UNIVERSITY of VIRGINIA

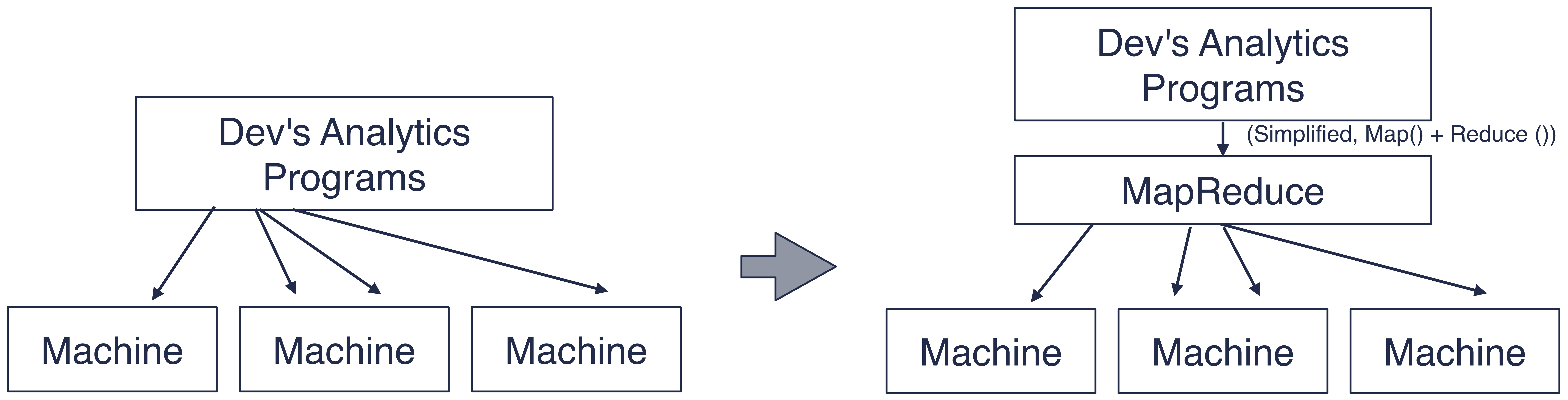
CS4740
CLOUD COMPUTING

RPC (Remote Procedure Call)

Prof. Chang Lou, UVA CS, Fall 2025

CONTEXT

- Previously: how we ease distributed analytics w/ MapReduce.
- The insight is to raise the level of **abstraction**.



CONTEXT

- How do we build distributed frameworks like MapReduce?
- Today we look at the most basic DS abstraction: RPC, the predominant communication abstraction in a DS.

AGENDA

– Motivation of RPC

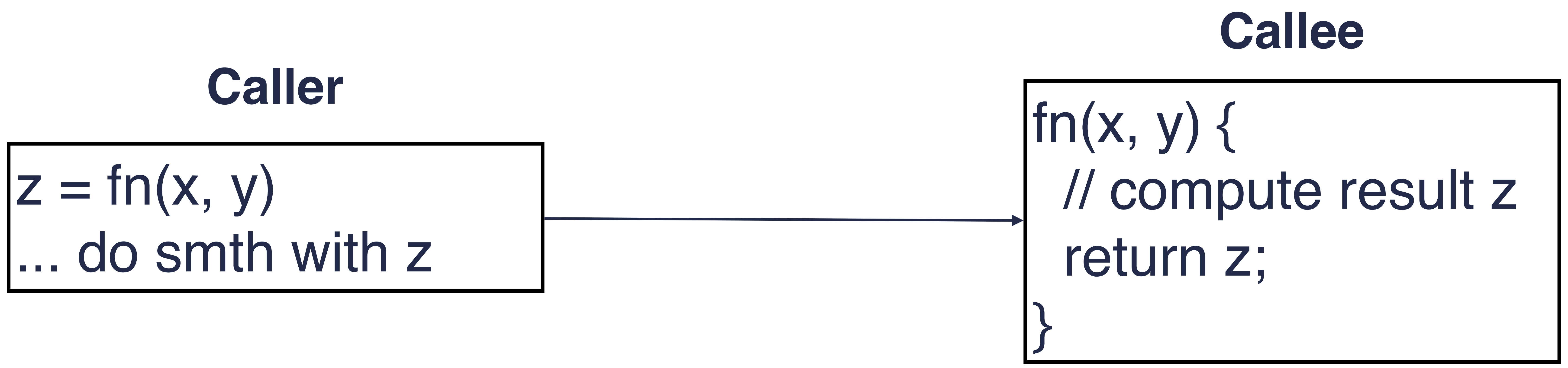
– How RPC works

– Dealing with Failures

What is "R"PC?

FROM LPC TO RPC

– Before talking about RPC (remote procedure call), let's talk about LPC (local procedure call).

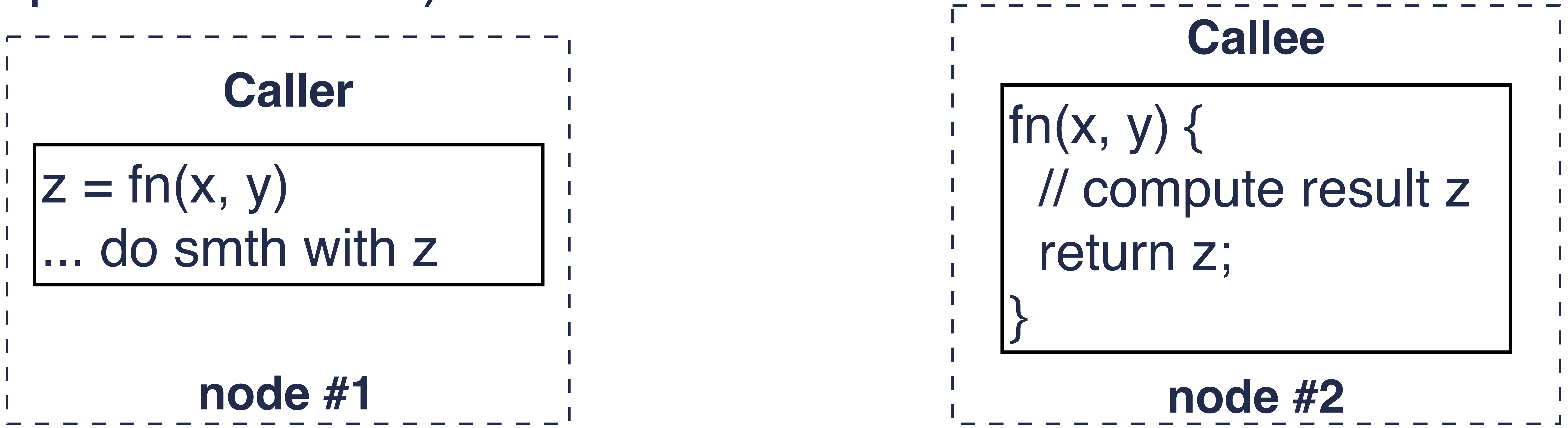


```
z = fn(x, y)
... do smth with z
```

```
fn(x, y) {
  // compute result z
  return z;
}
```

FROM LPC TO RPC

– Before talking about RPC (remote procedure call), let's talk about LPC (local procedure call).



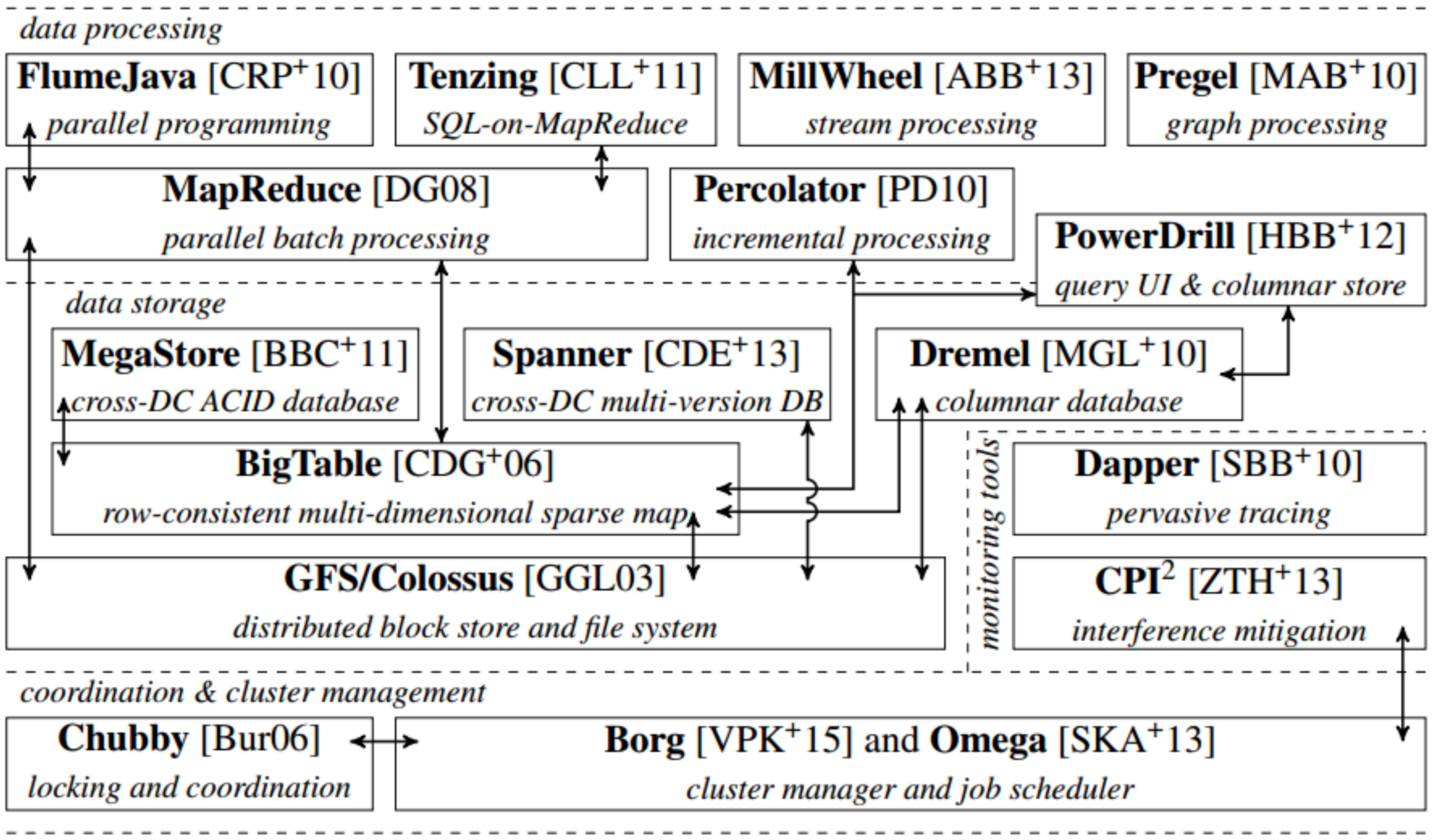
```
z = fn(x, y)
... do smth with z
```

```
fn(x, y) {
  // compute result z
  return z;
}
```

What if they are in two different machines?

EXAMPLE: (PART OF) GOOGLE INFRA STACK

Malte Schwarzkopf, "What does it take to make Google work at scale?"



IMPLEMENTING NEW PROCEDURE CALL

- Now procedure calls are no longer simple..
- What are the issues?

```
struct foomsg {
  u_int32_t len;
}
send_foo(int outsock, char* contents) {
  int msglen = sizeof(struct foomsg) +
                    strlen(contents);
  char* buf = malloc(msglen);
  struct foomsg* fm = (struct foomsg*)buf;
  fm->len = htonl(strlen(contents));
  memcpy(buf + sizeof(struct foomsg),
             contents, strlen(contents));
  write(outsock, buf, msglen);
}
```

IMPLEMENTING NEW PROCEDURE CALL

- Now procedure calls are no longer simple..
- What are the issues?
  - Lots of ugly boilerplate
  - Prone to bugs, vulnerabilities
  - Portability issues
  - Hard to understand/maintain/evolve

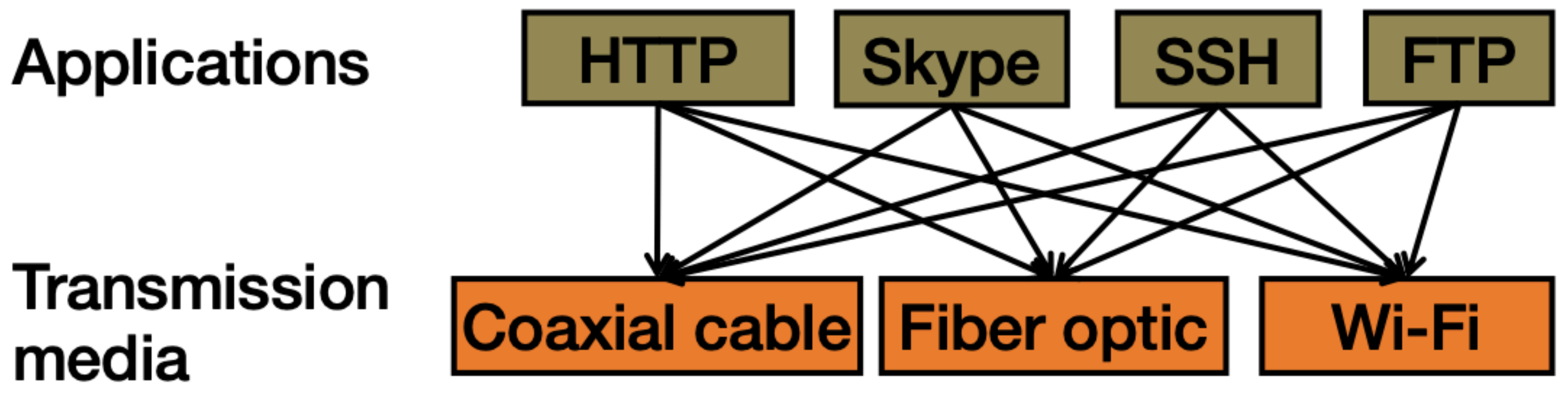
```
struct foomsg {
  u_int32_t len;
}
send_foo(int outsock, char* contents) {
  int msglen = sizeof(struct foomsg) +
                      strlen(contents);
  char* buf = malloc(msglen);
  struct foomsg* fm = (struct foomsg*)buf;
  fm->len = htonl(strlen(contents));
  memcpy(buf + sizeof(struct foomsg),
              contents, strlen(contents));
  write(outsock, buf, msglen);
}
```

WISDOM FROM THE PAST

— Before solutions in distributed systems were proposed, how was the problem of communication resolved in Internet?

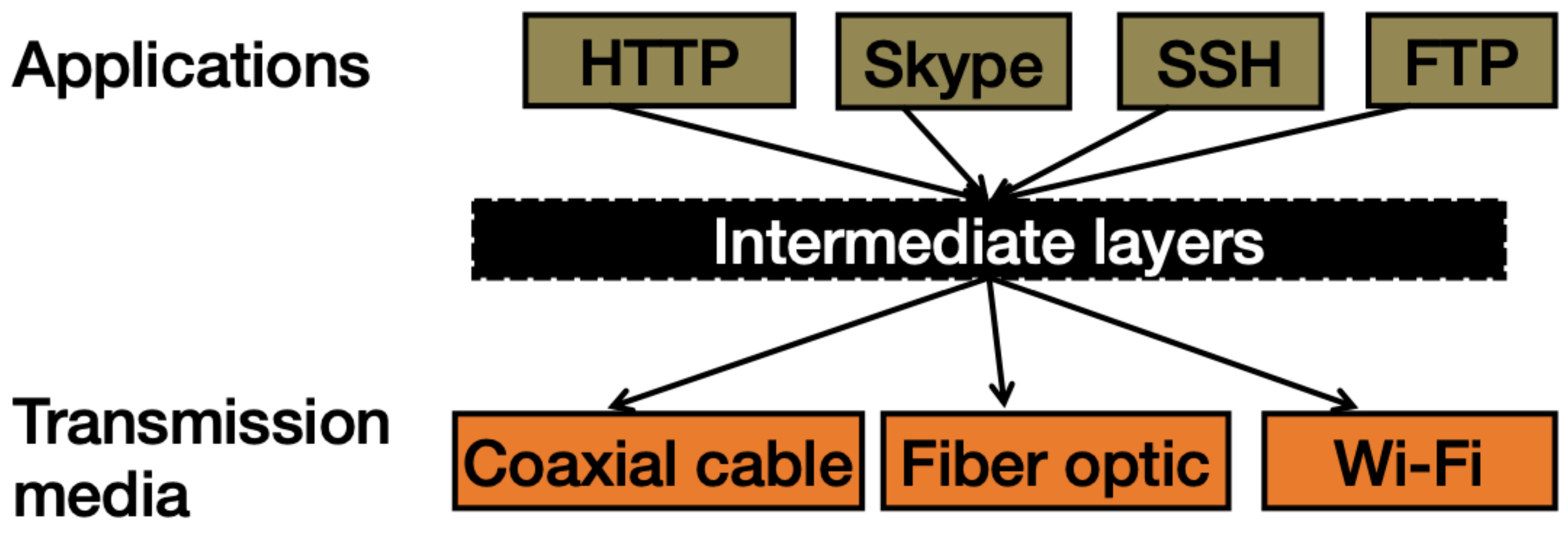
THE PROBLEM OF COMMUNICATION

- To coordinate, nodes must communicate.
- Problems
  - Re-implement every application for every new underlying transmission medium?
  - Change every application on any change to an underlying transmission medium?



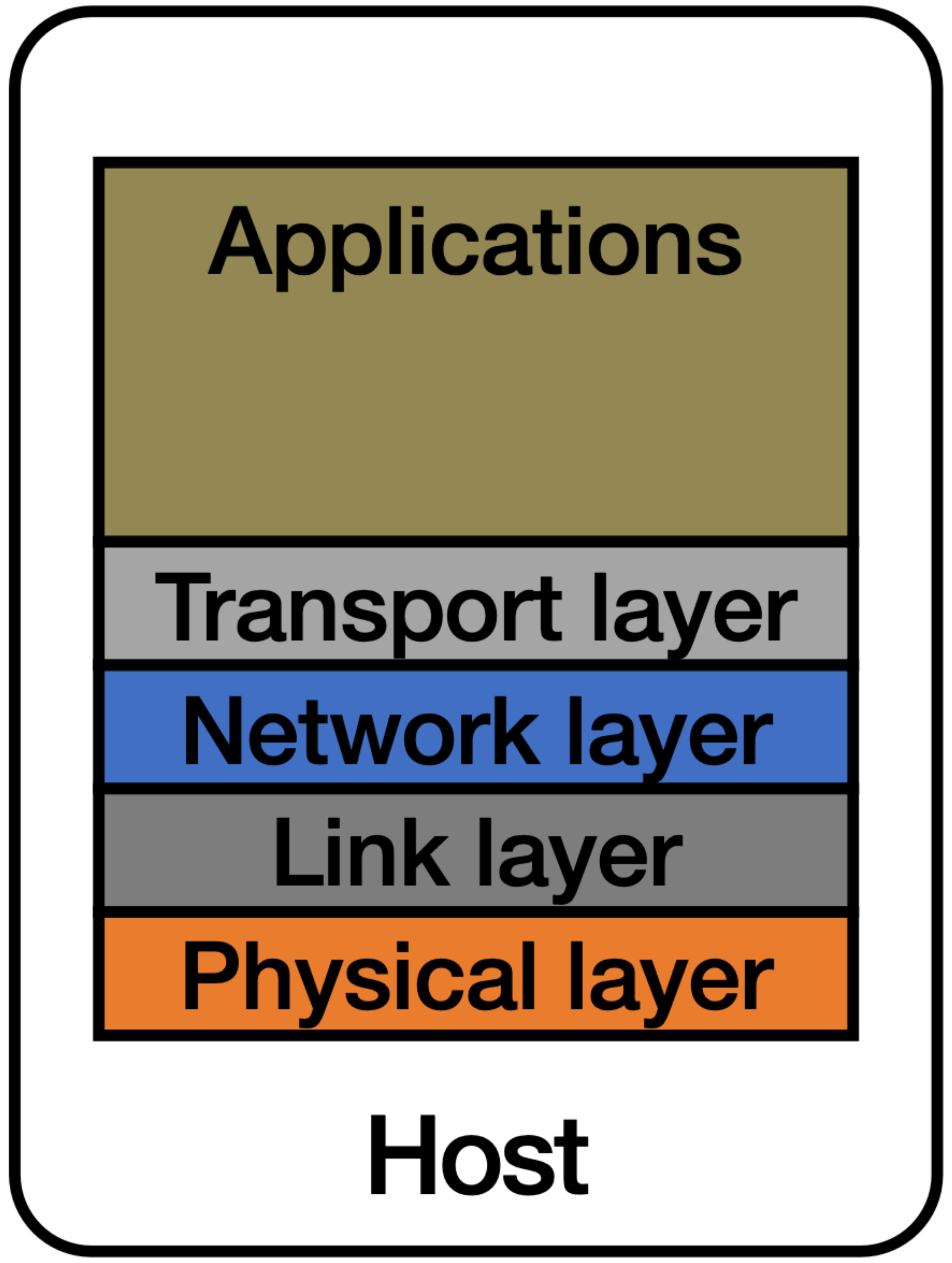
SOLUTION: LAYERING

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



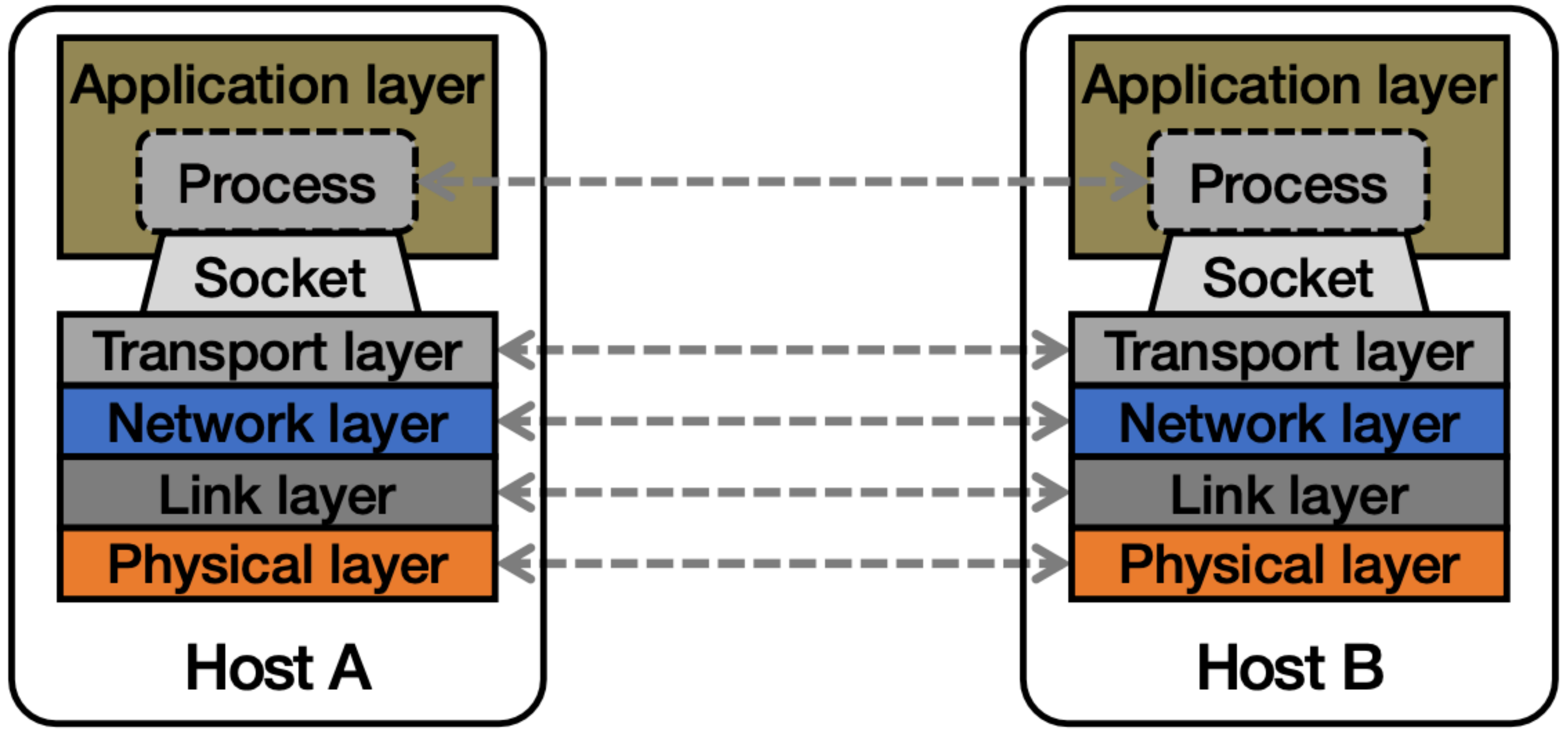
LAYERING IN THE INTERNET

- 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

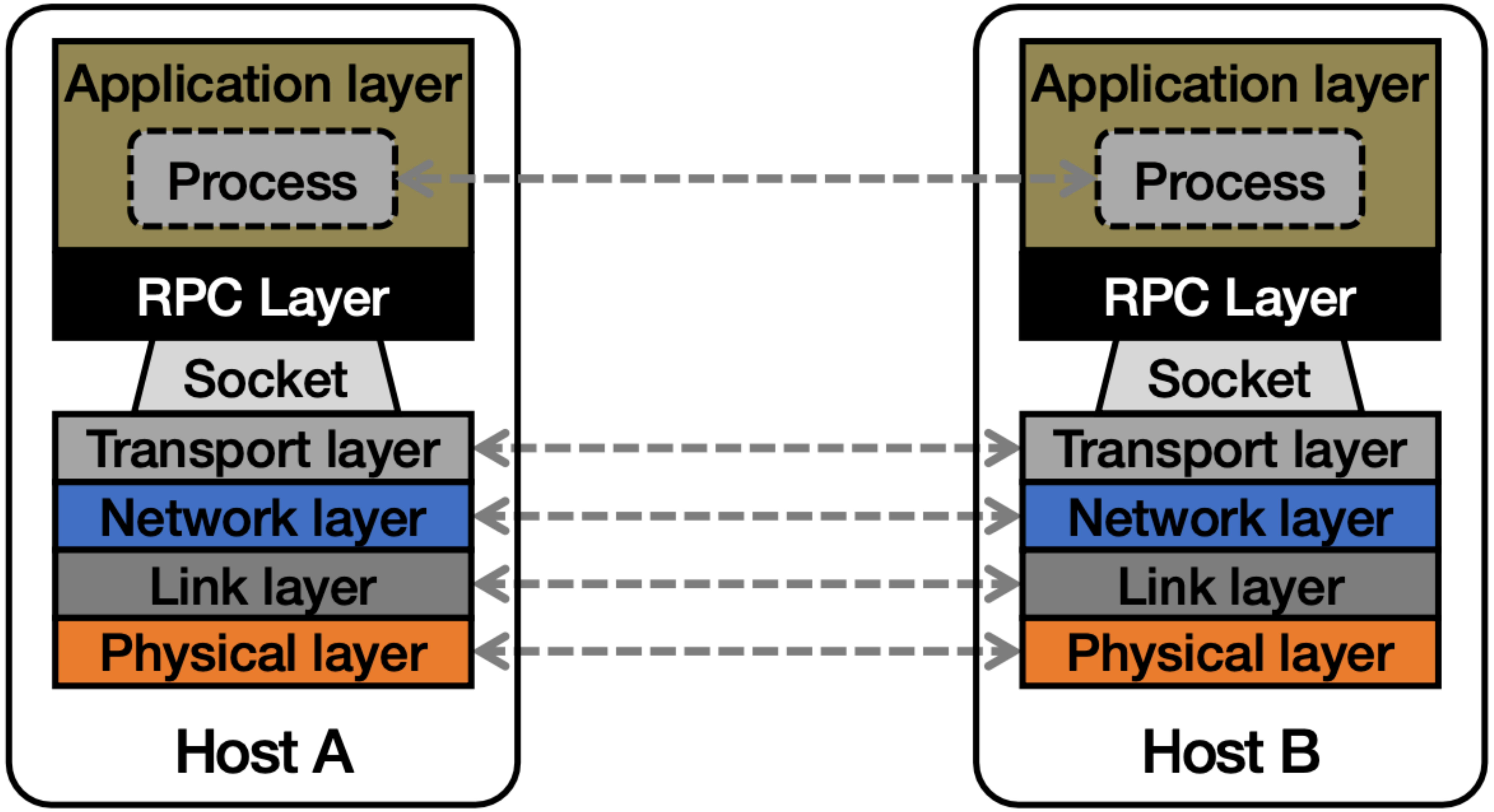


NETWORK SOCKET-BASED COMMUNICATION

- 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



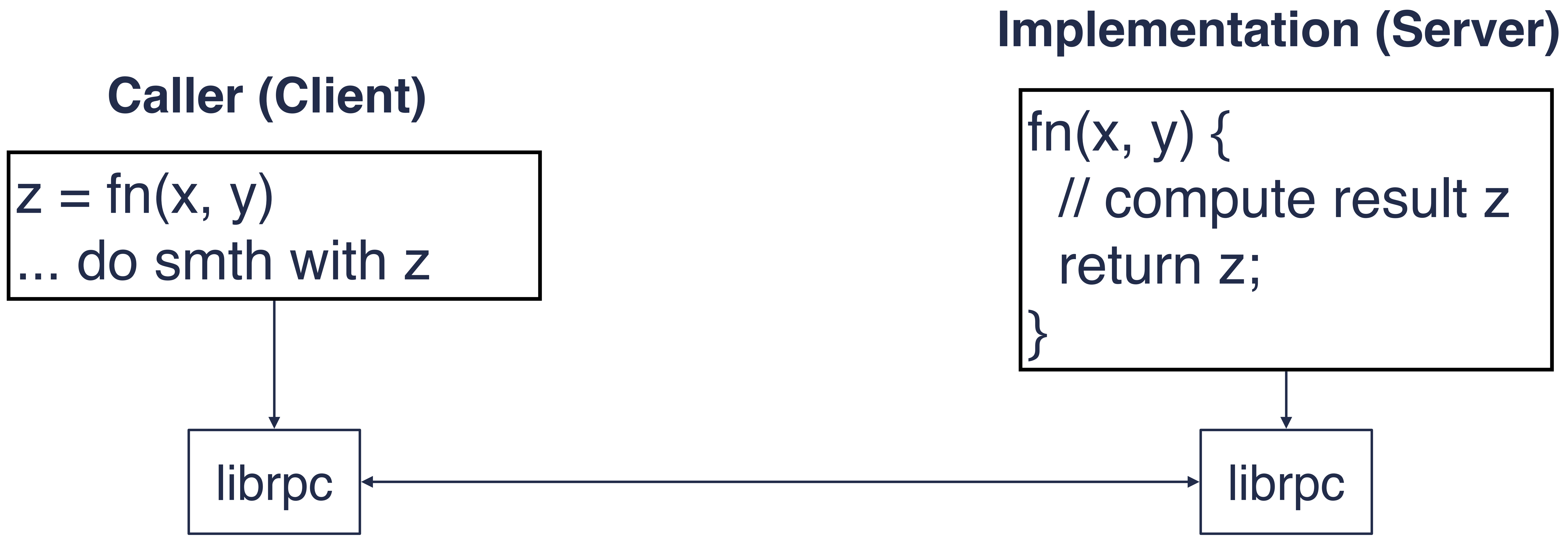
SOLUTION: ANOTHER LAYER!



RPC

(~1984 paper by Birrell, Nelson)

– Idea: Make network communication look like a local procedure call (LPC).

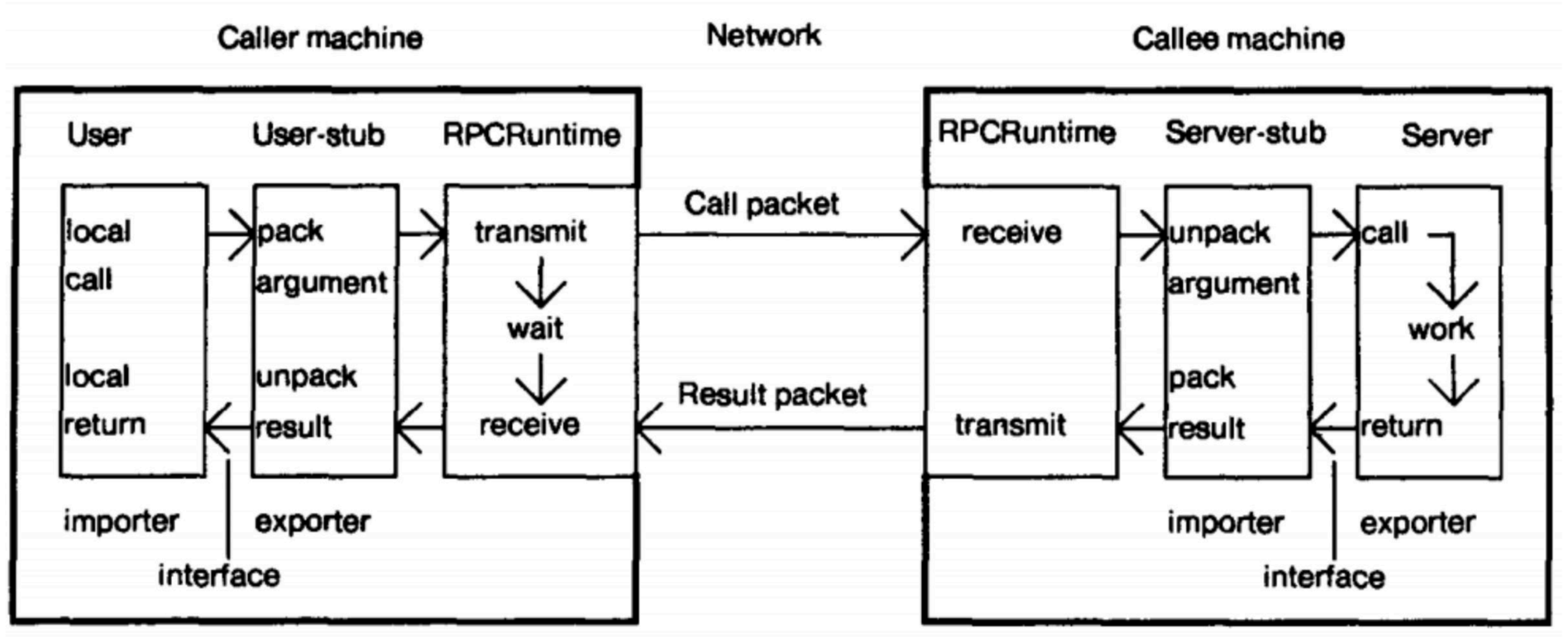


```
z = fn(x, y)
... do smth with z
```

```
fn(x, y) {
  // compute result z
  return z;
}
```

RPC ARCHITECTURE

figure taken from Nelson paper



BENEFITS?

- Easy to use and familiar to any programmer.
- Hides gory network/marshaling details that one would have to implement if doing, e.g., network-level communication, byte orders, ...
- Supports evolution of the communicating components independently.
- Allows for efficient packaging of arguments/return vals.
- Authentication support.
- Location independence.

PROBLEMS?
(or where distribution peeks through the LPC illusion)

- Latency
  - LPC: fast; RPC: can be slow.
  - So care must be taken when invoking RPCs.
- Pointer transfers
  - LPC: caller/callee share address space; RPC: no shared mem.
  - RPClib can't automatically decide what gets serialized and what doesn't.
- Failures
  - LPC: shared fate between caller and callee. RPC: caller and callee can fail independently (recall DS definition).
  - This is the critical challenge in DS and why cannot hide distribution.

ANOTHER PROBLEM: DIFFERENCES IN DATA REPRESENTATION

- 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

SOLUTION: INTERFACE DESCRIPTION LANGUAGE

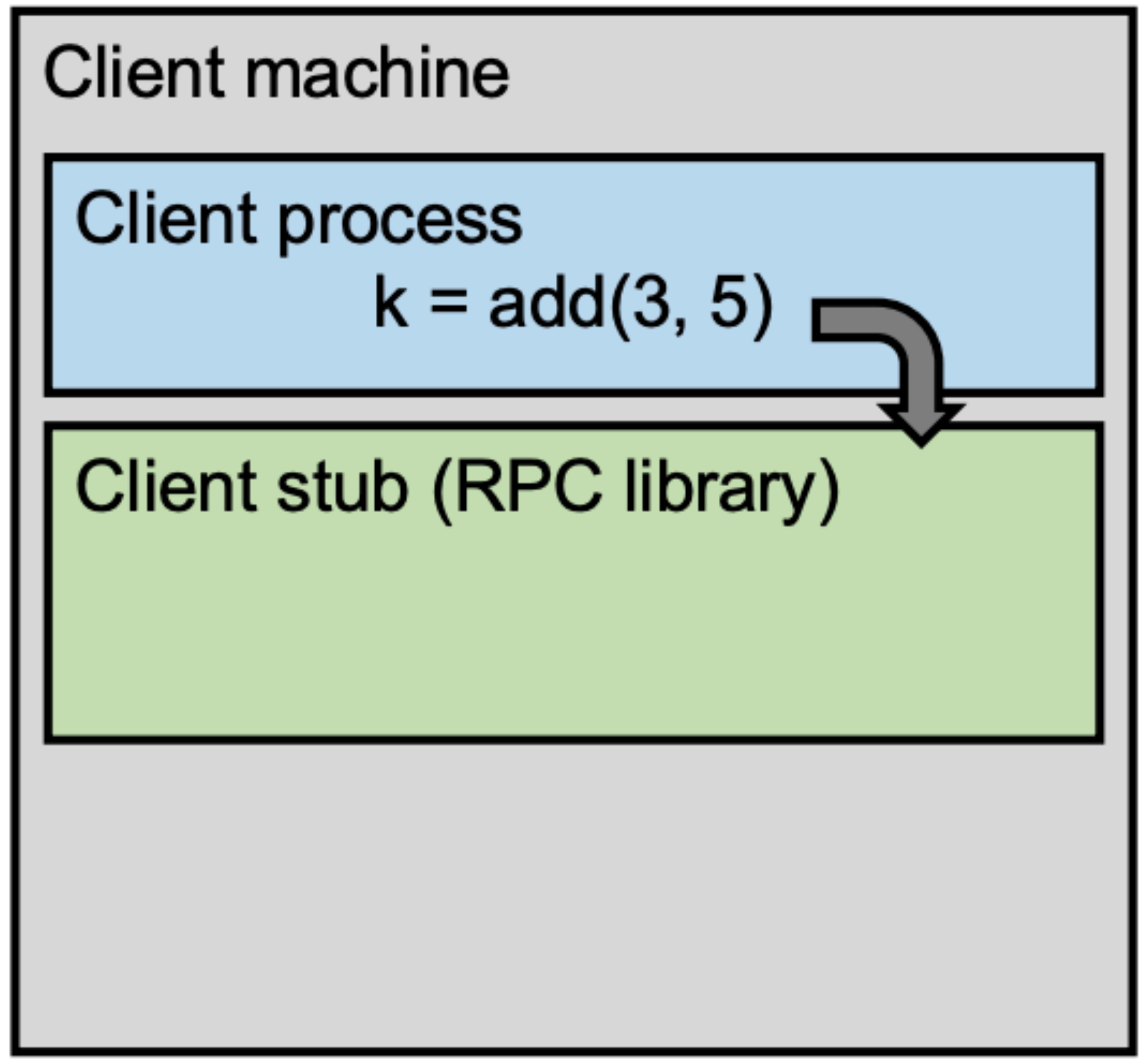
- Mechanism to pass procedure parameters and return values in a machine-independent way
- Programmer may write an interface description in the IDL
  - Defines API for procedure calls: names, parameter/return types
- Then runs an IDL compiler which generates:
  - Code to marshal (convert) native data types into machineindependent 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

EXAMPLE FROM GOOGLE'S PROTOBUF

```
message Person {
  optional string name = 1;
  optional int32 id = 2;
  optional string email = 3;

  enum PhoneType {
    PHONE_TYPE_UNSPECIFIED = 0;
    PHONE_TYPE_MOBILE = 1;
    PHONE_TYPE_HOME = 2;
    PHONE_TYPE_WORK = 3;
  }

  message PhoneNumber {
    optional string number = 1;
    optional PhoneType type = 2 [default = PHONE_TYPE_HOME];
  }

  repeated PhoneNumber phones = 4;
}
```

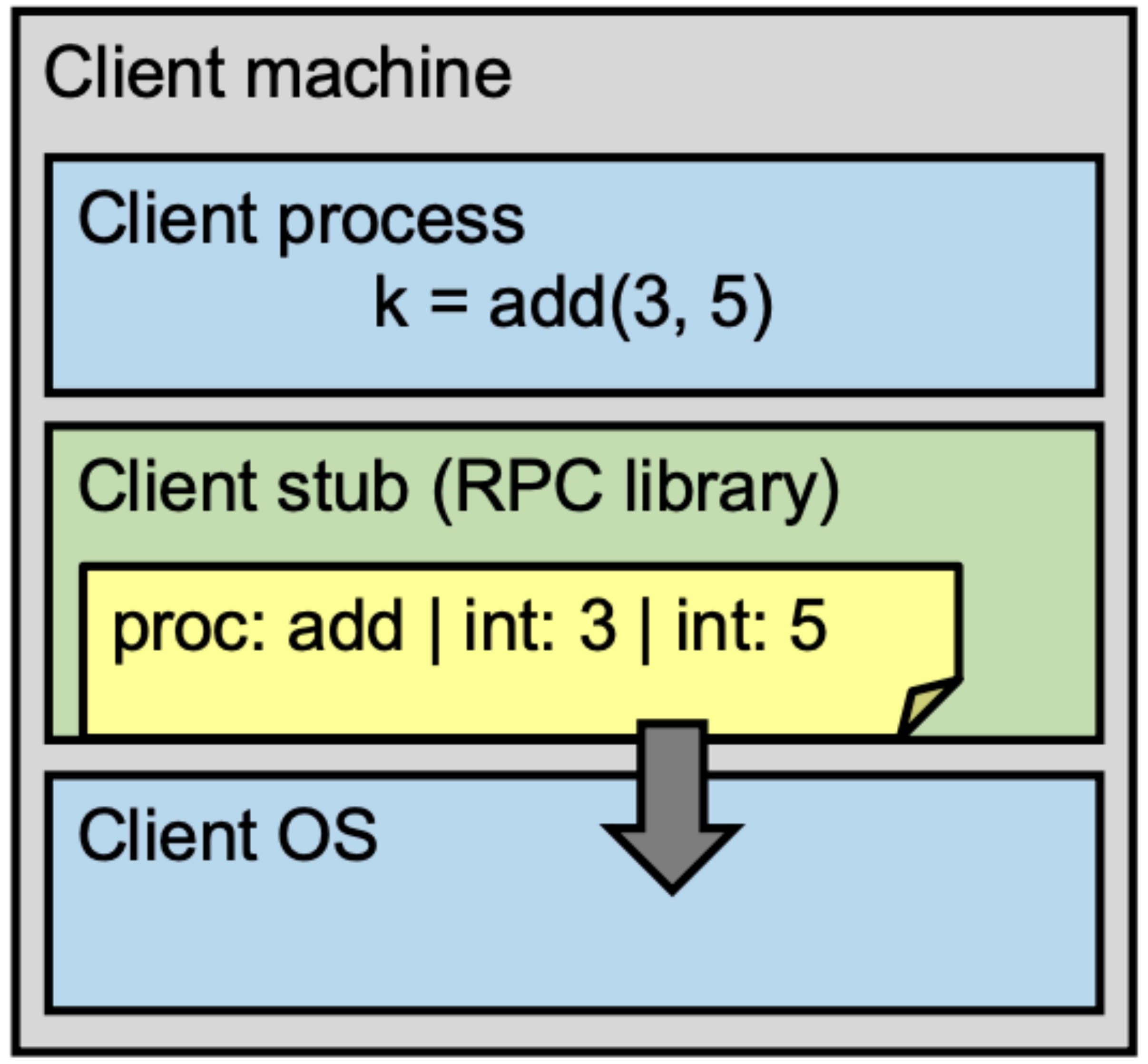
A DAY IN THE LIFE OF AN RPC

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



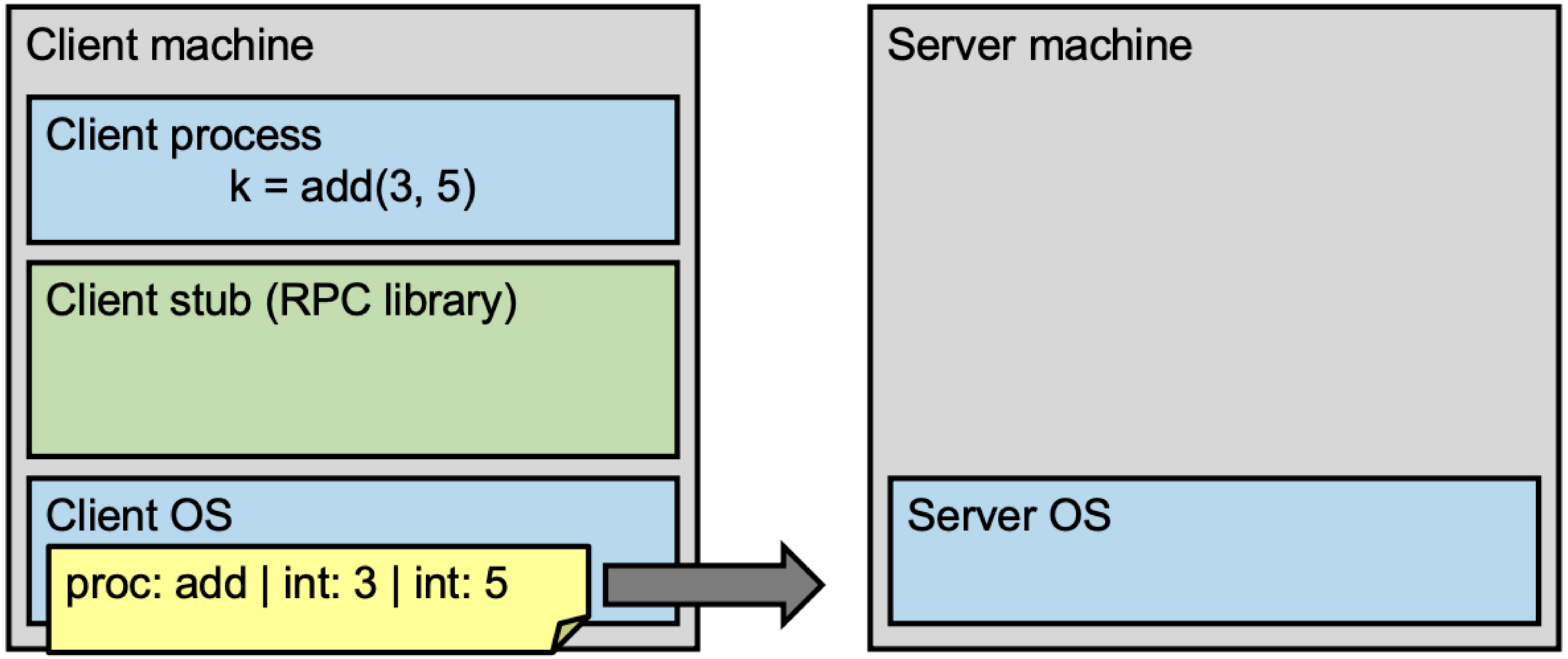
A DAY IN THE LIFE OF AN RPC

1. Client calls stub function (pushes parameters onto stack)
2. **Stub marshals parameters to a network message**



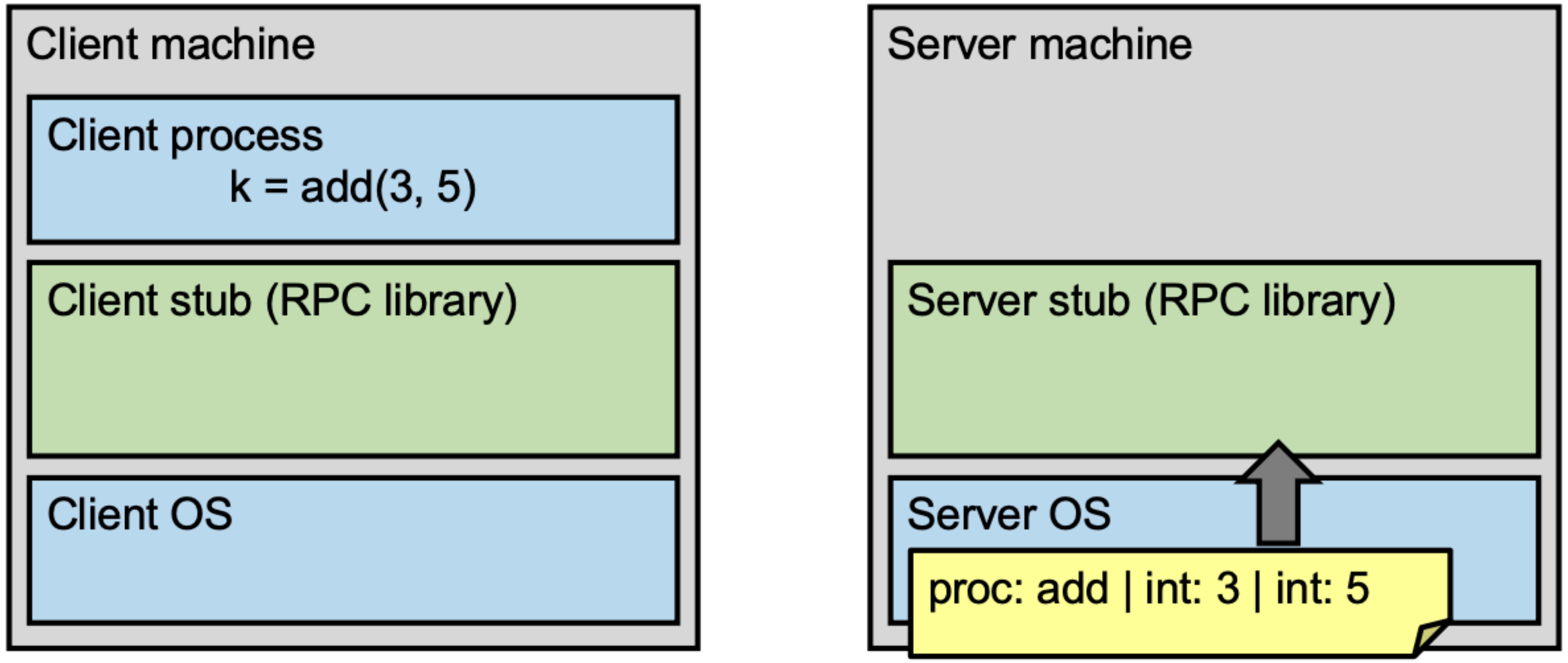
A DAY IN THE LIFE OF AN RPC

2. Stub marshals parameters to a network message
3. **OS sends a network message to the server**



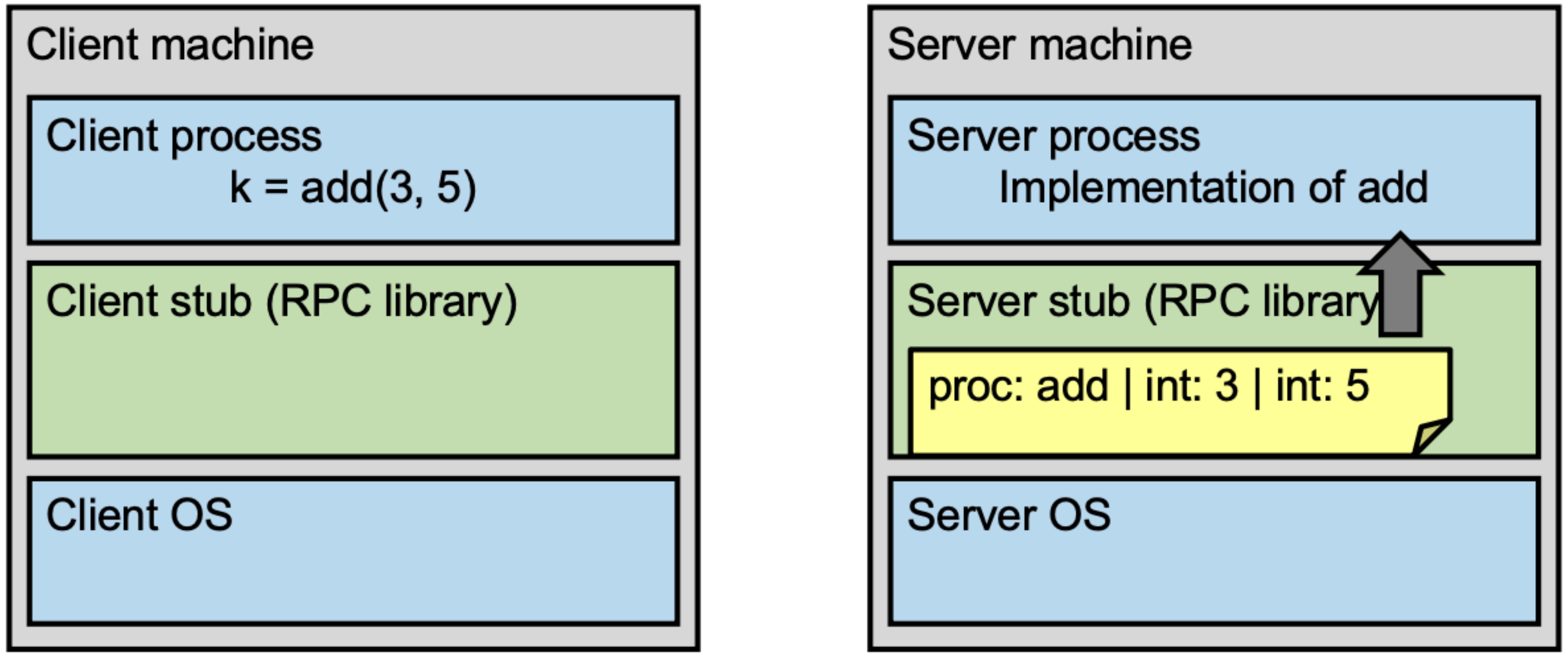
A DAY IN THE LIFE OF AN RPC

3. OS sends a network message to the server
4. **Server OS receives message, sends it up to stub**


Client machine
Client process
k = add(3, 5)
Client stub (RPC library)
Client OS
Server machine
Server stub (RPC library)
Server OS
proc: add | int: 3 | int: 5

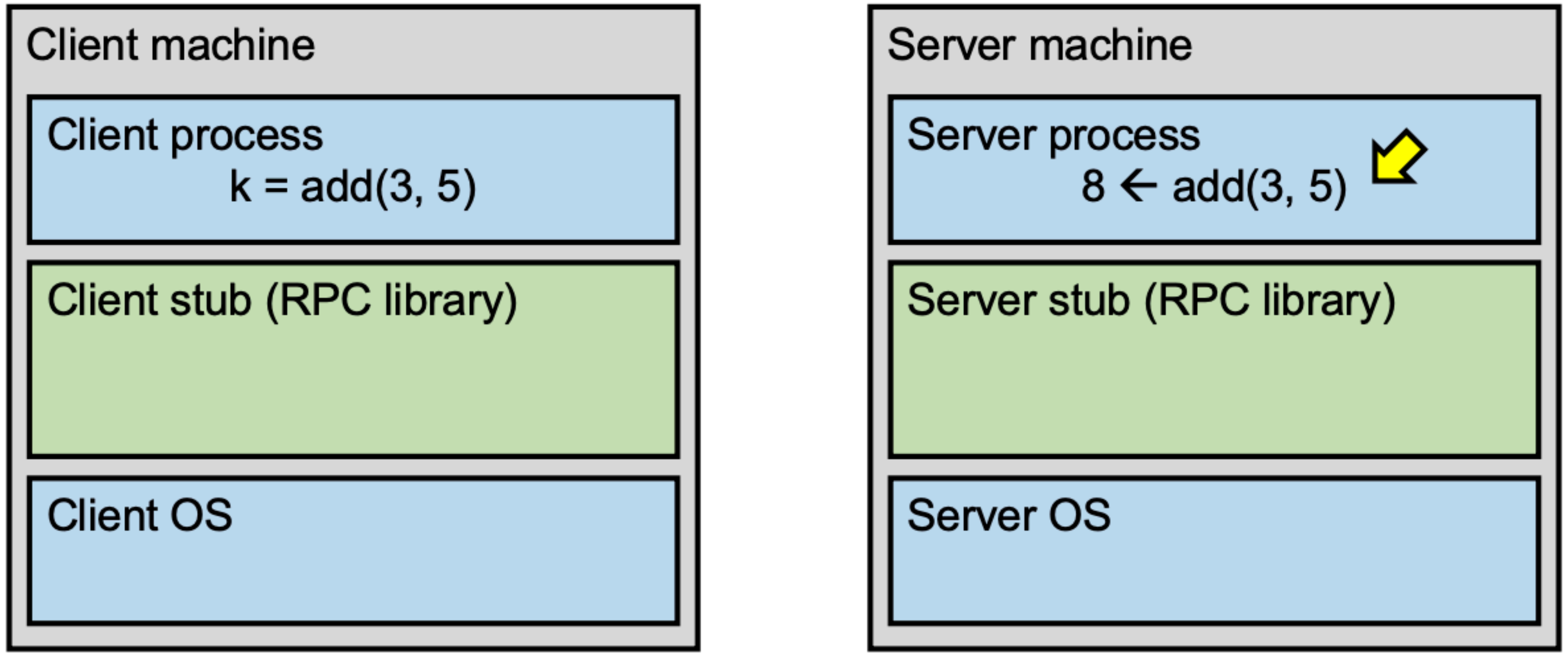
A DAY IN THE LIFE OF AN RPC

4. Server OS receives message, sends it up to stub
5. **Server stub unmarshals params, calls server function**



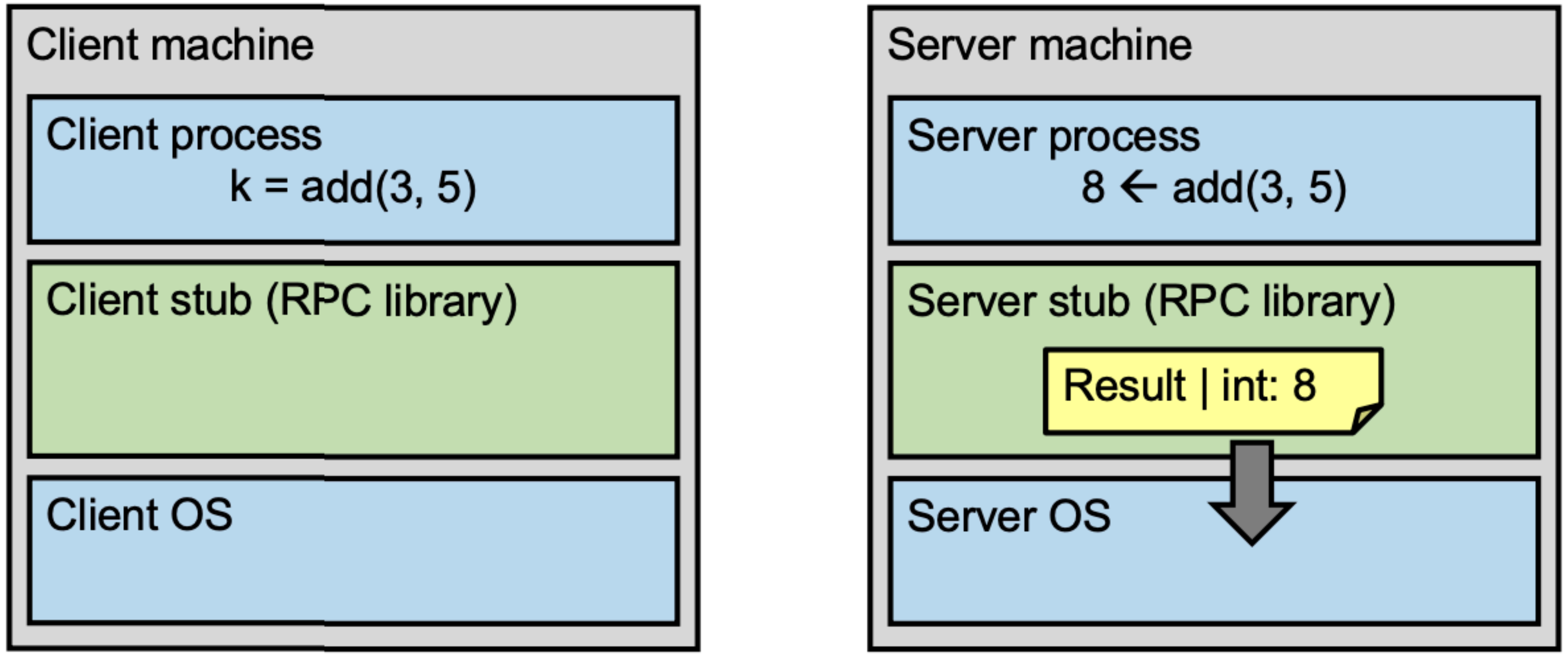
A DAY IN THE LIFE OF AN RPC

5. Server stub unmarshals params, calls server function
6. **Server function runs, returns a value**



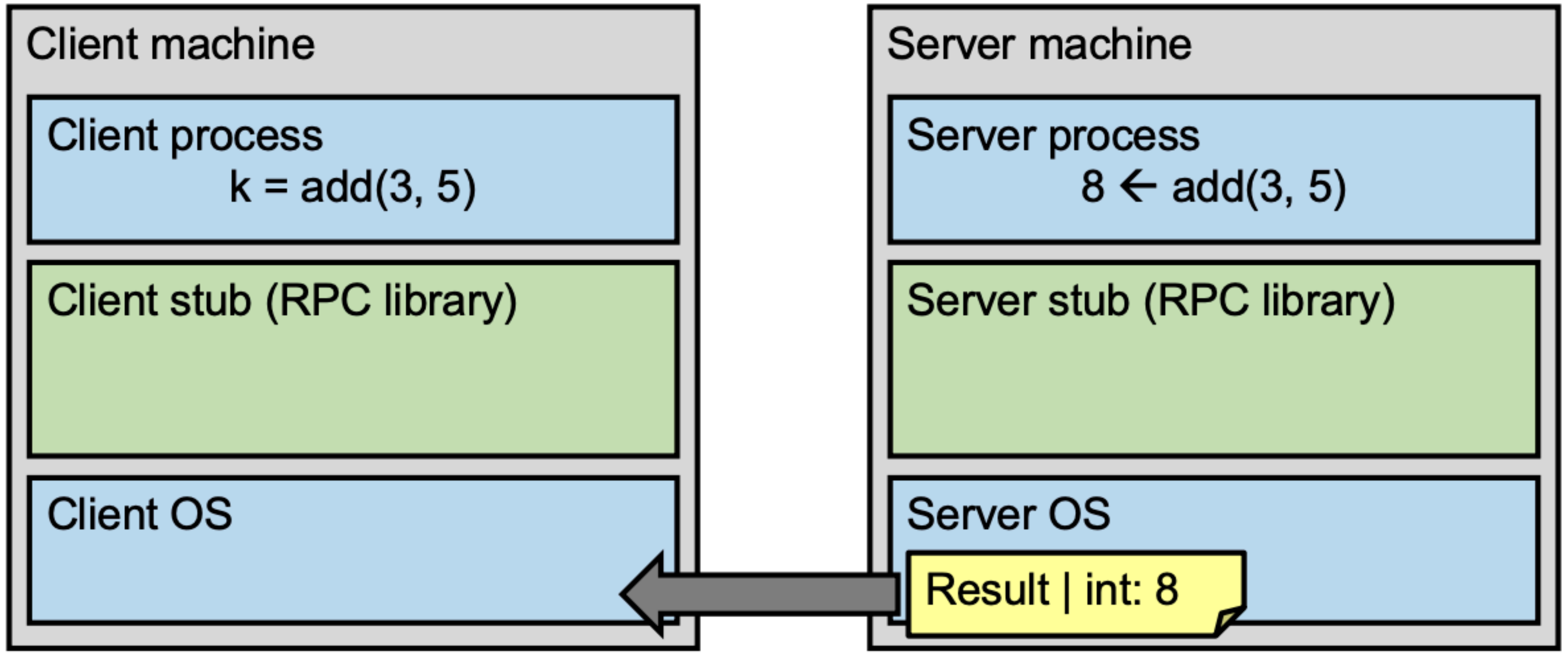
A DAY IN THE LIFE OF AN RPC

6. Server function runs, returns a value
7. **Server stub marshals the return value, sends message**



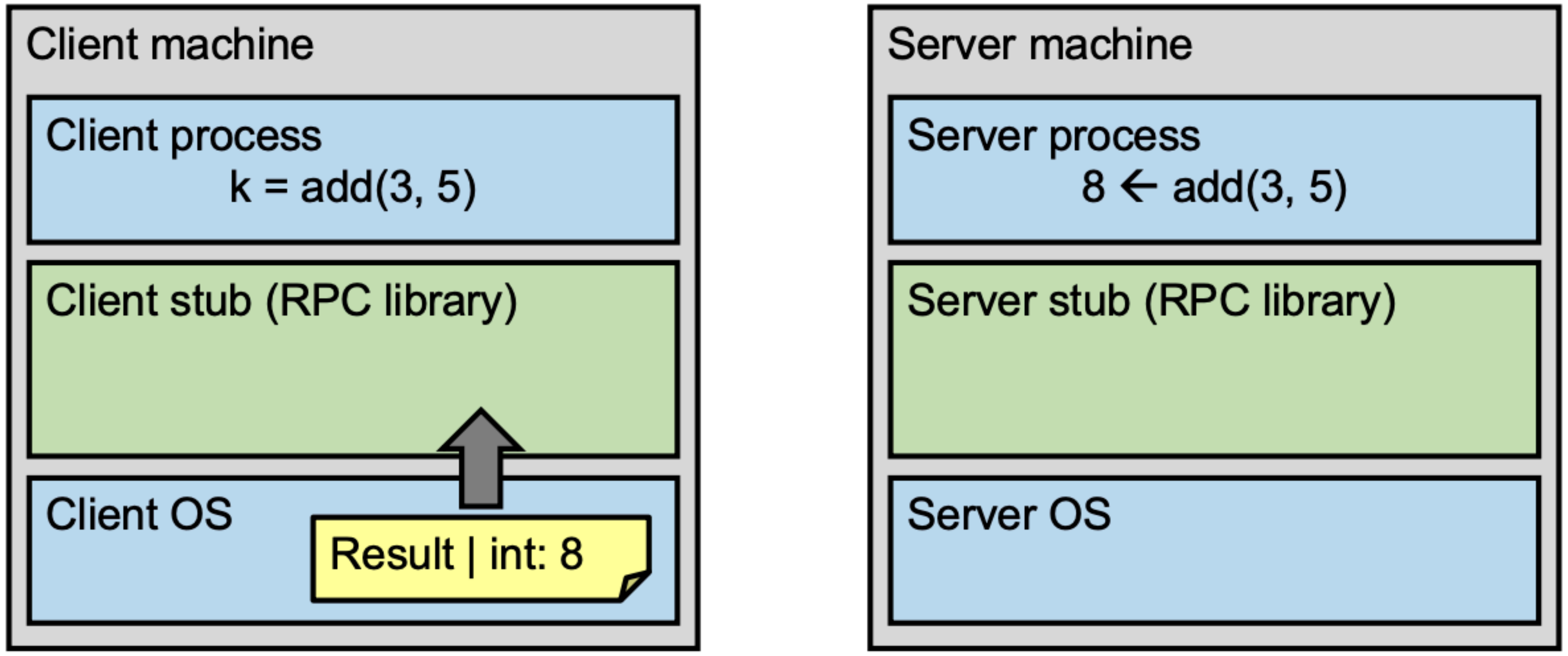
A DAY IN THE LIFE OF AN RPC

7. Server stub marshals the return value, sends message
8. **Server OS sends the reply back across the network**



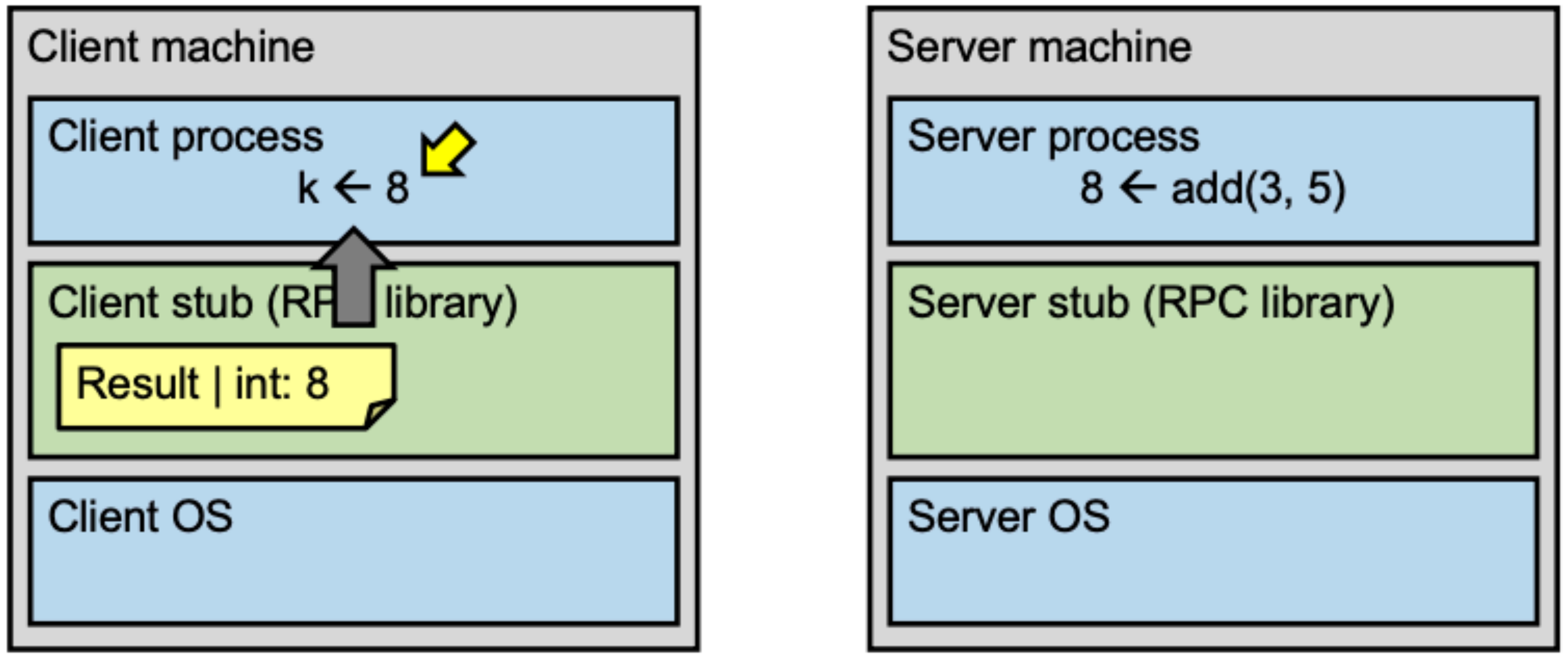
A DAY IN THE LIFE OF AN RPC

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

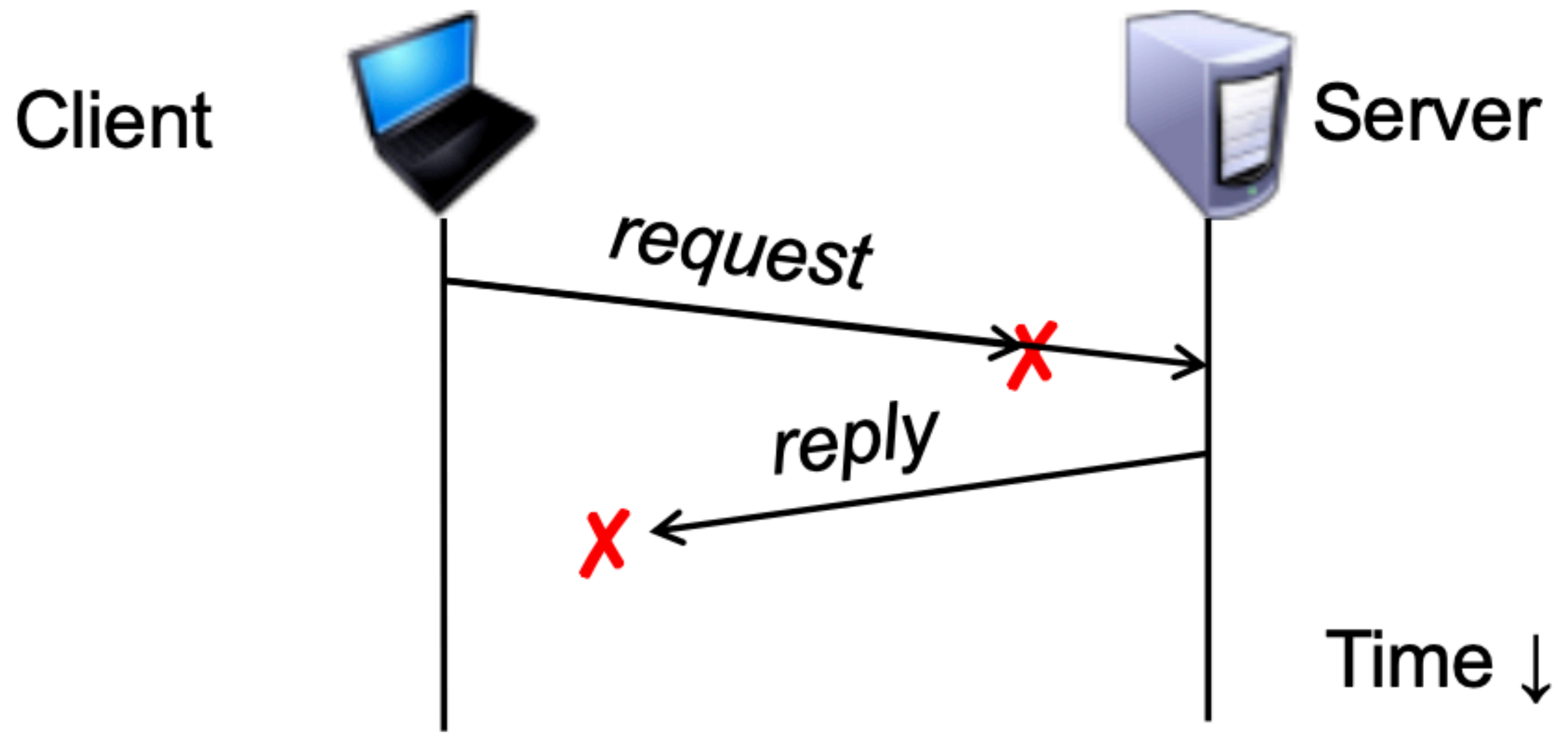
9. Client OS receives the reply and passes up to stub
10. Client stub unmarshals return value, returns to client



WHAT COULD POSSIBLY GO WRONG?

- Client may crash and reboot
- Packets may be dropped
  - Some individual packet loss in the Internet
  - Broken routing results in many lost packets
- Server may crash and reboot
- Network or server might just be very slow

FAILURES, FROM CLIENT'S PERSPECTIVE



The cause of the failure is hidden from the client!

RPC SEMANTICS

- At least once
- At most once
- Exactly once

AT LEAST ONCE

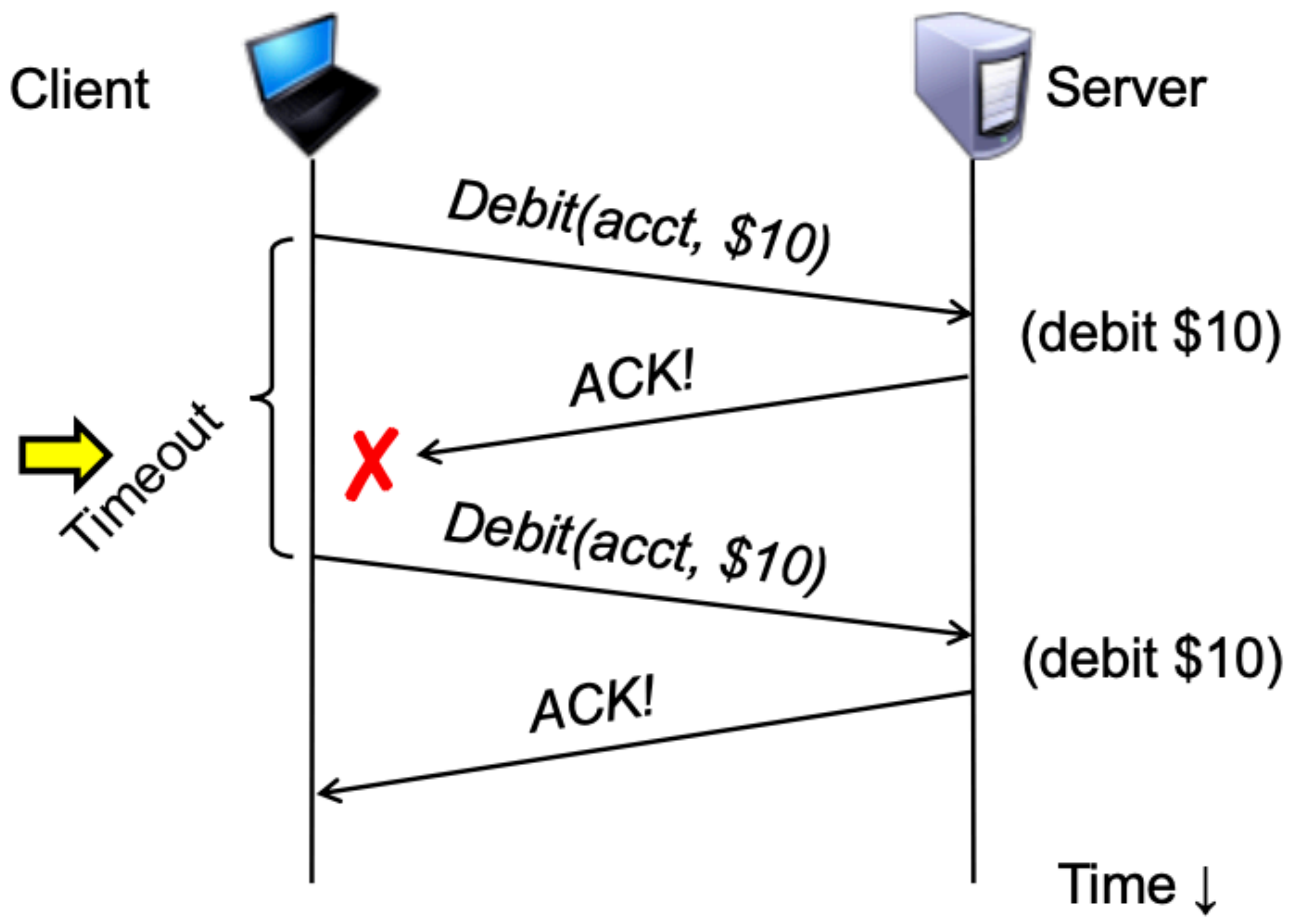
– Semantic: RPC is eventually executed at least once, but potentially multiple times.

AT LEAST ONCE

- Semantic: RPC is eventually executed at least once, but potentially multiple times.
- Simplest scheme for handling failures
- Implementation
  - Client keeps issuing RPC until gets a response from server (**retransmission**).
  - If failures (of net/server) are temporary, semantic satisfied.
  - Repeat the above a few times. Still no response? Return an error to the application

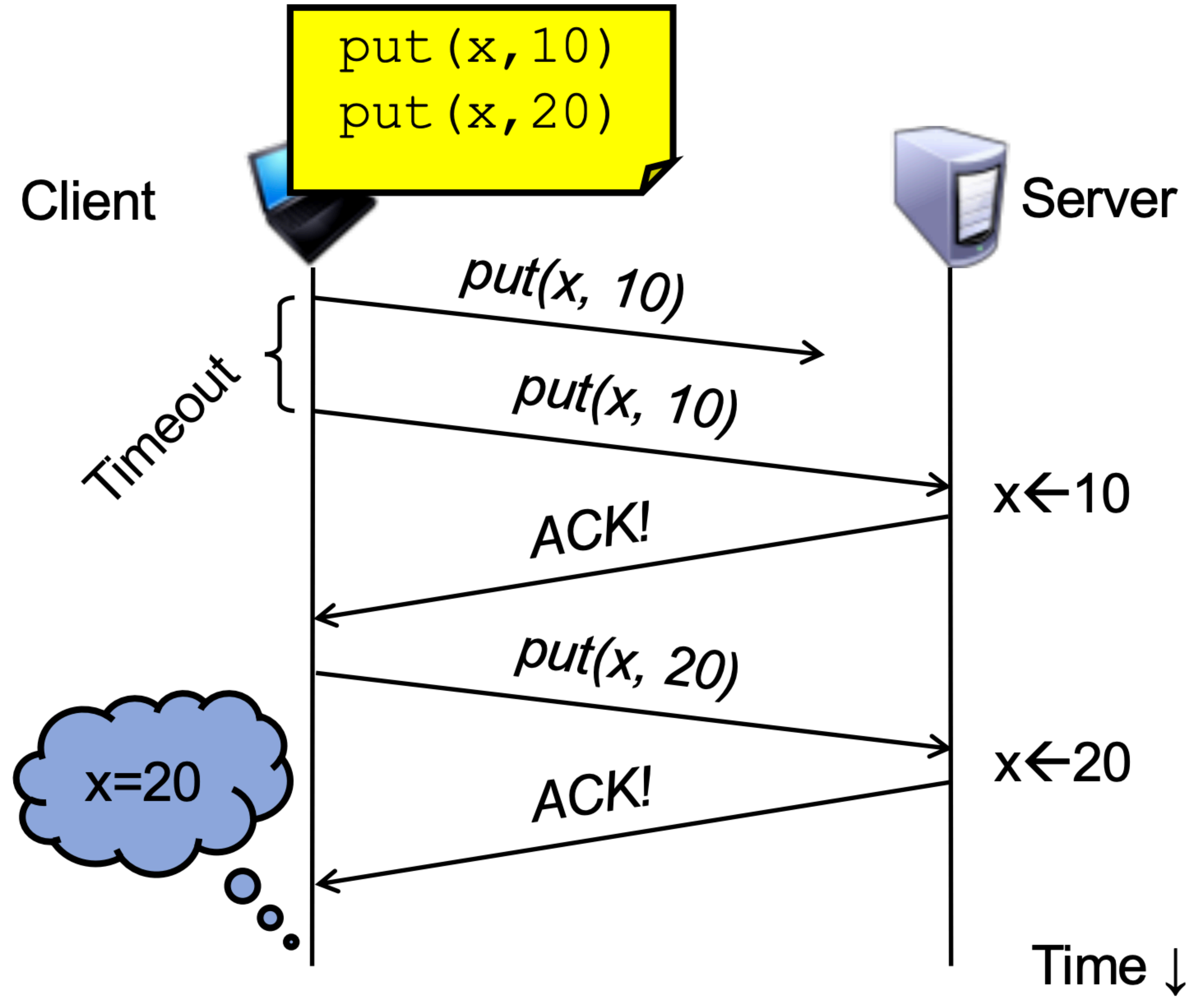
AT LEAST ONCE

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



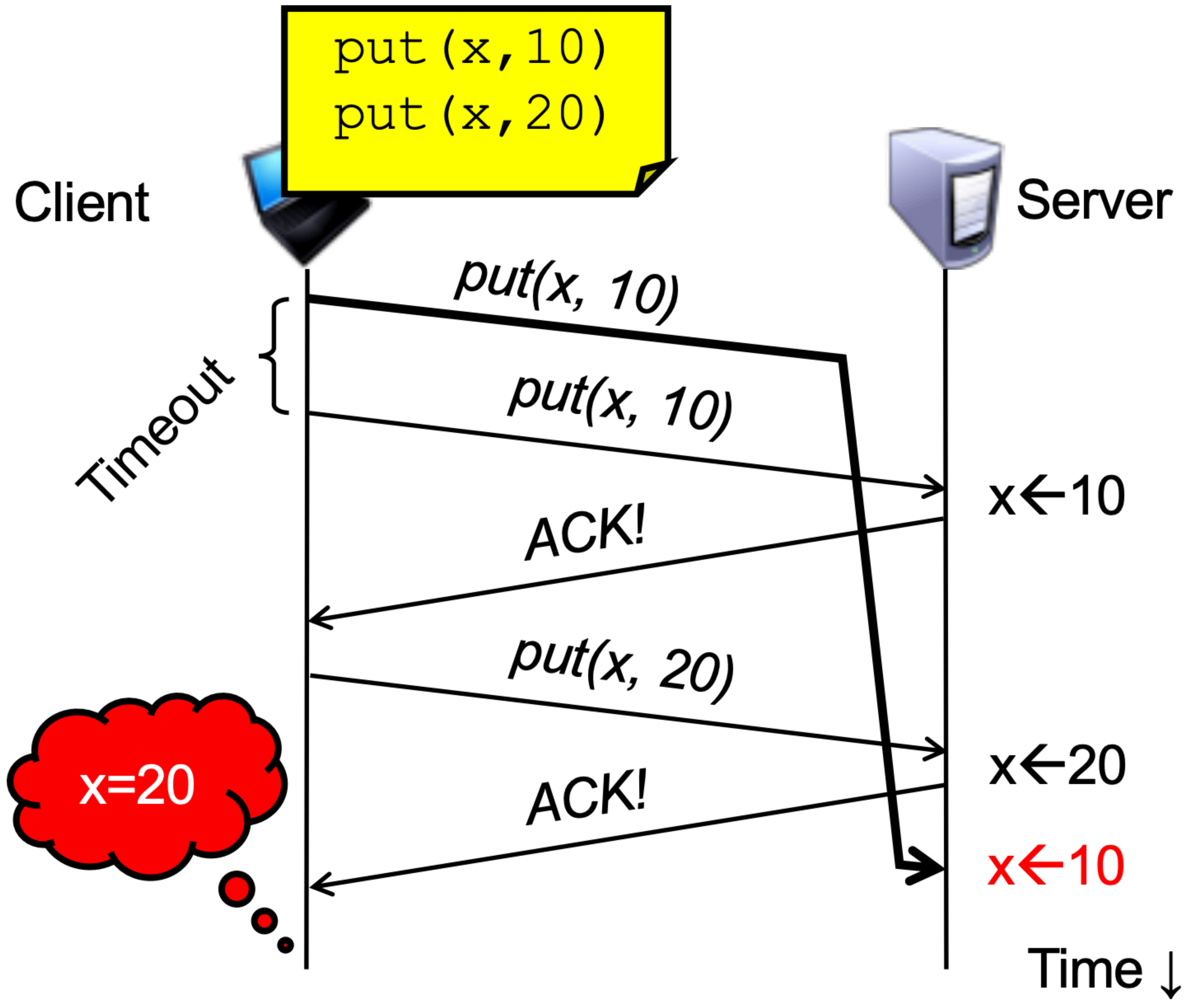
AT LEAST ONCE AND WRITES

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



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



AT MOST ONCE

– Semantic: RPC is executed zero or one times, not more.

AT MOST ONCE

- Semantic: RPC is executed zero or one times, not more.
- How to detect a duplicate request?
  - Test: Server stub sees same function, same arguments twice
  - **No!** Sometimes applications legitimately submit the same function with same augments, twice in a row

AT MOST ONCE

- Semantic: RPC is executed zero or one times, not more.
- Implementation:
    - Clients identify their requests with **unique transaction ID** (xid).
    - Server remembers xids to detect duplicates and squelch them.
- Problem: server failure at inopportune time can cause failure of the semantic. Give examples.

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

AT-MOST-ONCE: PROVIDING UNIQUE XIDS

- Combine a unique client ID (e.g., IP address) with the current time of day
- Combine unique client ID with a sequence number
  - Suppose client crashes and restarts. Can it reuse the same client ID?
- Big random number (probabilistic, not certain guarantee)

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

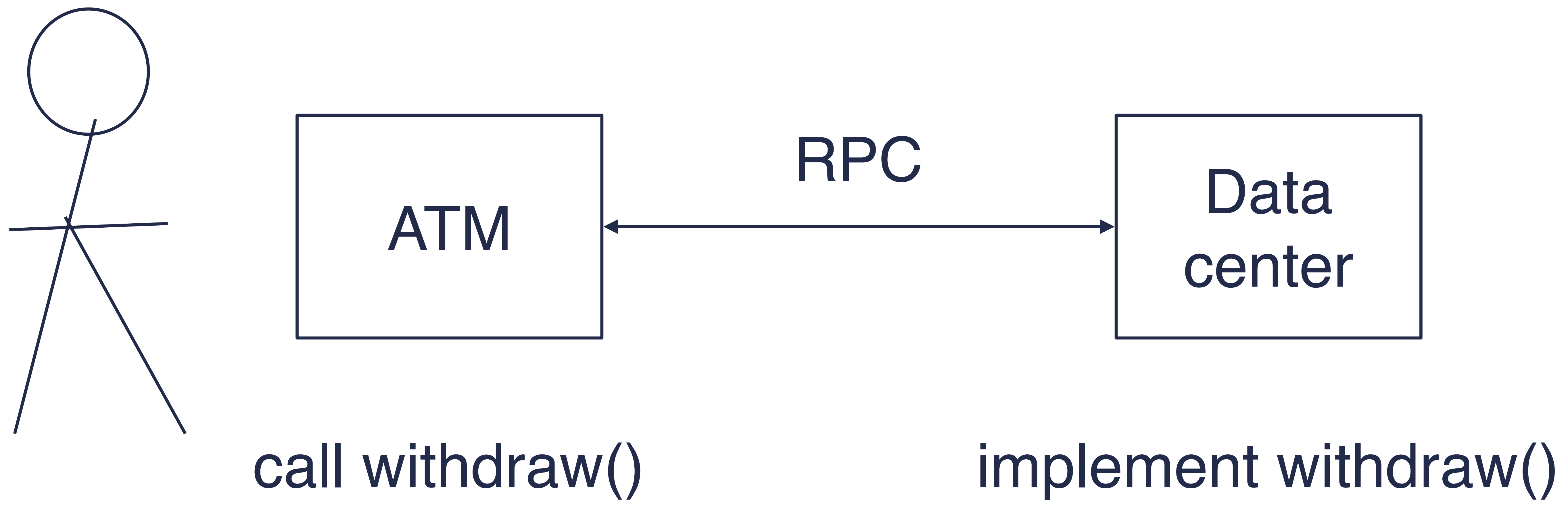
EXACTLY ONCE

- Semantic: RPC is executed once.
- This is the ideal (it resembles the LPC model most closely and it’s easiest to understand), but it’s surprisingly hard to implement.

- 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)

DESIGN AN ATM: WHAT COULD GO WRONG?

— When a person wants to withdraw cash from the ATM, the ATM sends an RPC to the bank. The bank first checks that the person has enough money in their account, and if so, deducts the money and confirms with the ATM. The ATM, in turn, should give the money to the user.

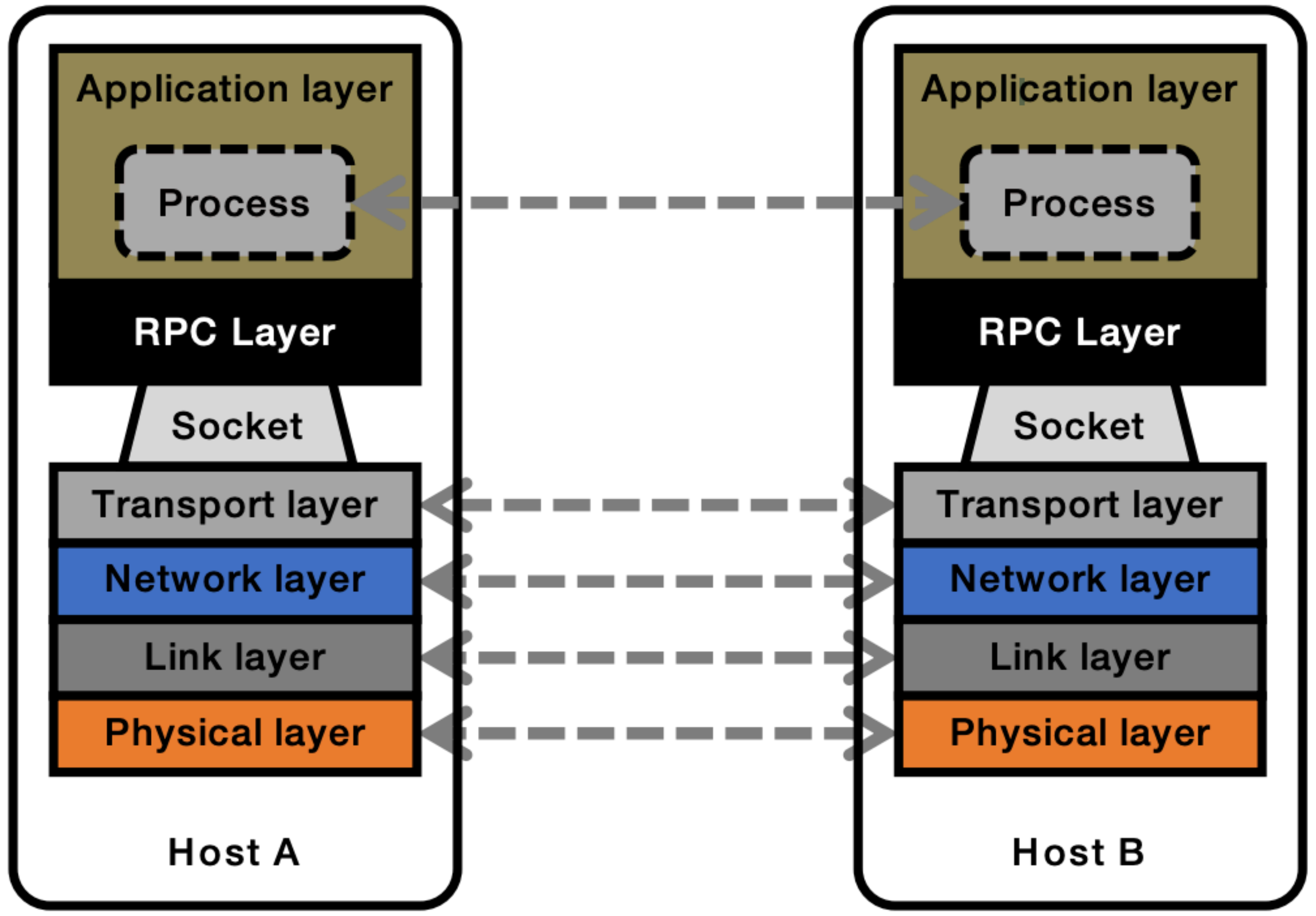


EXACTLY-ONCE FOR EXTERNAL ACTIONS?

- Imagine that remote operation triggers an external physical thing
  - e.g., dispense $100 from an ATM
- ATM could crash immediately before or after dispensing
  - ATM would lose its state, and
  - Don’t know which one happened (although can make window very small)
- Can’t achieve exactly-once in general, in presence of external actions

TAKEAWAYS

- Layering is your friend when building systems!
- Need to solve many challenges for seemingly "simple" abstractions
- Deal w/ RPC failures
  - At-least-once w/ retransmission
  - At-most-once w/ duplicate filtering
  - Exactly-once with:
    - at-least-once + at-most-once + fault tolerance + no external actions
- Next class: **transaction**



ACKNOWLEDGEMENT

THIS COURSE IS DEVELOPED HEAVILY BASED ON COURSE MATERIALS SHARED BY PROF. INDRANIL GUPTA, PROF. ROBERT MORRIS, PROF. MICHAEL FREEDMAN, PROF. KYLE JAMIESON, PROF. WYATT LLOYD AND PROF. ROXANA GEAMBASU. MANY APPRECIATIONS FOR GENEROUSLY SHARING THEIR MATERIALS AND TEACHING INSIGHTS.