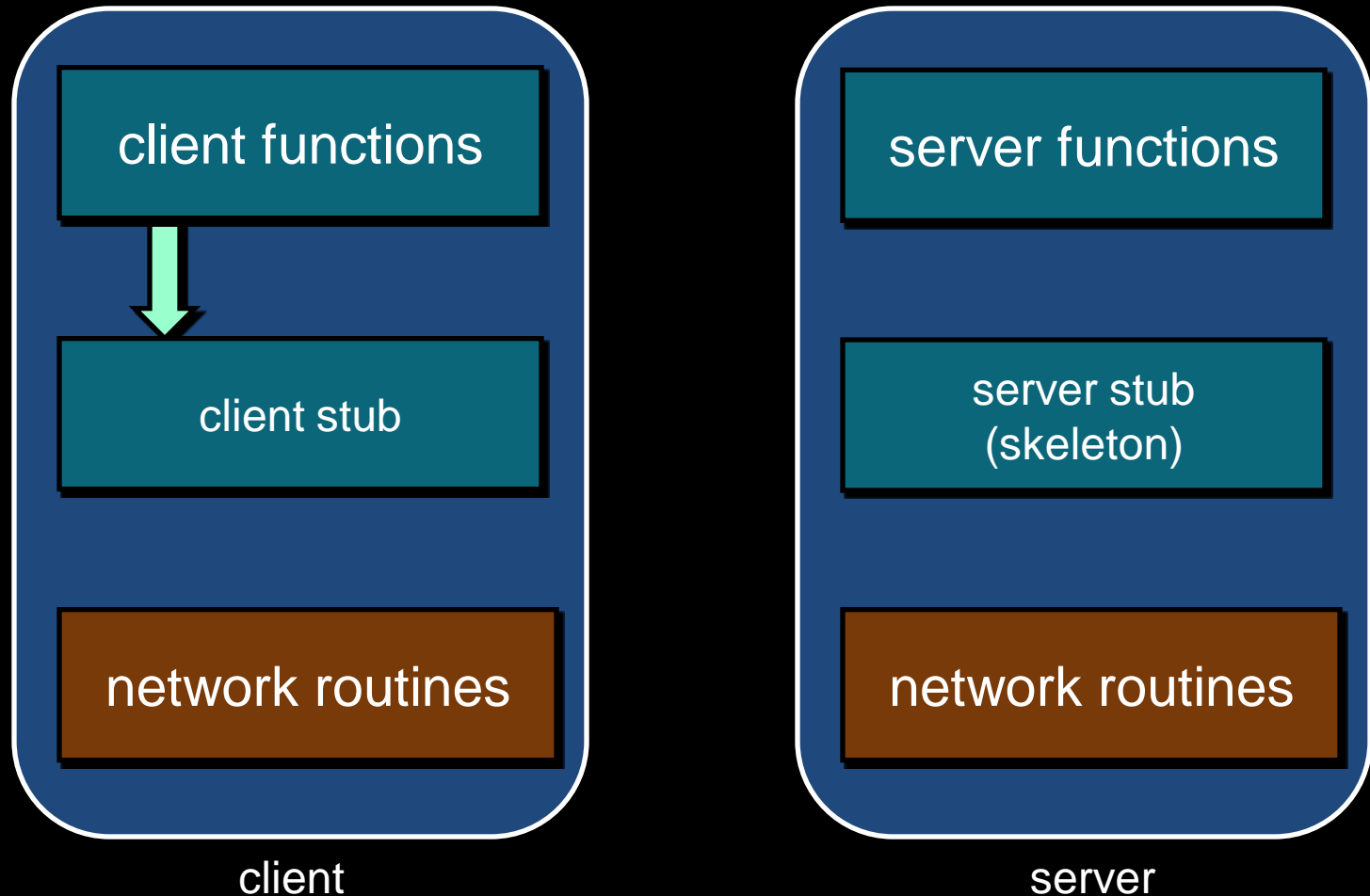
Create **stub functions** to make it appear to the user that the call is local

Stub function contains the function's interface

# Stub functions

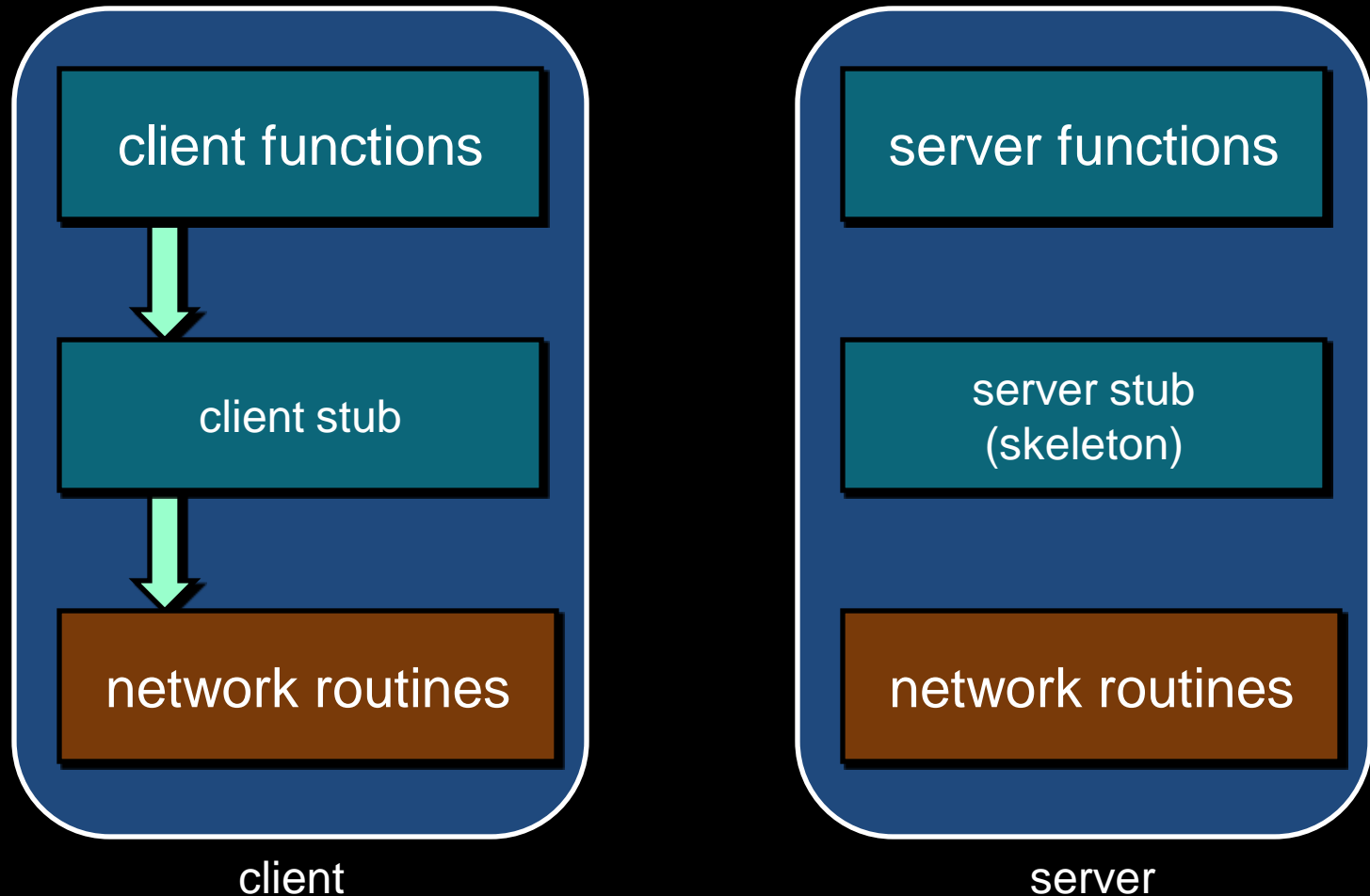
---

## 1. Client calls stub (params on stack)



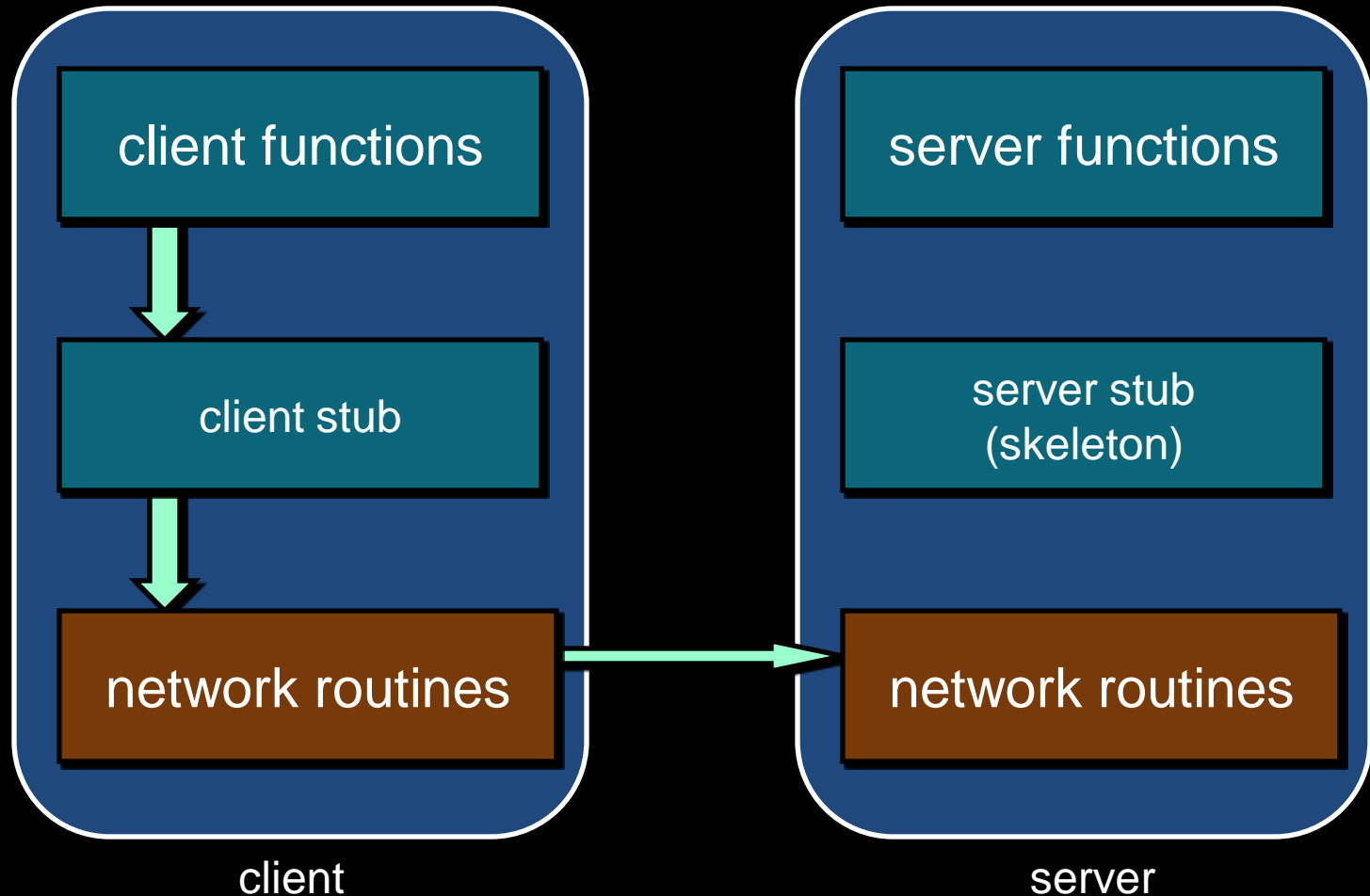
# Stub functions

## 2. Stub marshals params to net message



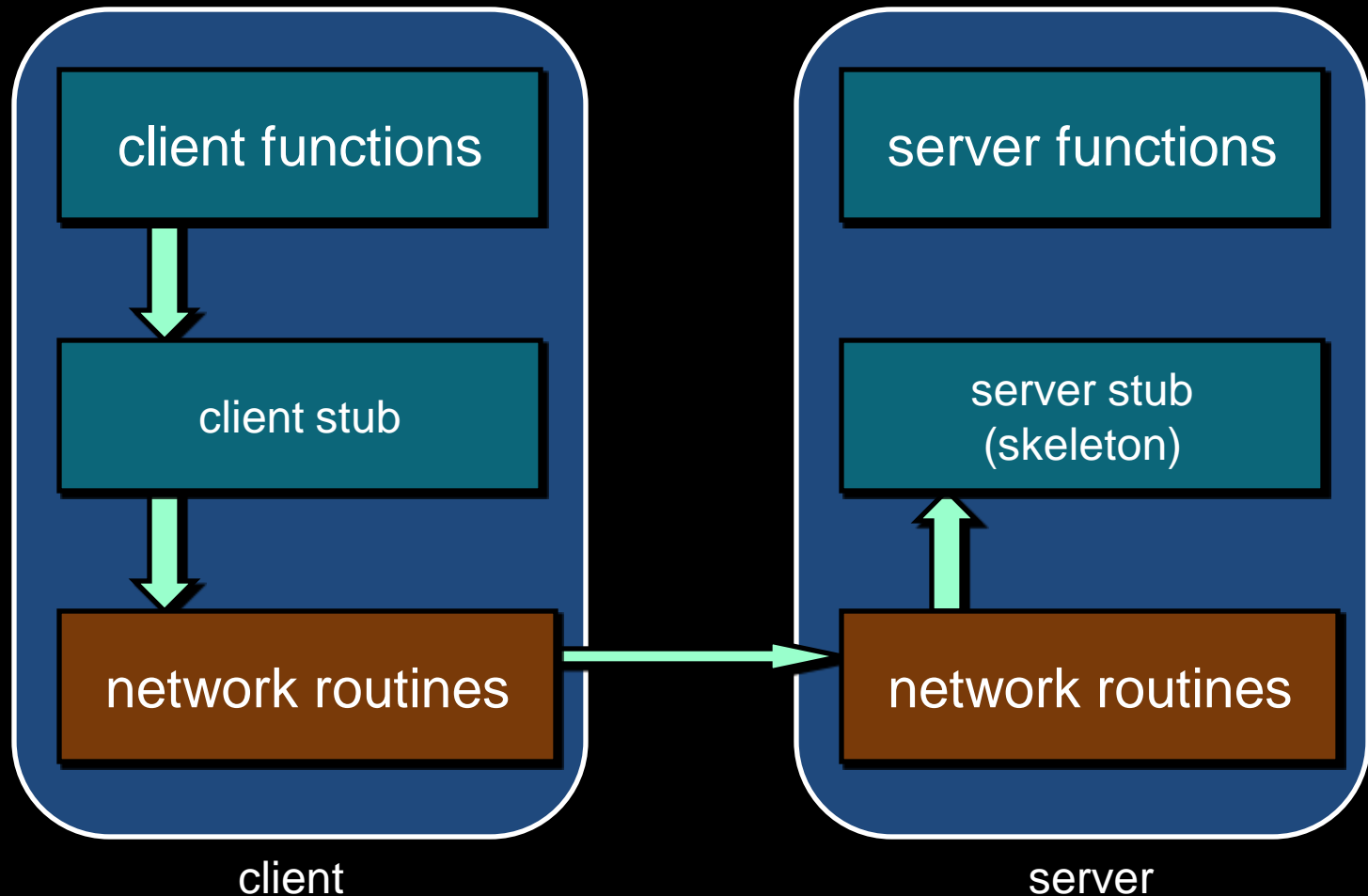
# Stub functions

## 3. Network message sent to server



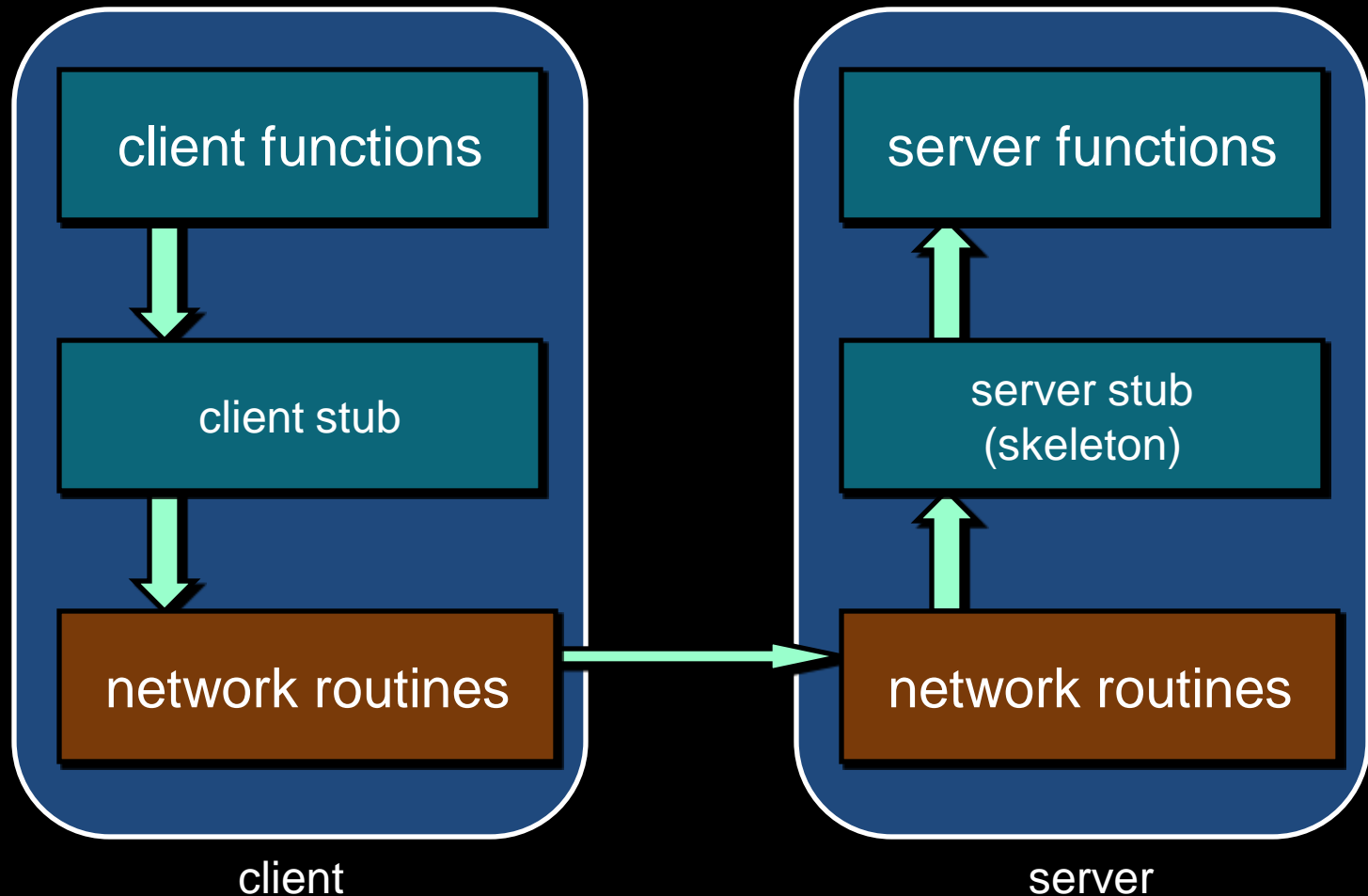
# Stub functions

## 4. Receive message: send it to server stub



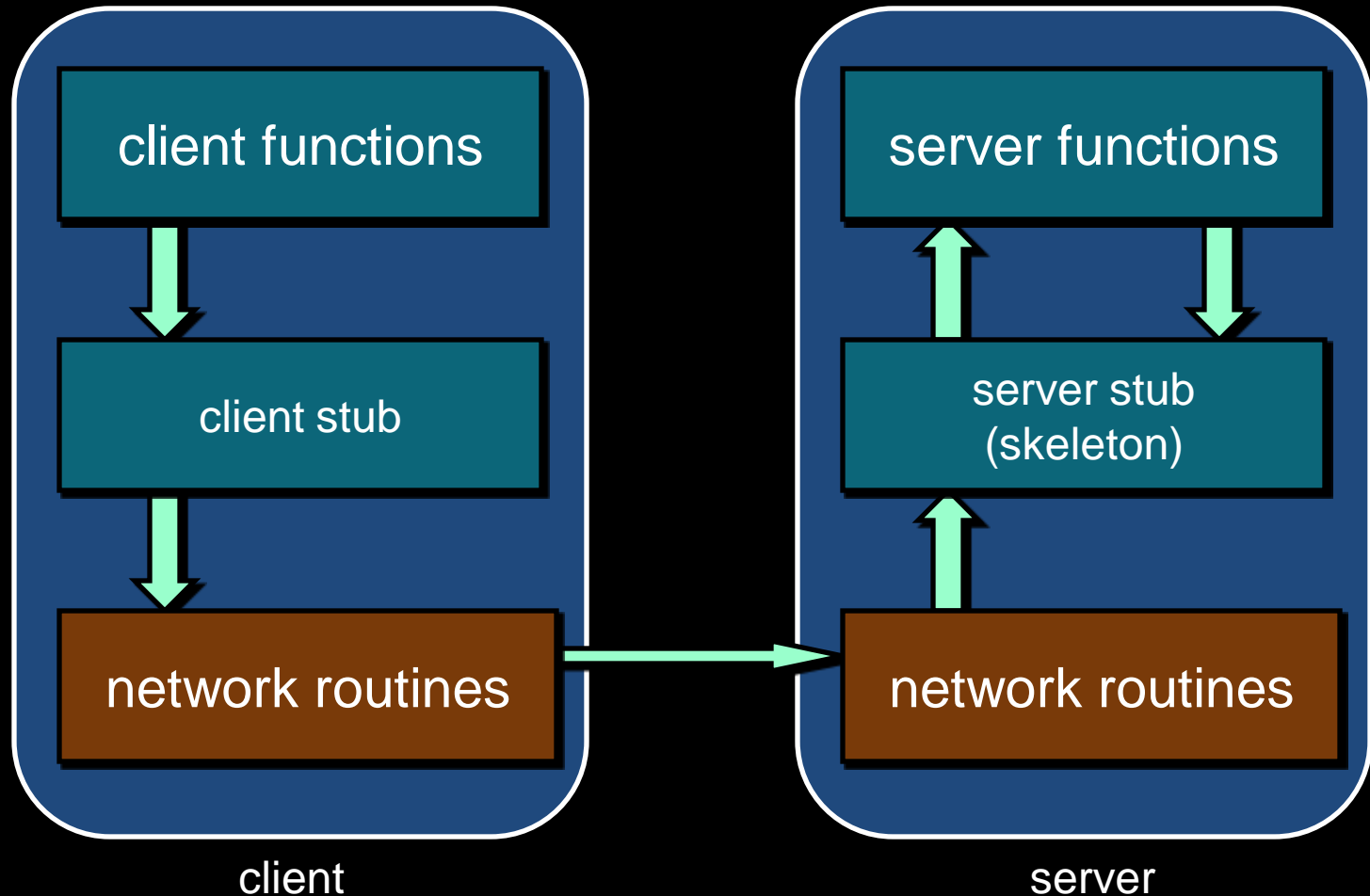
# Stub functions

## 5. Unmarshal parameters, call server function



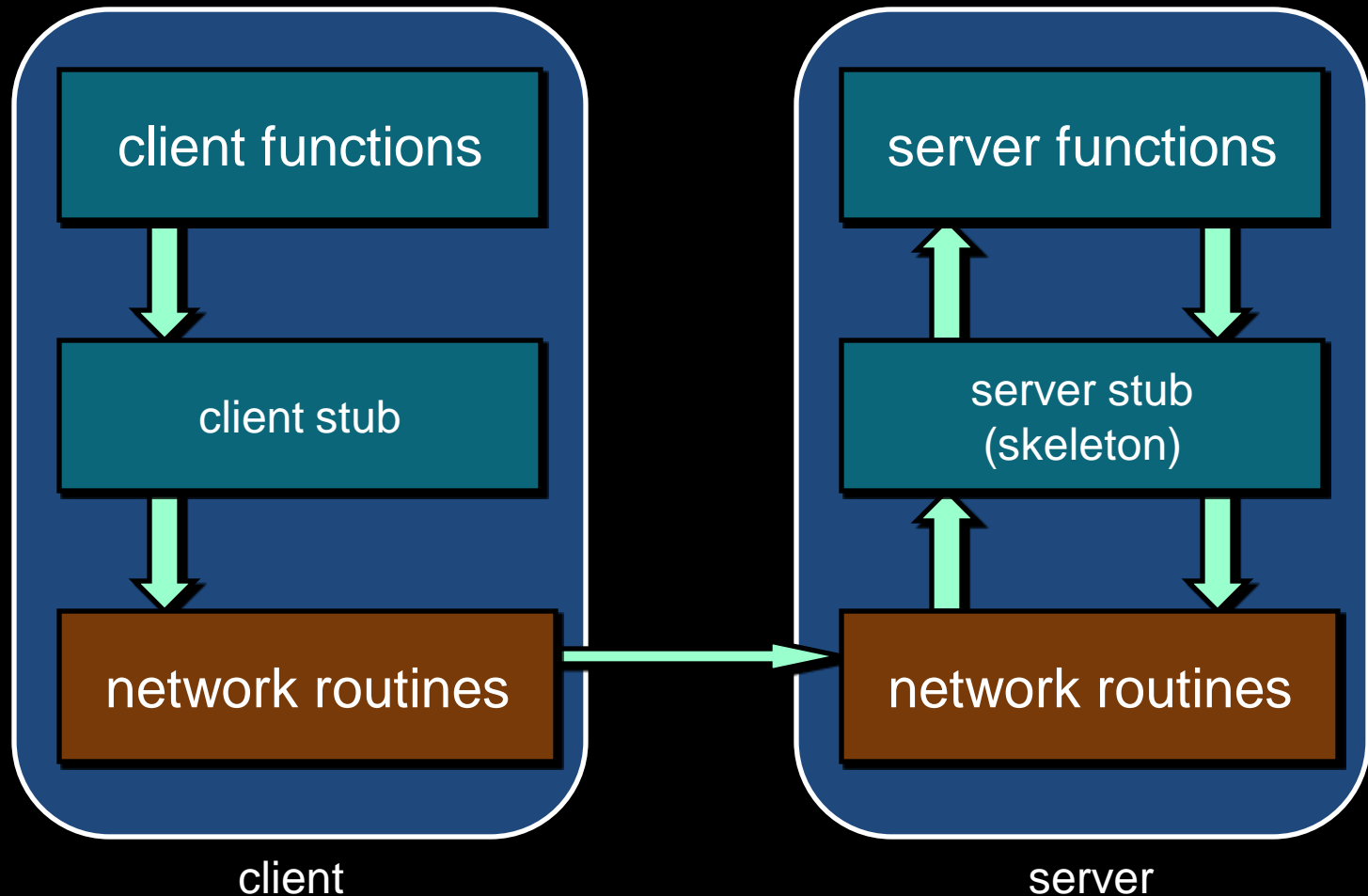
# Stub functions

## 6. Return from server function



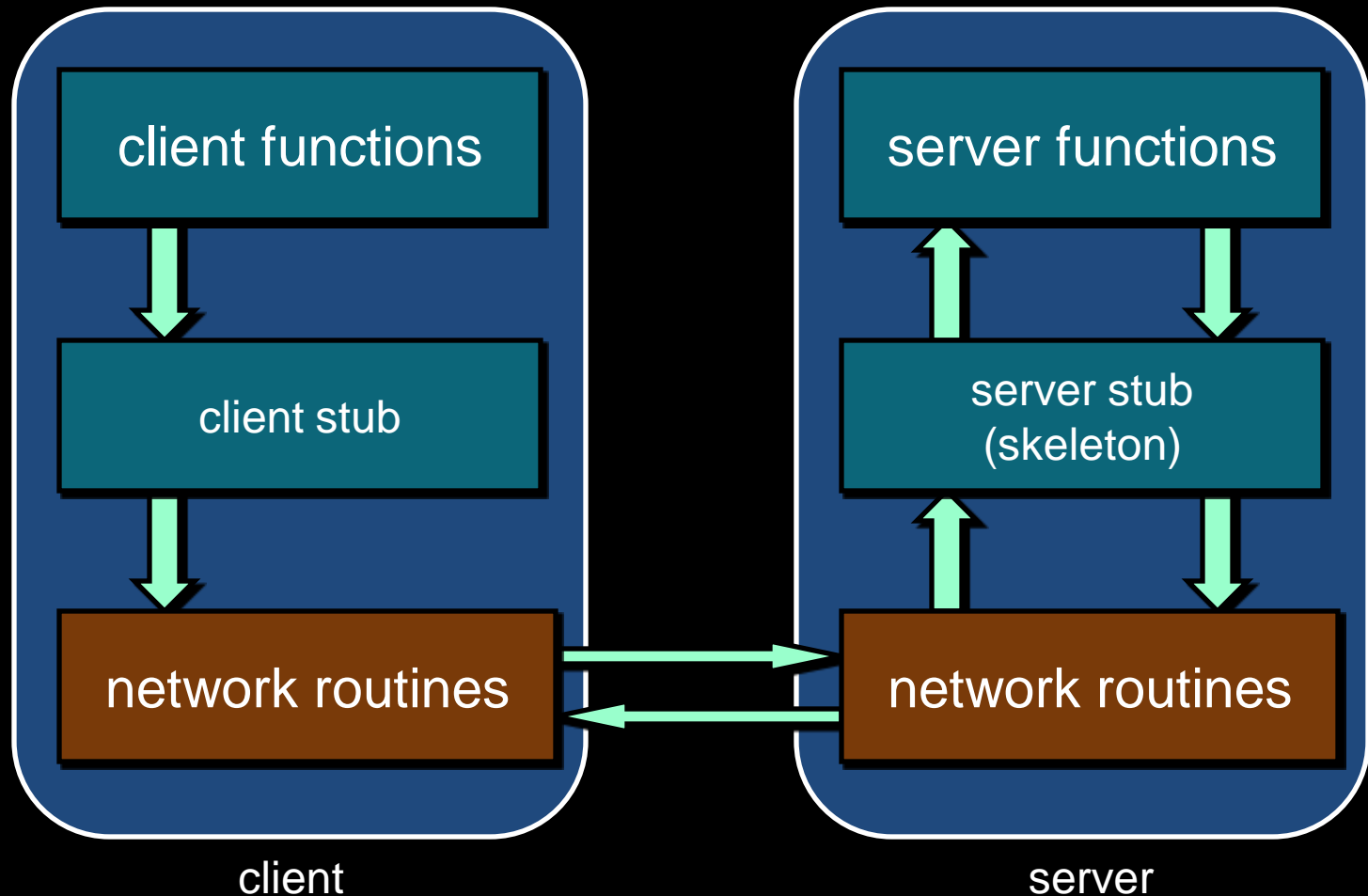
# Stub functions

## 7. Marshal return value and send message



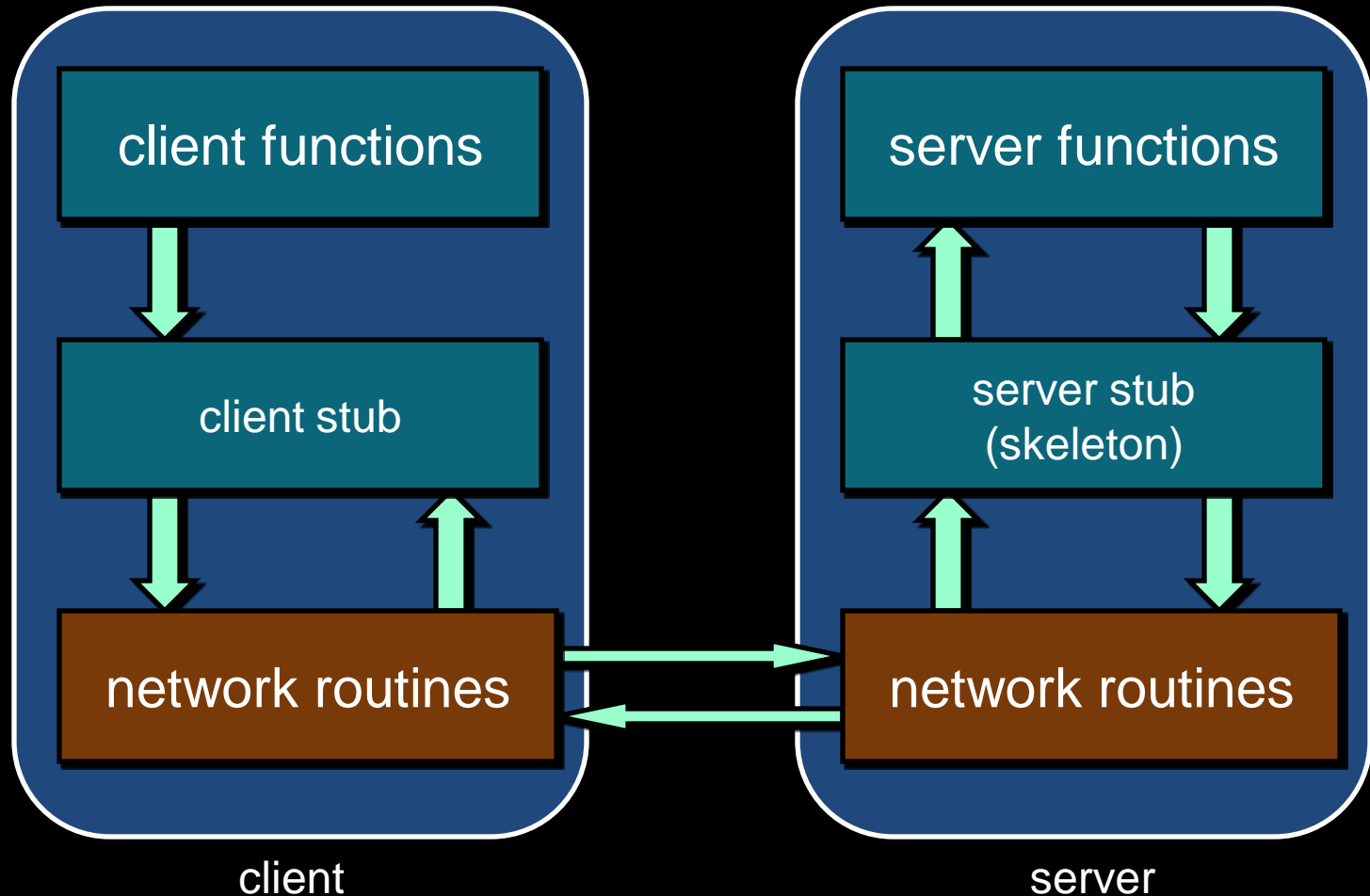
# Stub functions

## 8. Transfer message over network



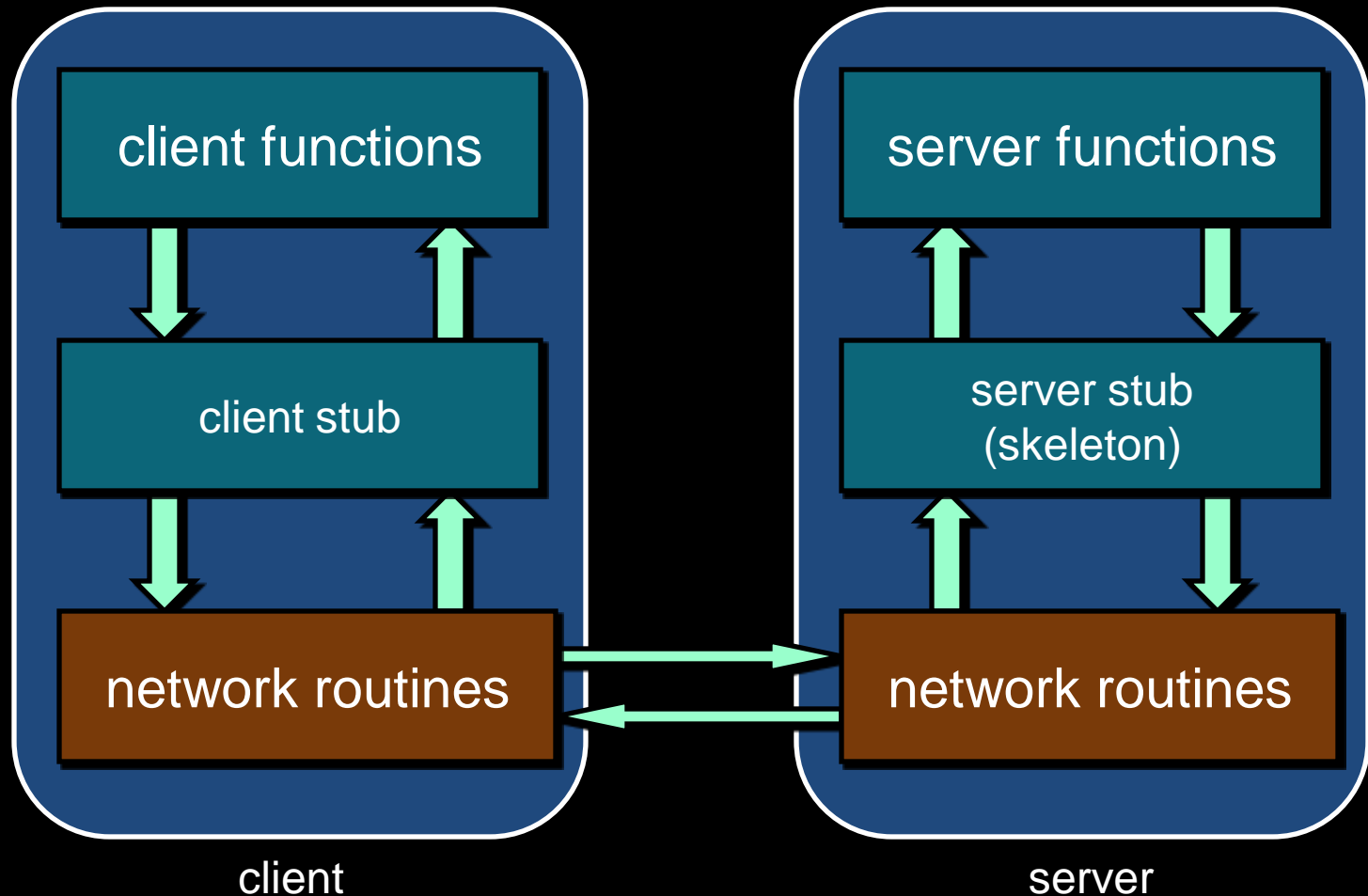
# Stub functions

## 9. Receive message: client stub is receiver



# Stub functions

## 10. Unmarshal return value, return to client code



# Benefits

---

- 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 issues

# Parameter passing

---

## Pass by value

- Easy: just copy data to network message

## Pass by reference

- Makes no sense without shared memory

# Pass by reference?

---

1. Copy items referenced to message buffer
2. Ship them over
3. Unmarshal data at server
4. Pass *local* pointer to server stub function
5. Send new values back

## To support complex structures

- Copy structure into pointerless representation
- Transmit
- Reconstruct structure with local pointers on server

# Representing data

---

No such thing as

***incompatibility problems*** on local system

Remote machine may have:

- Different byte ordering
- Different sizes of integers and other types
- Different floating point representations
- Different character sets
- Alignment requirements

# Representing data

---

IP (headers) forced all to use **big endian** byte ordering for 16- and 32-bit values

- Most significant byte in low memory
  - Sparc, 680x0, MIPS, PowerPC G5
  - Intel I-32 (x86/Pentium) use little endian

```
main() {
    unsigned int n;
    char *a = (char *)&n;

    n = 0x11223344;
    printf("%02x, %02x, %02x, %02x\n",
           a[0], a[1], a[2], a[3]);
}
```

Output on a Pentium:  
**44, 33, 22, 11**

Output on a PowerPC:  
**11, 22, 33, 44**

# Representing data

---

Need standard encoding to enable communication between heterogeneous systems

- e.g. Sun's RPC uses XDR (eXternal Data Representation)
- ASN.1 (ISO Abstract Syntax Notation)

# Representing data

---

## Implicit typing

- only values are transmitted, not data types or parameter info
- e.g., Sun XDR

## Explicit typing

- Type is transmitted with each value
- e.g., ISO's ASN.1, XML

# Where to bind?

---

Need to locate host and correct server process

# Where to bind? – Solution 1

---

Maintain centralized DB that can locate a host that provides a particular service  
*(Birrell & Nelson's 1984 proposal)*

# Where to bind? – Solution 2

---

A server on each host maintains a DB of *locally* provided services

Solution 1 is problematic for Sun NFS – identical file servers serve different file systems

# Transport protocol

---

TCP or UDP? Which one should we use?

- Some implementations may offer only one (e.g. TCP)
- Most support several
  - Allow programmer (or end user) to choose at runtime

# When things go wrong

---

- Local procedure calls do not fail
  - If they core dump, entire process dies
- More opportunities for error with RPC
- Transparency breaks here
  - Applications should be prepared to deal with RPC failure

# More issues

---

## Performance

- RPC is slower ... a lot slower

## Security

- messages visible over network
- Authenticate client
- Authenticate server

# Programming with RPC

---

## Language support

- Most programming languages (C, C++, Java, ...) have no concept of remote procedure calls
- Language compilers will not generate client and server stubs

### Common solution:

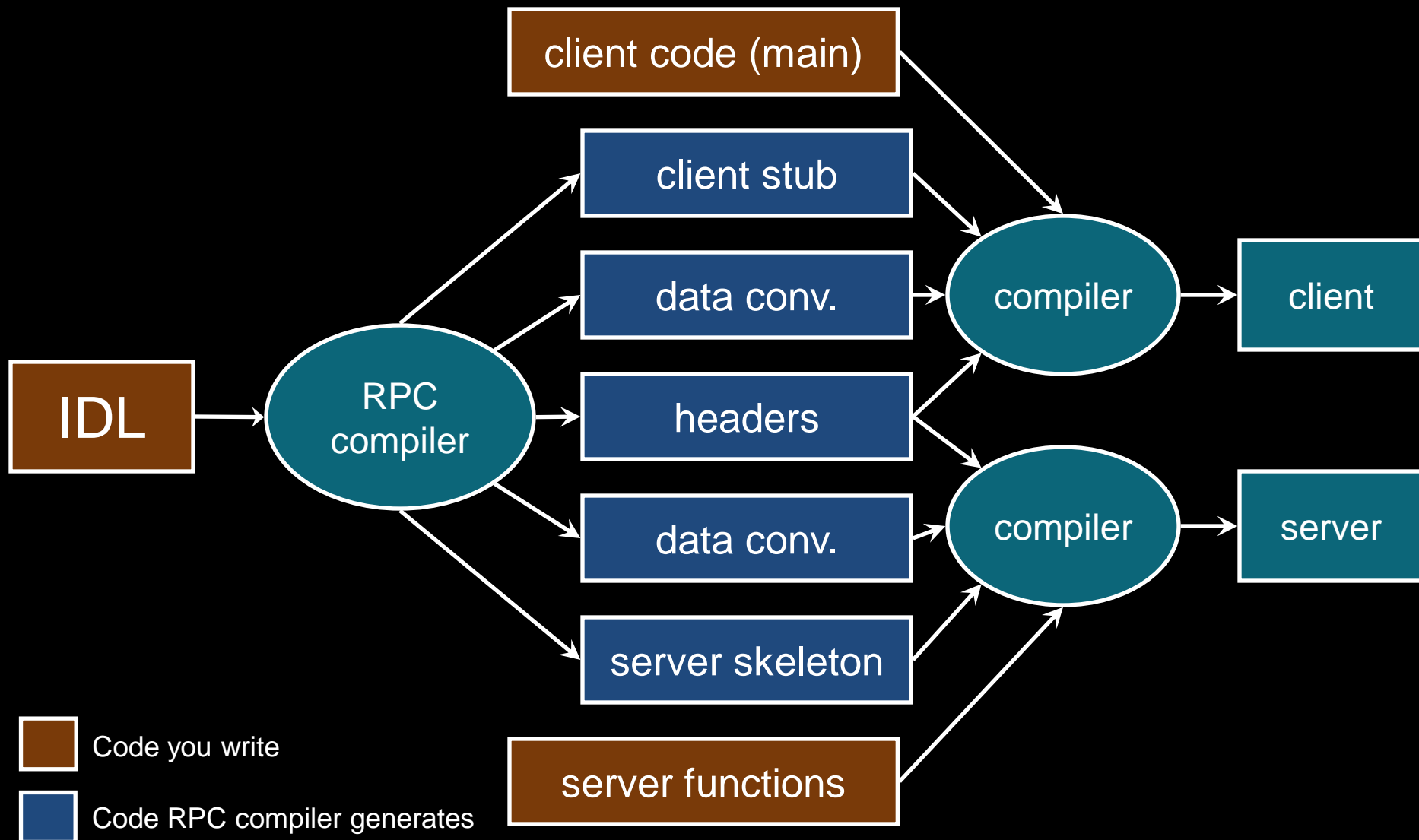
- Use a separate compiler to generate stubs (pre-compiler)

# Interface Definition Language

---

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

# RPC compiler



The End