Networking & RPC

CS6450: Distributed Systems

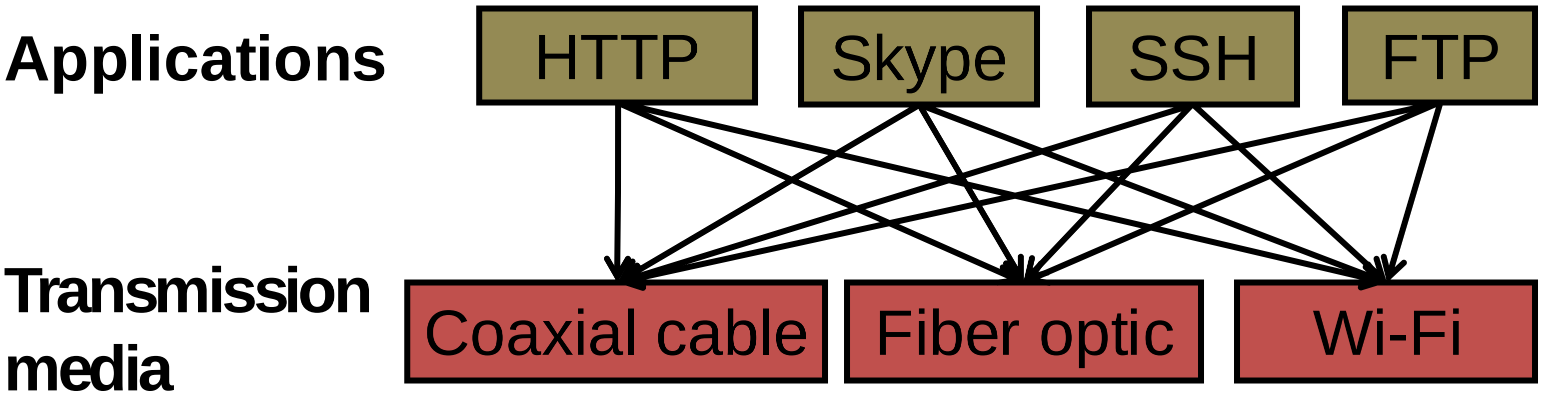
Slide Deck 2

Ryan Stutsman

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?*
    - *Which process on B is the intended receiver?*

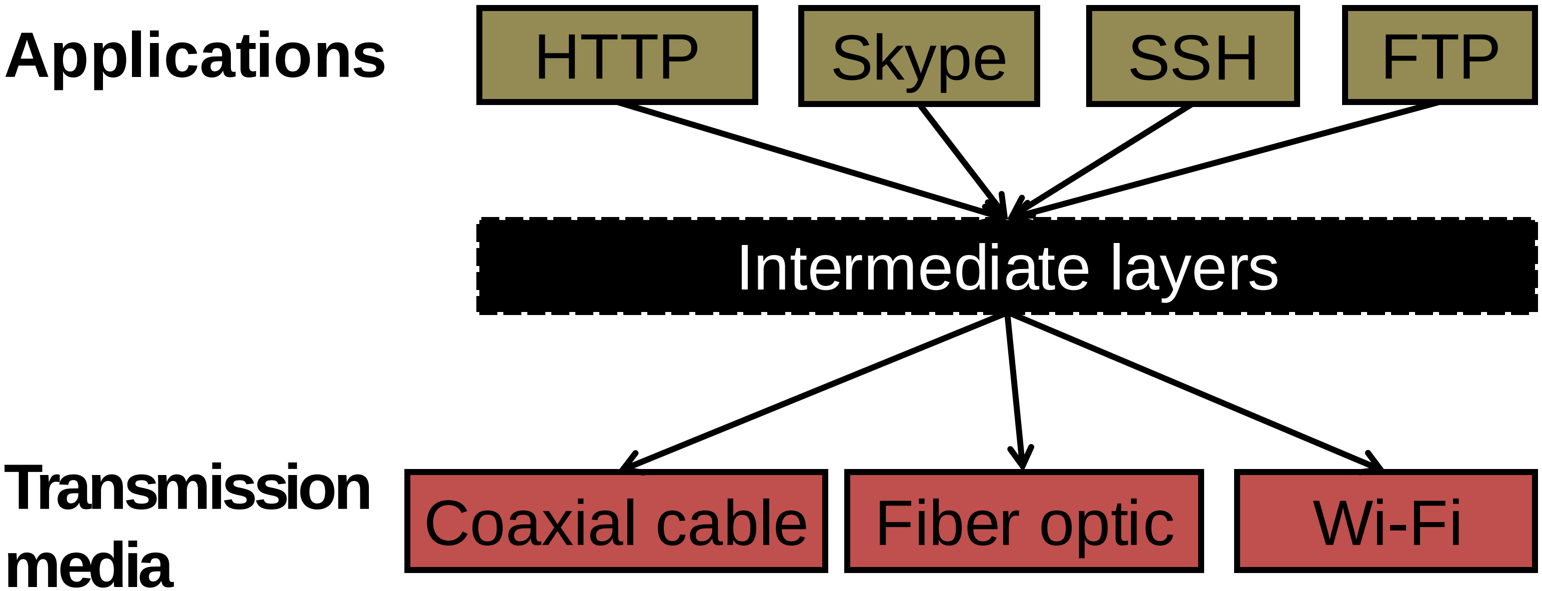
The problem of communication



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

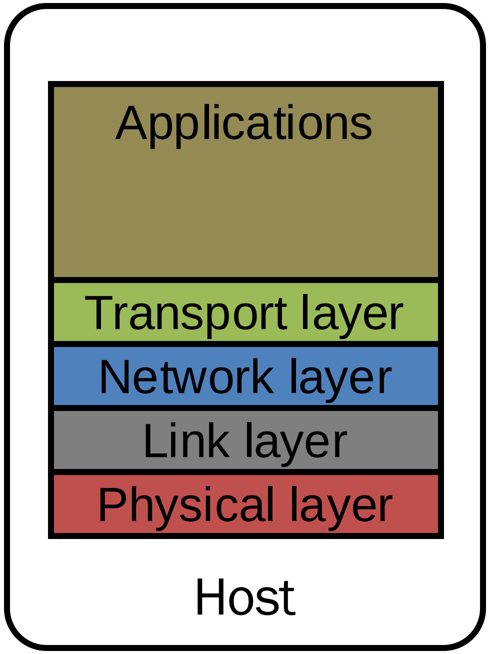
- **No!** But how does the Internet design avoid this?

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

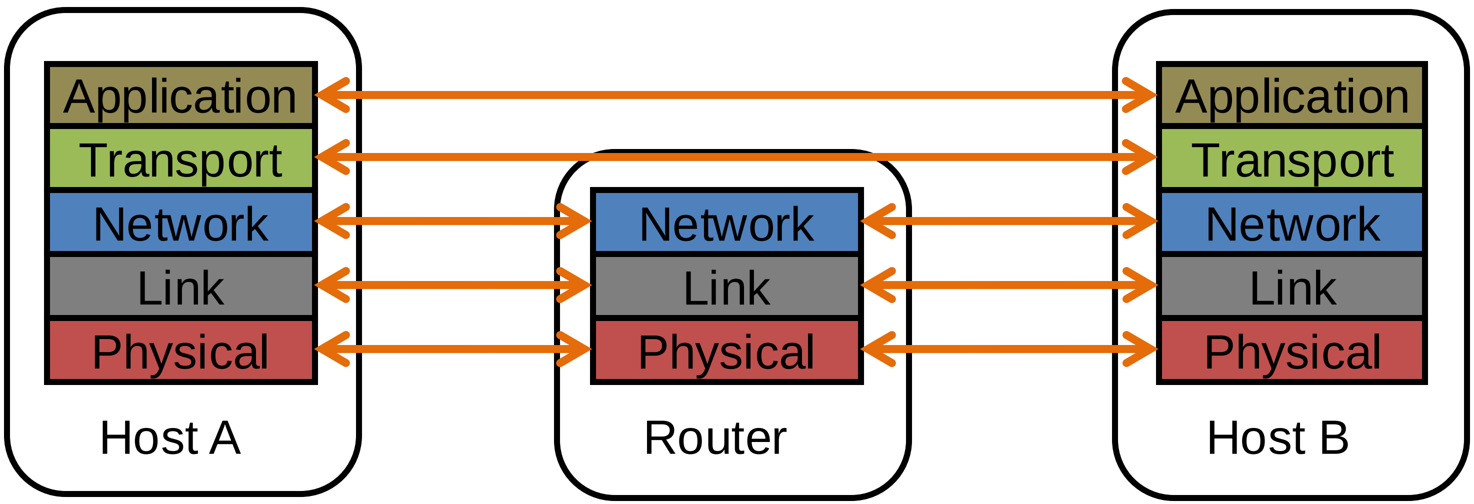
Layering in the Internet



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

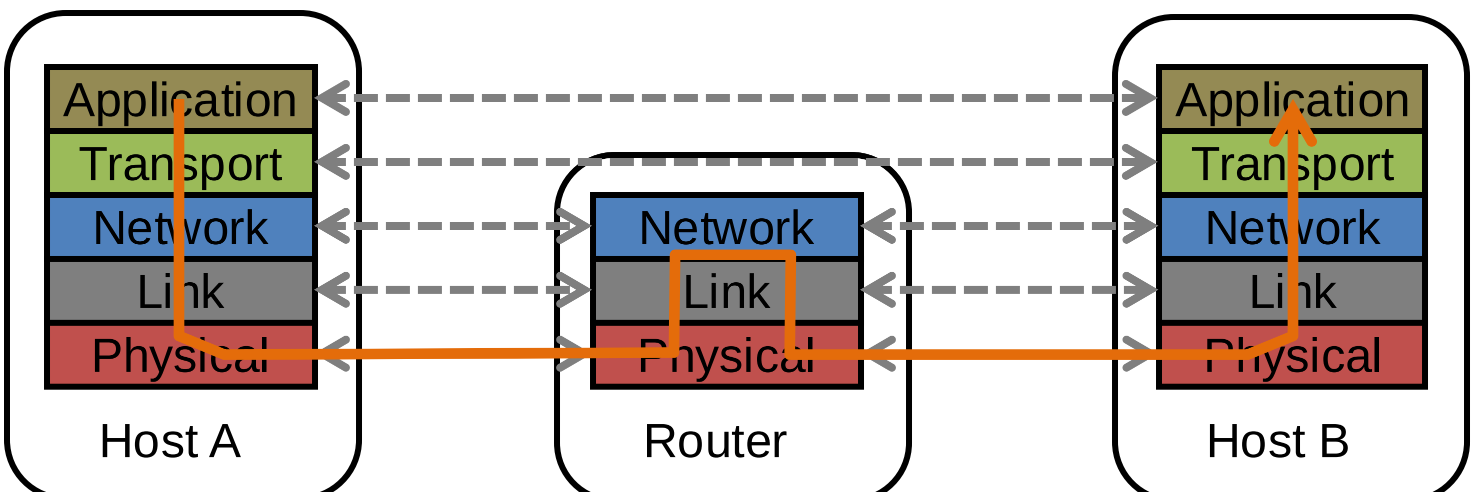
Logical communication between layers

- ***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***



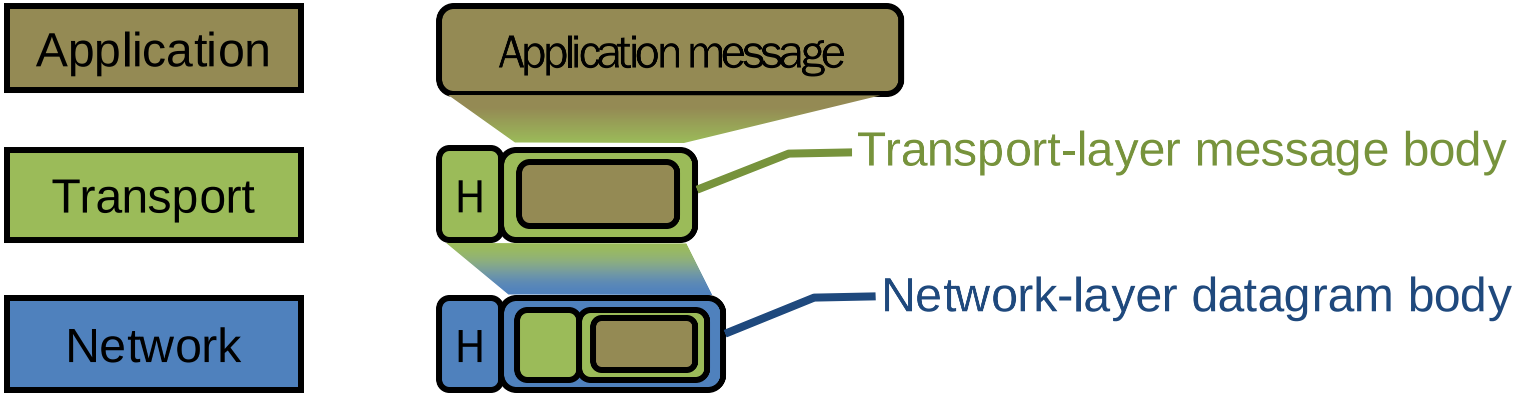
Physical communication

- 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



Transport: UDP

User Datagram Protocol

- Process-to-process datagrams/messages
- Does not handle loss, reordering, flow control, congestion control
- Unreliable, best effort

Header

- Source/Destination Port
- Length
- Checksum

Transport: TCP

Transmission Control Protocol

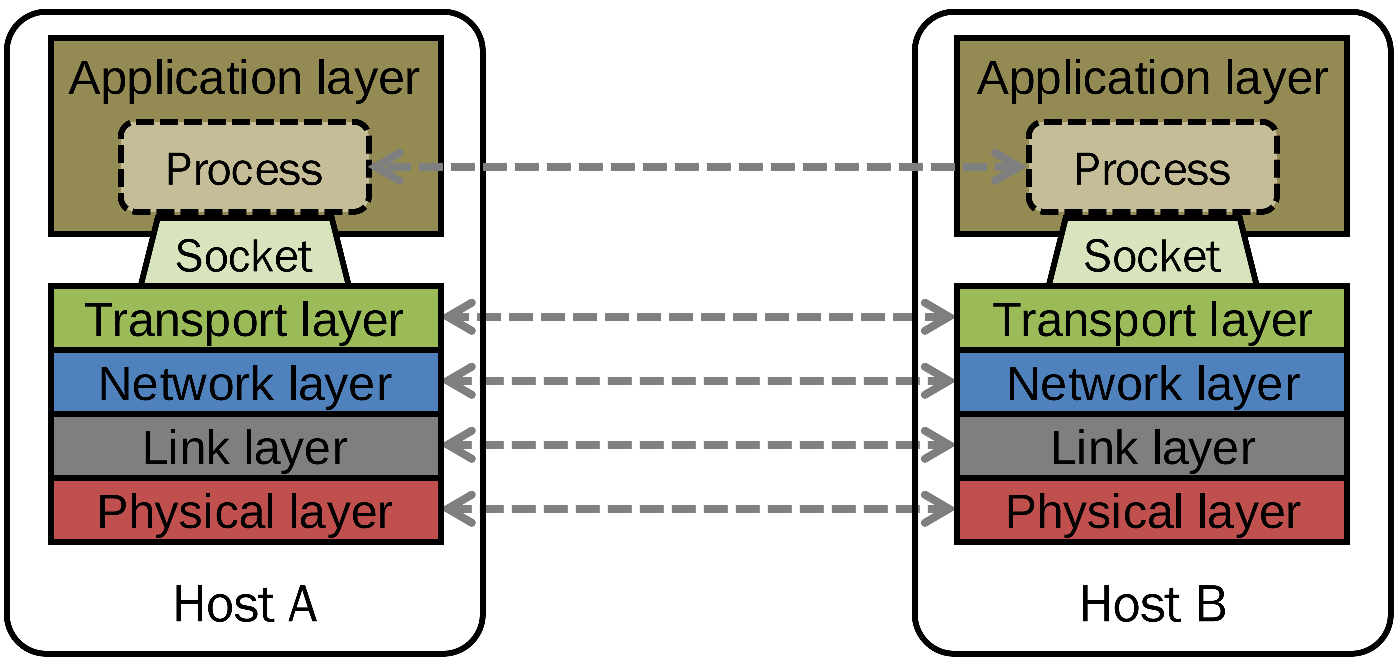
- Process-to-process “connections”: stream of bytes abstraction
- Reliable and in-order (using acks, retransmission, and dedup)

Header

- Source/Destination Port
- Sequence Number (loss detection and dedup)
- Ack Number (loss detection)
- Window Size (flow/congestion control)
- Checksum
- …

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: Poor Transparency

- **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?

```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

Today's outline

1. Network Sockets
2. Remote Procedure Call

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 assignments use RPC
  - Everyone does: Google (gRPC), Facebook (Thrift), Twitter, …

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

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 $\approx$ 10 cycles $\approx$ 3 ns
   - RPC in a data center takes $\approx$ 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:**
  - 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

- 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

- Mechanism to pass procedure parameters and return values in a machine-independent way
- And it eliminates hand-coding sockets/serialization code

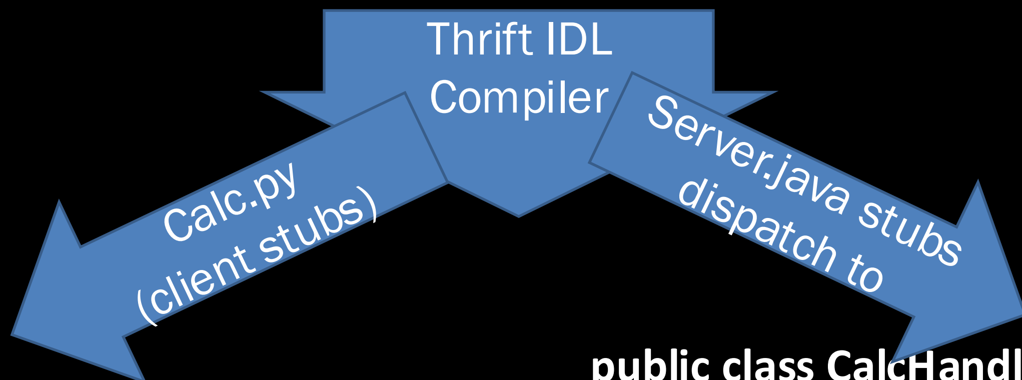
- Programmer may write an *interface description* in the IDL
  - Defines API for procedure calls: names, parameter/return types

- 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

```
service Calc extends shared.SharedService {
        void ping(),
        i32 add(1:i32 num1, 2:i32 num2),
        i32 calculate(1:i32 logid, 2:Work w)
                throws (1:InvalidOperation ouch),
}
```

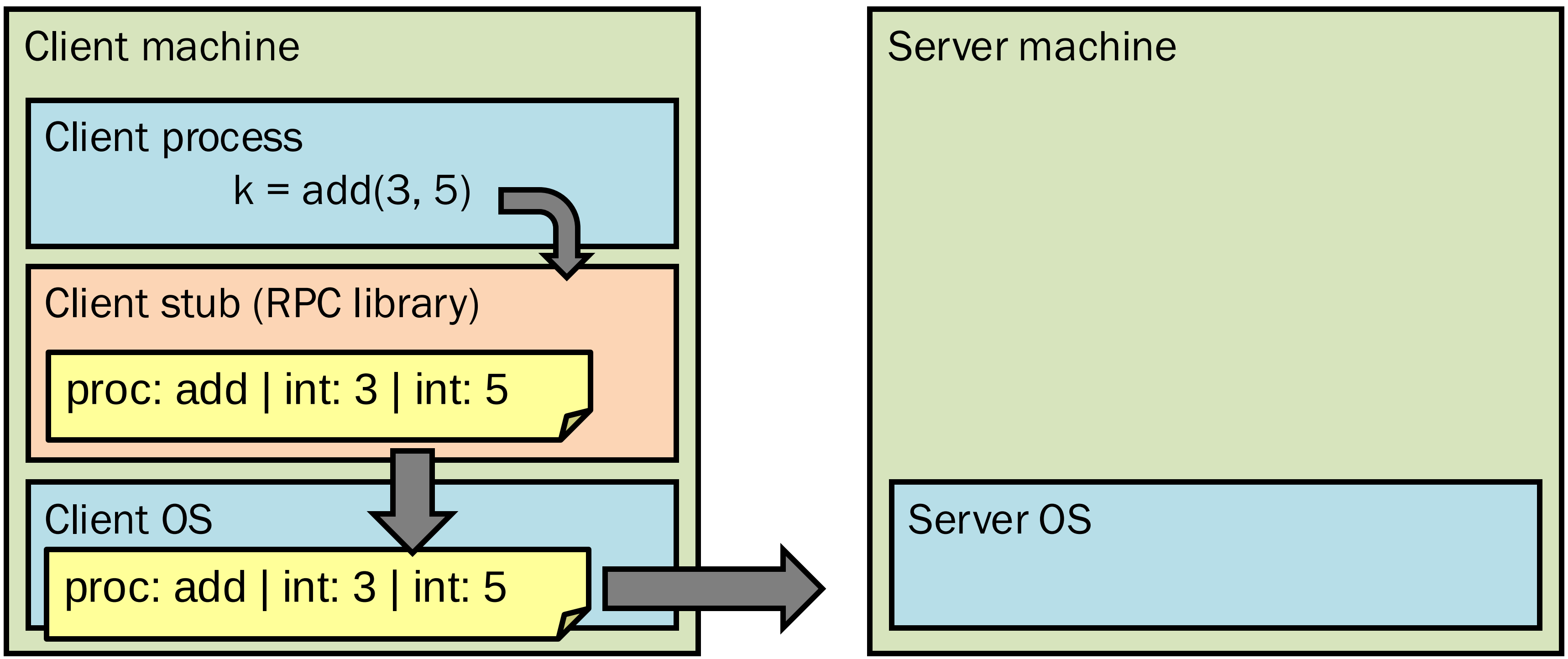



```
def main():
  t = TSocket('localhost', 9090)
  client = Calc.Client(t)
  t.open()
  client.ping()
  print('ping()')
  sum_ = client.add(1, 1)
```

```
public class CalcHandler
  implements Calc.Iface {
  void ping() {
    println("ping()");
  }

  int add(int n1, int n2) {
    return n1 + n2;
  }
}
```

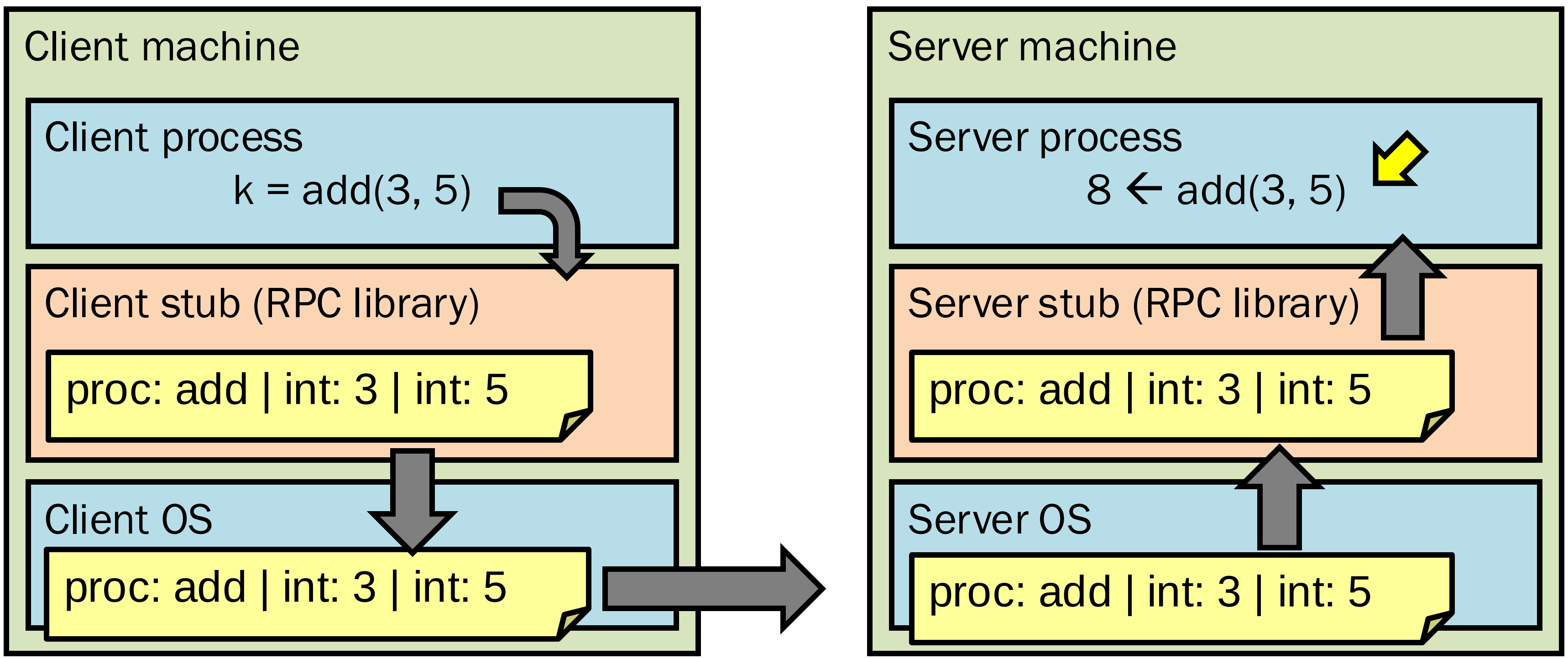
A day in the life of an RPC

1. Client calls stub function (pushes params onto stack)
2. Stub marshals parameters to a network message
3. OS sends a network message to the server



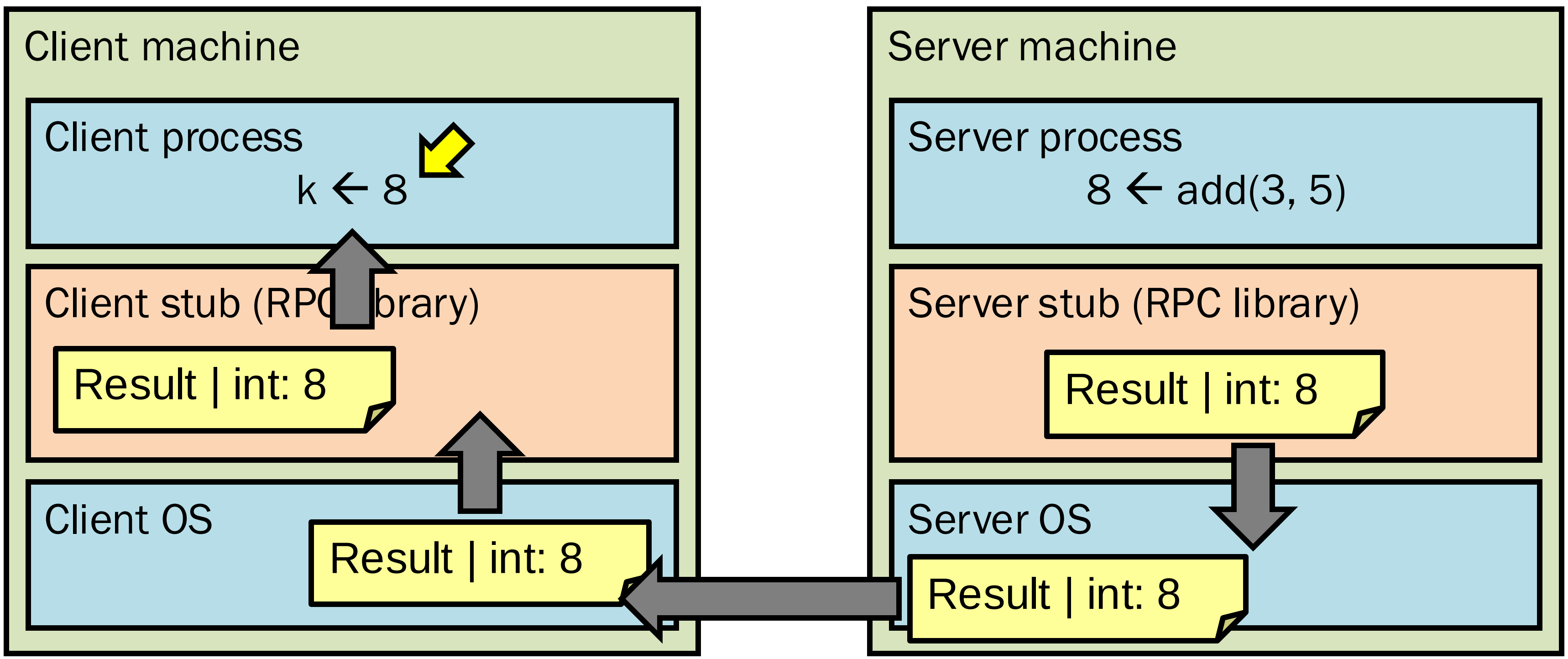
A day in the life of an RPC

4. Server OS receives message, sends it up to stub
5. Server stub unmarshals params, calls server function
6. Server function runs, returns a value



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
9. Client OS receives the reply and passes up to stub
10. Client stub unmarshals return value, returns to client



Failures & RPC Semantics

What could go wrong?

1. Client may **crash and reboot**

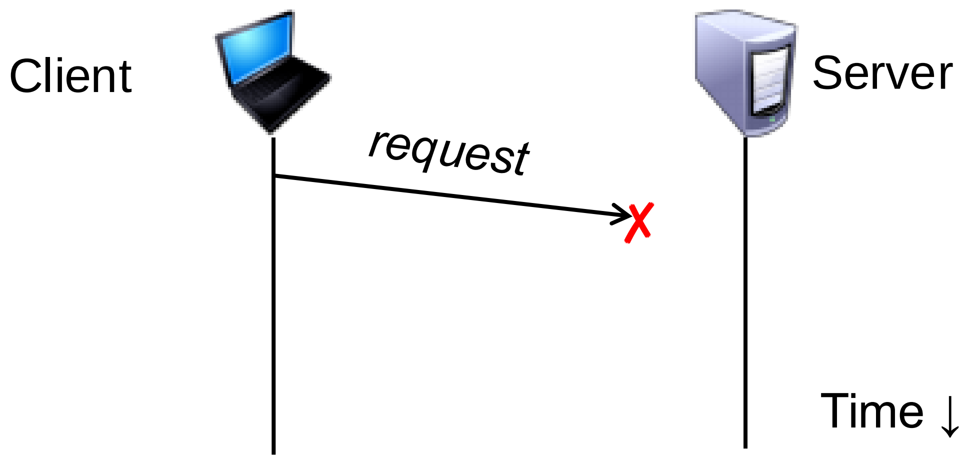
2. Packets may be **dropped**
   - Some individual **packet loss** in the Internet
   - **Broken routing** results in many lost packets

3. Server may **crash and reboot**

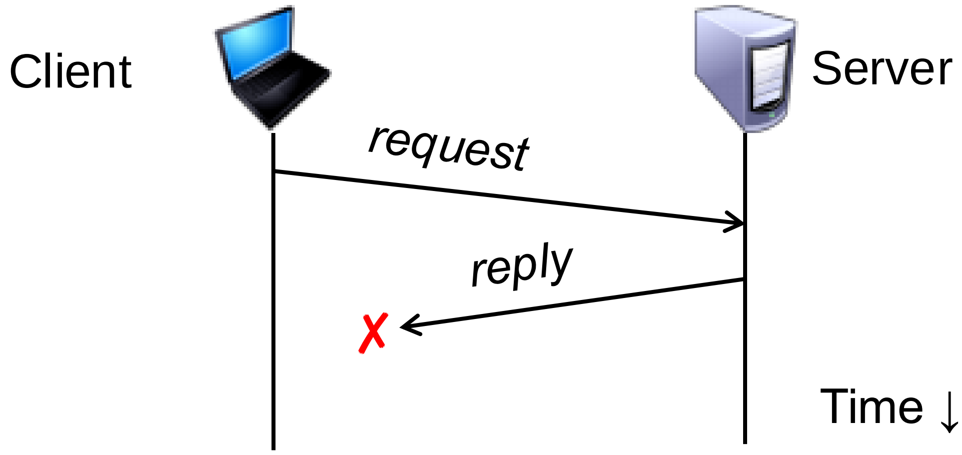
4. Network or server might just be **very slow**

All these may look the same to the client…

Failures, from client's perspective



Failures, from client's perspective



The cause of the failure is hidden from the client!

At-Least-Once Semantics

- Simple scheme for handling failures

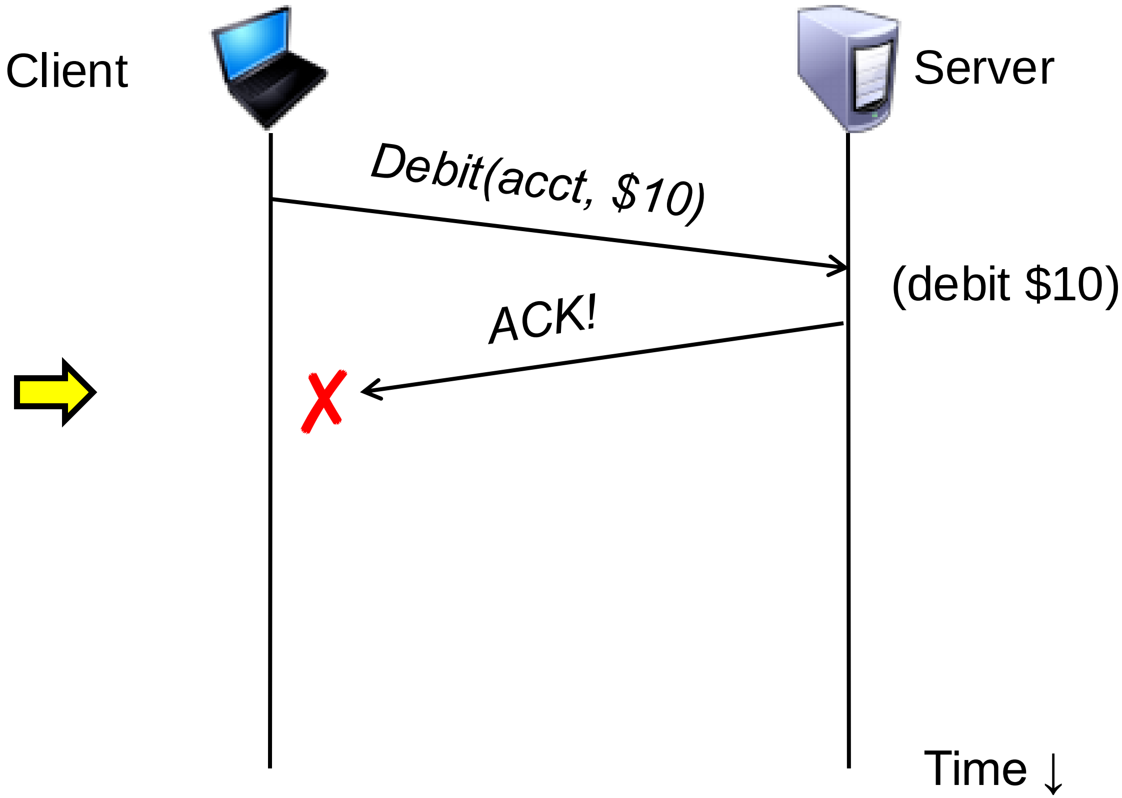
1. Client creates a unique seqnum for request so it can match the reply with its request

2. Sends request and **waits for a response**
   - Response takes the form of an *acknowledgement* message from the server stub

3. If no response arrives after a fixed *timeout*, then client stub go to (2)

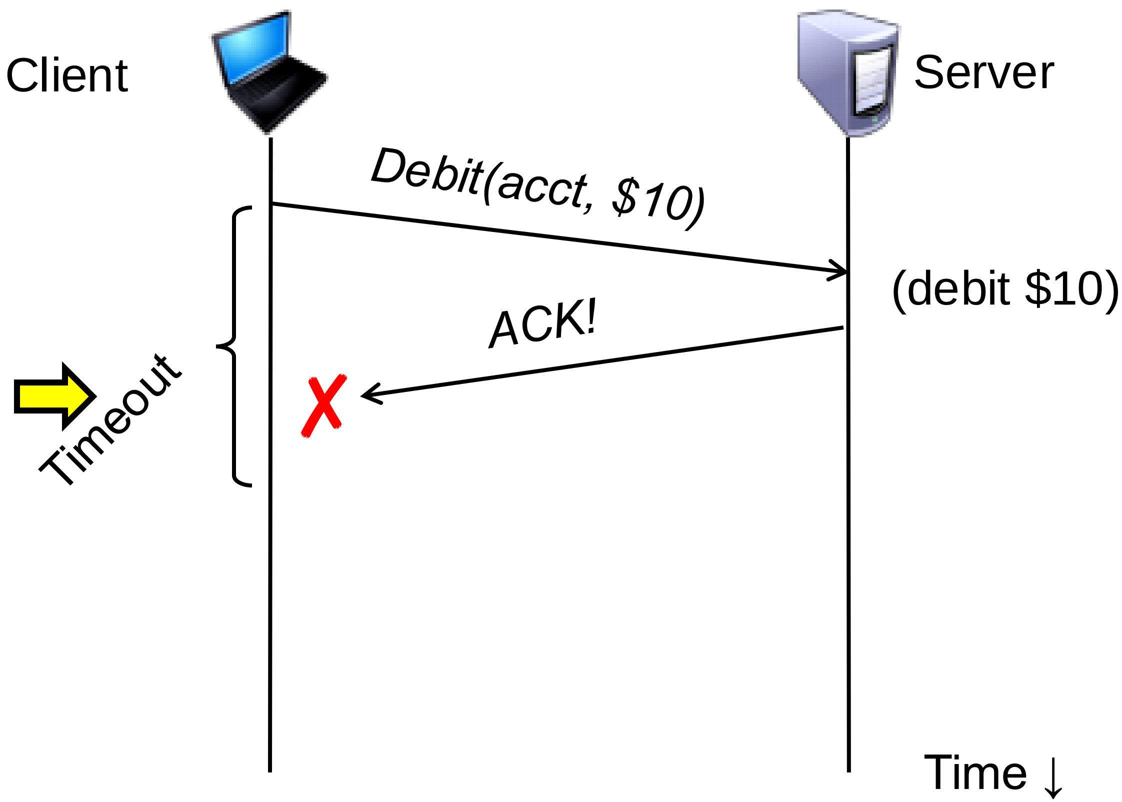
At-Least-Once and side effects

- Client sends a "debit $10 from bank account" RPC



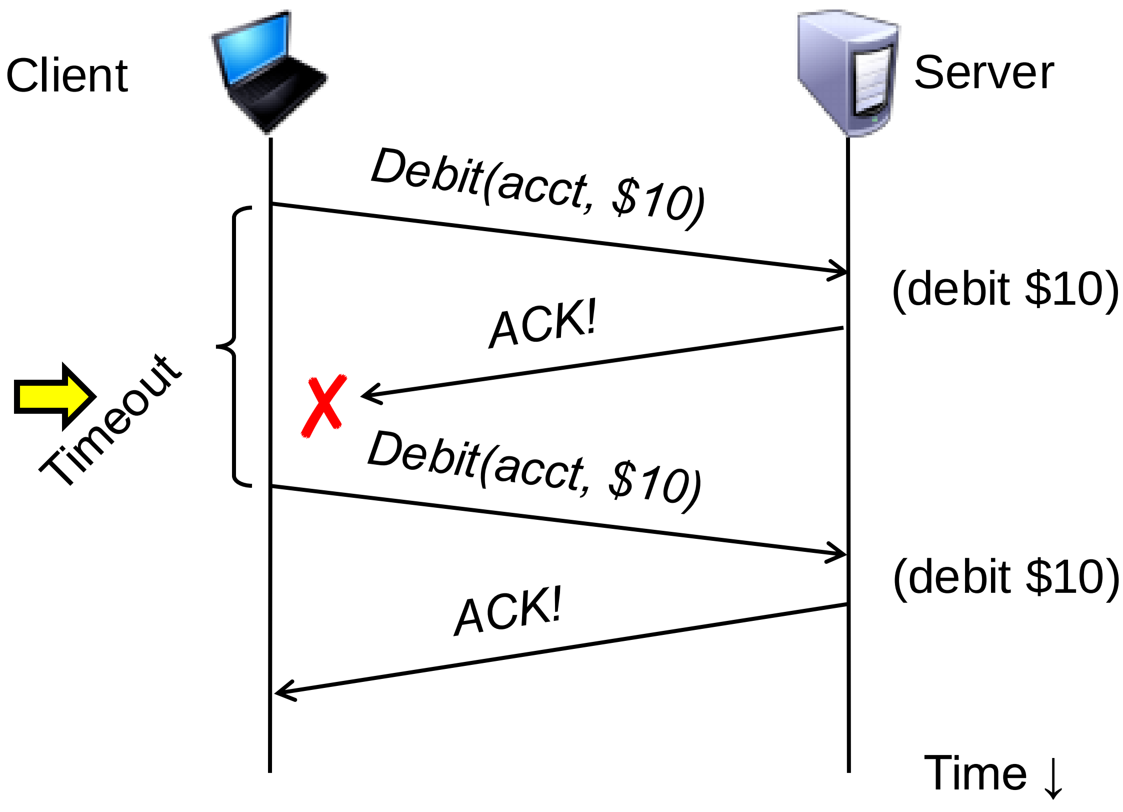
At-Least-Once and side effects

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



At-Least-Once and side effects

- Client sends a "debit $10 from bank account" RPC



Is At-Least-Once 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

Idempotence

Coulouris book:

An *idempotent operation* is an operation that can be performed repeatedly with the same effect as if it had been performed exactly once.

For example, an operation to add an element to a set is an idempotent operation because it will always have the same effect on the set each time it is performed, whereas an operation to append an item to a sequence is not an idempotent operation because it extends the sequence each time it is performed.

A server whose operations are all idempotent need not take special measures to avoid executing its operations more than once.

At-Most-Once scheme

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

- *How do we detect a duplicate request?*
  - Server sees same function, same arguments twice?
  - No good; submitting the same RPC twice might indicate the intent to do the thing twice!

At-Most-Once scheme

- *How do we detect a duplicate request?*
  - Client includes unique ***transaction ID*** (*xid*) with each one of its 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: Ensuring unique XIDs

- *How do we ensure that the xid is unique?*

1. Big random number

2. Combine unique client ID with a sequence number
    - e.g. 128-bit id chosen at random
    - Suppose the client crashes and restarts. *Can it reuse the same client ID?*

At-Most-Once: Discarding server state

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

- Suppose xid = ⟨unique client id, seqnum⟩
  - *e.g.* ⟨42, 1000⟩, ⟨42, 1001⟩, ⟨42, 1002⟩

- Server tracks most recently processed seqnum per client
  - If request seqnum < most recent, then discard request
  - If request seqnum == most recent, then send most recent reply
  - 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 do we 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?*

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

- Lab 2 doesn't handle server crash, so you don't need to worry about this (yet!)

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 processed request but server/net failed before reply came back
  - If you create a new connection and retry you lose at-most-once semantics

RPC Transport: TCP or UDP?

TCP

- Stream but can send large chunks
- Reliable connection
- Orders pipelined requests
- All packets acked
- ✅ Block client if server busy
- ✅ Block on congestion and "incast"

UDP

- Messages "framed" but size limit
- Apps must handle loss anyway
- Reordering; must handle anyway
- ✅ Can use reply as ack
- ❌ Client must guess on when to retry: lost? busy? congested?

RPC and Assignments

- Go's RPC semantics aren't enough for Lab 3
  - If one node doesn't respond, client re-sends to another
    - Go RPC can't detect this kind of duplicate
  - Breaks at-most-once semantics
- You will need to explicitly detect duplicates using something like what we've talked about

Exactly-once?

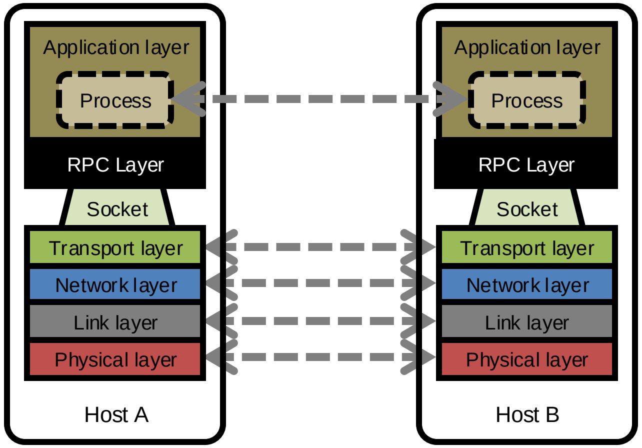
- Need retransmission of at least once scheme
- Plus the duplicate filtering of at most once scheme
- Plus story for making server reliable
  - Even if server fails, it needs to continue with full state
  - To survive **server** crashes, server should log to disk results of completed RPCs (to suppress duplicates) and **effects/record of effects must be atomic**
- But, client crashes are still problematic even with these steps!
  - Client needs to record pending RPCs on disk?
    - So it can replay them with the same unique identifier
  - This may not be enough – in the real world clients have users
  - Users may retry elsewhere using a different client
    - Open a new tab and try there, hit refresh, etc.
  - Can client even restore itself to correct points in code?
  - Often, client crash isn't handled, so degrades to at-most-once

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
- **Can't achieve exactly-once in general,** in the presence of external actions

Summary: Networking & RPCs

- Layers are our friends!
- RPCs are everywhere
- Covers over 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 + atomicity + no external
      actions



```
package calc
type AddArgs struct {
  Left  int
  Right int
}
```

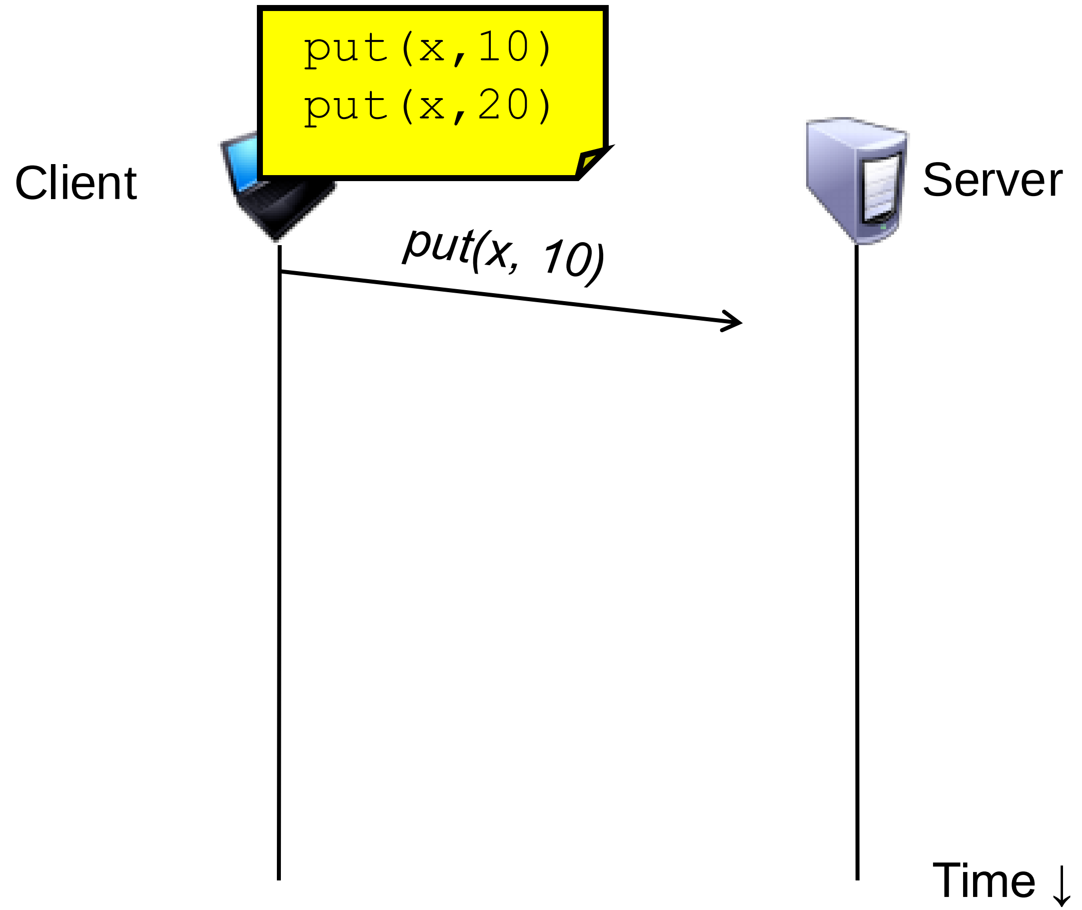
```
type Calc int

func (c *Calc) Add(args *calc.AddArgs,
                                  reply *int) error {
  *reply = args.Left + args.Right
  Printf("Server just added %v\n", args)
  return nil
}

func main() {
  c := new(Calc)
  rpc.Register(c)
  rpc.HandleHTTP()
  l, e := net.Listen("tcp",
                    "localhost:1234")
  if e != nil {
    log.Fatal("listen error:", e)
  }
  http.Serve(l, nil)
}
```

```go
package calc
type AddArgs struct {
  Left  int
  Right int
}
```

```go
func connect() *rpc.Client {
  client, err := rpc.DialHTTP("tcp",
                    "localhost:1234")
  if err != nil {
    log.Fatal("dialing:", err)
  }
  return client
}

var client *rpc.Client

func Add(left int, right int) int {
  args := &calc.AddArgs{left, right}
  var reply int
  err := client.Call("Calc.Add",
                   args, &reply)
  if err != nil {
    log.Fatal("arith error:", err)
  }
  return reply
}

func main() {
  client = connect()
  r := Add(3, 4)
  fmt.Printf("3 + 4 = %v\n", r)
}
```

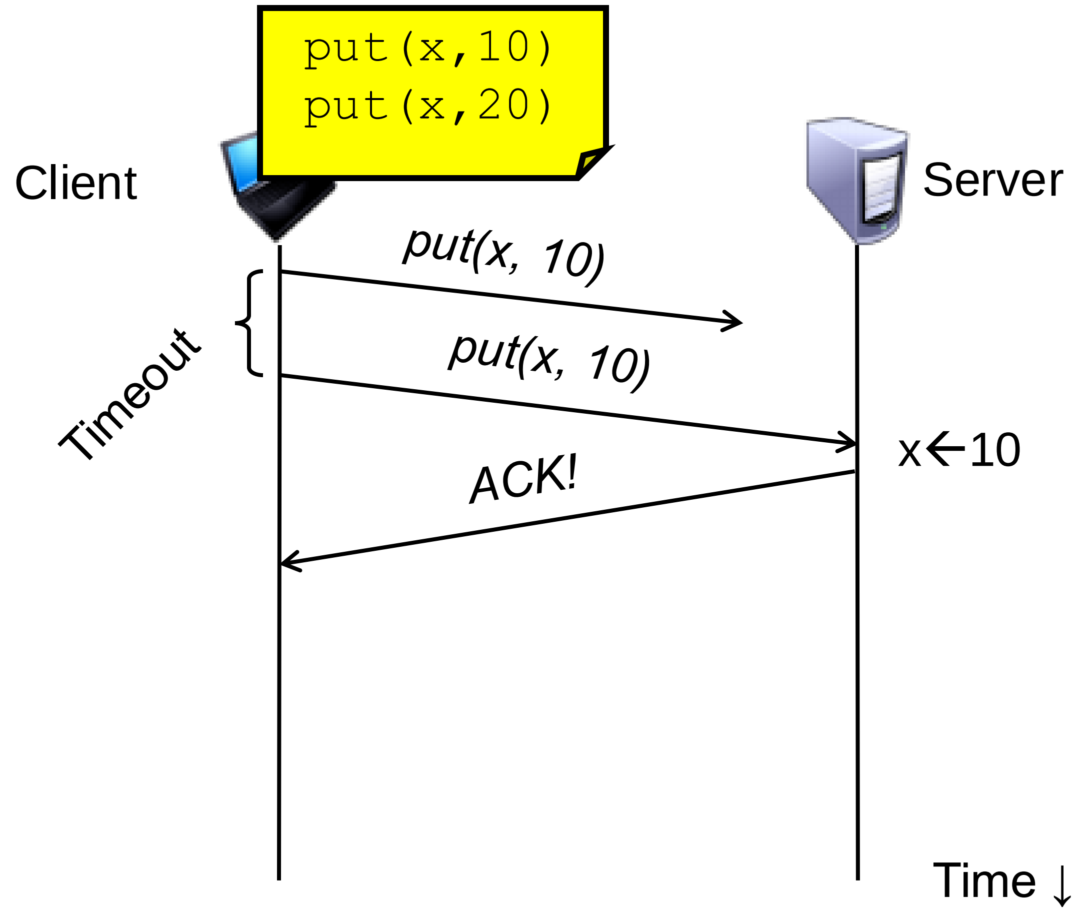
At-Least-Once and writes

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



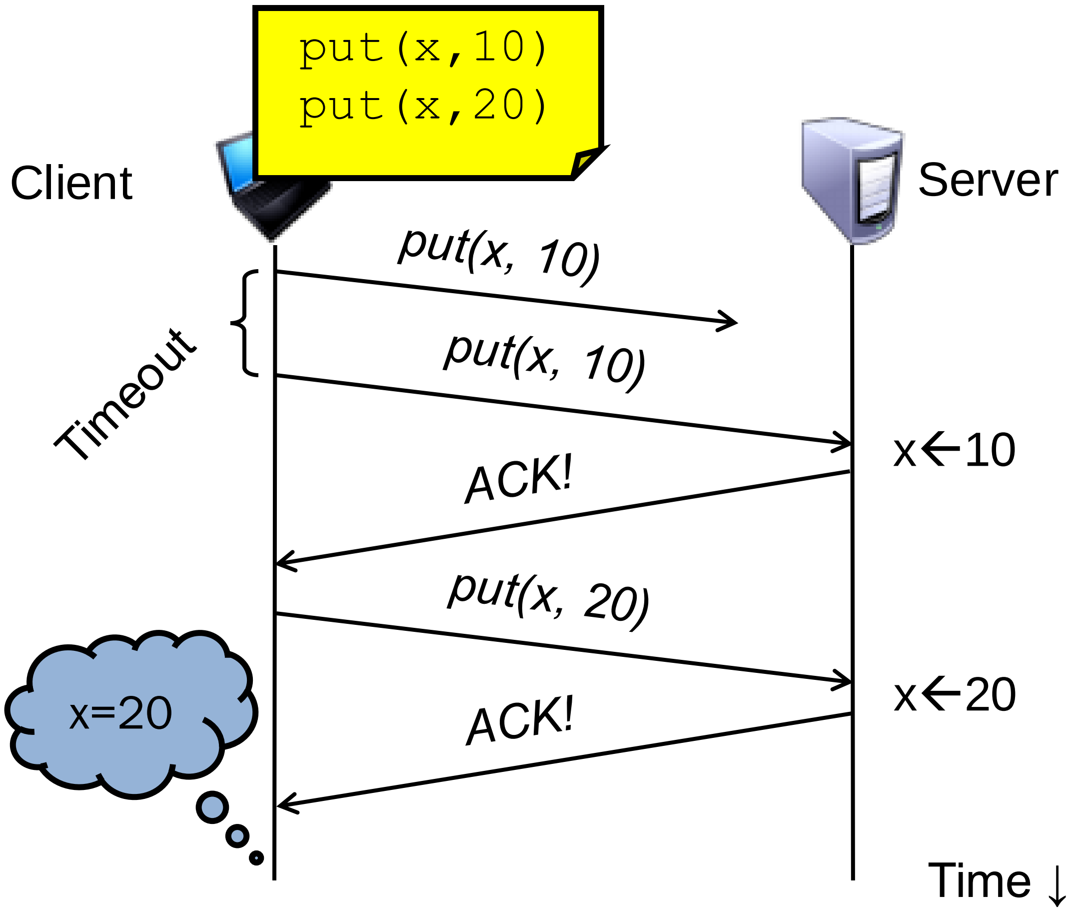
At-Least-Once and writes

- 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*