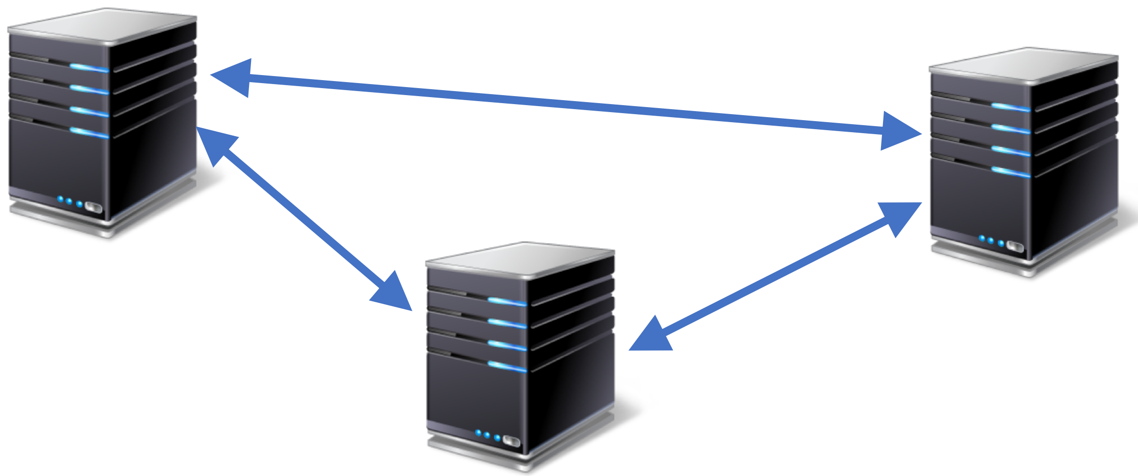
Remote Procedure Call (RPC)

CS 675: Distributed Systems (Spring 2020)

Lecture 2

Yue Cheng

Context



- Multiple computers
- Connected by a network
- Doing something together

- A *distributed system* is many cooperating computers that appear to users as a single service

Today's outline

- Today– *How can processes on different cooperating computers exchange information?*

1. Network sockets
2. Remote procedure call RPC
3. RPCs in Go

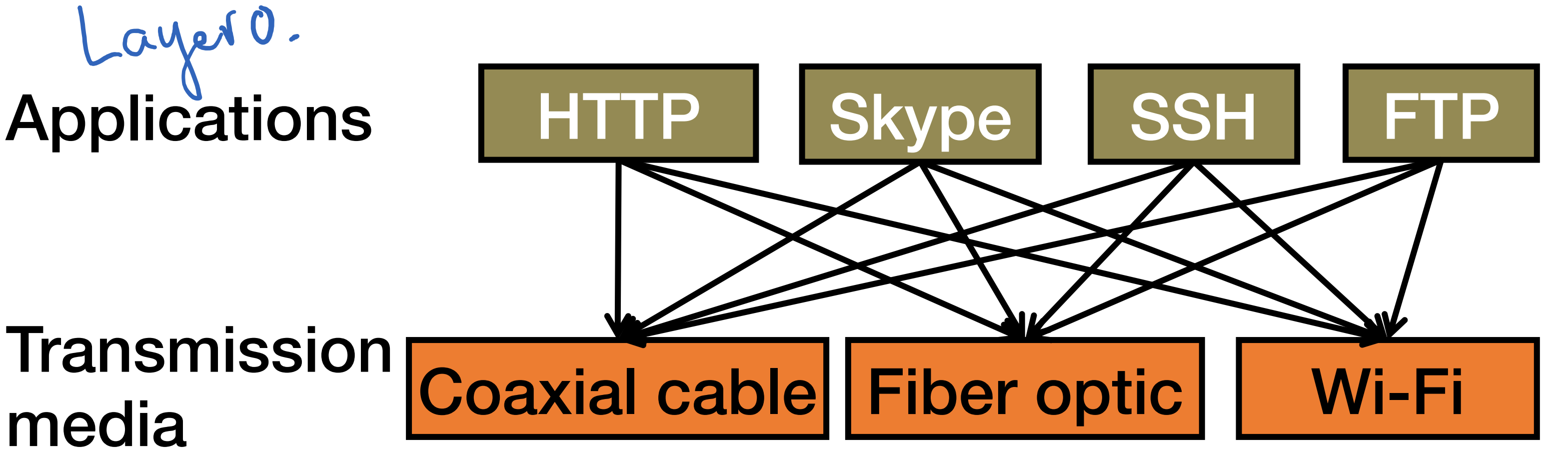
The problem of communication

- Process on Host A wants to talk to process on Host B

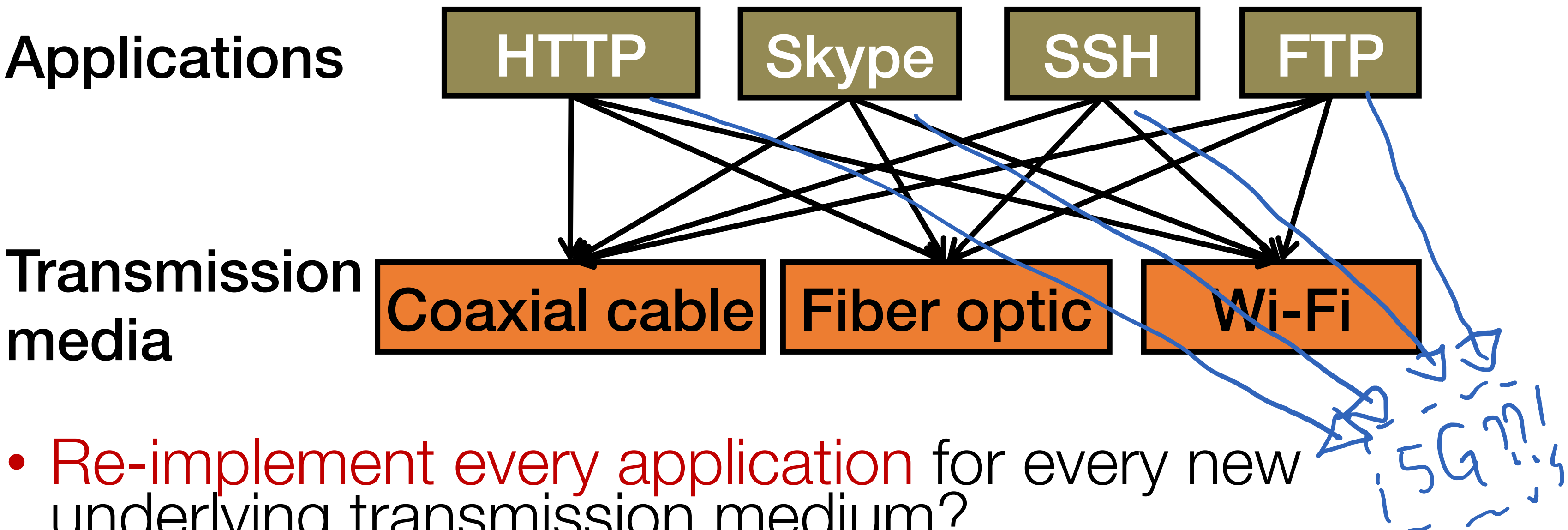
Spec. Protocol.

  - 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 bits?
    - How does receiver know which is the last bit?
    - How many bits long is a number?

The problem of communication

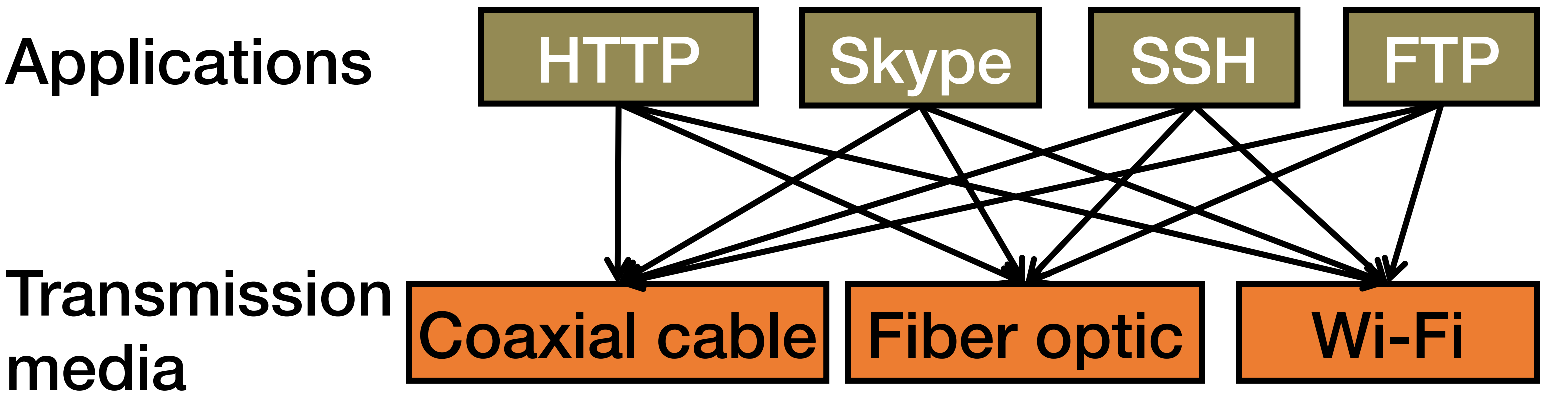


The problem of communication



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

The problem of communication

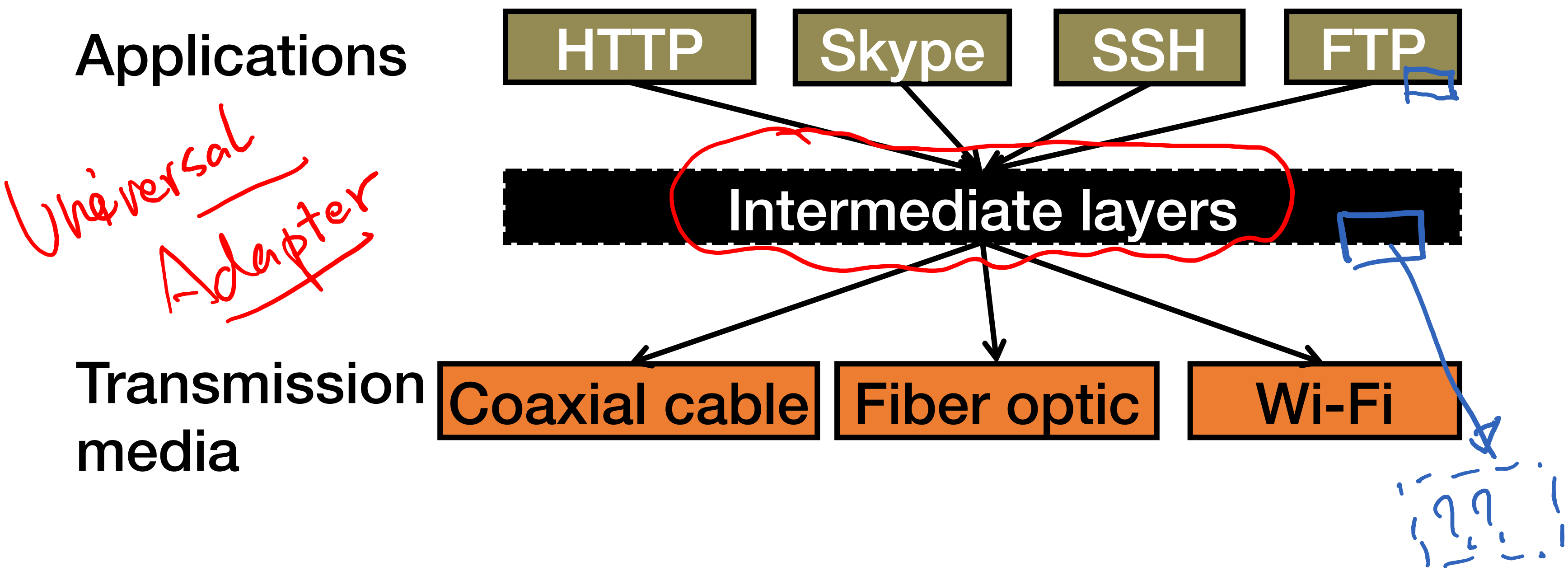


- Re-implement every application for every new underlying transmission medium?
- Change every application on any change to an underlying transmission medium?

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

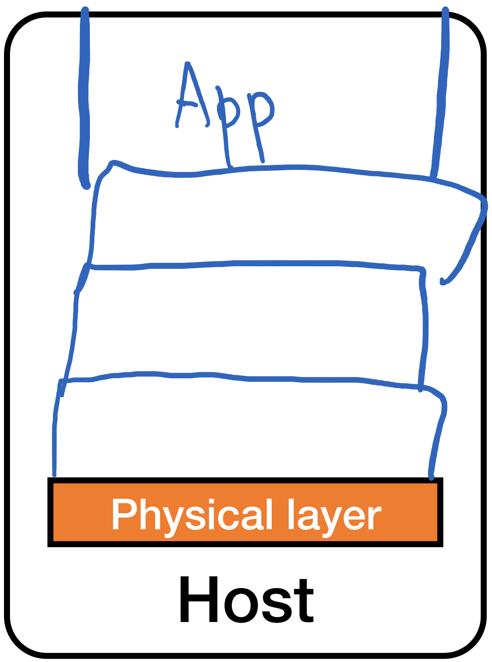
Solution: Layering

Modular Design.



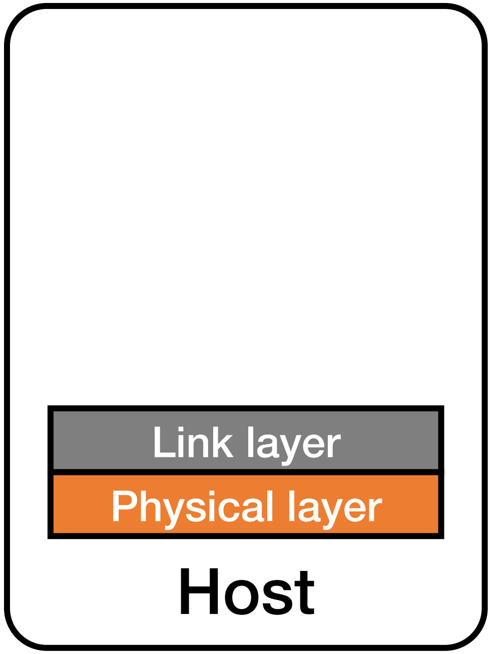
- Intermediate layers provide a set of abstractions for applications and media
- New applications or media need only implement for intermediate layer's interface

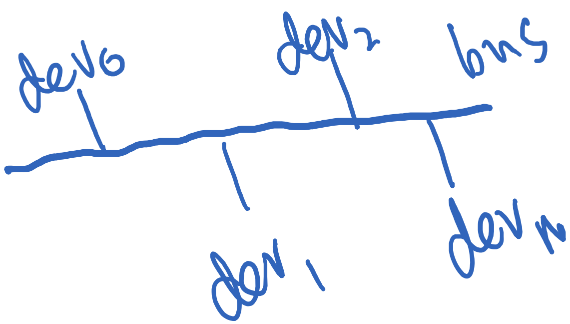
Layering in the Internet



- Physical: Moves bits between two hosts connected by a physical link

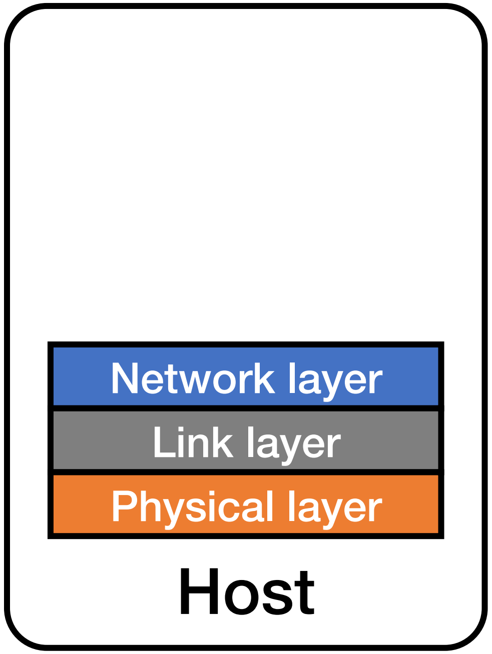
Layering in the Internet





- Link: Enables end hosts to exchange atomic messages with each other
- Physical: Moves bits between two hosts connected by a physical link

Layering in the Internet

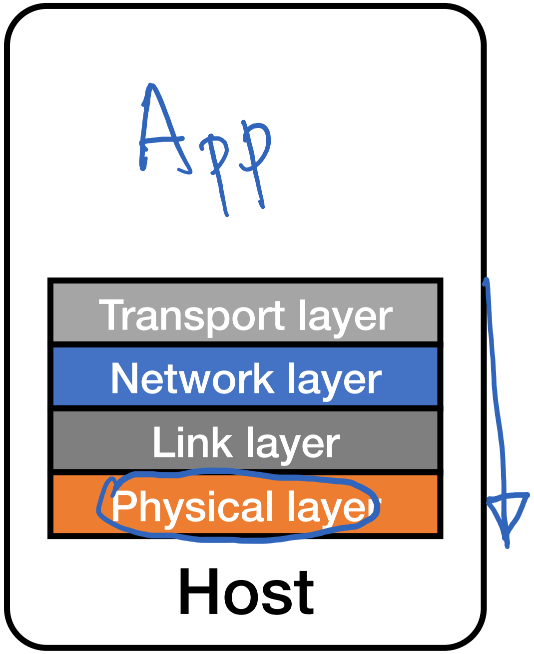


- Network: Deliver packets to destinations on other (heterogeneous) networks

- Link: Enables end hosts to exchange atomic messages with each other

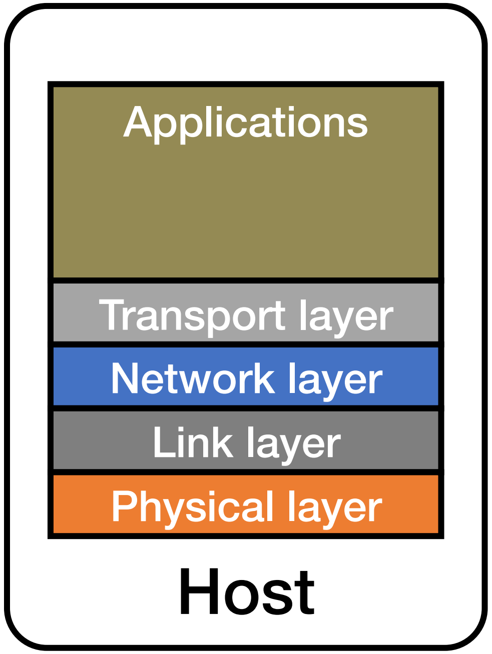
- Physical: Moves bits between two hosts connected by a physical link

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

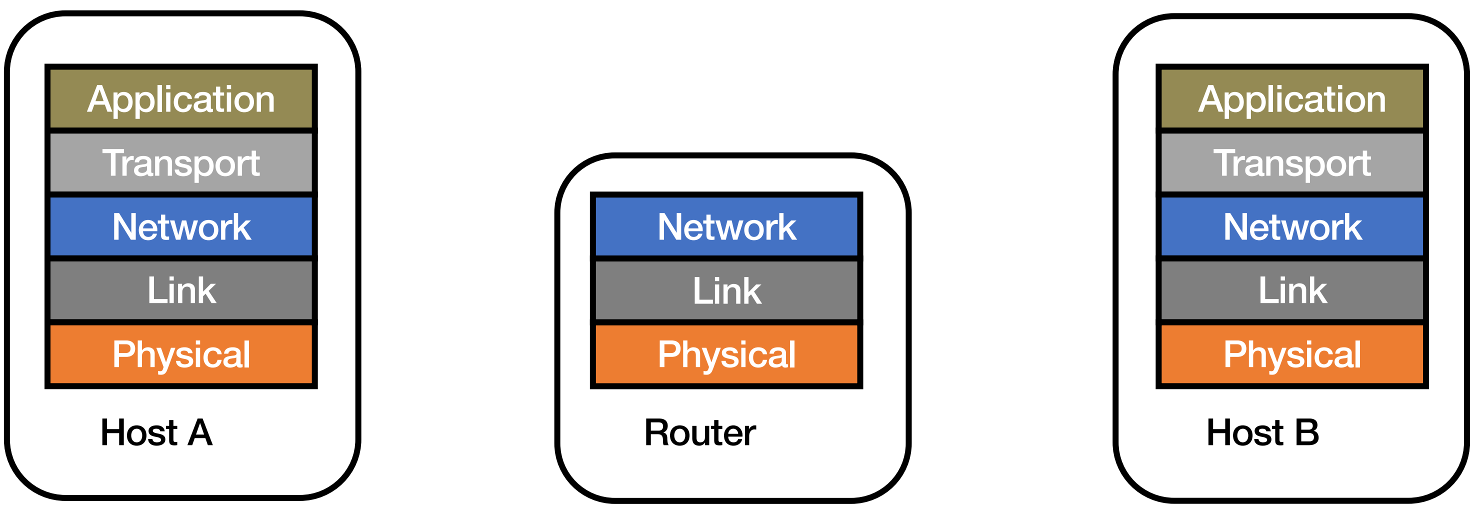
Layering in the Internet



- 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

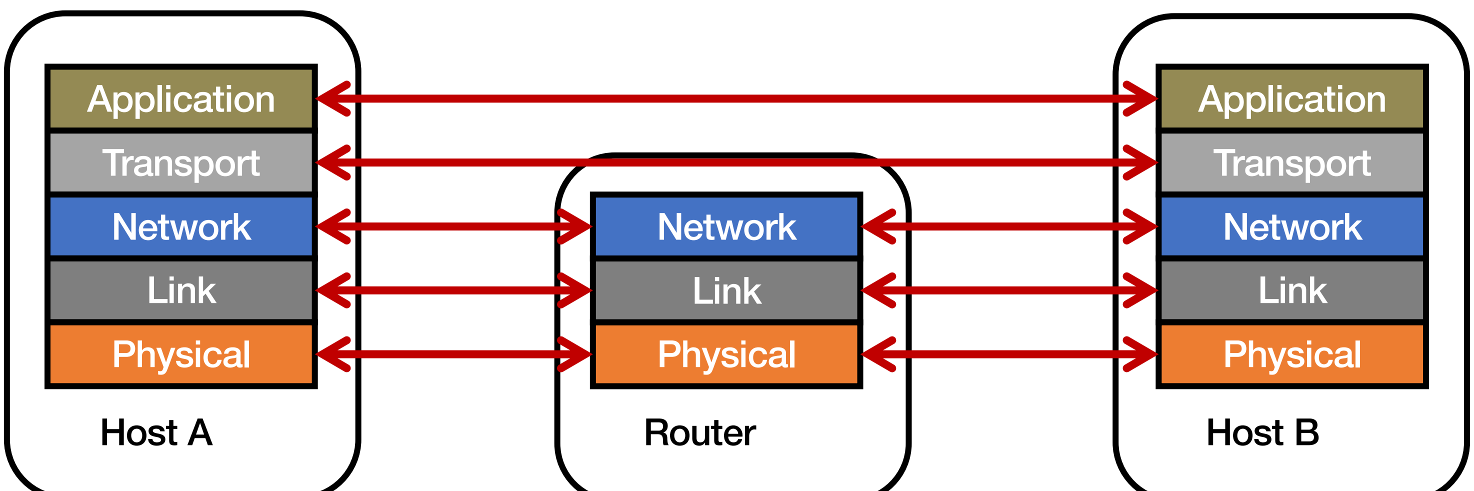
- How to forge agreement on the meaning of the bits exchanged between two hosts?



Logical communication between layers

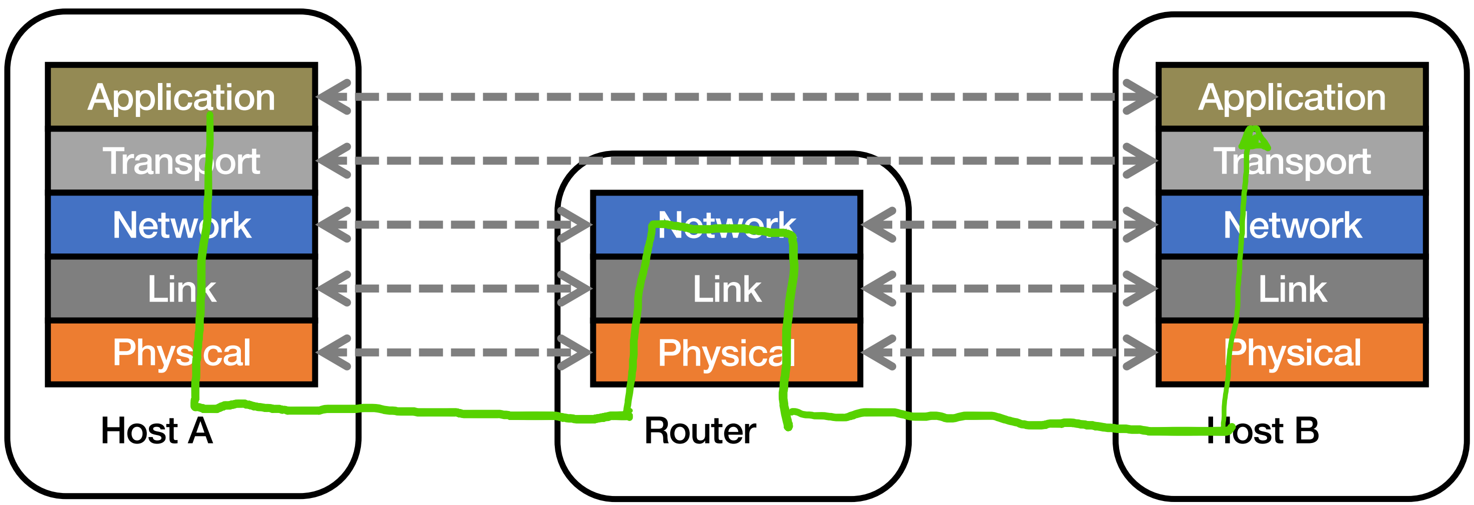
- How to forge agreement on the meaning of the bits exchanged between two hosts?

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



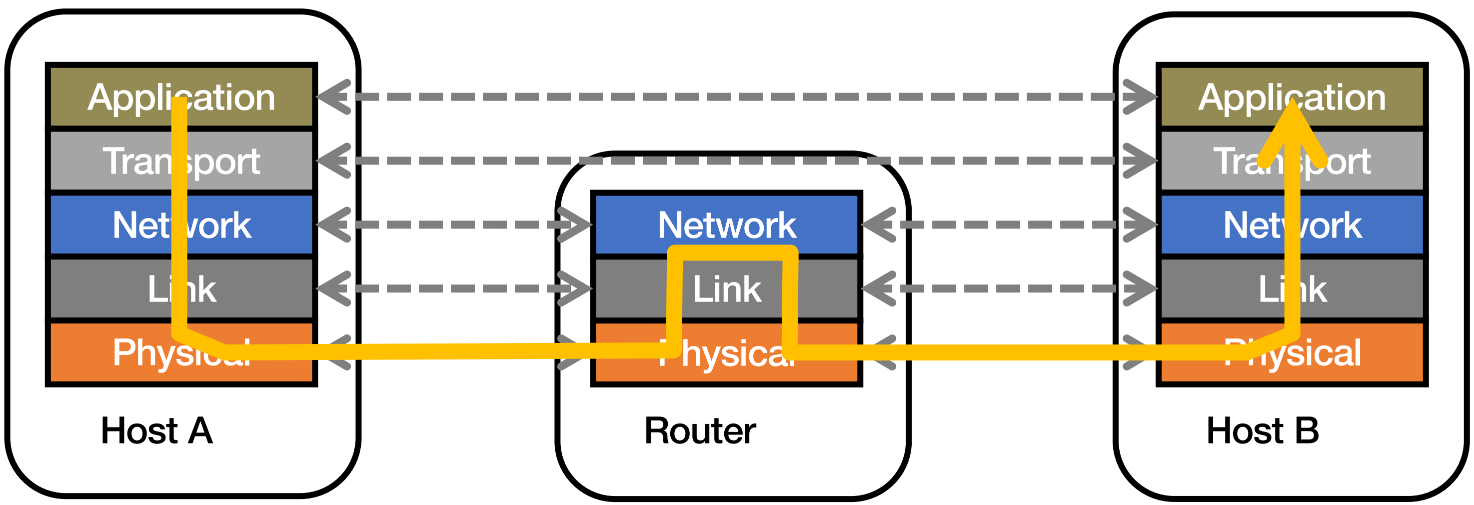
Physical communication

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



Physical communication

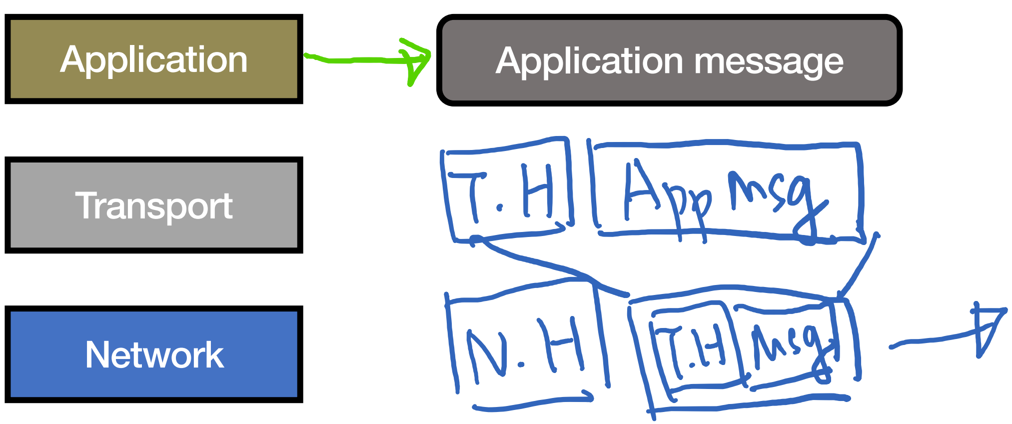
- Communication goes down to the physical network
- Then from network peer to peer
- Then up to the relevant application



Communication between layers

- How do peer protocols coordinate with each other?

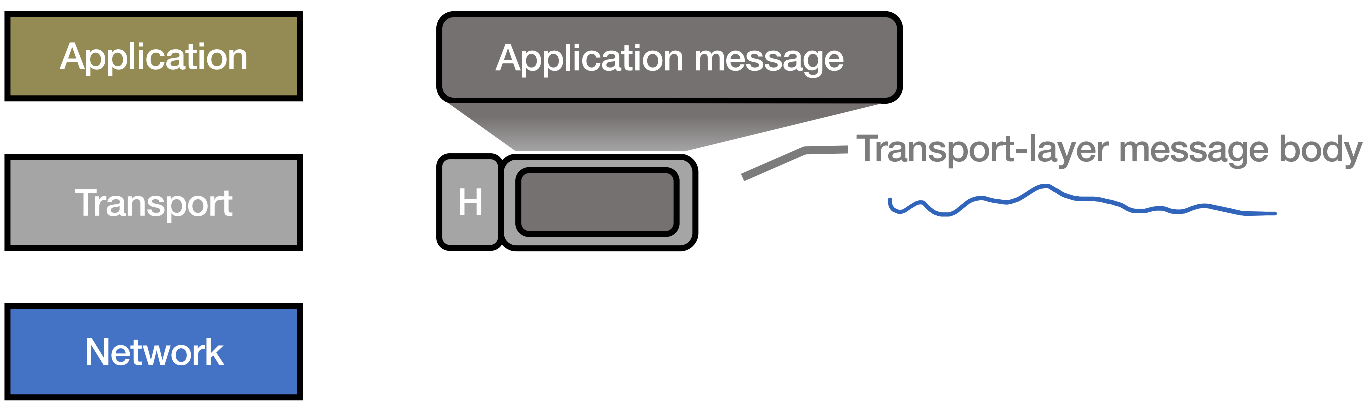
- Layer attaches its own header (H) to communicate with peer
  - Higher layers' headers, data encapsulated inside message
    - Lower layers don't generally inspect higher layers' headers



Communication between layers

- How do peer protocols coordinate with each other?

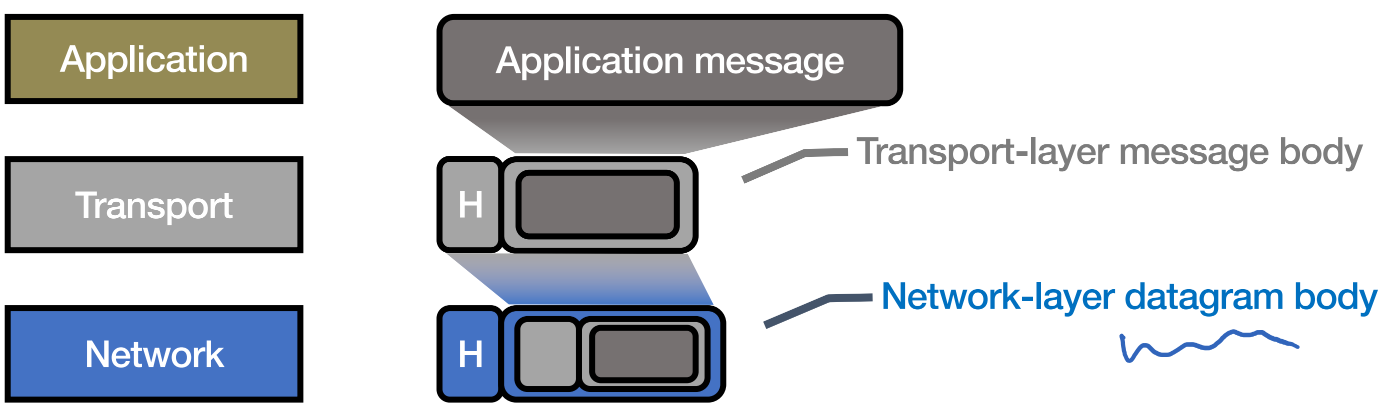
- Layer attaches its own header (H) to communicate with peer
  - Higher layers' headers, data encapsulated inside message
    - Lower layers don't generally inspect higher layers' headers



Communication between layers

- How do peer protocols coordinate with each other?

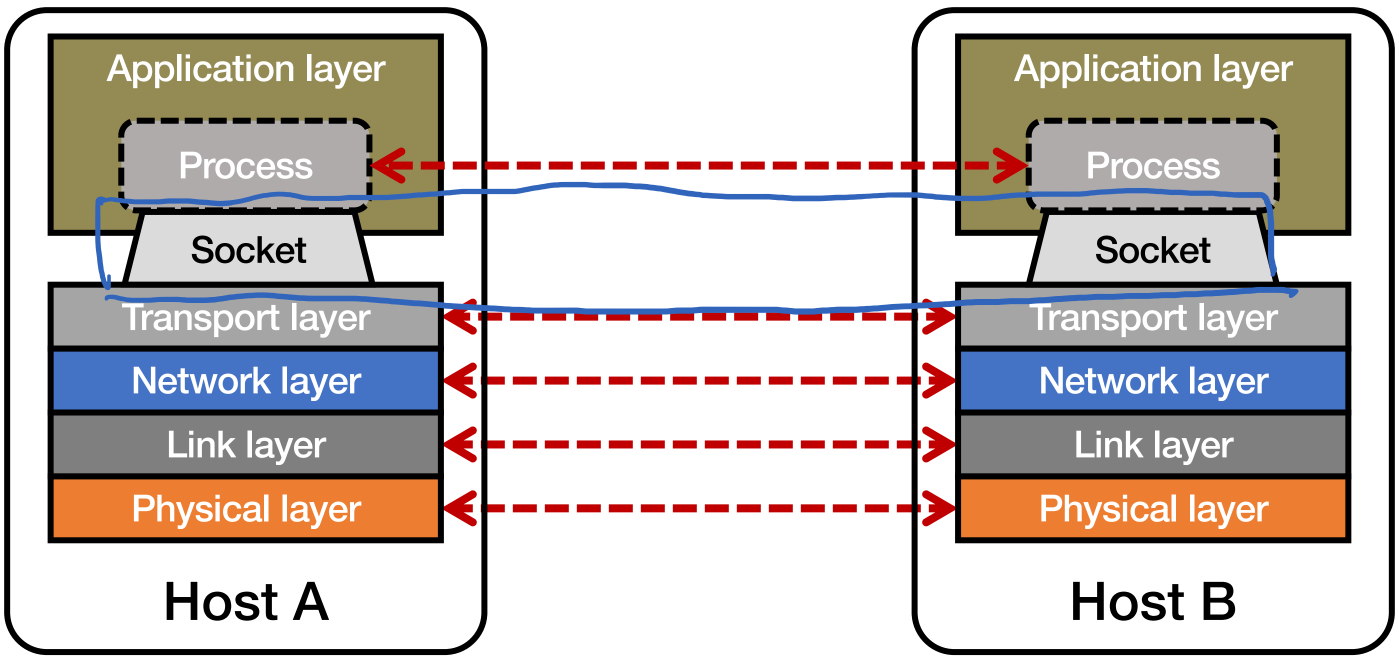
- Layer attaches its own header (H) to communicate with peer
  - Higher layers' headers, data encapsulated inside message
    - Lower layers don't generally inspect higher layers' headers



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



Network sockets: Summary

- 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

- **`put(key,value)`** → message with sockets?

S1

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

TCP

S2

```
// 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
```

S3

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

S4

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

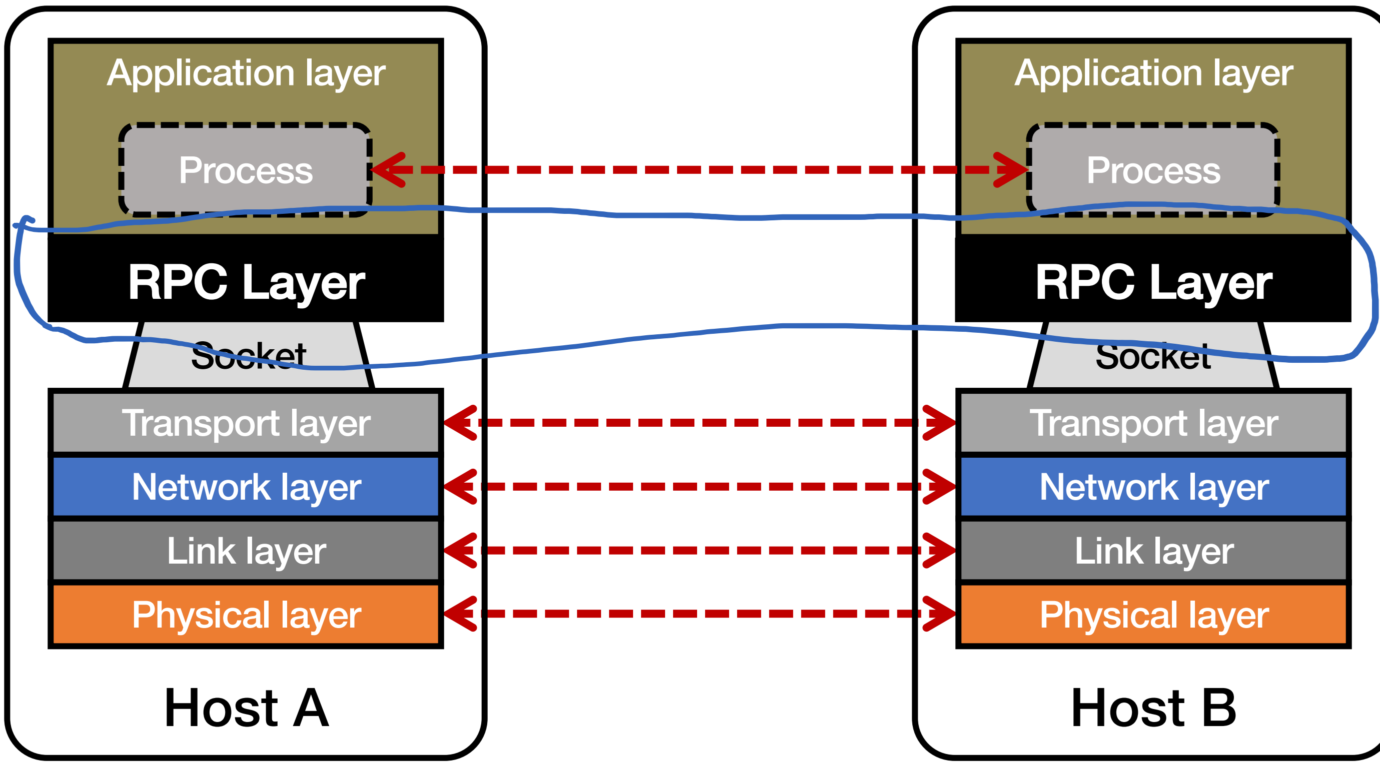
```c
// Create a socket for the client
if ((sockfd = socket (AF_INET, SOCK_STREAM, 0)) < 0) {
  perror("Socket creation");
  exit(2);
}

// 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);
```

Sockets don't provide transparency

Takeaway: Socket programming still not ideal (great)

- Lots for the programmer to deal with every time
  - 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 written in Go and Host B's process is in C++?
  - What to do with those bytes?
- Still pretty painful... Have to worry a lot about the network

Solution: Another layer!



Today's outline

1. Network sockets

2. Remote procedure call

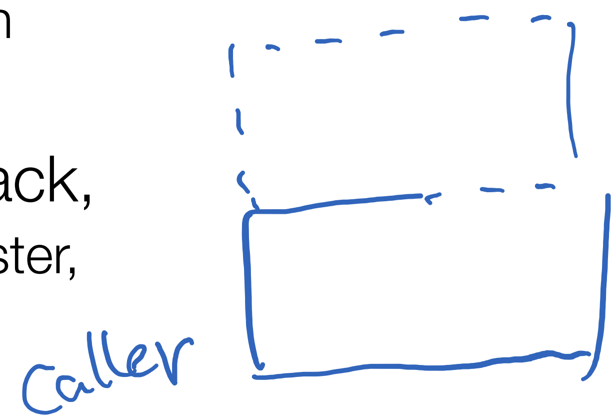
3. RPCs in Go

Motivation: 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
- Programming Labs use Go RPC

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



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: make communication appear like a local procedure call: transparency for procedure calls – way less painful than sockets…

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?

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?

RPC issues

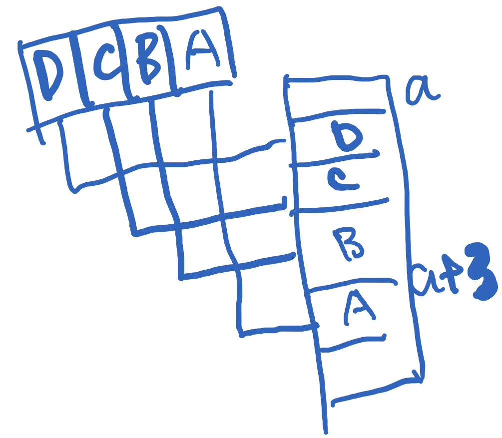
1. Heterogeneity
   - Client needs to rendezvous with the server
   - Server must dispatch to the required function
     - What if server is different type of machine?

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

3. Performance
   - Procedure call takes takes ≈ 10 cycles ≈ 3 ns
   - RPC in a data center takes ≈ 10 μs (10^3× slower)
     - In the wide area, typically 10^6× slower

Problem: Differences in data representation

endianness.



- Not an issue for local procedure calls

- 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

Problem: Differences in programming support

gRPC
thrift

- Language support varies:
  - Many programming languages have no inbuilt way of extracting values from complex types
    - C, C++
    - Effectively need sockets glue code underneath
  - Some languages have support that enables RPC
    - Python, Go
    - Exploit type system for some help

Solution: Interface Description Language

Google Protobuf

- 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

Solution: Interface Description Language

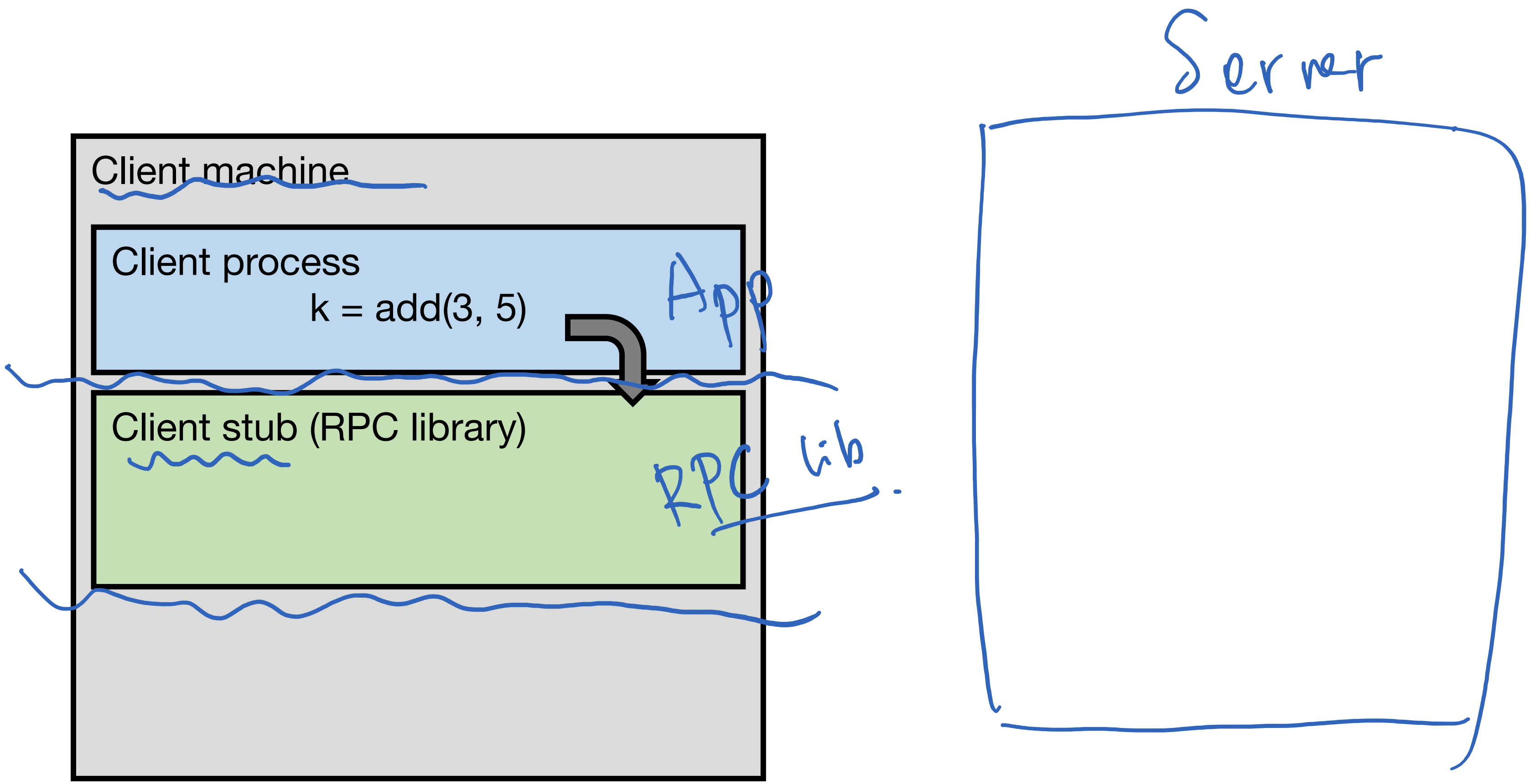
- Mechanism to pass procedure parameters and return values in a machine-independent way

- 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

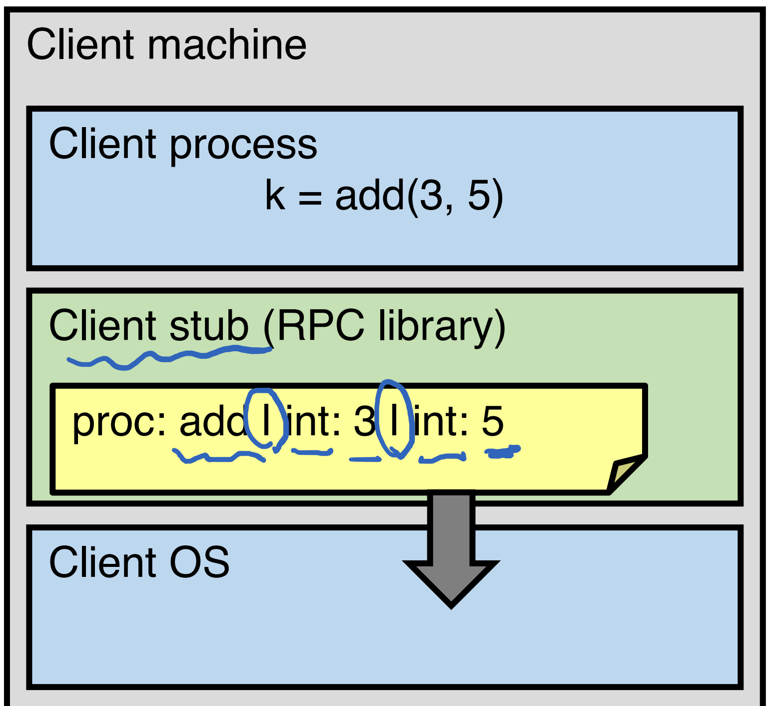
1. Client calls stub function (pushes parameters onto stack)



A day in the life of an RPC

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

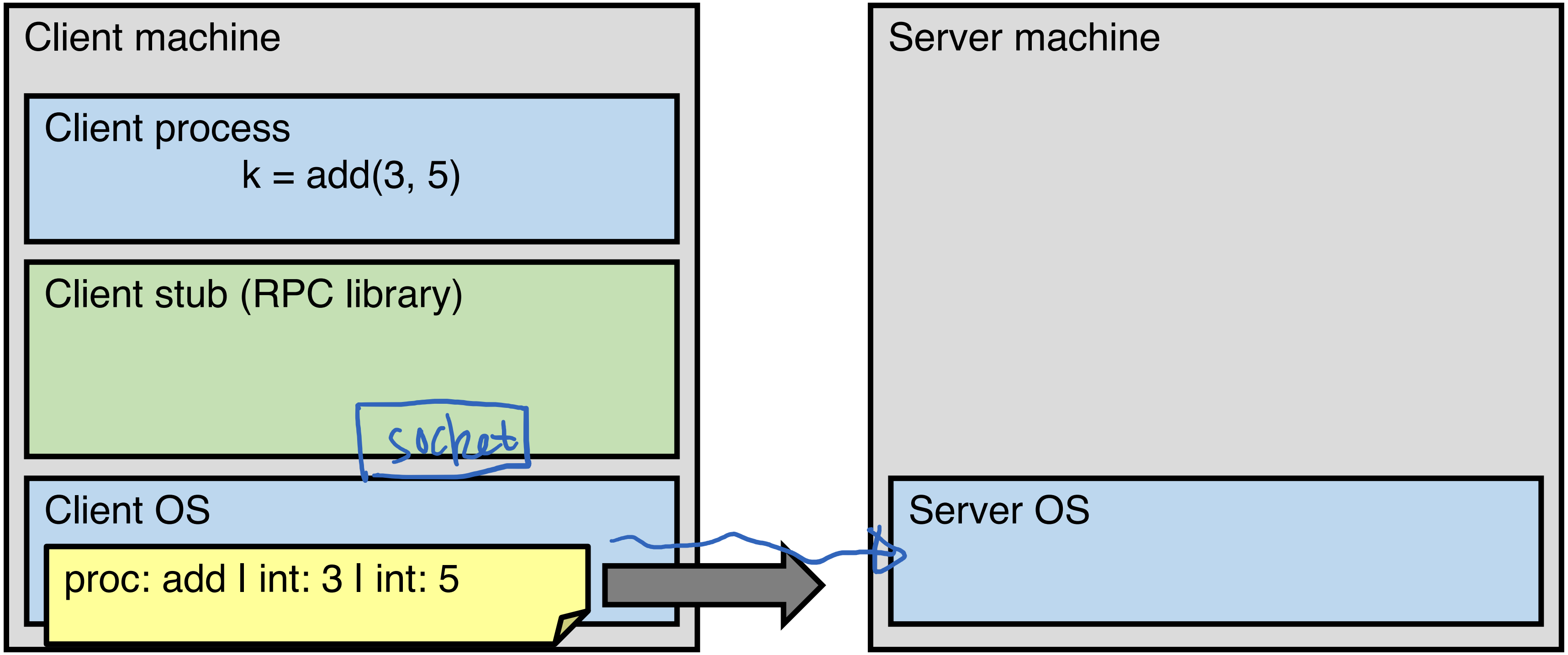
2. Stub marshals parameters to a network message



A day in the life of an RPC

2. Stub marshals parameters to a network message

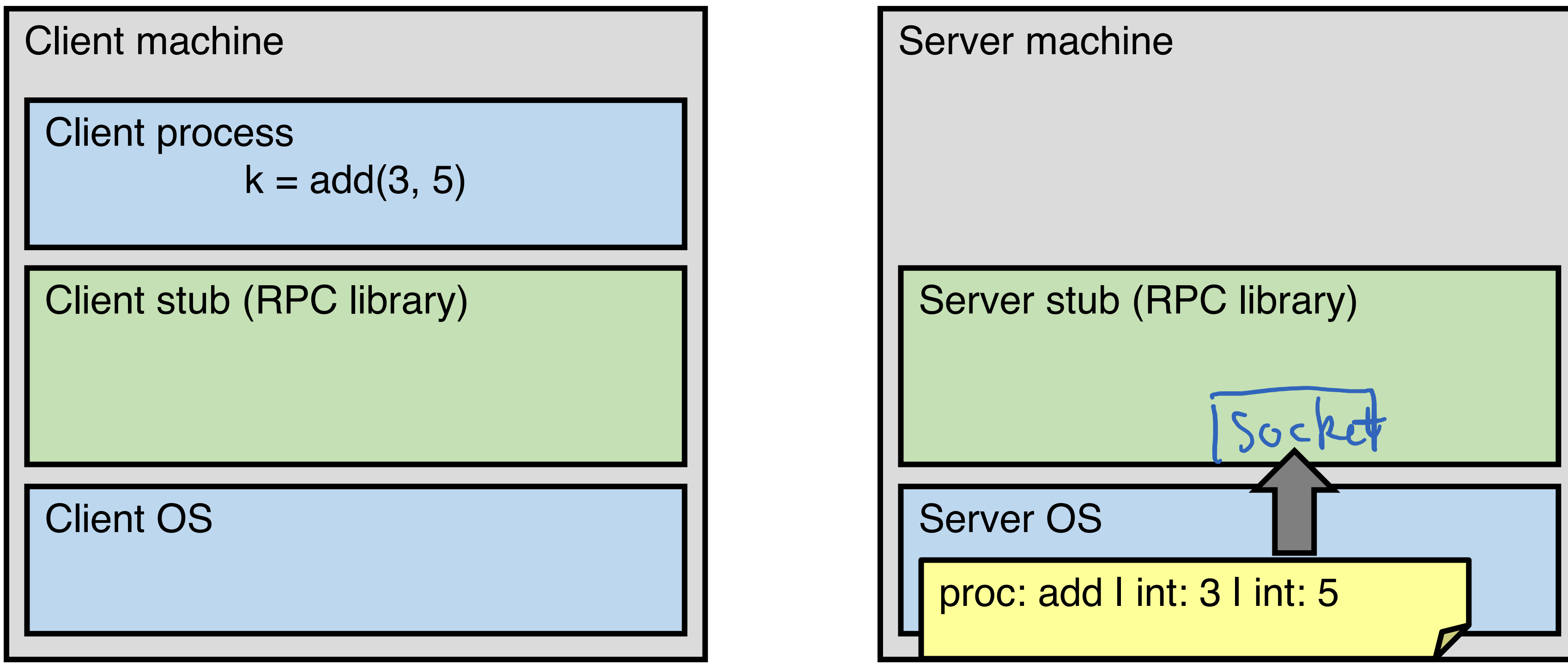
3. OS sends a network message to the server



A day in the life of an RPC

3. OS sends a network message to the server

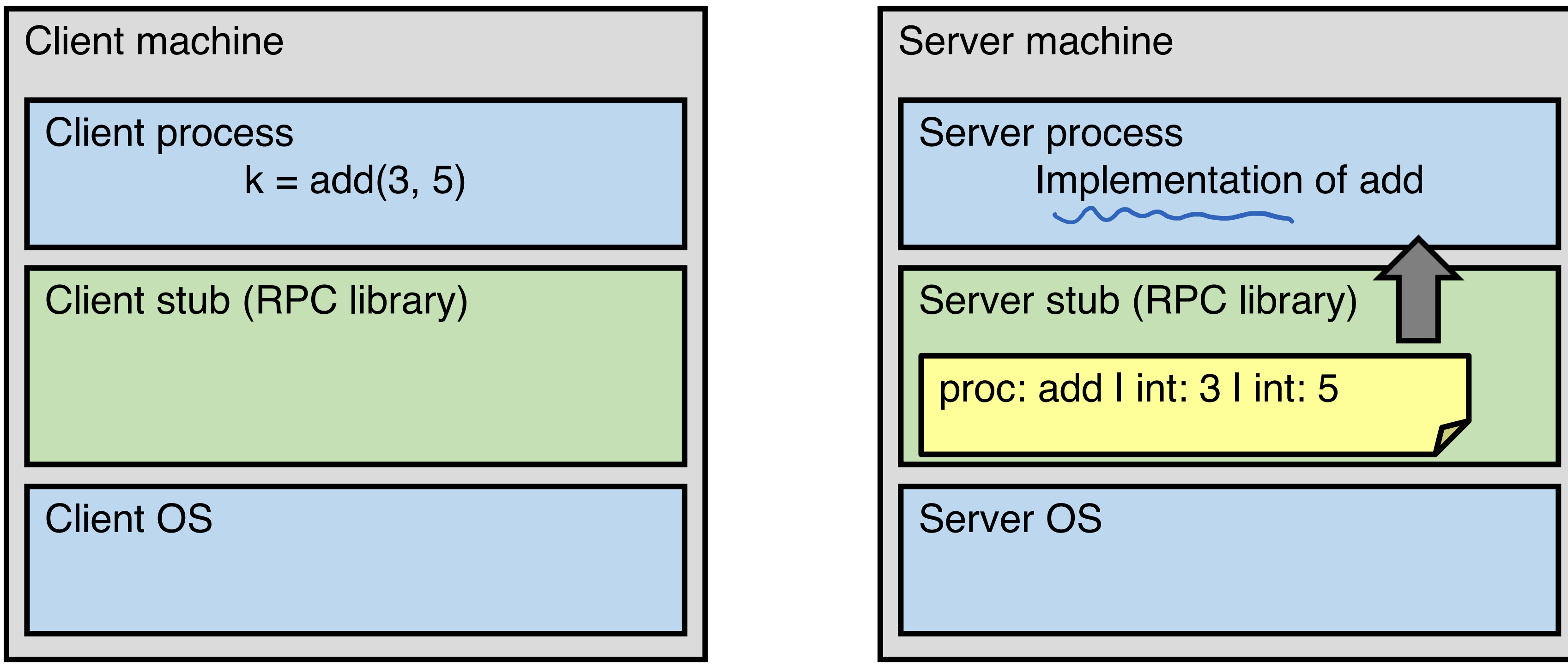
4. Server OS receives message, sends it up to stub



A day in the life of an RPC

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

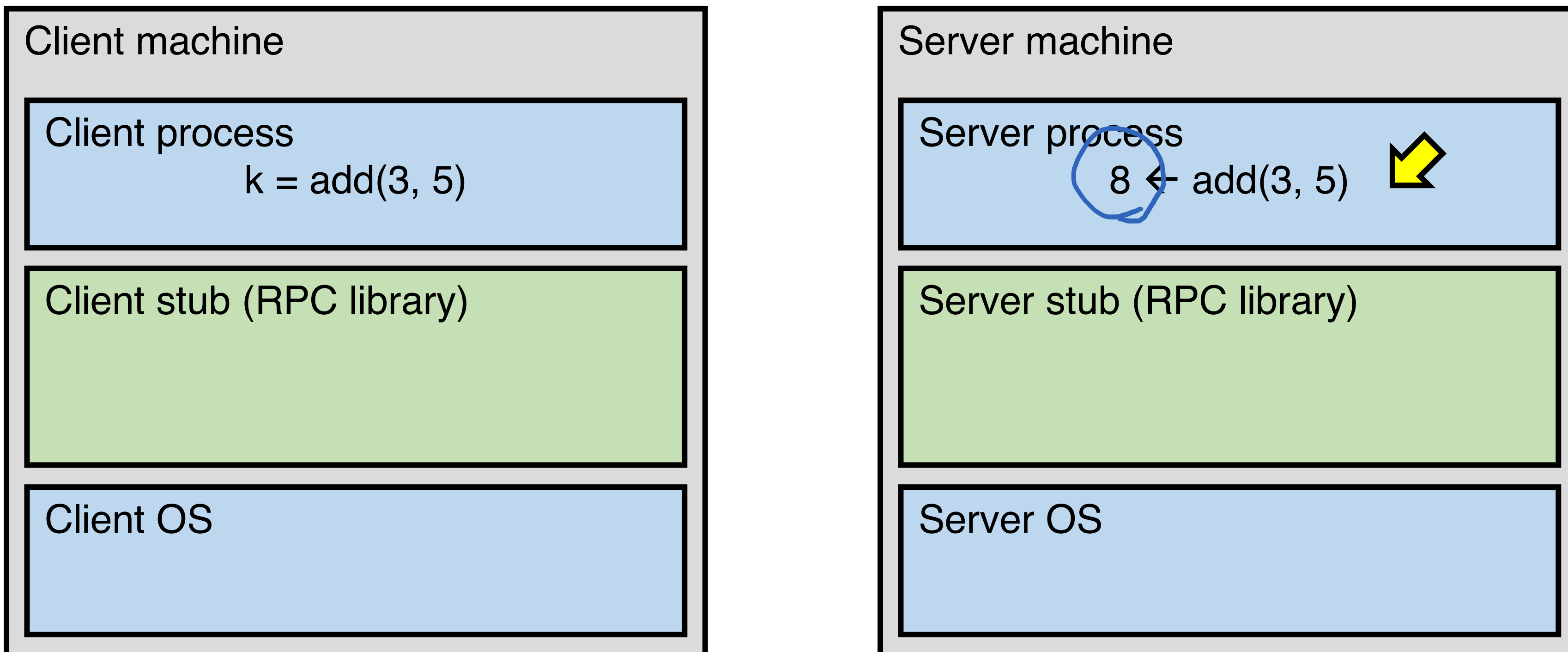
5. Server stub unmarshals params, calls server function



A day in the life of an RPC

5. Server stub unmarshals params, calls server function

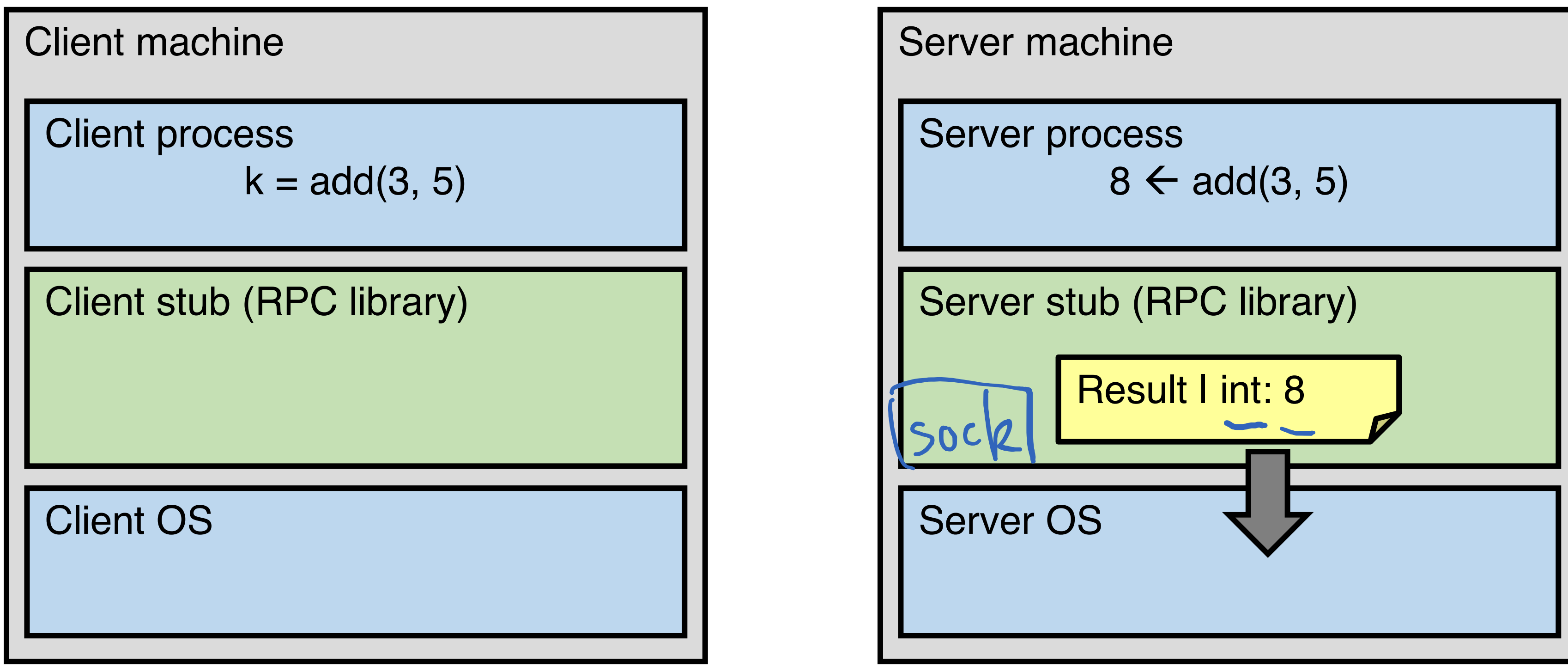
6. Server function runs, returns a value



A day in the life of an RPC

6. Server function runs, returns a value

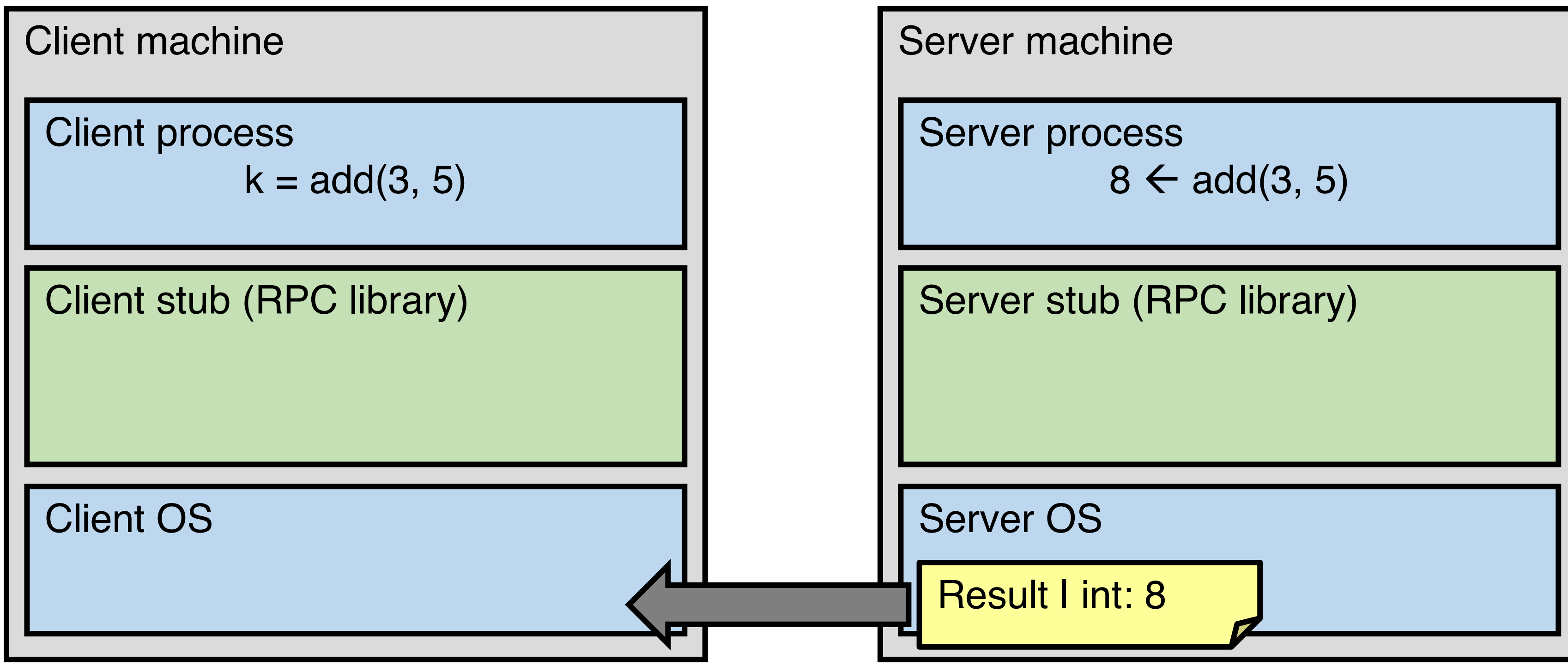
7. Server stub marshals the return value, sends message



A day in the life of an RPC

7. Server stub marshals the return value, sends message

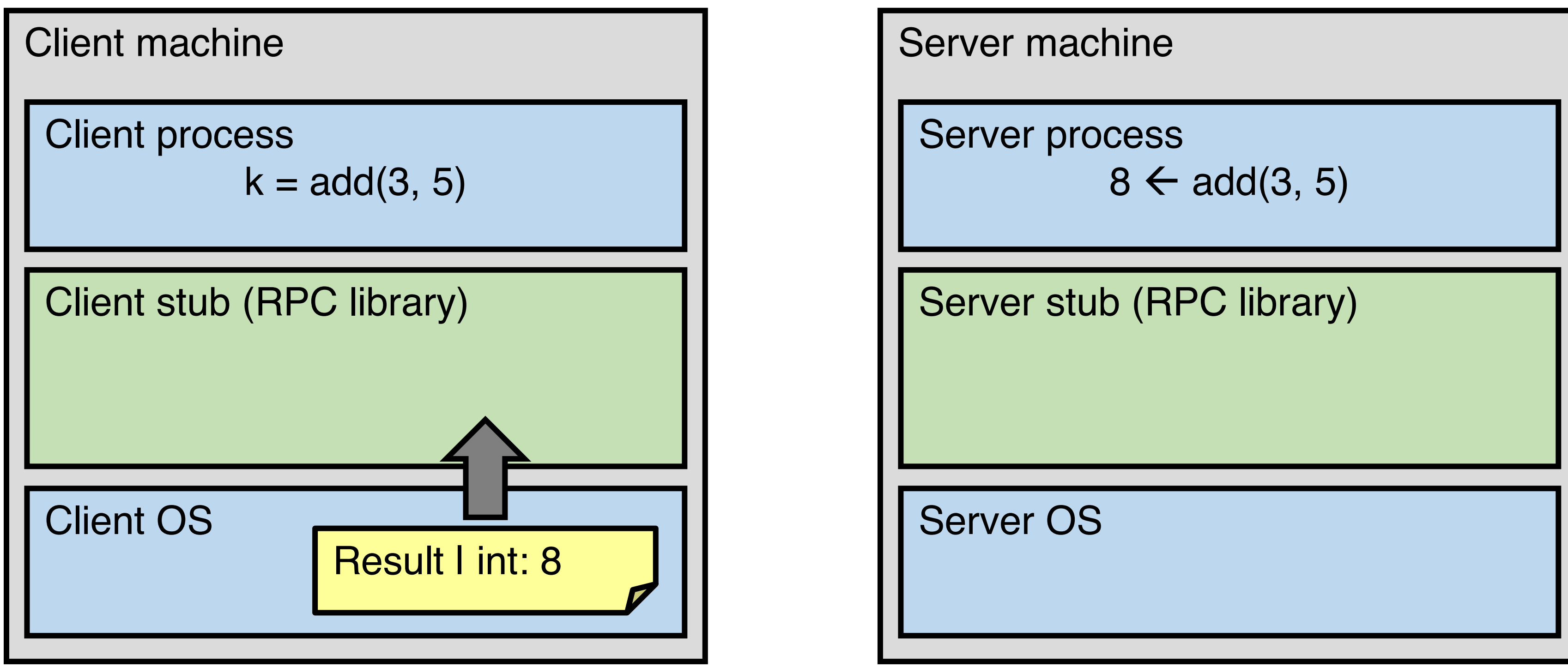
8. Server OS sends the reply back across the network



A day in the life of an RPC

8. Server OS sends the reply back across the network

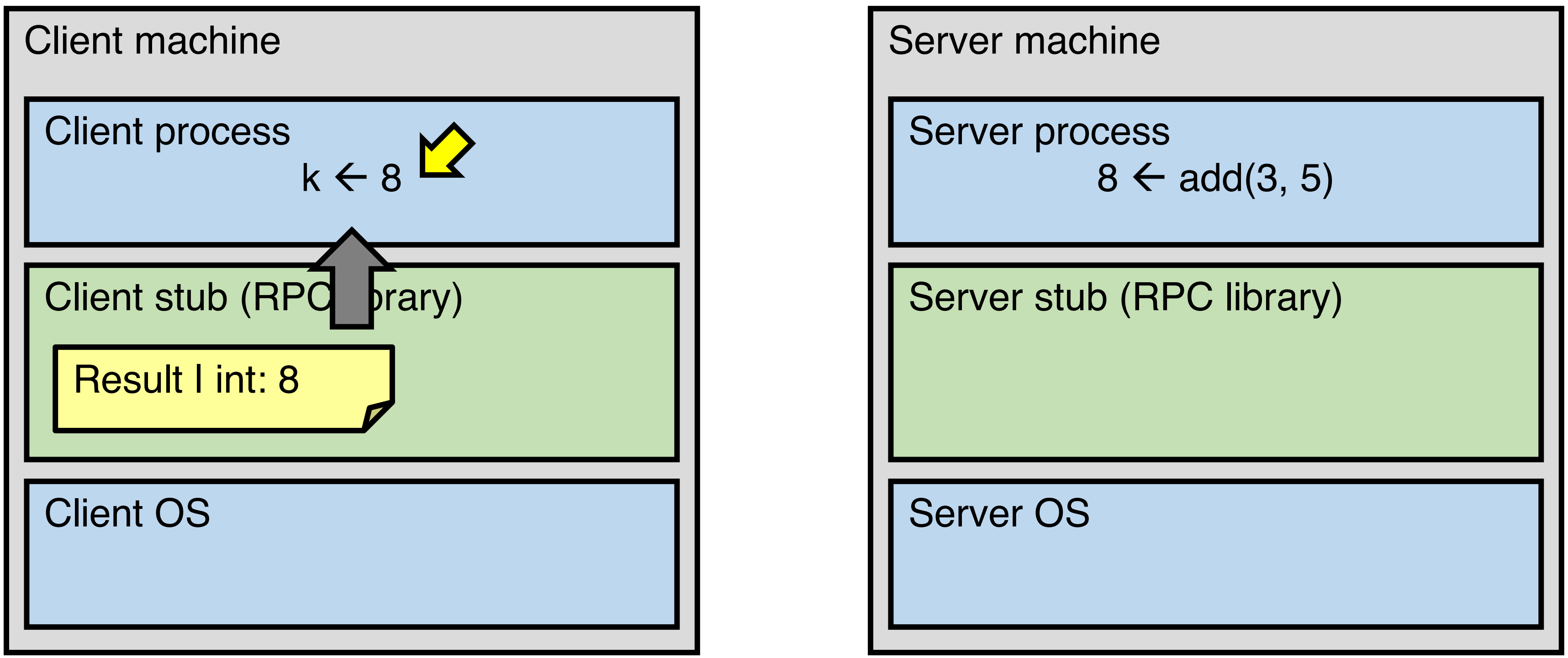
9. Client OS receives the reply and passes up to stub



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 sockets

2. Remote procedure call
    - Heterogeneity – use IDL w/ compiler
    - Failure

3. RPCs in Go

What could possibly go wrong?

1. Client may crash and reboot

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

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

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

What could possibly go wrong?

1. Client may crash and reboot

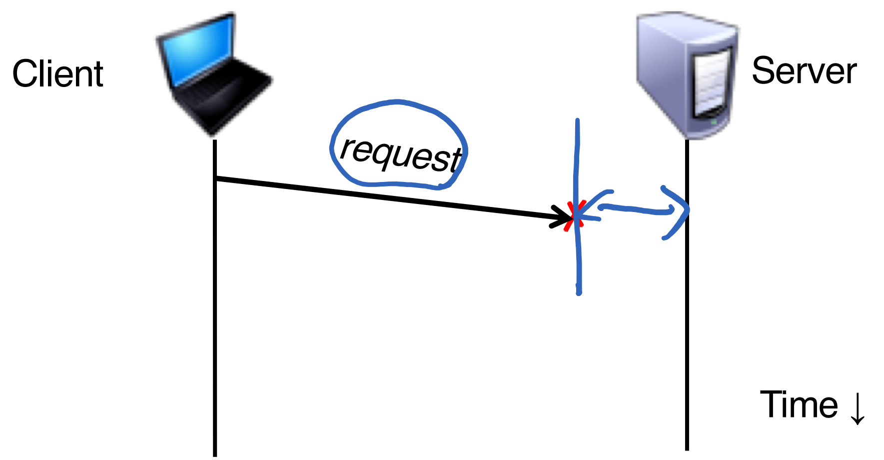
2. Packets may be dropped
   - Some individual packet loss in the Internet
   - Broken routing results in many lost packets

3. Server may crash and reboot

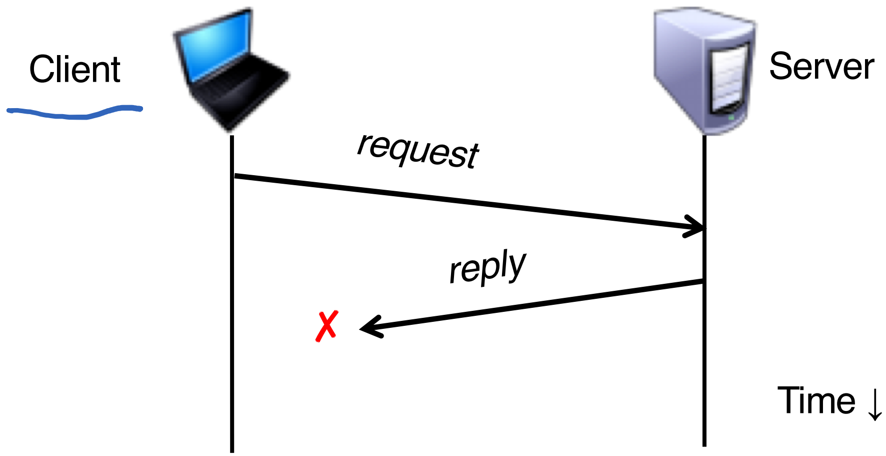
4. Network or server might just be very slow

All of these may **look the same** to the client…

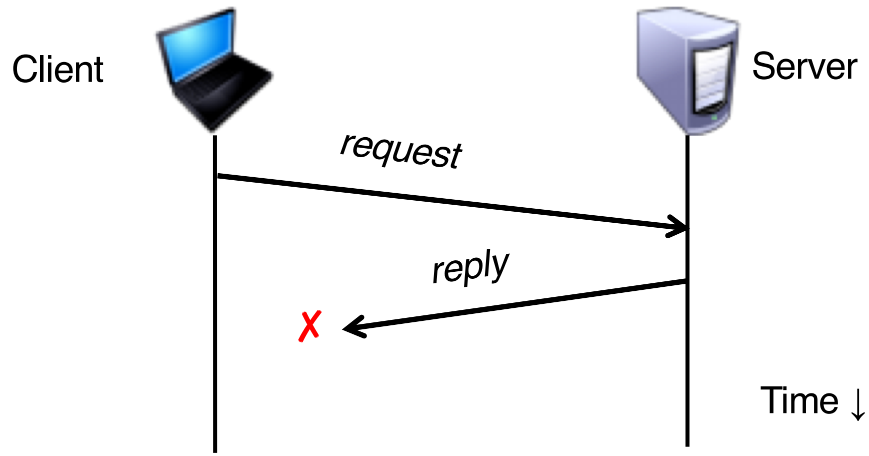
Failures, from client's perspective



Failures, from client's perspective



Failures, from client's perspective



The cause of the failure is hidden from the client!

At-Least-Once scheme

- Simplest scheme for handling failures

1. Client stub waits for a response, for a while
   - Response is an acknowledgement message from the server stub

At-Least-Once scheme

- Simplest scheme for handling failures

1. Client stub waits for a response, for a while
   - Response is an acknowledgement message from the server stub

retry

2. If no response arrives after a fixed timeout time period, then client stub re-sends the request

At-Least-Once scheme

- Simplest scheme for handling failures

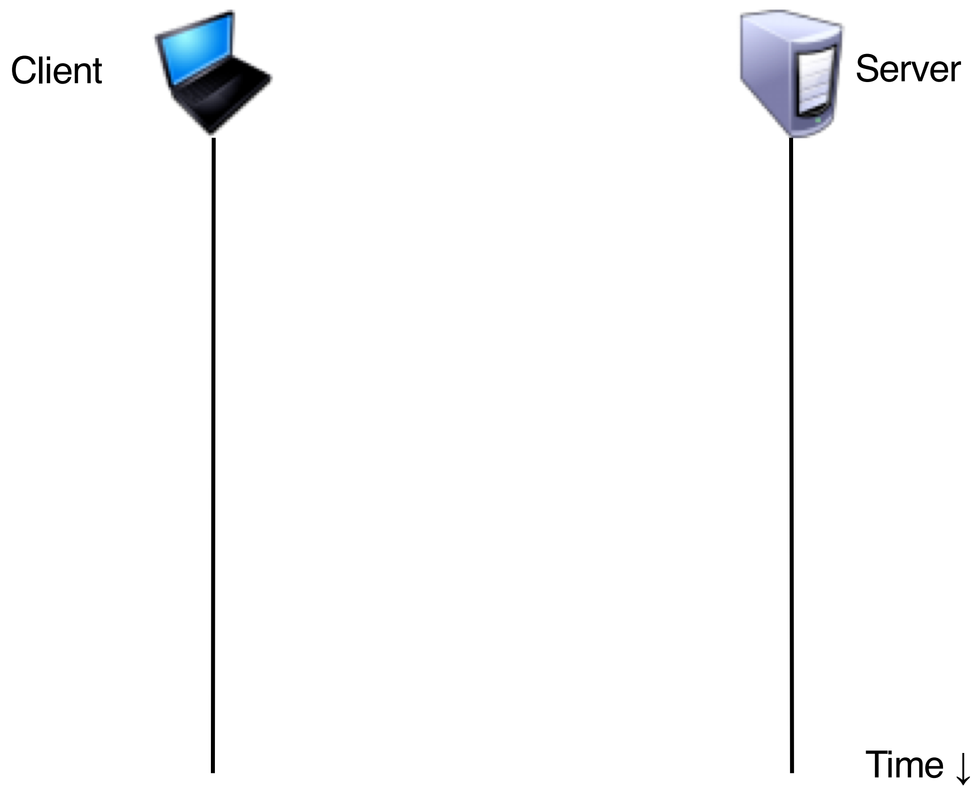
1. Client stub waits for a response, for a while
    - Response is an acknowledgement message from the server stub

2. If no response arrives after a fixed timeout time period, then client stub re-sends the request

- Repeat the above a few times
    - Still no response? Return an error to the application

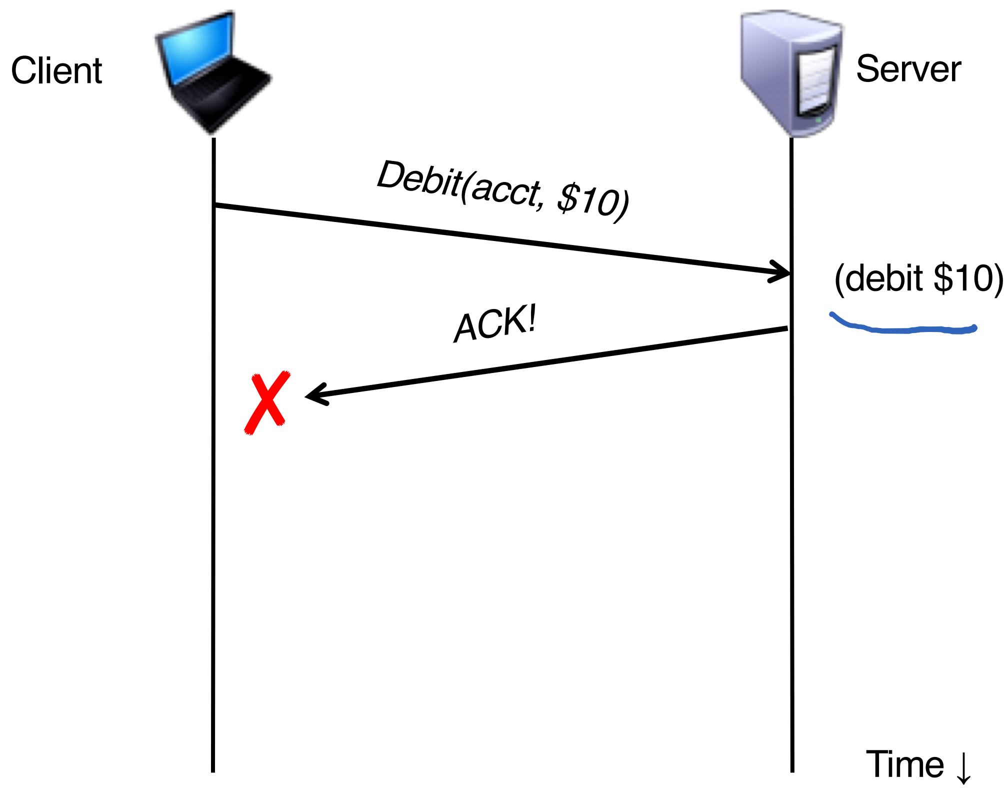
At-Least-Once and side effects

- Client sends a “debit $10 from bank account” RPC



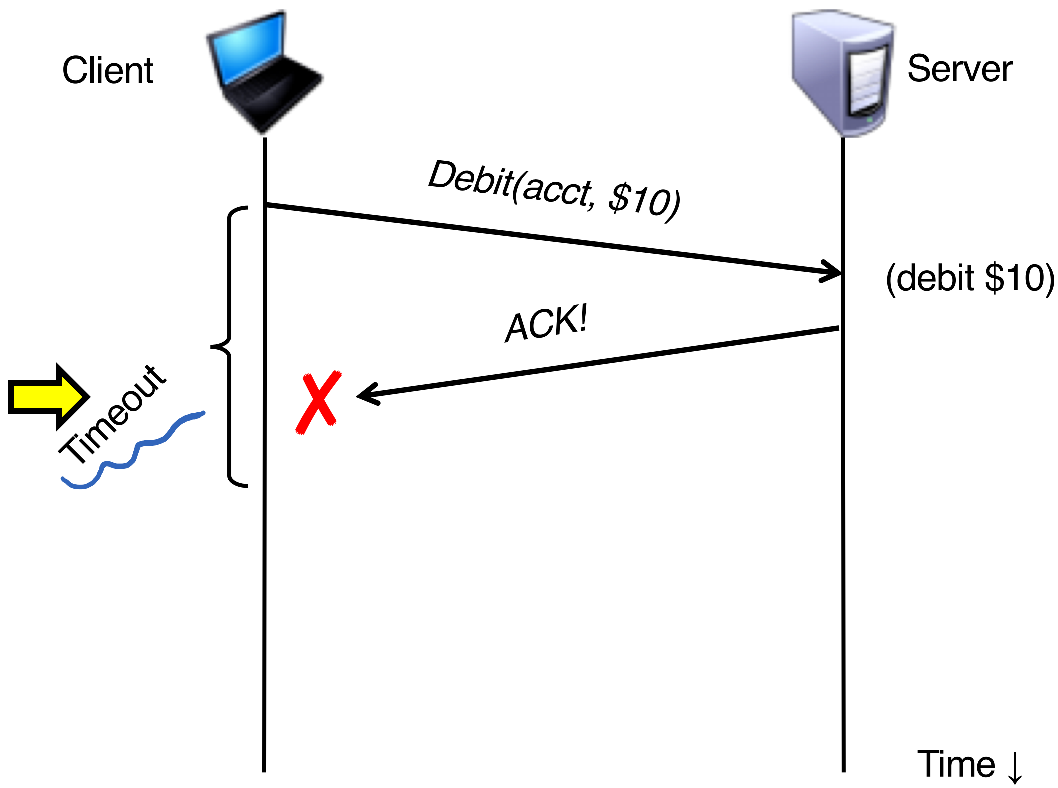
At-Least-Once and side effects

- Client sends a “debit $10 from bank account” RPC



At-Least-Once and side effects

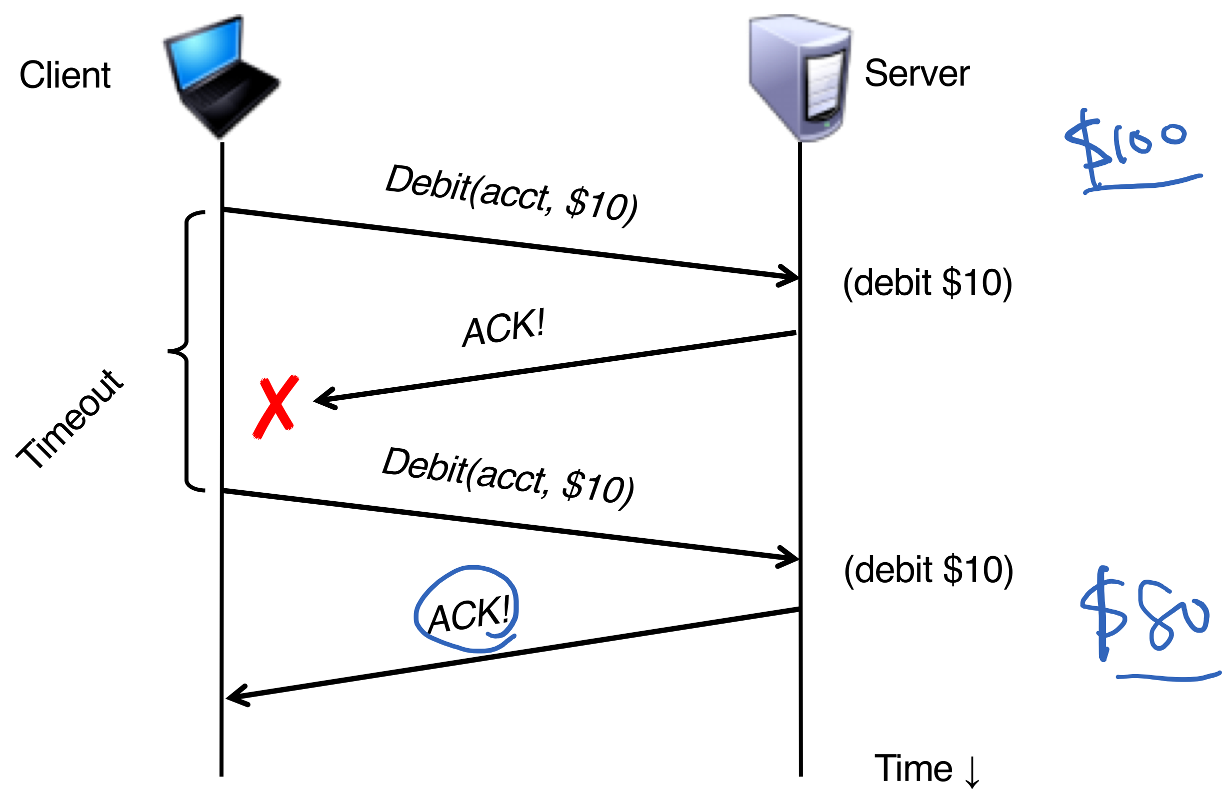
- Client sends a “debit $10 from bank account” RPC



At-Least-Once and side effects

duplicate

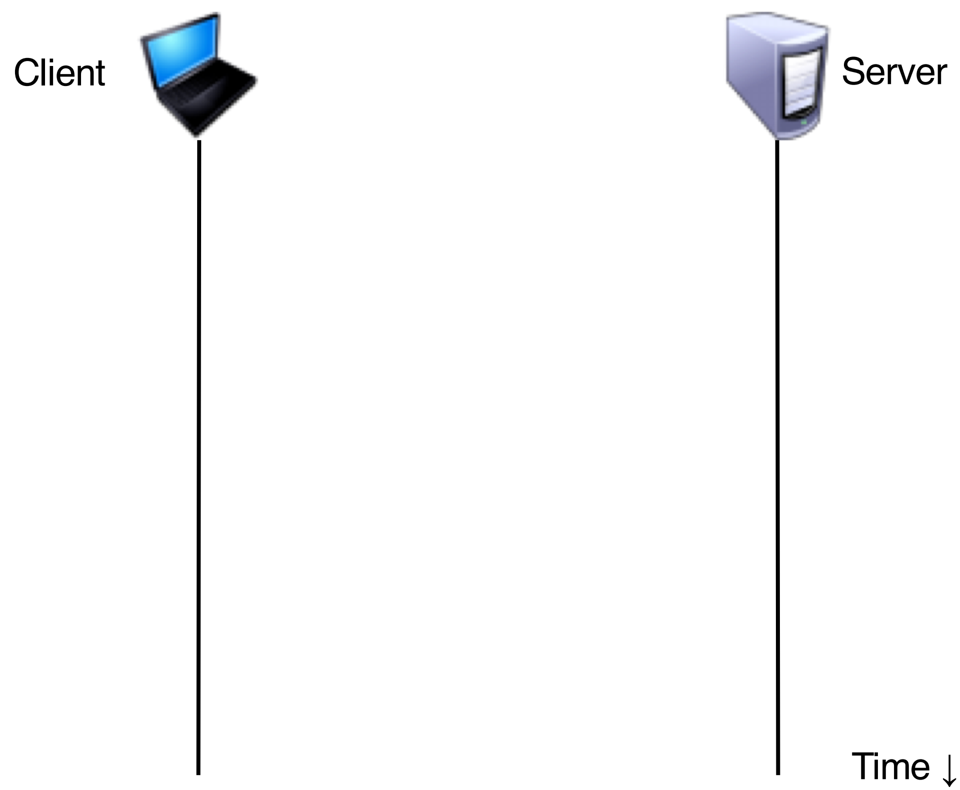
- Client sends a “debit $10 from bank account” RPC



At-Least-Once and writes

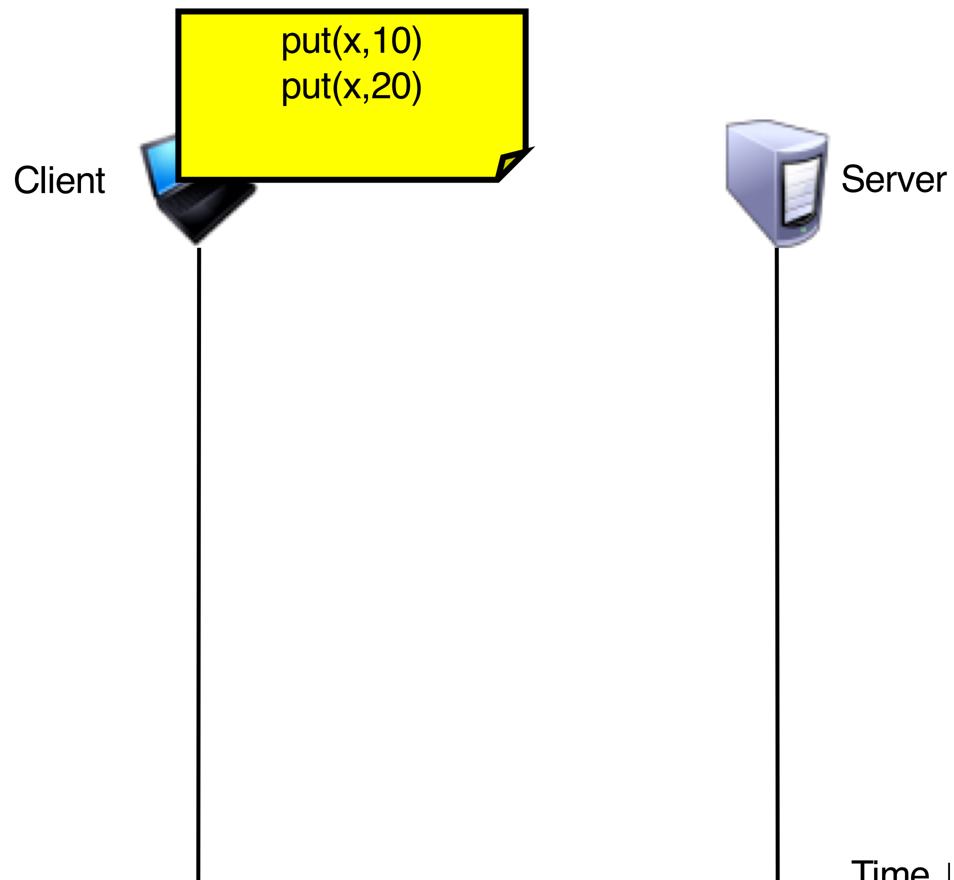
key

- `put(x, value)`, then `get(x)`: expect answer to be *value*



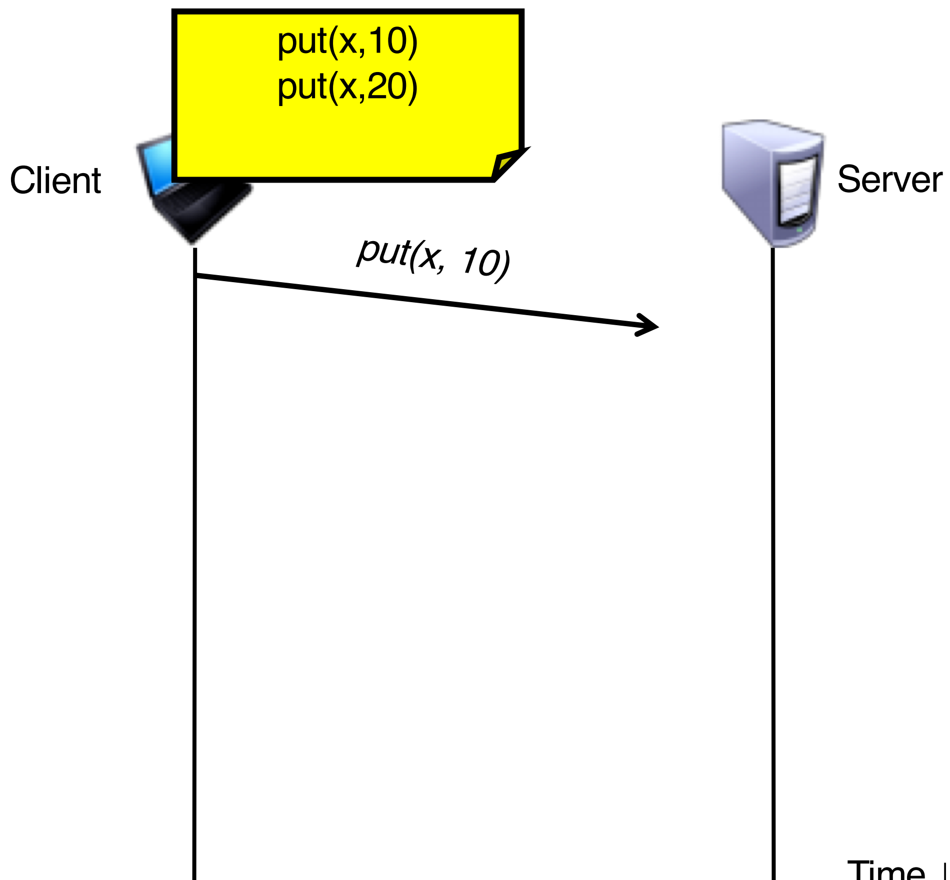
At-Least-Once and writes

- `put(x, value)`, then `get(x)`: expect answer to be *value*



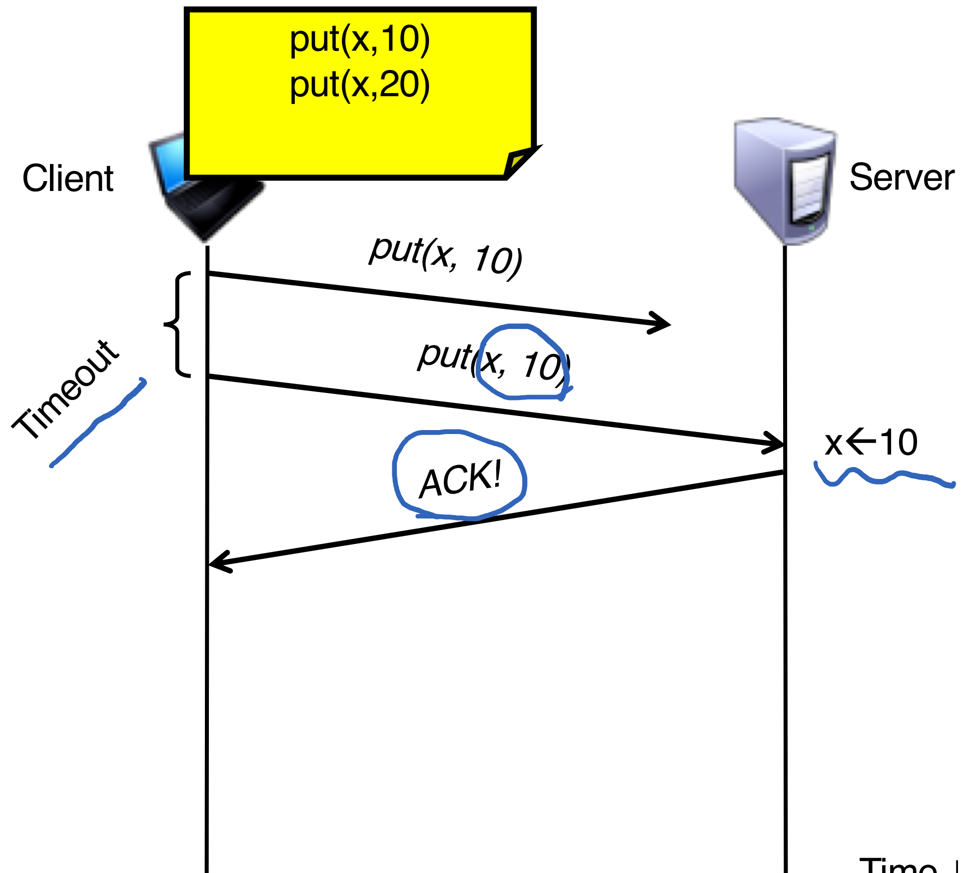
At-Least-Once and writes

- `put(x, value)`, then `get(x)`: expect answer to be *value*



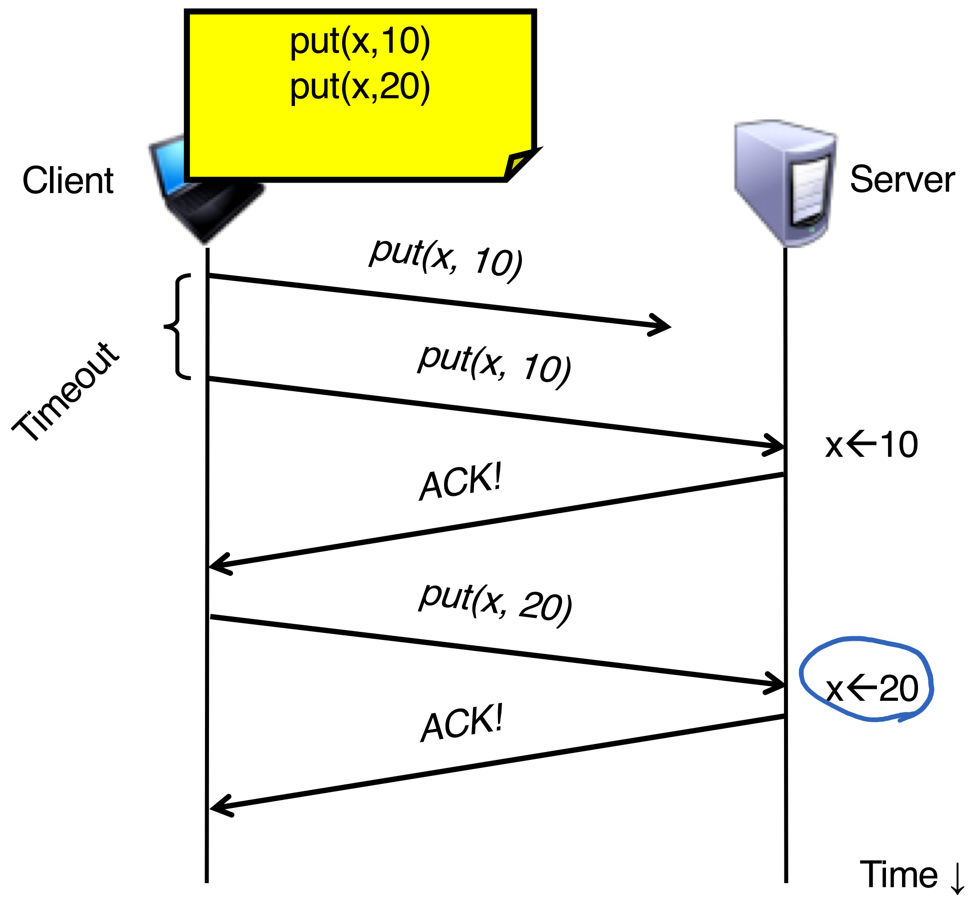
At-Least-Once and writes

- `put(x, value)`, then `get(x)`: expect answer to be *value*



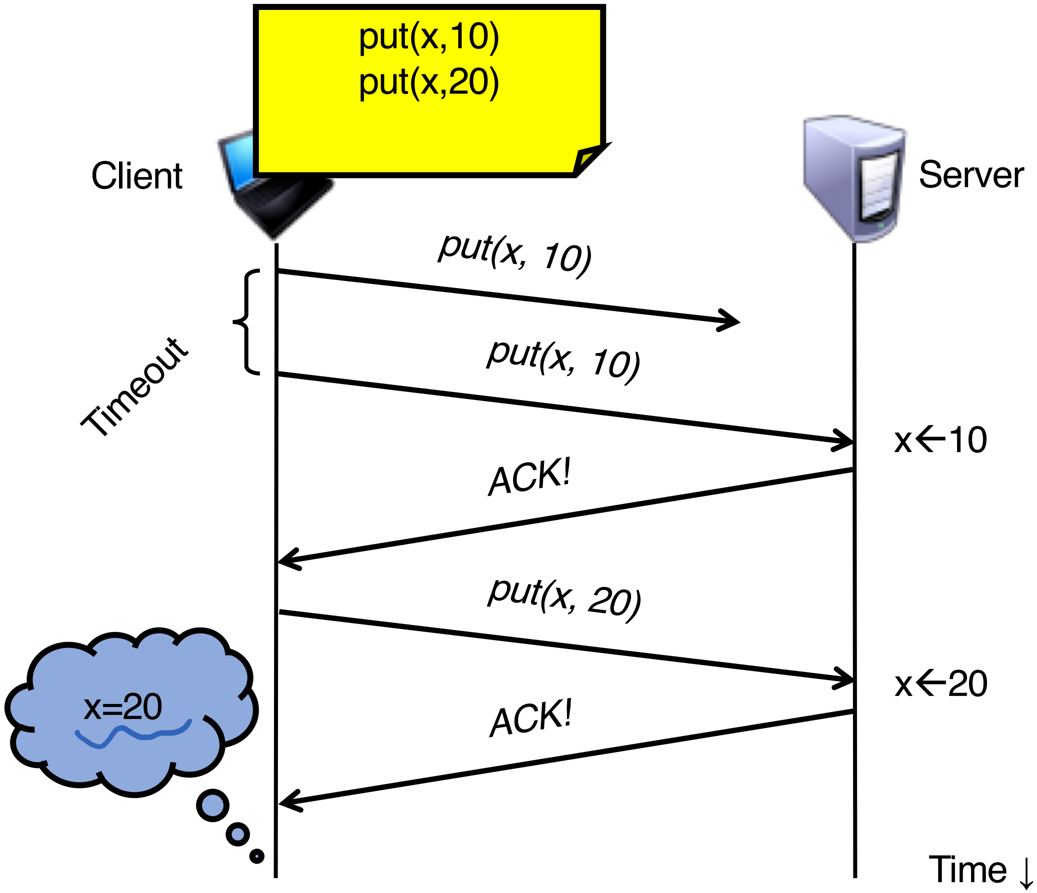
At-Least-Once and writes

- `put(x, value)`, then `get(x)`: expect answer to be *value*



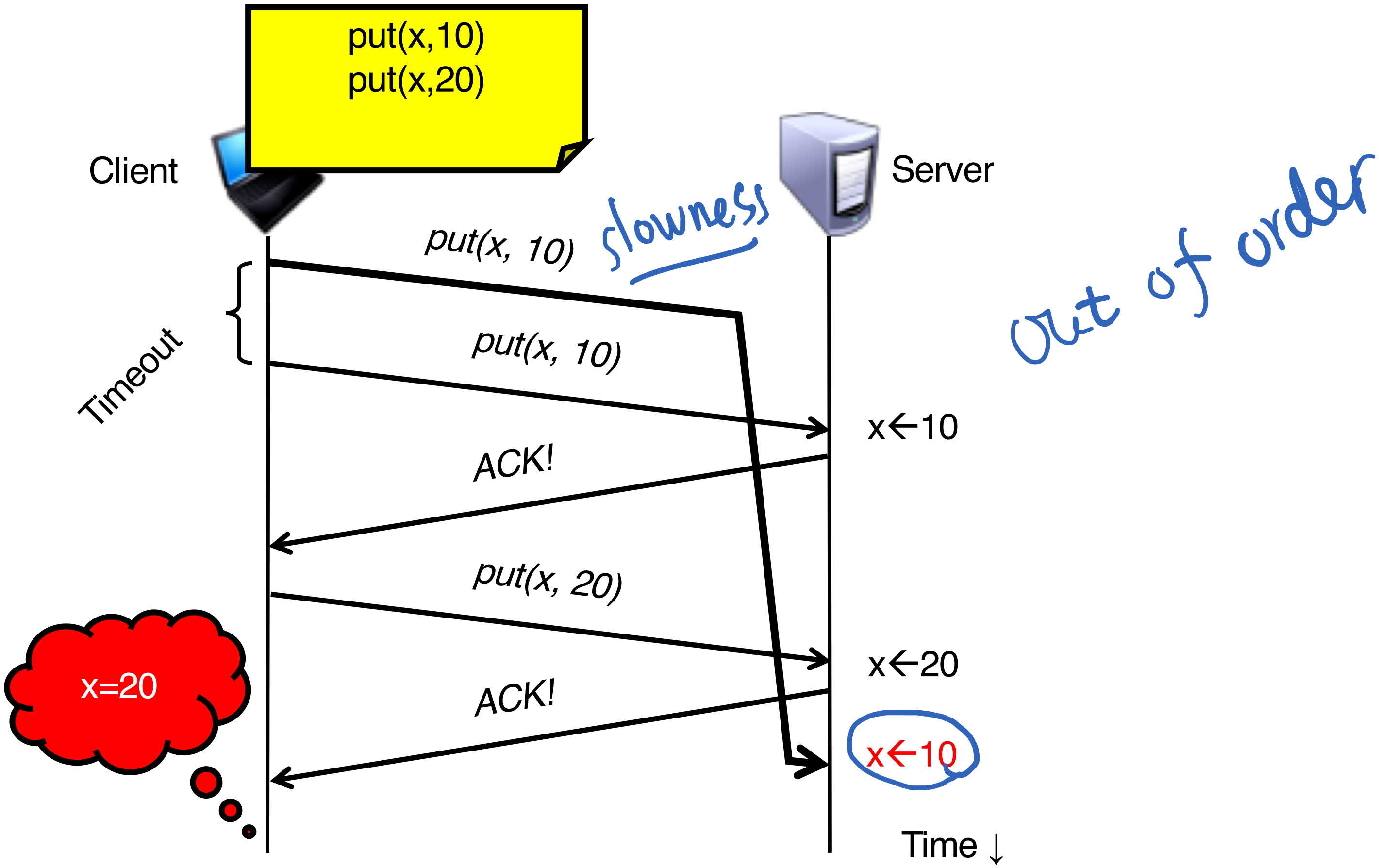
At-Least-Once and writes

- `put(x, value)`, then `get(x)`: expect answer to be *value*



At-Least-Once and writes

- Consider a client storing key-value pairs in a database
  - `put(x, value)`, then `get(x)`: expect answer to be *value*



So, is At-Least-Once ever okay?

- **Yes:** If they are read-only operations with no side effects
  - e.g., read a key's value in a database

- **Yes:** If the application has its own functionality to cope with duplication and reordering

At-Most-Once scheme

de-duplication

- Idea: server RPC code detects duplicate requests
  - Returns previous reply instead of re-running handler

At-Most-Once scheme

- Idea: server RPC code detects duplicate requests
  - Returns previous reply instead of re-running handler

- How to detect a duplicate request?
  - Test: Server sees same function, same arguments twice

At-Most-Once scheme

- Idea: server RPC code detects duplicate requests
  - Returns previous reply instead of re-running handler

- How to detect a duplicate request?
  - Test: Server sees same function, same arguments twice
    - No! Sometimes applications legitimately submit the same function with same augments, twice in a row

At-Most-Once scheme

- How to detect a duplicate request?
  - Client includes unique transaction ID (xid) with each RPC requests
  - Client uses same xid for retransmitted requests

At-Most-Once scheme

- How to detect a duplicate request?
  - Client includes unique transaction ID (xid) with each RPC requests
  - Client uses same xid for retransmitted requests

```
At-Most-Once Server
if seen[xid]:
     retval = old[xid]
else:
     retval = handler()
     old[xid] = retval
     seen[xid] = true
return retval
```

At-Most-Once: Providing unique XIDs

1. Combine a unique client ID (e.g., IP address) with the current time of day

At-Most-Once: Providing unique XIDs

1. Combine a unique client ID (e.g., IP address) with the current time of day

monotonically ++

2. Combine unique client ID with a sequence # number
   - Suppose client crashes and restarts. Can it reuse the same client ID?

At-Most-Once: Providing unique XIDs

1. Combine a unique client ID (e.g., IP address) with the current time of day

2. Combine unique client ID with a sequence number
   - Suppose client crashes and restarts. Can it reuse the same client ID?

3. Big random number (probabilistic, not certain guarantee)

At-Most-Once: Discarding server state

- Problem: `seen` and `old` arrays will grow without bound

At-Most-Once: Discarding server state

- Problem: `seen` and `old` arrays will grow without bound

- Observation: By construction, when the client gets a response to a particular xid, it will never re-send it

At-Most-Once: Discarding server state

- Problem: `seen` and `old` arrays will grow without bound

- Observation: By construction, when the client gets a response to a particular xid, it will never re-send it

- Client could tell server "I'm done with xid x – delete it"
  - Have to tell the server about each and every retired xid
    - Could piggyback on subsequent requests

At-Most-Once: Discarding server state

- Problem: `seen` and `old` arrays will grow without bound

- Observation: By construction, when the client gets a response to a particular xid, it will never re-send it

- Client could tell server "I'm done with xid x – delete it"
  - Have to tell the server about each and every retired xid
    - Could piggyback on subsequent requests

Significant overhead if many RPCs are in flight, in parallel

At-Most-Once: Discarding server state

- Problem: `seen` and `old` arrays will grow without bound

At-Most-Once: Discarding server state

- Problem: `seen` and `old` arrays will grow without bound

- Suppose `xid = ⟨`unique client id, sequence no.⟩
  - e.g., ⟨42, 1000⟩, ⟨42, 1001⟩, ⟨42, 1002⟩

At-Most-Once: Discarding server state

- Problem: `seen` and `old` arrays will grow without bound

- Suppose `xid =` ⟨unique client id, sequence no.⟩
  - e.g., ⟨42, 1000⟩, ⟨42, 1001⟩, ⟨42, 1002⟩

- Client includes "seen all replies ≤ X" with every RPC
  - Much like TCP sequence numbers, acks

At-Most-Once: Discarding server state

- Problem: `seen` and `old` arrays will grow without bound
- Suppose `xid = `⟨unique client id, sequence no.⟩
  - e.g., ⟨42, 1000⟩, ⟨42, 1001⟩, ⟨42, 1002⟩
- Client includes "seen all replies ≤ X" with every RPC
  - Much like TCP sequence numbers, acks
- How does the client know that the server received the information about retired RPCs?
  - Each one of these is cumulative: later seen messages subsume earlier ones

At-Most-Once: Concurrent requests

- Problem: How to handle a duplicate request while the original is still executing?
  - Server doesn't know reply yet. Also, we don't want to run the procedure twice
- Idea: Add a `pending` flag per executing RPC
  - Server waits for the procedure to finish, or ignores

At-Most-Once: Server crash and restart

- Problem: Server may crash and restart

- Does server need to write its tables to disk?

At-Most-Once: Server crash and restart

- Problem: Server may crash and restart

- Does server need to write its tables to disk?

- Yes! On server crash and restart:
  - If `old[]`, `seen[]` tables are only in memory:
    - Server will forget, accept duplicate requests

Go’s net/rpc is at-most-once

- Opens a TCP connection and writes the request
  - TCP may retransmit but server's TCP receiver will filter out duplicates internally, with sequence numbers
  - No retry in Go RPC code (i.e. will not create a second TCP connection)

Go's net/rpc is at-most-once

- Opens a TCP connection and writes the request
  - TCP may retransmit but server's TCP receiver will filter out duplicates internally, with sequence numbers
  - No retry in Go RPC code (i.e. will not create a second TCP connection)
- However: Go RPC returns an error if it doesn't get a reply
  - Perhaps after a TCP timeout
  - Perhaps server didn't see request
  - Perhaps server processed request but server/net failed before reply came back

Exactly-once?

- Need retransmission of at least once scheme

Exactly-once?

- Need retransmission of at least once scheme
- + Plus the duplicate filtering of at most once scheme
  - To survive client crashes, client needs to record pending RPCs on disk
    - So it can replay them with the same unique identifier

Exactly-once?

- Need retransmission of at least once scheme
- Plus the duplicate filtering of at most once scheme
  - To survive client crashes, client needs to record pending RPCs on disk
    - So it can replay them with the same unique identifier
- Plus story for making server reliable
  - Even if server fails, it needs to continue with full state
  - To survive server crashes, server should log to disk results of completed RPCs (to suppress duplicates)

Exactly-once for external actions?

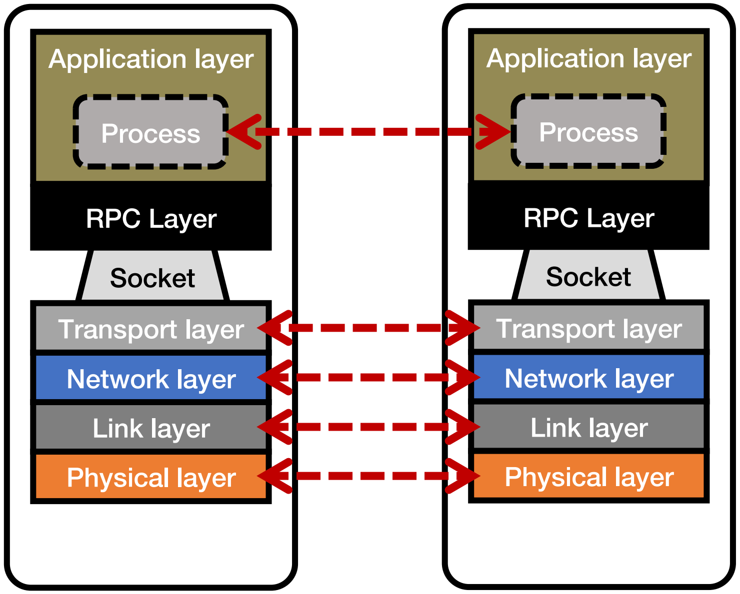
- Imagine that the remote operation triggers an external physical thing
  - e.g., dispense $100 from an ATM

- The ATM could crash immediately before or after dispensing and lose its state
  - Don't know which one happened
    - Can, however, make this window very small

- So can't achieve exactly-once in general, in the presence of external actions

Summary: Network comm. and RPCs

- Layers are our friends!
- RPCs are everywhere
- Necessary issues surrounding machine heterogeneity
- Subtle issues around failures
  - At-least-once w/ retransmission
  - At-most-once w/ duplicate filtering
    - Discard server state w/ cumulative acks
  - Exactly-once with:
    - at-least-once + at-most-once + fault tolerance + no external actions



Today's outline

1. Network sockets

2. Remote procedure call

3. RPCs in Go

Go RPCs

net/rpc

- Implementation in built-in library net/rpc

Go RPCs

- Implementation in built-in library net/rpc

- Write stub receiver methods of the form
  - `func (t *T) MethodName(args T1, reply *T2) error`

Go RPCs

- Implementation in built-in library net/rpc

- Write stub receiver methods of the form
  - `func (t *T) MethodName(args T1, reply *T2) error`

- Register receiver methods

Go RPCs

- Implementation in built-in library net/rpc
- Write stub receiver methods of the form
  - `func (t *T) MethodName(args T1, reply *T2) error`
- Register receiver methods
- Create a listener (i.e., server) that accepts requests

Writing a WordCount RPC server in Go

```go
type WordCountServer struct {
	addr string
}

type WordCountRequest struct {
	Input string
}

type WordCountReply struct {
	Counts map[string]int
}
```

Writing a WordCount RPC server in Go

```go
type WordCountServer struct {
	addr string
}

type WordCountRequest struct {
	Input string
}

type WordCountReply struct {
	Counts map[string]int
}
```

server

```go
func (*WordCountServer) Compute(
		request WordCountRequest,
		reply *WordCountReply) error {
	counts := make(map[string]int)
	input := request.Input
	tokens := strings.Fields(input)
	for _, t := range tokens {
		counts[t] += 1
	}
	reply.Counts = counts
	return nil
}
```

++

Writing a WordCount RPC server in Go

```go
type WordCountServer struct {
	addr string
}

type WordCountRequest struct {
	Input string
}

type WordCountReply struct {
	Counts map[string]int
}
```

```go
func (*WordCountServer) Compute(
		request WordCountRequest,
		reply *WordCountReply) error {
	counts := make(map[string]int)
	input := request.Input
	tokens := strings.Fields(input)
	for _, t := range tokens {
		counts[t] += 1
	}
	reply.Counts = counts
	return nil
}
```

Writing a WordCount RPC server in Go

"localhost:8888"

```
func (server *WordCountServer) Listen() {
      rpc.Register(server)
      listener, err := net.Listen("tcp", server.addr)
      checkError(err)
      go func() {
            rpc.Accept(listener)
      }()
}
```

Writing a WordCount RPC server in Go

```
func (server *WordCountServer) Listen() {
	rpc.Register(server)
	listener, err := net.Listen("tcp", server.addr)
	checkError(err)
	go func() {
		rpc.Accept(listener)
	}()
}
```

Writing a WordCount RPC server in Go

```
func (server *WordCountServer) Listen() {
      rpc.Register(server)
      listener, err := net.Listen("tcp", server.addr)
      checkError(err)
      go func() {
            rpc.Accept(listener)
      }()
}
```

Writing a WordCount RPC server in Go

```go
func (server *WordCountServer) Listen() {
	rpc.Register(server)
	listener, err := net.Listen("tcp", server.addr)
	checkError(err)
	go func() {
		rpc.Accept(listener)
	}()
}
```

WordCount client

```
func makeRequest(input string, serverAddr string) (map[string]int, error) {
      client, err := rpc.Dial("tcp", serverAddr)
      checkError(err)
      args := WordCountRequest{input}
      reply := WordCountReply{make(map[string]int)}
      err = client.Call("WordCountServer.Compute", args, &reply)
      if err != nil {
            return nil, err
      }
      return reply.Counts, nil
}
```

WordCount client

```
func makeRequest(input string, serverAddr string) (map[string]int, error) {
	client, err := rpc.Dial("tcp", serverAddr)
	checkError(err)
	args := WordCountRequest{input}
	reply := WordCountReply{make(map[string]int)}
	err = client.Call("WordCountServer.Compute", args, &reply)
	if err != nil {
		return nil, err
	}
	return reply.Counts, nil
}
```

WordCount client

```
func makeRequest(input string, serverAddr string) (map[string]int, error) {
	client, err := rpc.Dial("tcp", serverAddr)
	checkError(err)
	args := WordCountRequest{input}
	reply := WordCountReply{make(map[string]int)}
	err = client.Call("WordCountServer.Compute", args, &reply)
	if err != nil {
		return nil, err
	}
	return reply.Counts, nil
}
```

WordCount client

```go
func makeRequest(input string, serverAddr string) (map[string]int, error) {
	client, err := rpc.Dial("tcp", serverAddr)
	checkError(err)
	args := WordCountRequest{input}
	reply := WordCountReply{make(map[string]int)}
	err = client.Call("WordCountServer.Compute", args, &reply)
	if err != nil {
		return nil, err
	}
	return reply.Counts, nil
}
```

WordCount client-server

```go
func main() {
	serverAddr := "localhost:8888"
	server := WordCountServer{serverAddr}
	server.Listen()
	input1 := "hello I am good hello bye bye bye bye good night hello"
	wordcount, err := makeRequest(input1, serverAddr)
	checkError(err)
	fmt.Printf("Result: %v\n", wordcount)
}
```

WordCount client-server

```
func main() {
	serverAddr := "localhost:8888"
	server := WordCountServer{serverAddr}
	server.Listen()
	input1 := "hello I am good hello bye bye bye bye good night hello"
	wordcount, err := makeRequest(input1, serverAddr)
	checkError(err)
	fmt.Printf("Result: %v\n", wordcount)
}
```

```
Result: map[hello:3 I:1 am:1 good:2 bye:4 night:1]
```

Is this synchronous or asynchronous?

```go
func makeRequest(input string, serverAddr string) (map[string]int, error)
{
	client, err := rpc.Dial("tcp", serverAddr)
	checkError(err)
	args := WordCountRequest{input}
	reply := WordCountReply{make(map[string]int)}
	err = client.Call("WordCountServer.Compute", args, &reply)
	if err != nil {
		return nil, err
	}
	return reply.Counts, nil
}
```

Making client asynchronous

```
func makeRequest(input string, serverAddr string) chan Result {
      client, err := rpc.Dial("tcp", serverAddr)
      checkError(err)
      args := WordCountRequest{input}
      reply := WordCountReply{make(map[string]int)}









      return ch
}
```

Making client asynchronous

```go
func makeRequest(input string, serverAddr string) chan Result {
	client, err := rpc.Dial("tcp", serverAddr)
	checkError(err)
	args := WordCountRequest{input}
	reply := WordCountReply{make(map[string]int)}
	ch := make(chan Result)
	go func() {
		err := client.Call("WordCountServer.Compute", args, &reply)
		if err != nil {
			ch <- Result{nil, err} // something went wrong
		} else {
			ch <- Result{reply.Counts, nil} // success
		}
	}()
	return ch
}
```

Making client asynchronous

```
func makeRequest(input string, serverAddr string) *Call {
	client, err := rpc.Dial("tcp", serverAddr)
	checkError(err)
	args := WordCountRequest{input}
	reply := WordCountReply{make(map[string]int)}
	return client.Go("WordCountServer.Compute", args, &reply, nil)
}
```

Making client asynchronous

```
func makeRequest(input string, serverAddr string) *Call {
	client, err := rpc.Dial("tcp", serverAddr)
	checkError(err)
	args := WordCountRequest{input}
	reply := WordCountReply{make(map[string]int)}
	return client.Go("WordCountServer.Compute", args, &reply, nil)
}
```

```
call := makeRequest(...)
<-call.Done
checkError(call.Error)
handleReply(call.Reply)
```