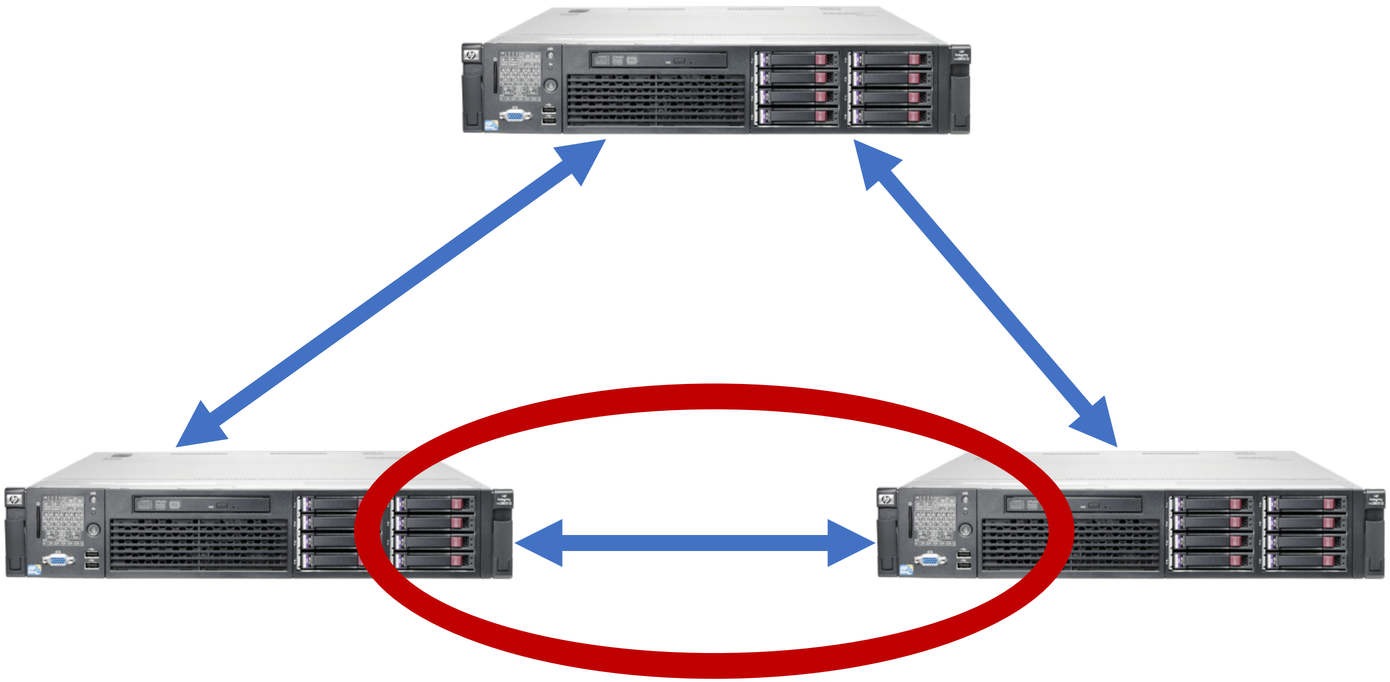
Network Communication and Remote Procedure Calls

جامعة الملك عبدالله
للعلوم والتقنية
King Abdullah University of
Science and Technology

CS 240: *Computing Systems and Concurrency*
Lecture 3

Marco Canini

Distributed Systems, What?



1) Multiple computers
2) Connected by a network
3) Doing something together

Today's outline

- *How can processes on different cooperating computers* ***communicate with each other over the network?***

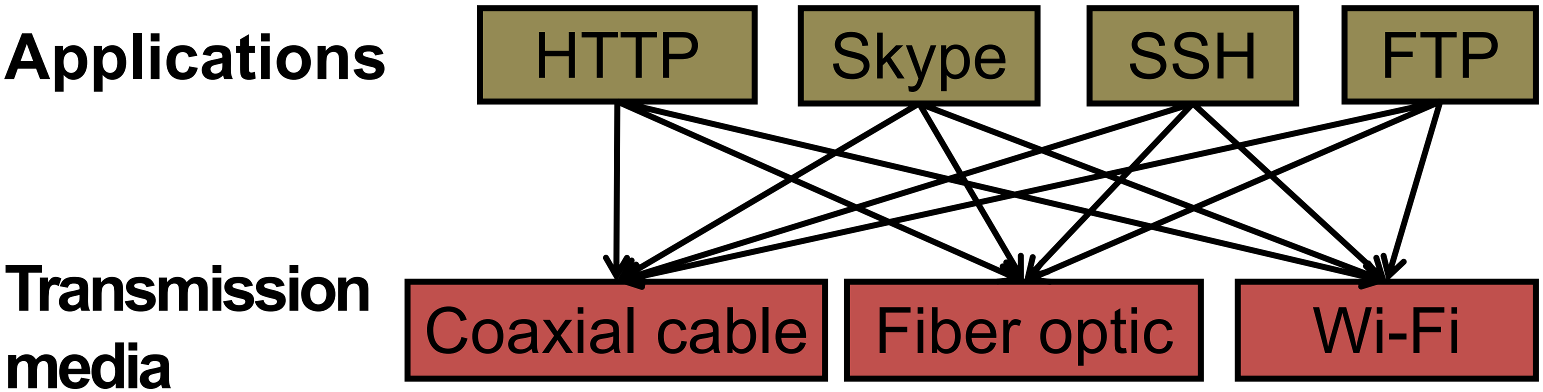
1. **Network Communication**

2. Remote Procedure Call (RPC)

The problem of communication

- Process on **Host A** wants to talk to process on **Host B**
  - A and B must agree on the **meaning** of the bits being sent and received **at many different levels**, including:
    - *How many volts is a 0 bit, a 1 bit?*
    - *How does receiver know which is the last bit?*
    - *How many bits long is a number?*

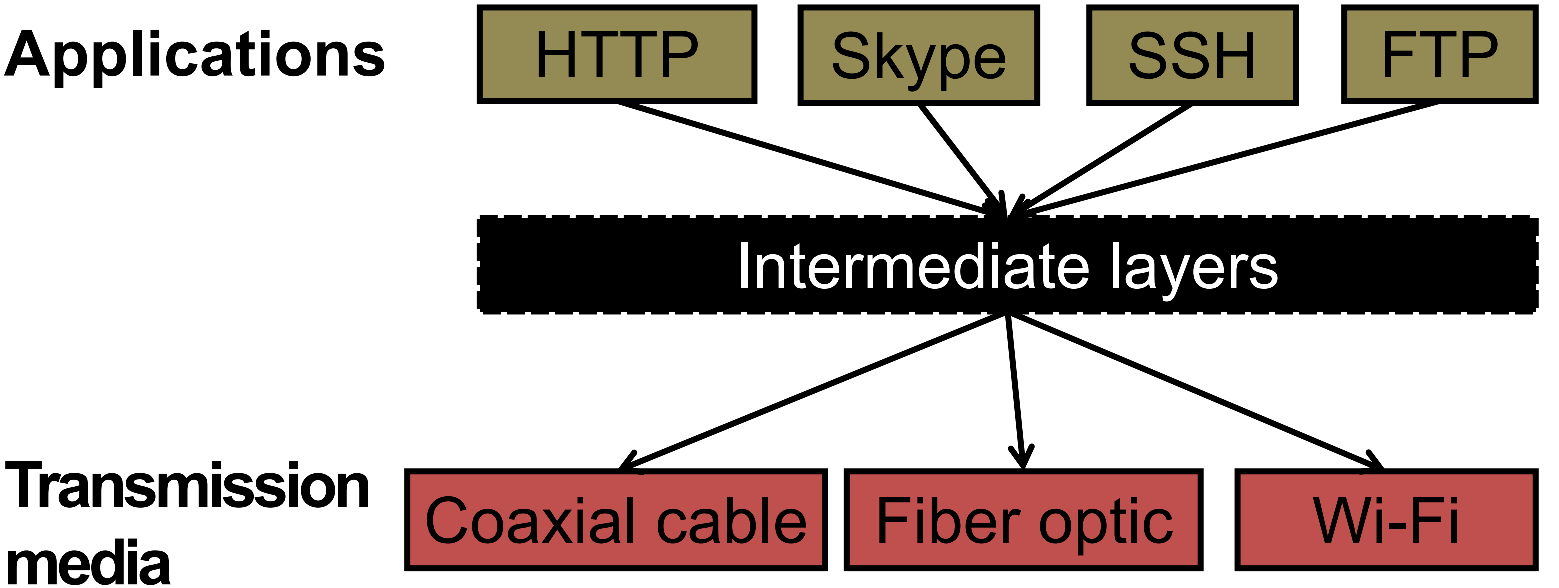
The problem of communication



- **Re-implement every application** for every new underlying transmission medium?
- **Change every application** on any change to an underlying transmission medium?

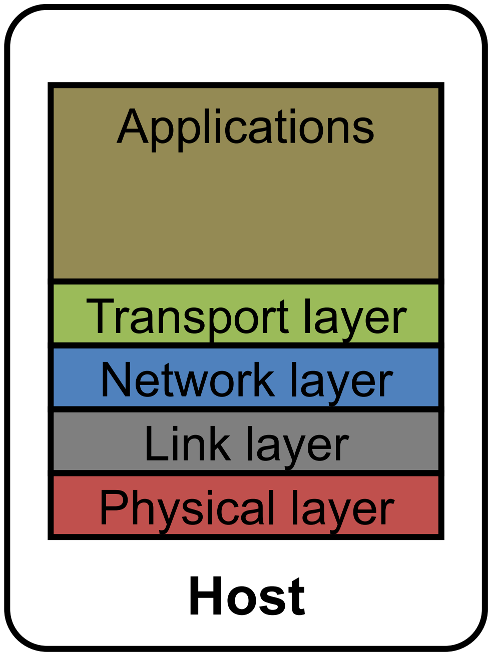
- **No!** But how does the Internet design avoid this?

Solution: Layering



- Intermediate ***layers*** provide a set of abstractions for applications and media
- New applications or media need only implement for intermediate layer's interface

Layering in the Internet

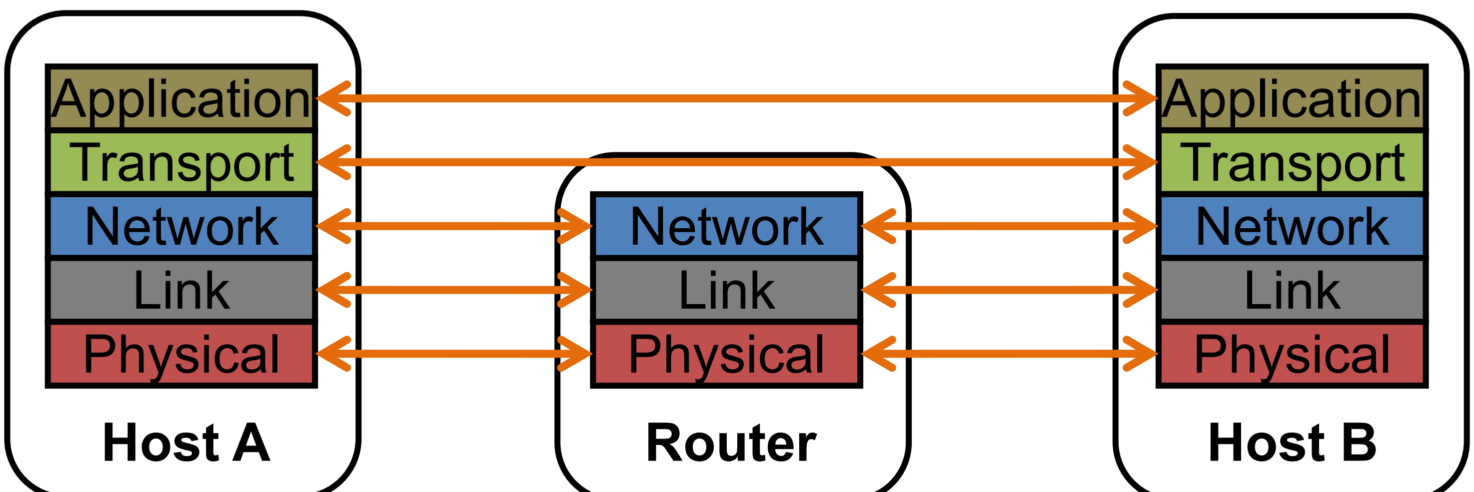


- **Transport:** Provide end-to-end communication between processes on different hosts
- **Network:** Deliver packets to destinations on other (heterogeneous) networks
- **Link:** Enables end hosts to exchange atomic messages with each other
- **Physical:** Moves bits between two hosts connected by a physical link

Logical communication between layers

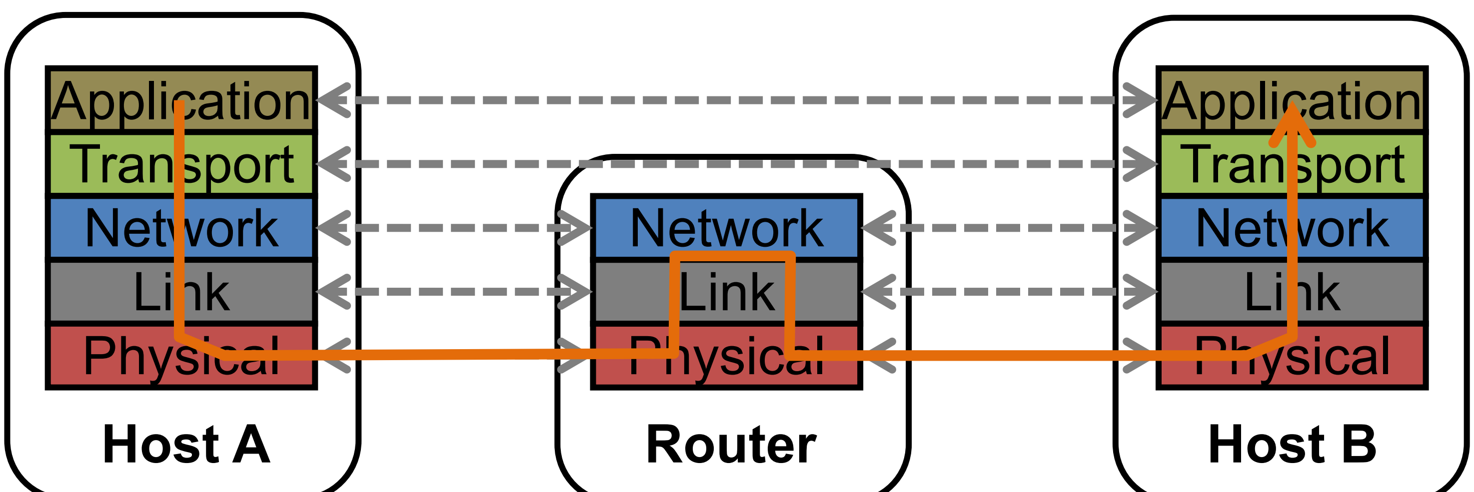
- *How to **forge agreement** on the **meaning** of the bits exchanged between two hosts?*

- ***Protocol:*** Rules that governs the format, contents, and meaning of messages
  - Each layer on a host interacts with its **peer** host's corresponding layer via the ***protocol interface***



Physical communication

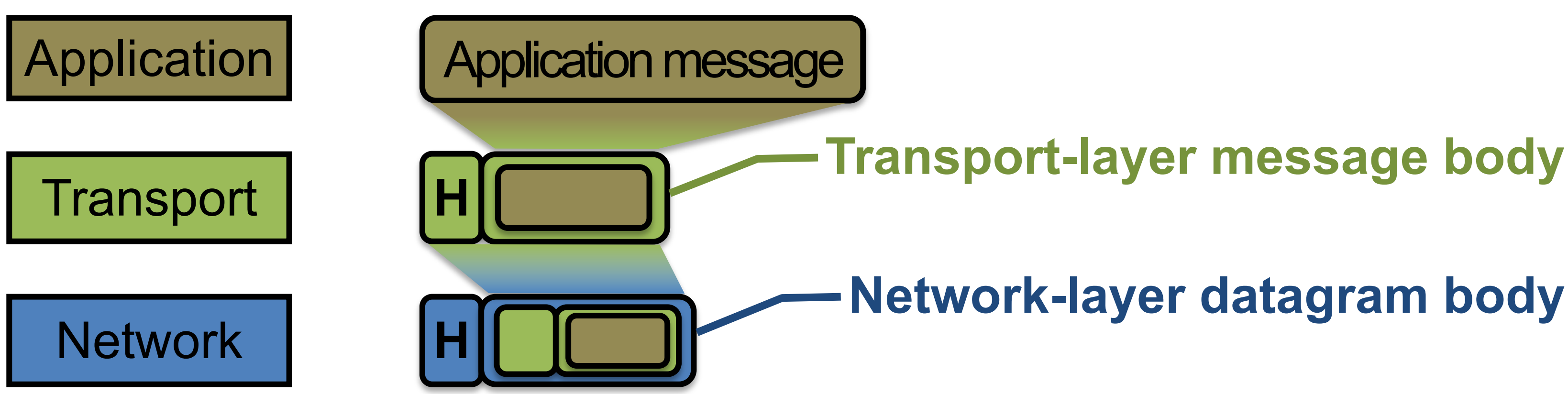
- Communication goes down to the **physical network**
- Then from **network** peer to peer
- Then up to the **relevant application**



Communication between peers

- *How do peer protocols coordinate with each other?*

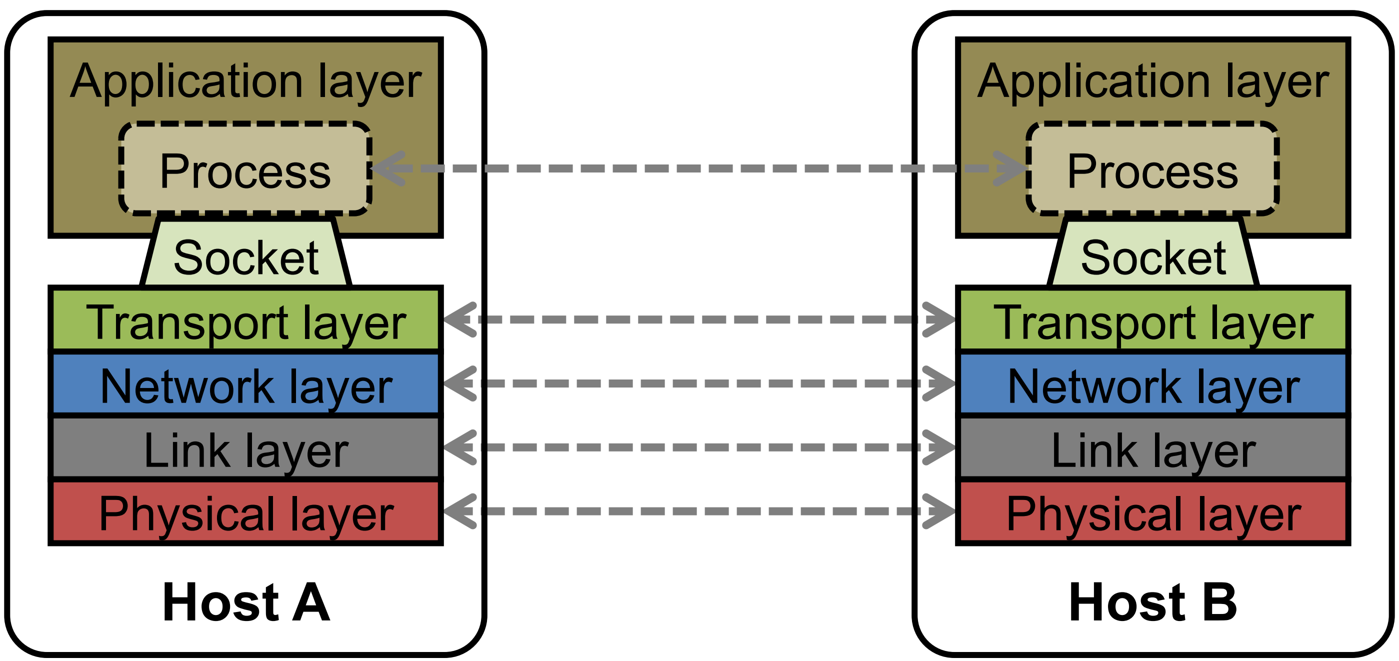
- Layer attaches its own ***header*** (**H**) to communicate with peer
  - Higher layers' headers, data ***encapsulated*** inside message
    - Lower layers don't generally inspect higher layers' headers



Network socket-based communication

- ***Socket:*** The interface the OS provides to the network
  – Provides inter-process **explicit message exchange**

- Can build distributed systems atop sockets: send(), recv()
  – *e.g.*: **`put(key,value)`** → message



```
// Create a socket for the client
if ((sockfd = socket (AF_INET, SOCK_STREAM, 0)) < 0) {
  perror("Socket creation");
  exit(2);
}

// Set server address and port
memset(&servaddr, 0, sizeof(servaddr));
servaddr.sin_family = AF_INET;
servaddr.sin_addr.s_addr = inet_addr(argv[1]);
servaddr.sin_port = htons(SERV_PORT); // to big-endian

// Establish TCP connection
if (connect(sockfd, (struct sockaddr *) &servaddr,
            sizeof(servaddr)) < 0) {
  perror("Connect to server");
  exit(3);
}

// Transmit the data over the TCP connection
send(sockfd, buf, strlen(buf), 0);
```

Network sockets: not great

- **Principle of transparency:** Hide that resource is physically distributed across multiple computers
  - Access resource same way as locally
  - Users can't tell where resource is physically located

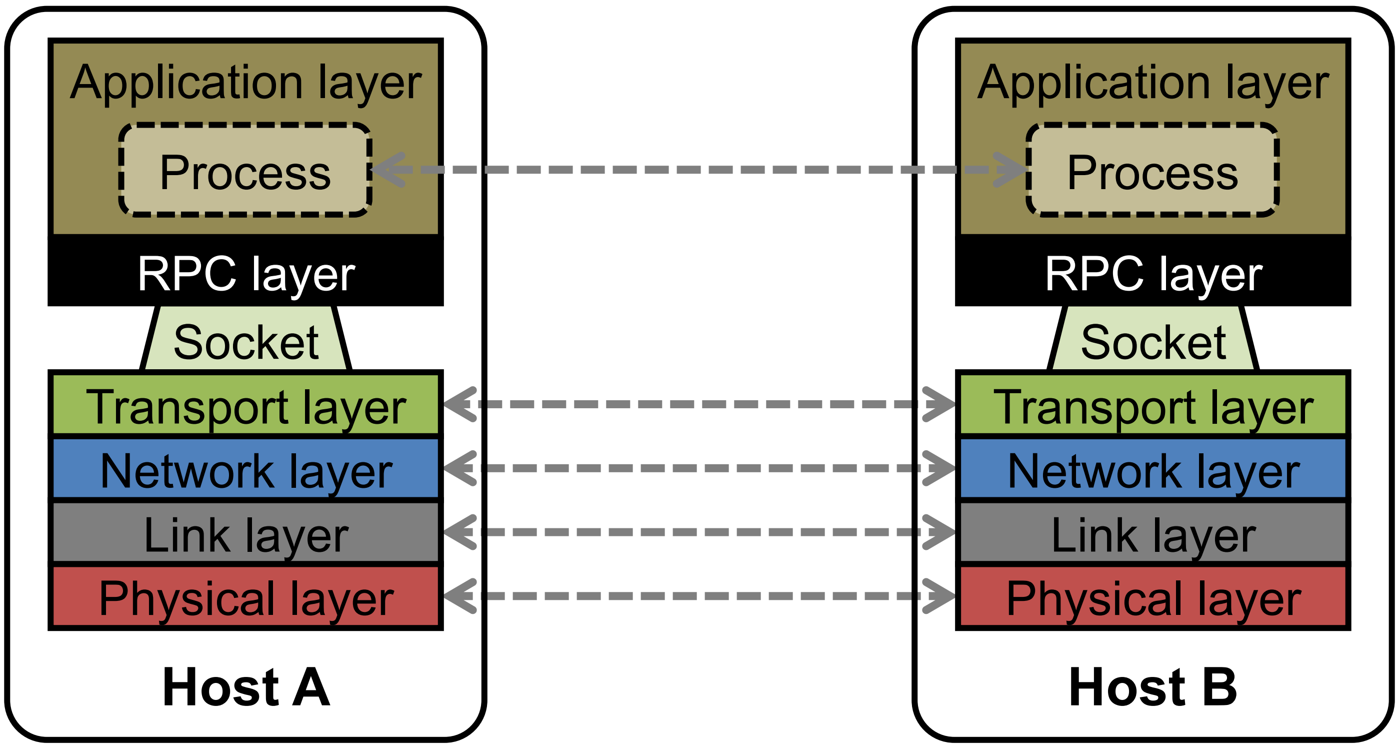
Network sockets provide apps with **point-to-point communication** between processes

Sockets don't provide transparency

Also, lots to deal with, have to worry a lot about the network

- How to separate different requests on the same connection?
- How to write bytes to the network / read bytes from the network?
  - What if Host A's process is in Go and Host B's process is in C++?
- What to do with those bytes?

Solution: Another layer!



Today’s outline

1. Network Communication
2. **Remote Procedure Call (RPC)**

Why RPC?

- The typical programmer is trained to write single-threaded code that runs in **one place**

- **Goal:** Easy-to-program network communication that makes client-server communication **transparent**
  - Retains the “feel” of writing centralized code
    - Programmer needn’t think about the network

Everyone uses RPCs

- Course programming assignments use RPC

- Google gRPC
- Facebook/Apache Thrift
- Twitter Finagle
- …

What's the goal of RPC?

- Within a single program, running in a single process, recall the well-known notion of a **procedure call**:
  - ***Caller*** pushes arguments onto stack,
    - jumps to address of ***callee*** function
  - ***Callee*** reads arguments from stack,
    - executes, puts return value in register,
    - returns to next instruction in caller

RPC's Goal: To make communication appear like a **local** procedure call: **transparency** for procedure calls

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

RPC issues

1. **Heterogeneity**
   – Client needs to **rendezvous** with the server
   – Server must **dispatch** to the required function
     • What if server is **different** type of machine?

2. **Failure**
   – What if messages get **dropped?**
   – What if client, server, or network **fails?**

3. **Performance**
   – Procedure call takes ≈ 10 cycles ≈ 3 ns
   – RPC in a data center takes ≈ 10 μs ($10^3 \times$ slower)
     • In the wide area, typically $10^6 \times$ slower

Problem: Differences in data representation

- Not an issue for **local** procedure call

- For a remote procedure call, a **remote machine may:**
  - Run process written in a **different language**
  - Represent data types using **different sizes**
  - Use a **different byte ordering** (*endianness*)
  - Represent floating point numbers **differently**
  - Have **different data alignment** requirements
    - *e.g.*, 4-byte type begins only on 4-byte memory boundary

Solution: Interface Description Language

- Mechanism to pass procedure parameters and return values in a **machine-independent way**

- Programmer may write an ***interface description*** in the IDL
  - Defines API for procedure calls: names, parameter/return types

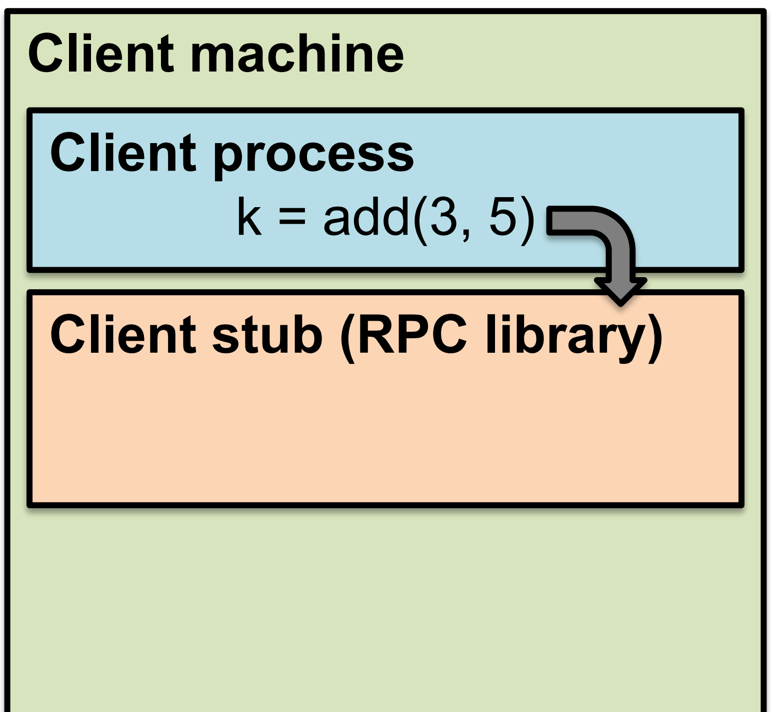
- Then runs an ***IDL compiler*** which generates:
  - Code to ***marshal*** (convert) native data types into machine-independent byte streams
    - And vice-versa, called ***unmarshaling***

  - **Client stub:** Forwards local procedure call as a request to server

  - **Server stub:** Dispatches RPC to its implementation

A day in the life of an RPC

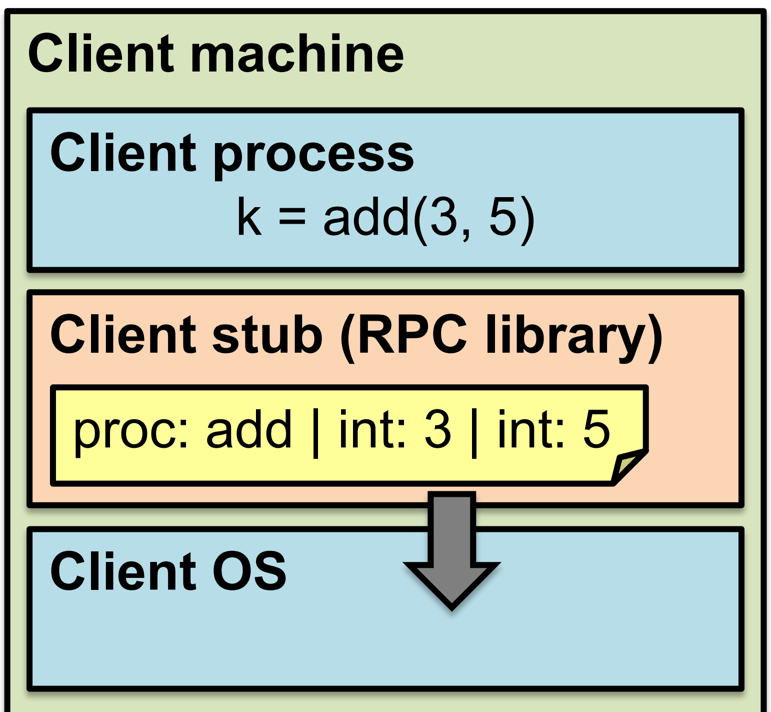
1. Client calls stub function (pushes params onto stack)



A day in the life of an RPC

1. Client calls stub function (pushes params onto stack)

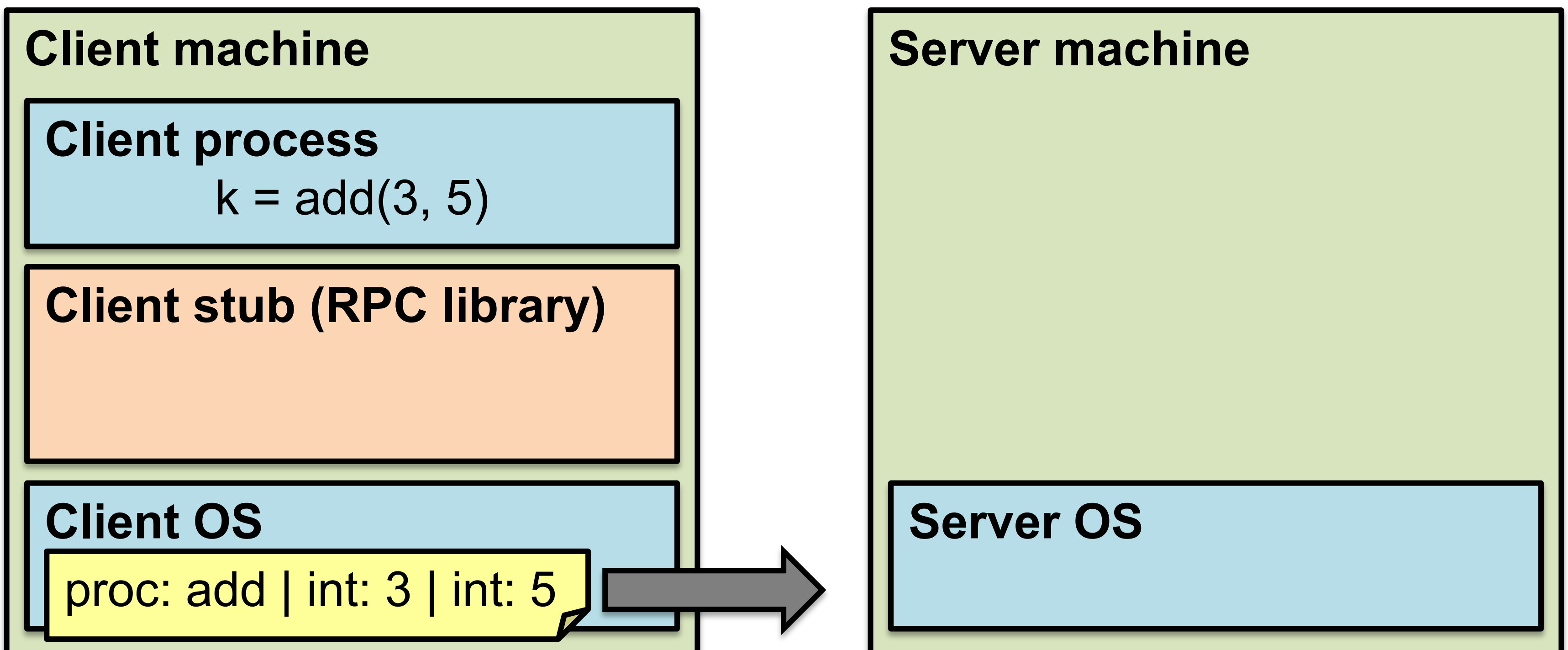
2. **Stub marshals parameters to a network message**



A day in the life of an RPC

2. Stub marshals parameters to a network message

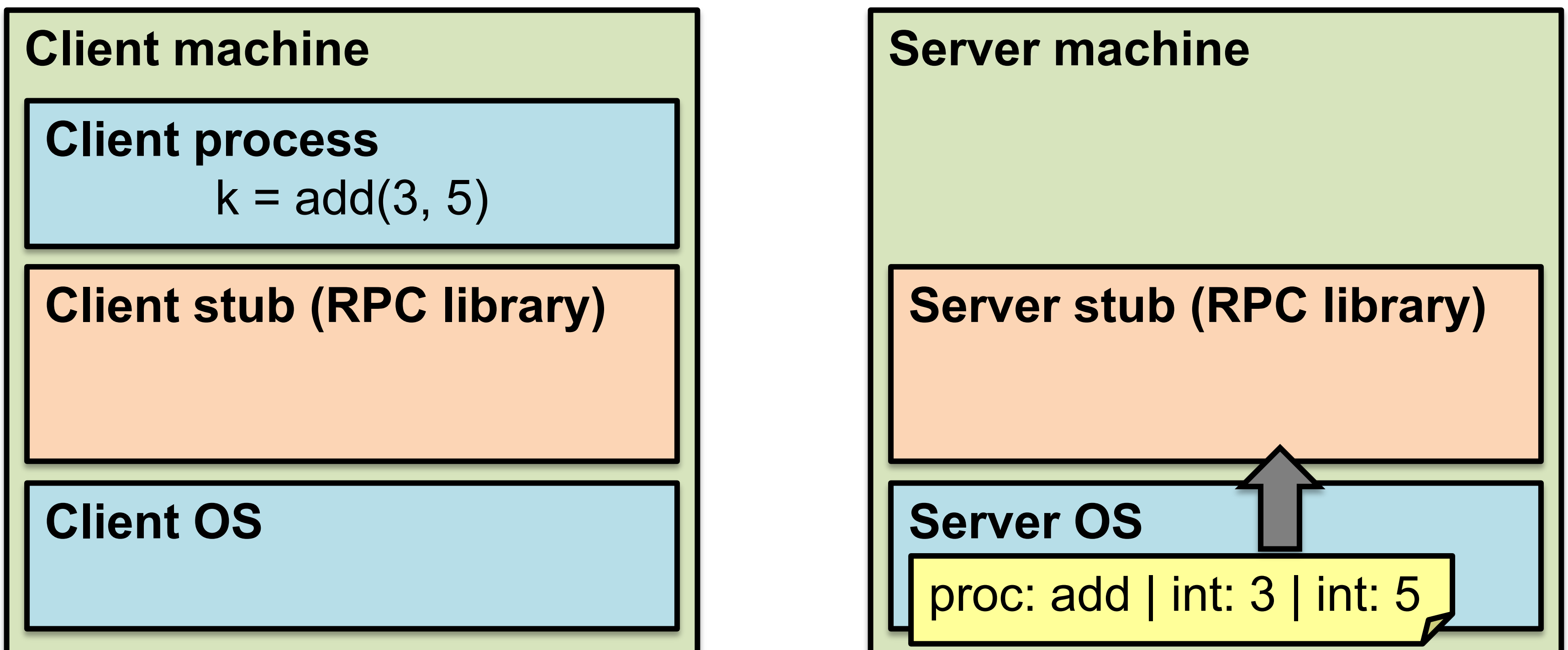
3. **OS sends a network message to the server**



A day in the life of an RPC

3. OS sends a network message to the server

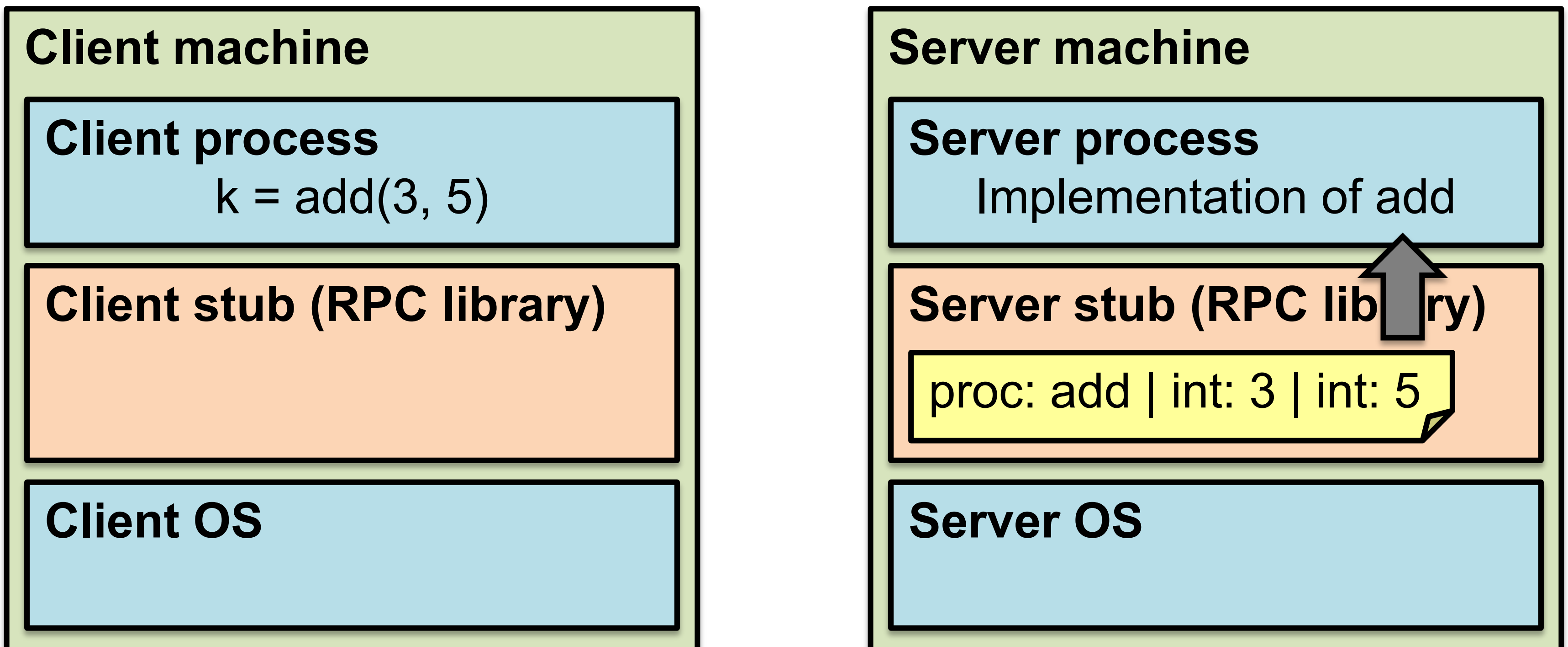
4. **Server OS receives message, sends it up to stub**



A day in the life of an RPC

4. Server OS receives message, sends it up to stub

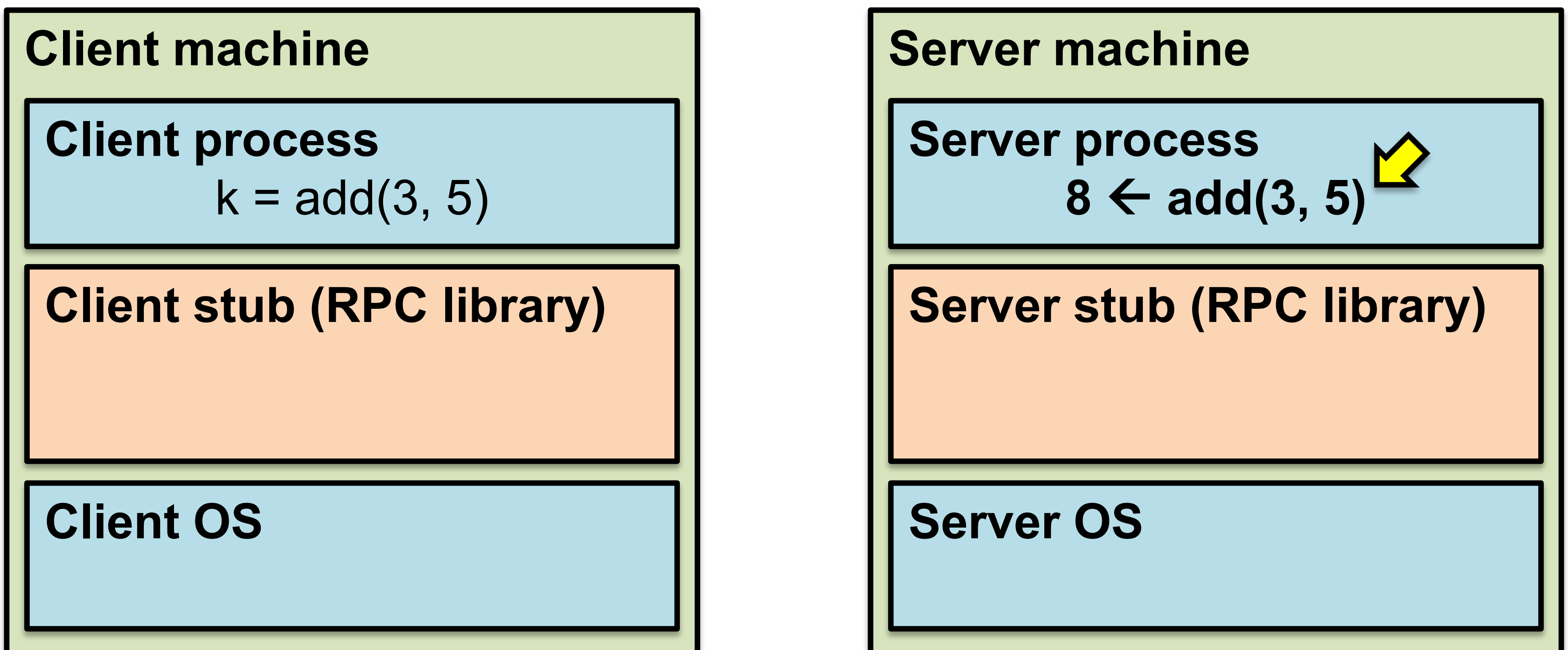
5. Server stub unmarshals params, calls server function



A day in the life of an RPC

5. Server stub unmarshals params, calls server function

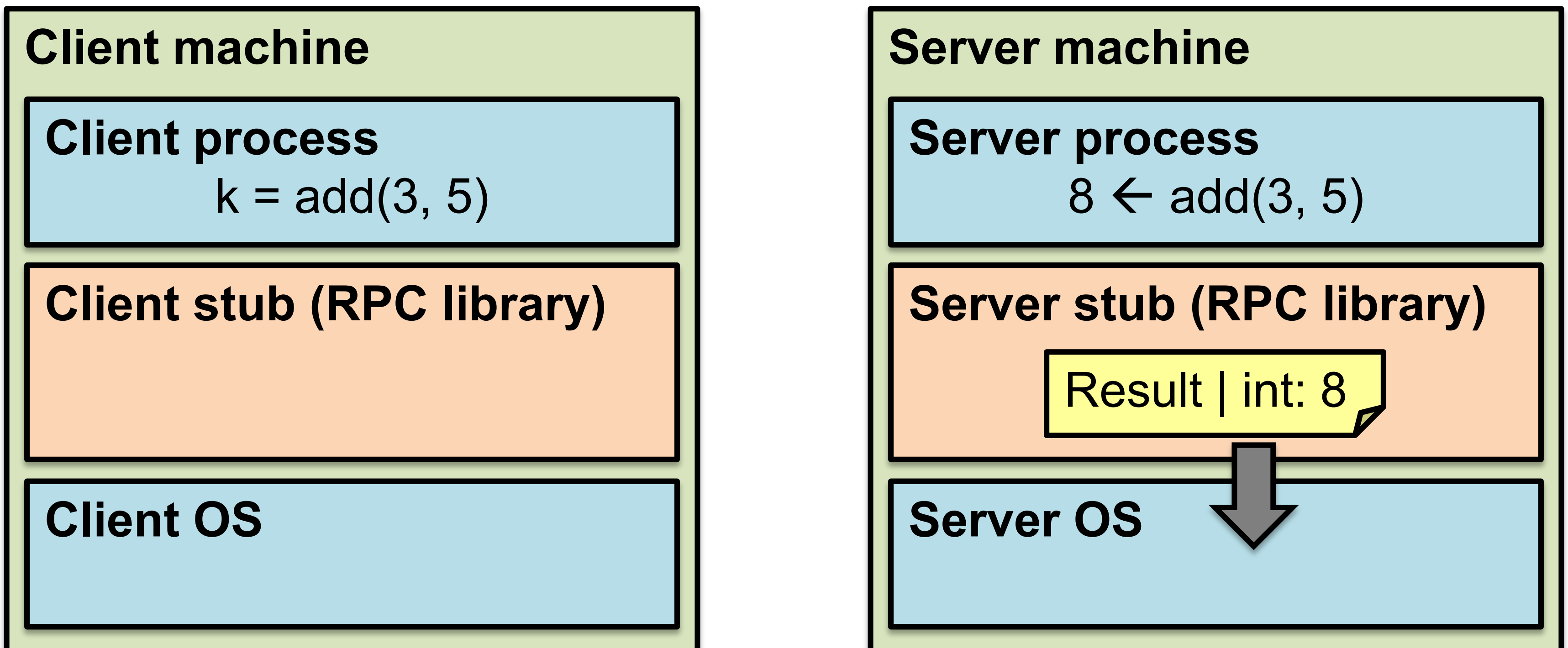
6. Server function runs, returns a value



A day in the life of an RPC

6. Server function runs, returns a value

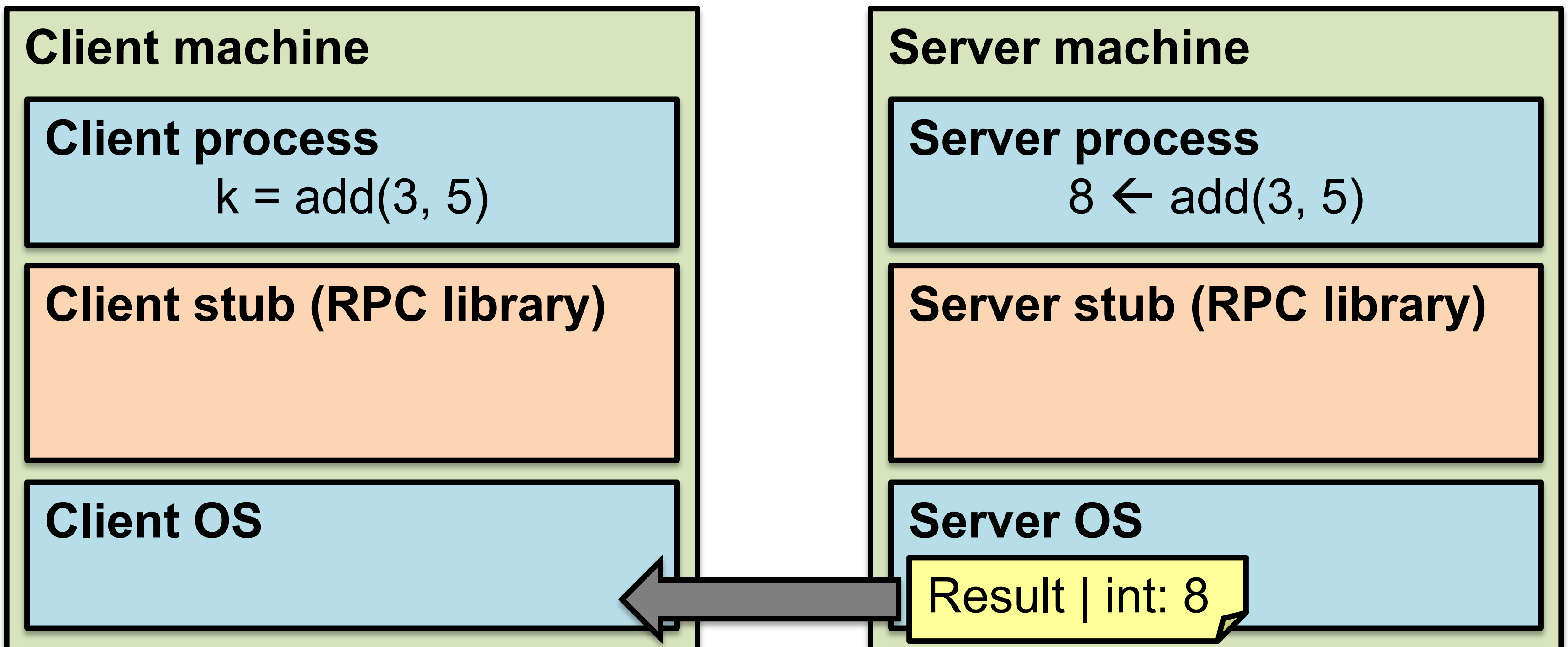
7. **Server stub marshals the return value, sends msg**



A day in the life of an RPC

7. Server stub marshals the return value, sends msg

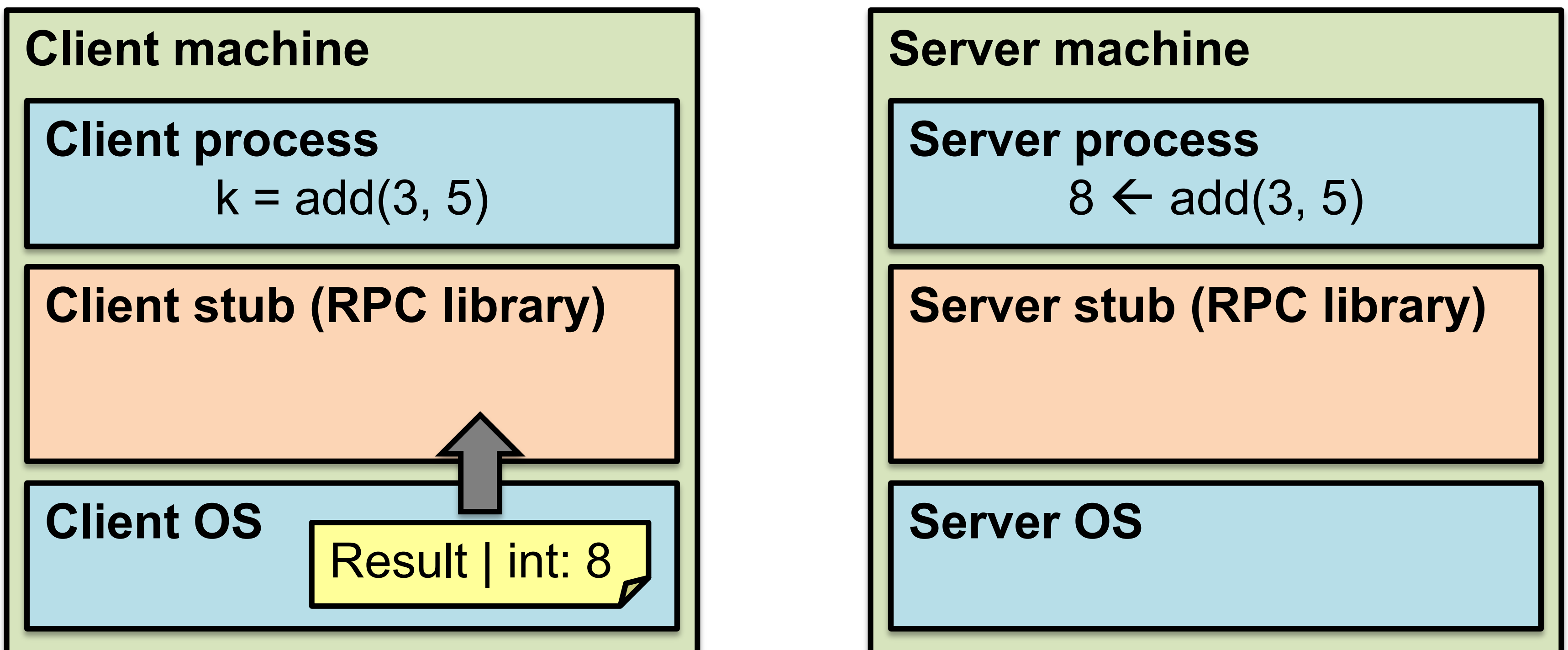
8. **Server OS sends the reply back across the network**



A day in the life of an RPC

8. Server OS sends the reply back across the network

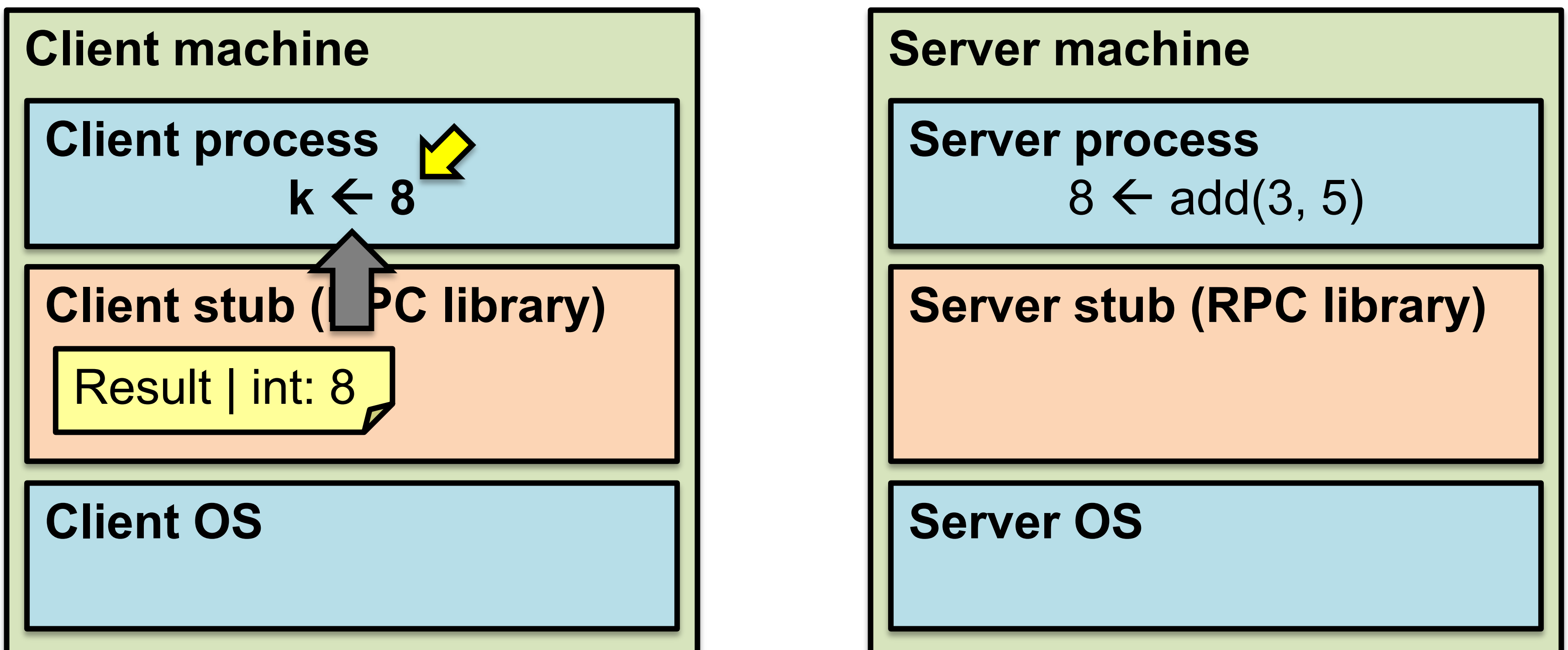
9. Client OS receives the reply and passes up to stub



A day in the life of an RPC

9. Client OS receives the reply and passes up to stub

10. Client stub unmarshals return value, returns to client



The server stub is really two parts

- ***Dispatcher***
  – Receives a client's RPC request
    - **Identifies** appropriate server-side method to invoke

- ***Skeleton***
  – **Unmarshals** parameters to server-native types
  – **Calls** the local server procedure
  – **Marshals** the response, sends it back to the dispatcher

- **All this is hidden from the programmer**
  – Dispatcher and skeleton may be integrated
    - Depends on implementation

Today's outline

1. Network Communication

2. **Remote Procedure Call (RPC)**
   – Heterogeneity – use IDL w/ compiler
   – **Failure**

What could *possibly* go wrong?

1. Client may **crash and reboot**

2. Packets may be **dropped**
   – Some individual **packet loss** in the Internet
   – **Broken routing** results in many lost packets

3. Server may **crash and reboot**

4. Network or server might just be **very slow**

All these may look the same to the client…

Summary: RPCs and Net. Comm.

- Layers are our friends!
- RPCs are everywhere
- **Necessary** issues surrounding machine heterogeneity
- **Subtle** issues around **failures**
  - … Next time!!!