CS 5450

Remote Procedure Calls

Vitaly Shmatikov

Abstractions

- Abstractions for communication
  - TCP masks some of the pain of communicating over unreliable IP
- Abstractions for computation

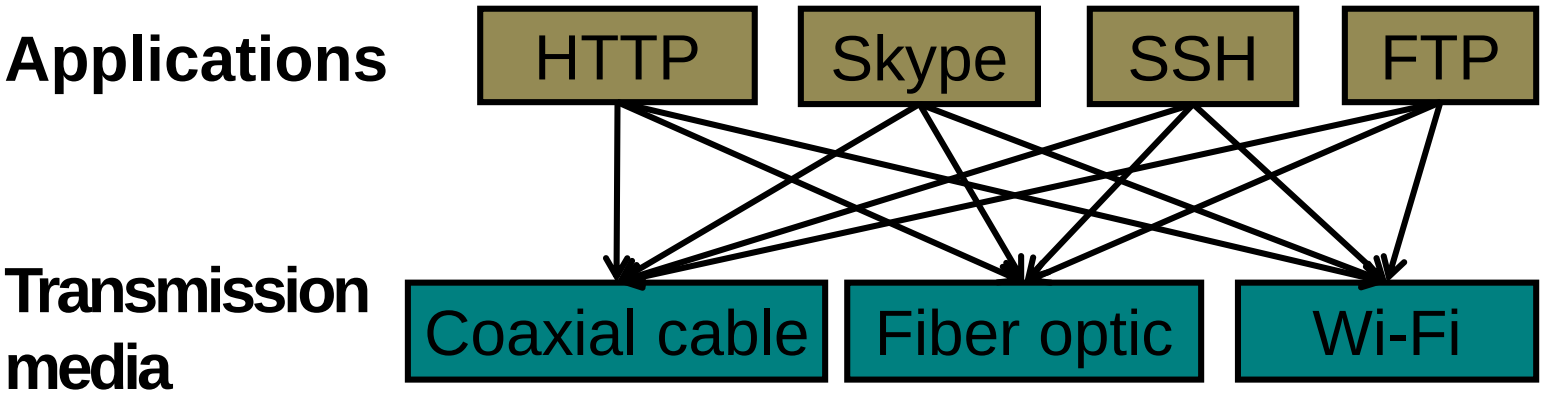
Goal:

programming abstraction that enables splitting work across multiple networked computers

The Problem of Communication

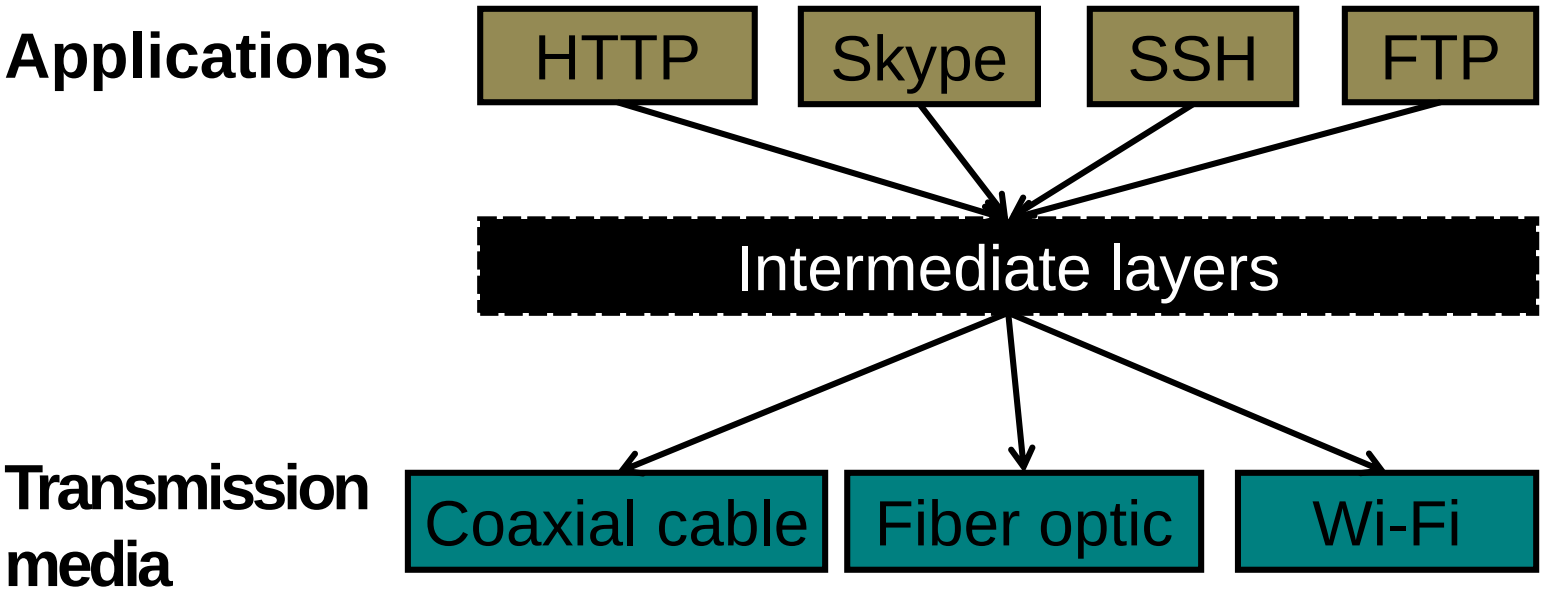
- Process on Host A wants to talk to process on Host B
- A and B must agree on the meaning of the bits being sent and received at many different levels, including:
  - How many volts is a 0 bit, a 1 bit?
  - How does receiver know which is the last bit?
  - How many bits long is a number?

The Problem of Communication



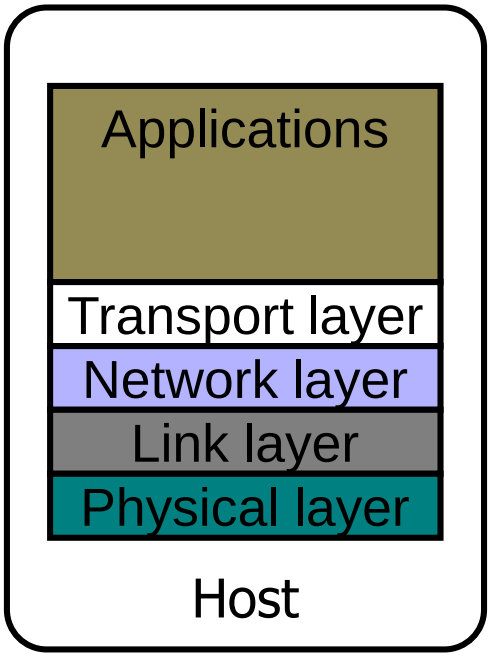
- Re-implement every application for every new underlying transmission medium?
  - Consequence: change every application on any change to an underlying transmission medium
- No!

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

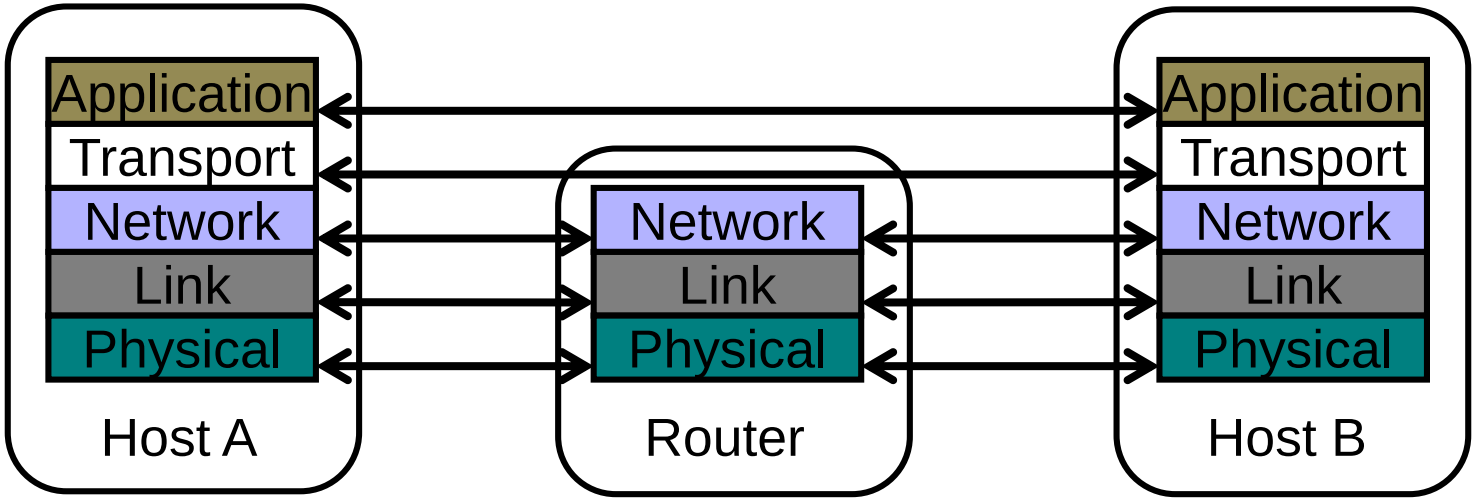
Layering in the Internet



- 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

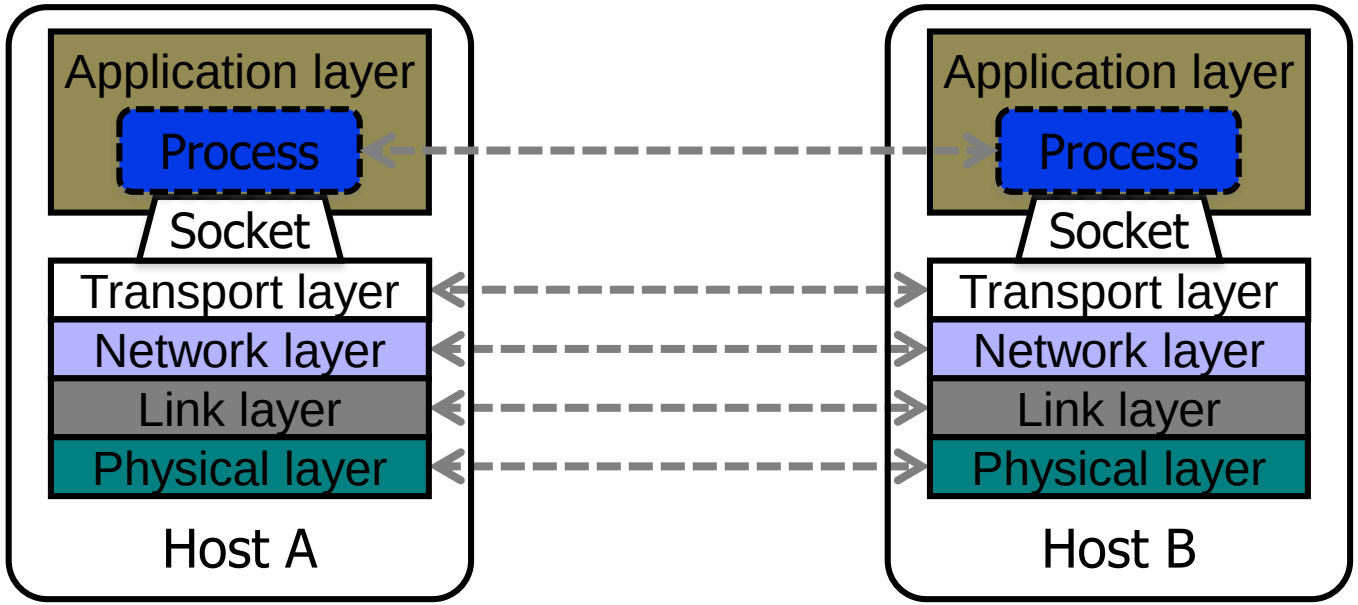
Communication Between Hosts

- 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



Socket-Based Communication

- Socket: the interface OS provides to the network
  - Explicit inter-process message exchange
- Can build distributed systems atop sockets: send(), recv()



Principle of Transparency

- Principle of transparency: Hide that resource is physically distributed across multiple computers
  - Access resource the same way as locally
  - Users can't tell where resource is physically located
- Network sockets provide applications with point-to-point communications between processes
  - Is this transparent?

```
// 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;
servad
servad

// 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

Remote Procedure Call

- A type of client/server communication
- Attempts to make remote procedure calls look like local ones

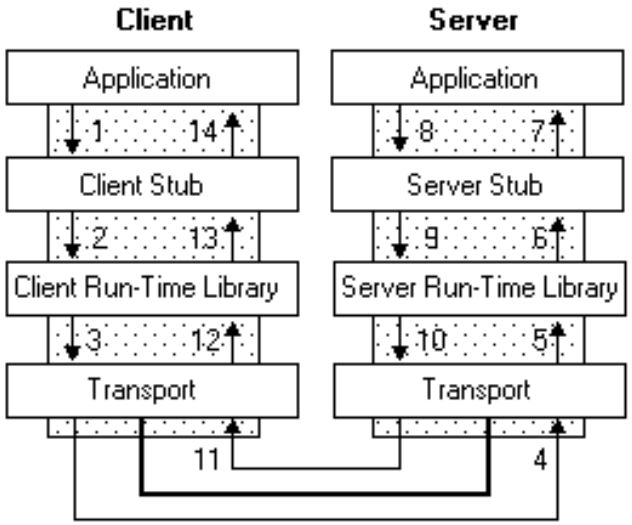


figure from Microsoft MSDN

```
{ ...
   foo()
}
void foo() {
  invoke_remote_foo()
}
```

RPC Goals

- Ease programming
- Hide complexity
- Automate task of implementing distributed computation
- Familiar model for programmers (just make a function call)

RPC Properties

- A remote procedure call makes a call to a remote service look like a local call
- RPC makes transparent whether server is local or remote
- RPC allows applications to become distributed transparently
- RPC makes architecture of remote machine transparent

RPC Issues

- Heterogeneity
  - Client needs to rendezvous with the server
  - Server must dispatch to the required function
  - Different address spaces, data representation
- Failure
  - What if messages get dropped?
  - What if client, server, or network fails?
- Performance
  - Procedure call takes ≈ 10 cycles ≈ 3 ns
  - RPC in a data center takes ≈ 10 µs (103 times slower)
  - In the wide area, typically 106 times slower

Differences in Data Representation

- Not an issue for local procedure call
- For an RPC, 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

Stubs: Obtaining Transparency

- Compiler generates from API stubs for a procedure on the client and server
- Client stub
  - **Marshals** arguments into machine-independent format
  - Sends request to server
  - Waits for response
  - **Unmarshals** result and returns to caller
- Server stub
  - **Unmarshals** arguments and builds stack frame
  - Calls procedure
  - Server stub **marshals** results and sends reply

Marshaling and Unmarshaling

- Example of marshaling: convert into string in a predefined byte order, followed by string length
- Floating point...
- Nested structures?
  - Design question for the RPC system – support them or not?
- Complex data structures?
  - Some RPC systems let you send lists and maps as first-order objects

Stubs and IDLs

- RPC stubs do the work of marshaling and unmarshaling data
- But how do they know how to do it?
- Typically: write a description of the function signature using an IDL -- interface definition language
  - Lots of these. Some look like C, some look like XML ... details don't matter much.

Interface Description Language

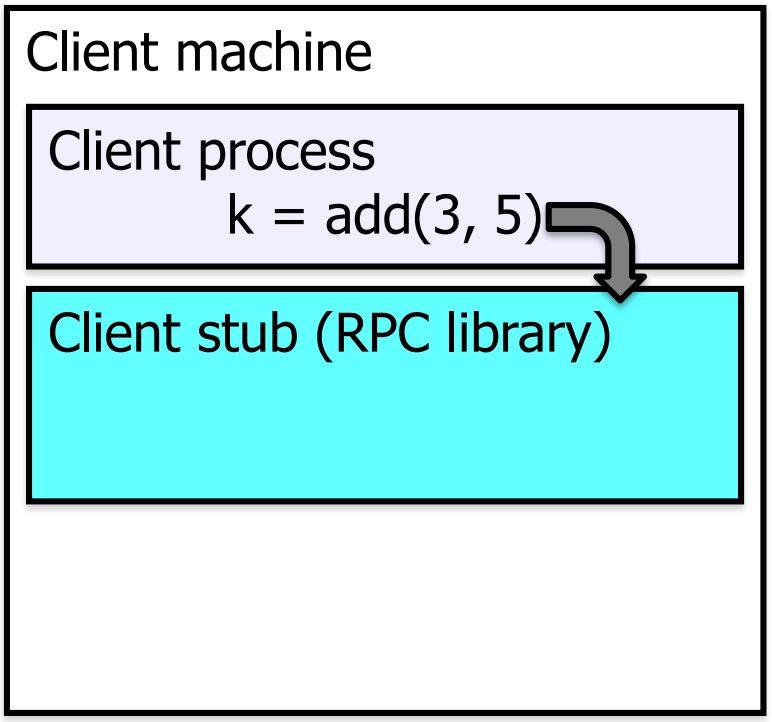
- Mechanism to pass procedure parameters and return values in a machine-independent way
- Use IDL to define API for procedure calls: names, parameter/return types
- IDL compiler generates:
  - Code to marshal (convert) and unmarshal native data types into machine-independent byte streams
  - Client stub: forwards local procedure call as a request to server
  - Server stub: dispatches RPC to its implementation

Server Stub Structure

- 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

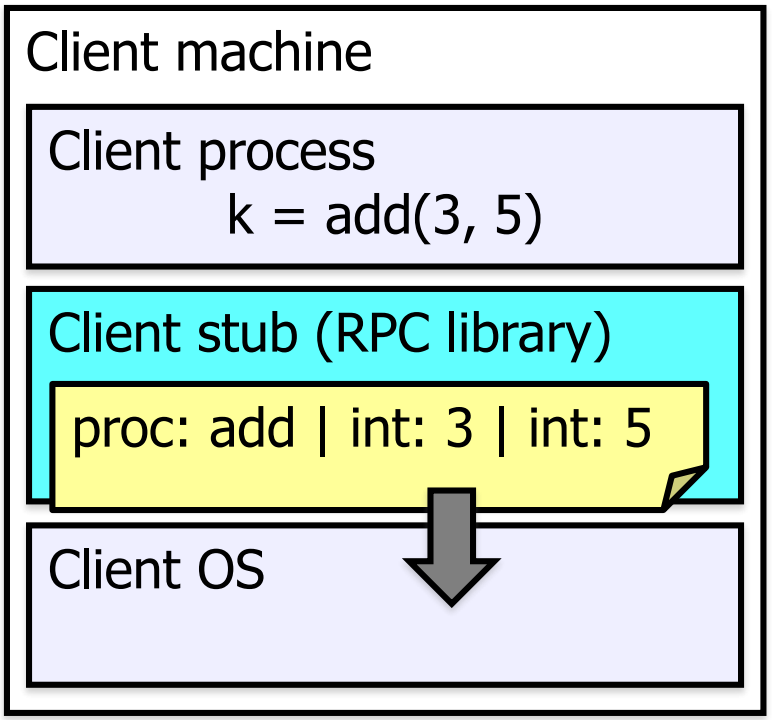
A Day in the Life of RPC

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



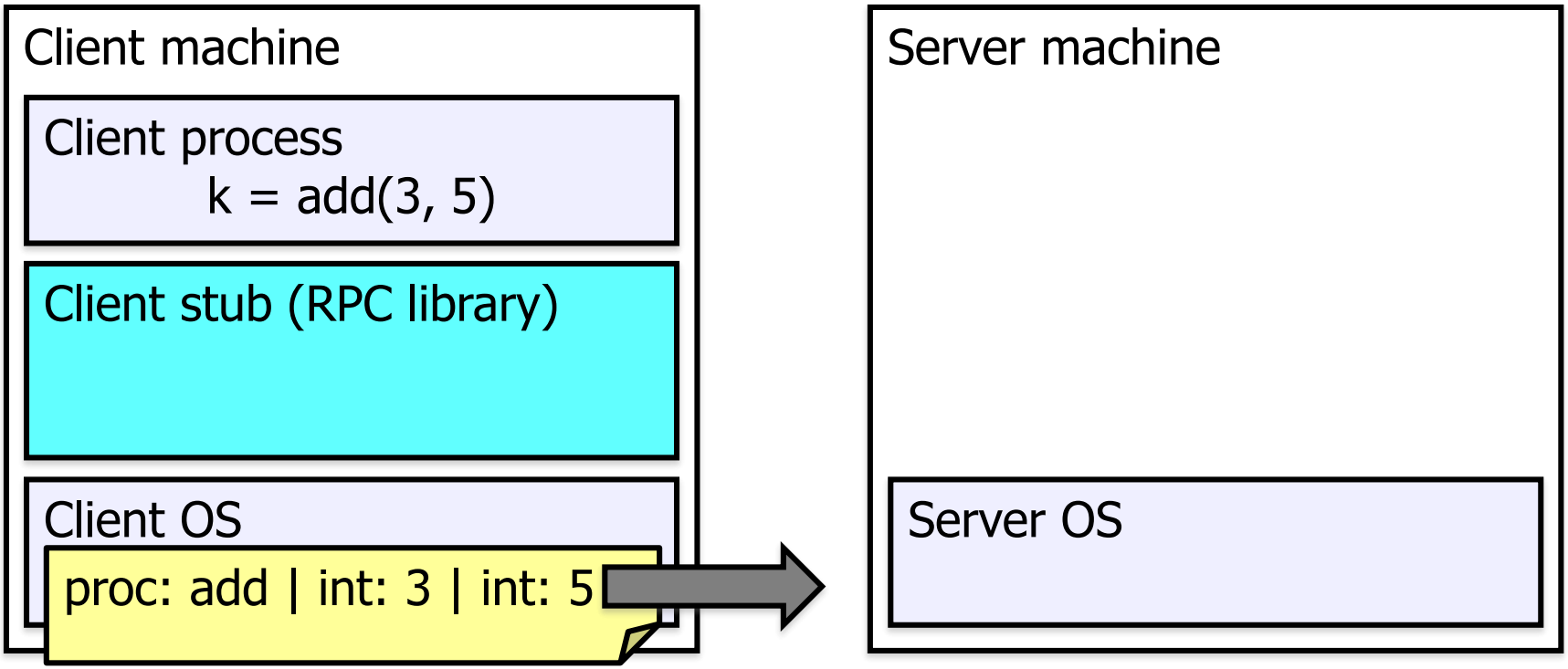
A Day in the Life of RPC

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



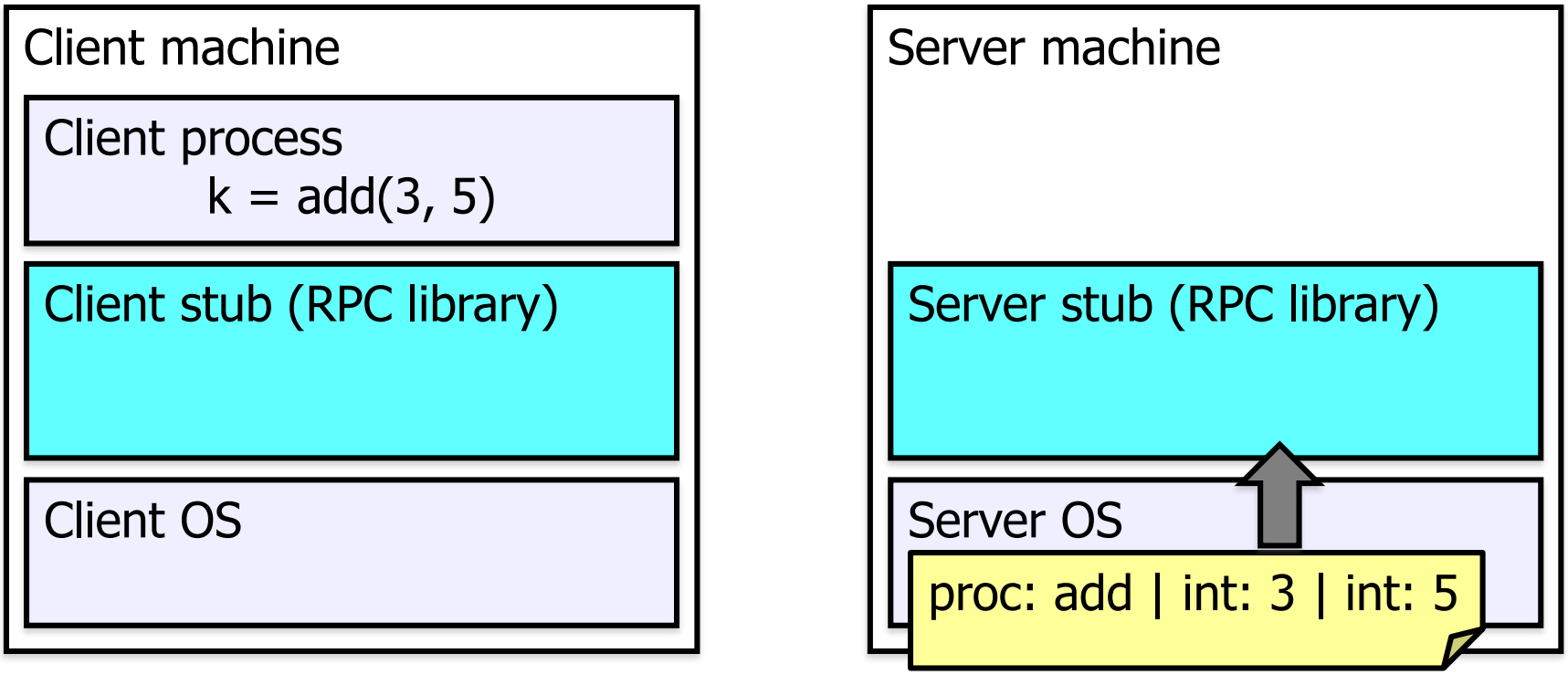
A Day in the Life of RPC

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



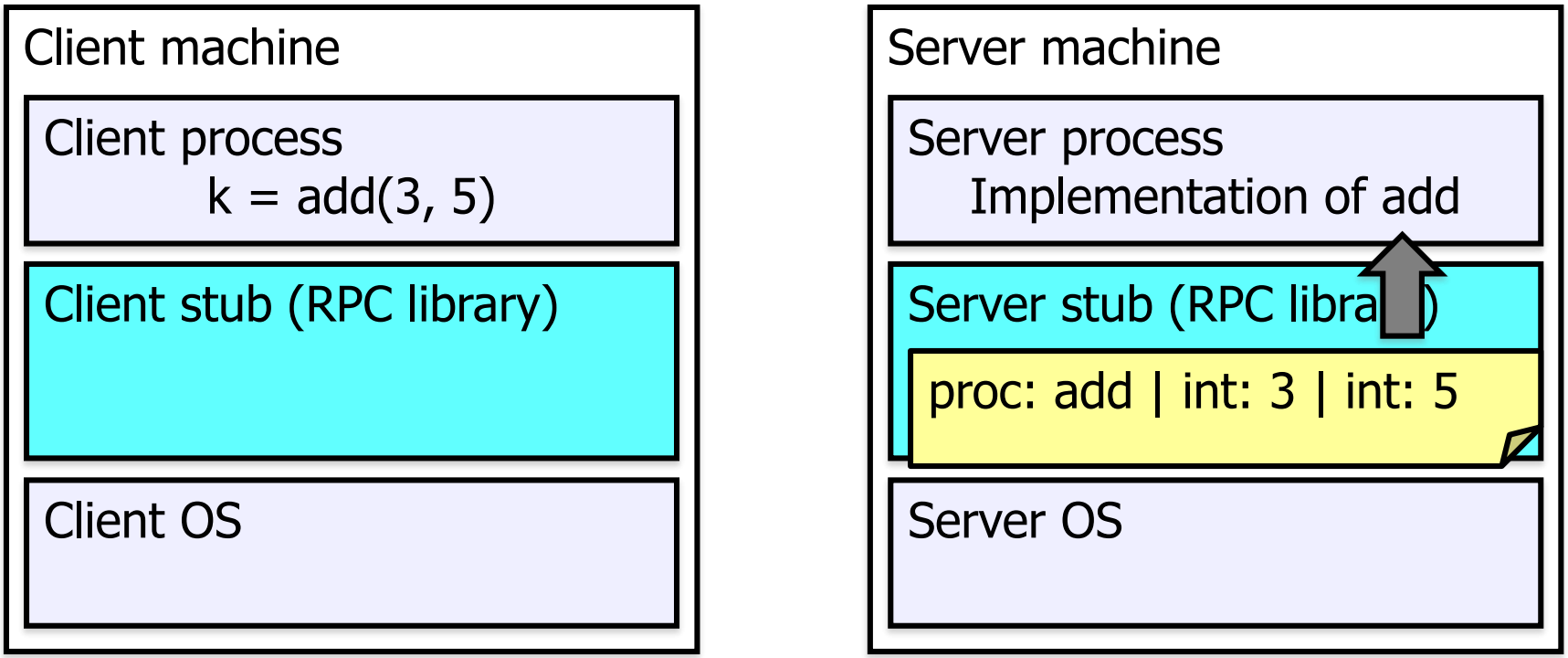
A Day in the Life of RPC

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



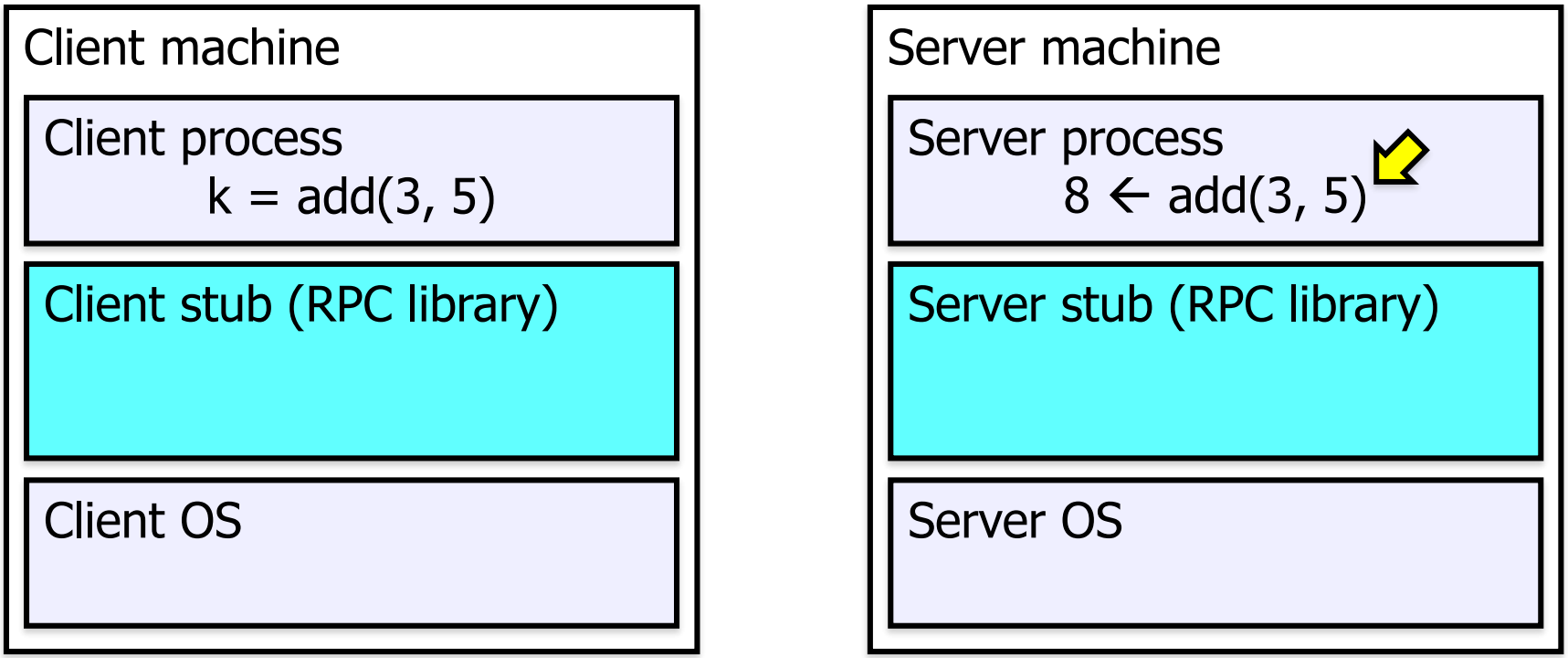
A Day in the Life of RPC

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



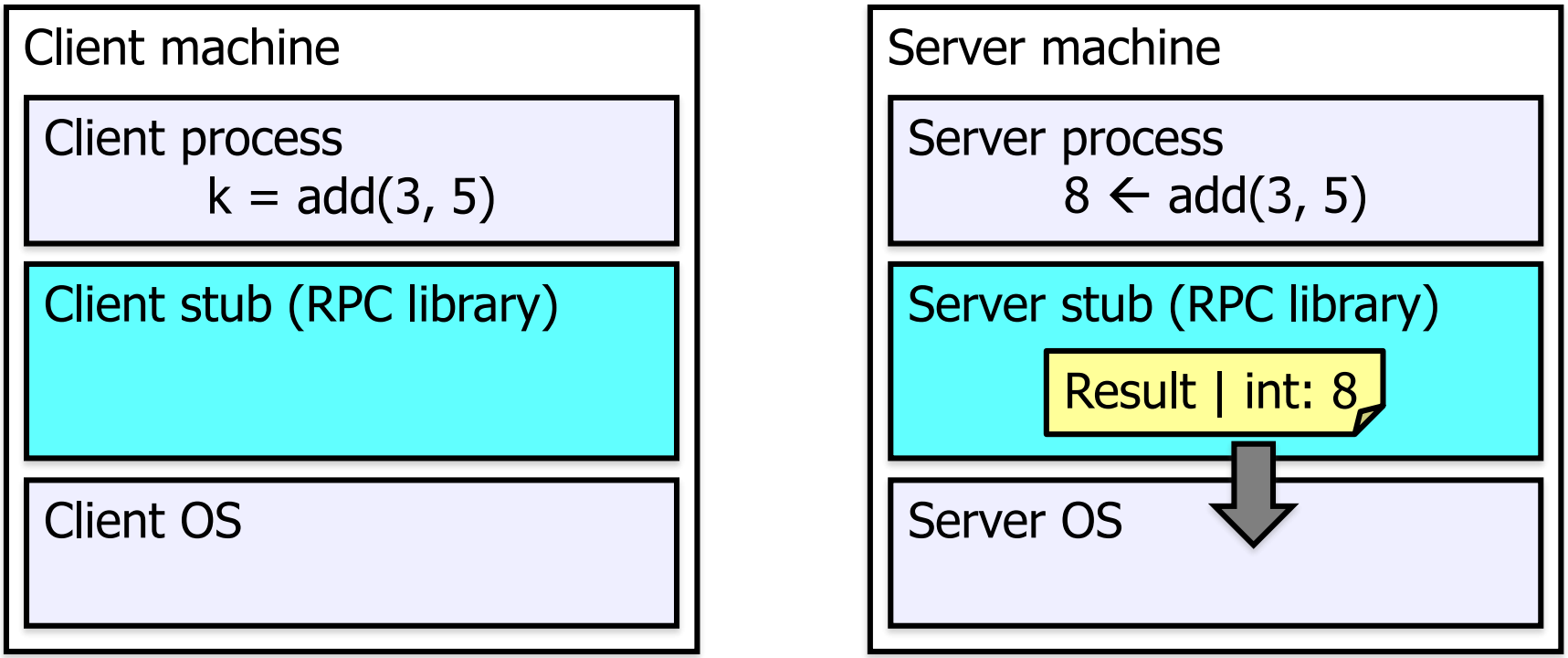
A Day in the Life of RPC

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



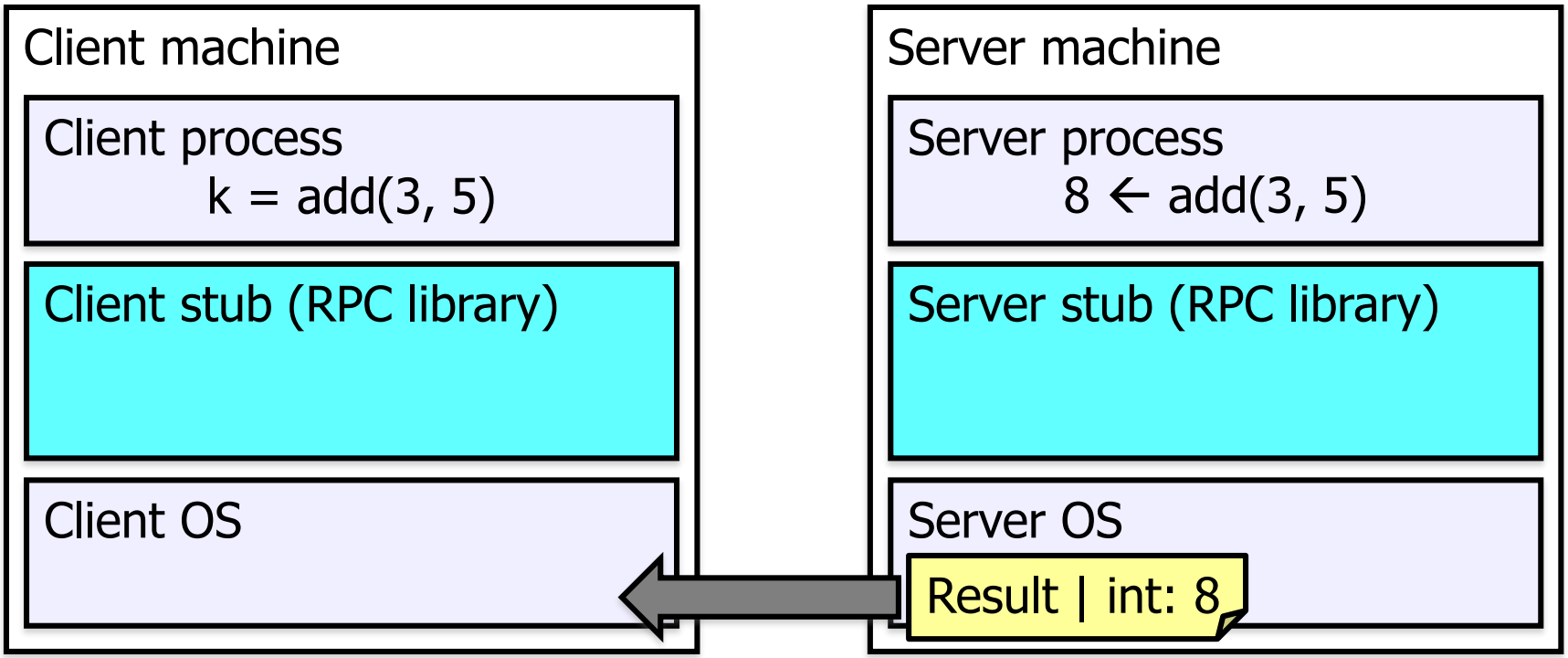
A Day in the Life of RPC

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



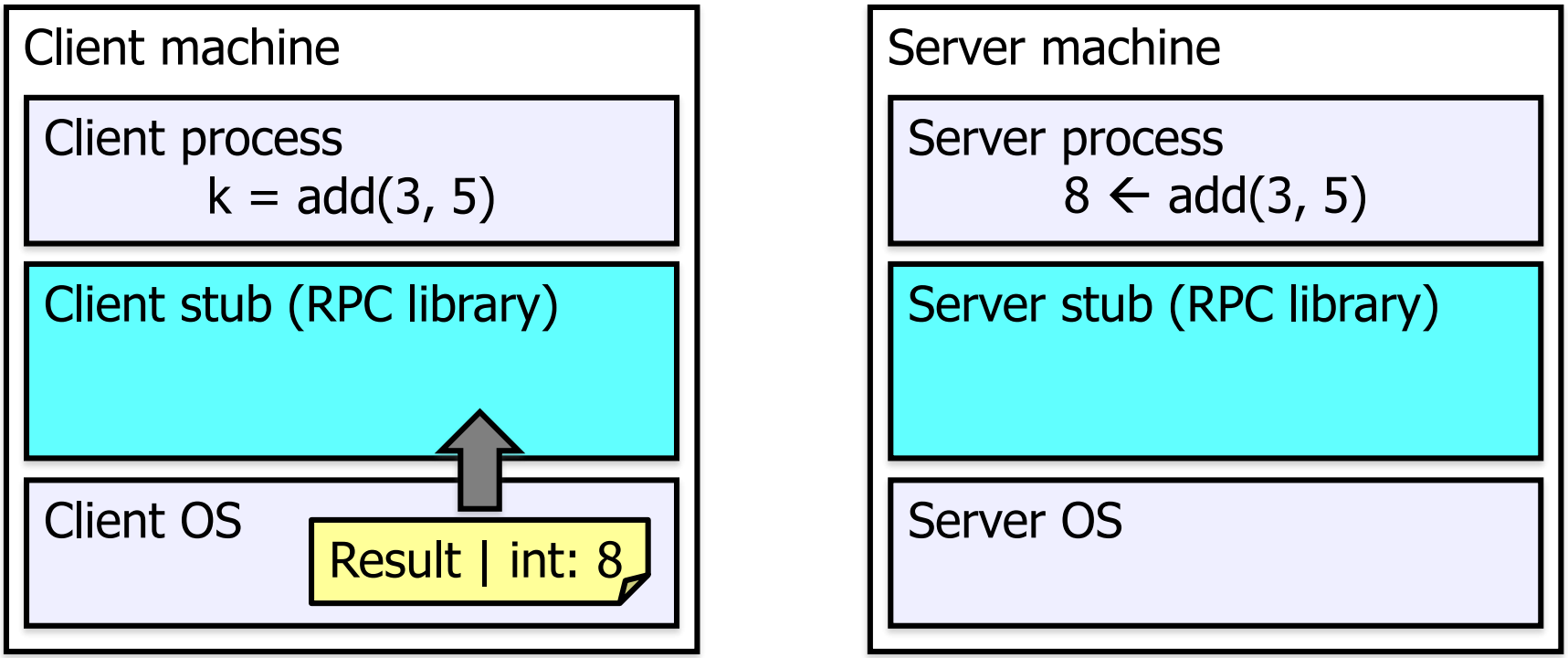
A Day in the Life of RPC

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



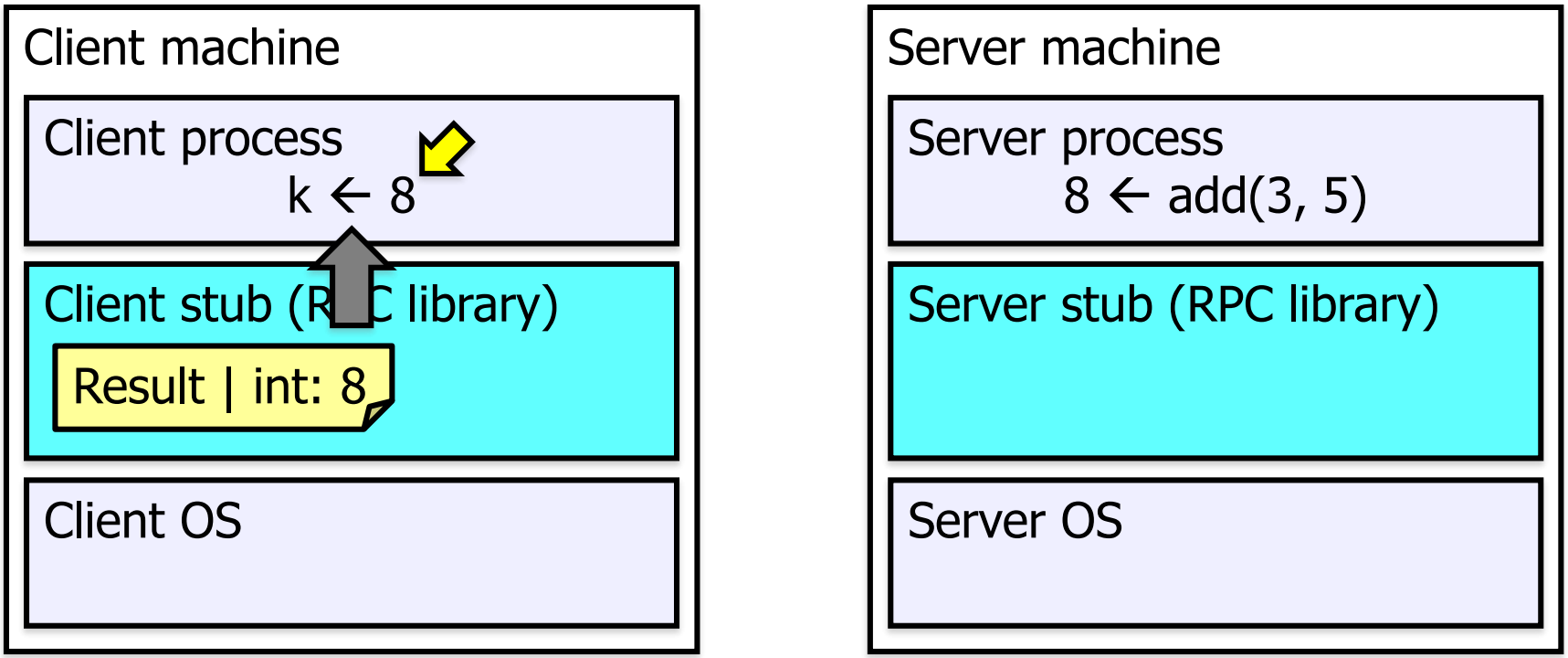
A Day in the Life of RPC

8. Server OS sends the reply back across the network
9. Client OS receives the reply and passes up to stub

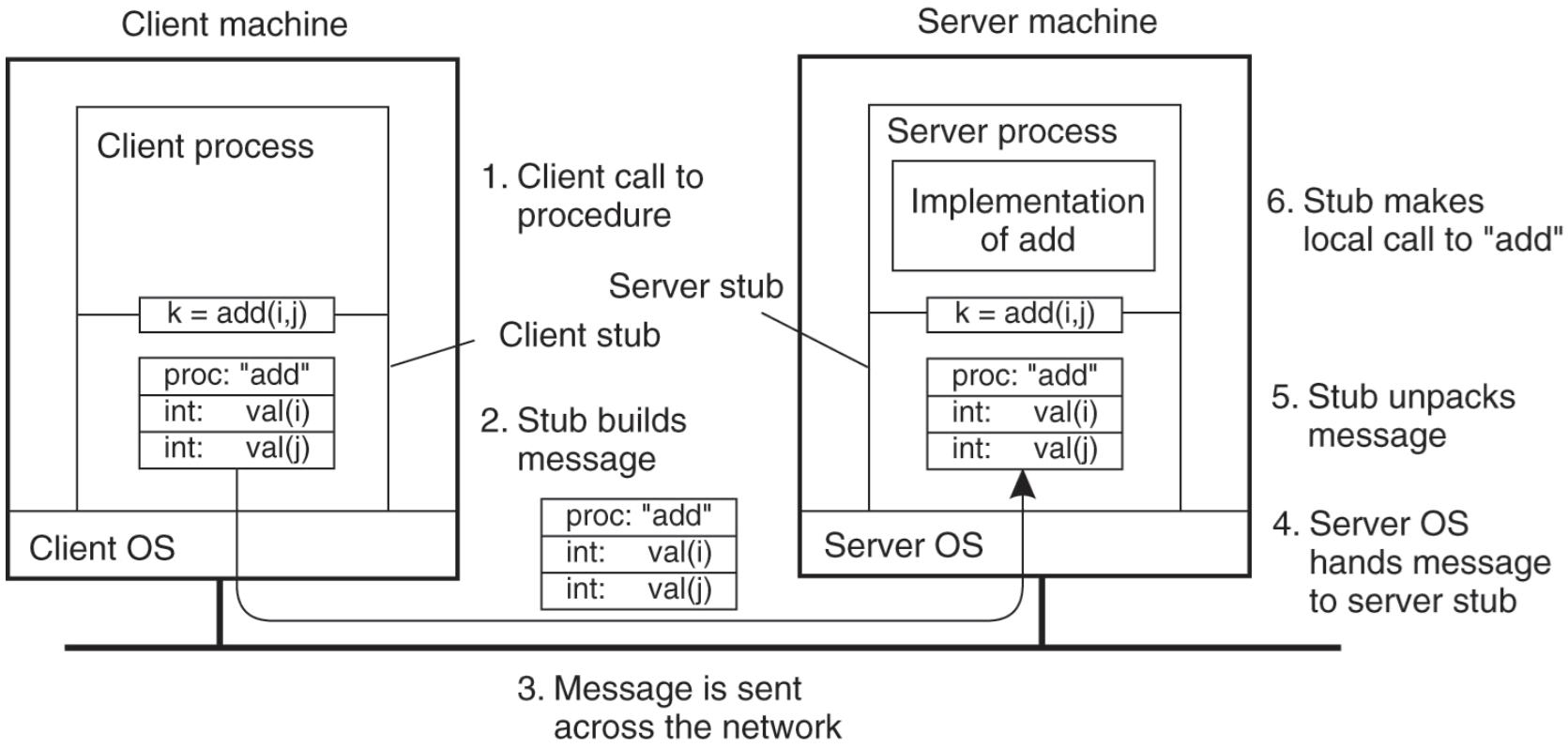


A Day in the Life of RPC

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

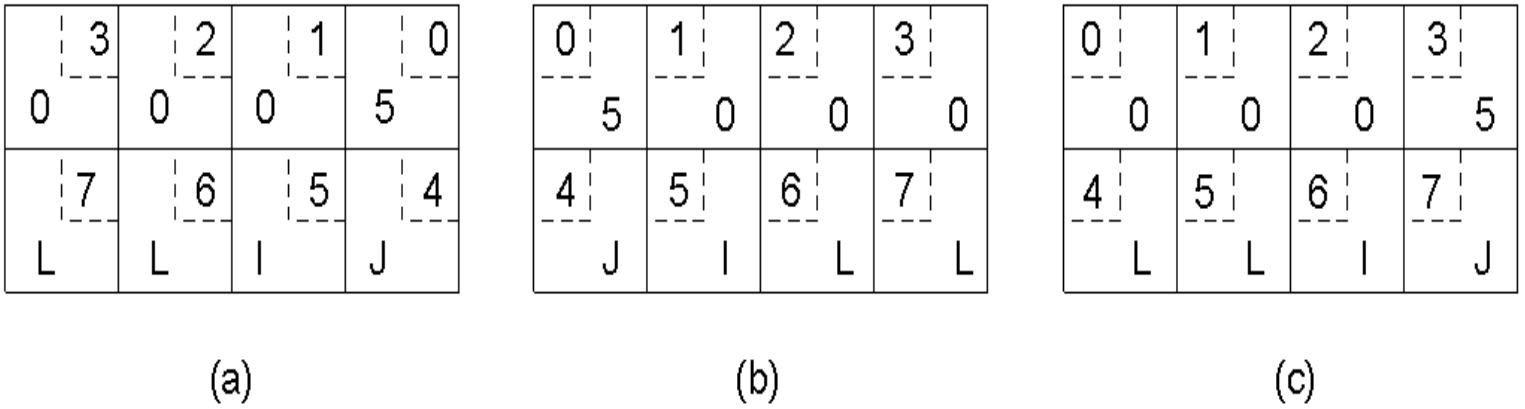


Passing Value Parameters (1)



Doing a remote computation through RPC

Passing Value Parameters (2)



a) Original message on x86
b) The message after receipt on the SPARC
c) The message after being inverted (the little numbers in boxes indicate the address of each byte)

Passing Reference Parameters

- Replace with pass by copy/restore
  - Need to know size of data to copy
    - Difficult in some programming languages
  - Solves the problem only partially
    - What about data structures containing pointers?
    - Access to memory in general?
- Distributed shared memory
  - People have kind of given up on it - it turns out often better to admit that you're doing things remotely

RPC vs. LPC

Achieving transparency is difficult!

- Partial failures
- Latency
- Memory access

RPC Failures

- Request from client to server lost
- Reply from server to client lost
- Server crashes after receiving request
- Client crashes after sending request

These look the same to client

Partial Failures

- Local computing:

  if machine fails, application fails
- Distributed computing:

  if a machine fails, part of application fails -

  cannot tell the difference between a machine failure and network failure
- How to make partial failures transparent to client?

Strawman Solution

- Make remote behavior identical to local behavior: every partial failure results in complete failure
  - Abort and reboot the whole system
  - Wait patiently until system is repaired
- Problems with this solution:
  - Many catastrophic failures
  - Clients block for long periods
  - System might not be able to recover

Real Solution: Break Transparency

Possible semantics for RPC

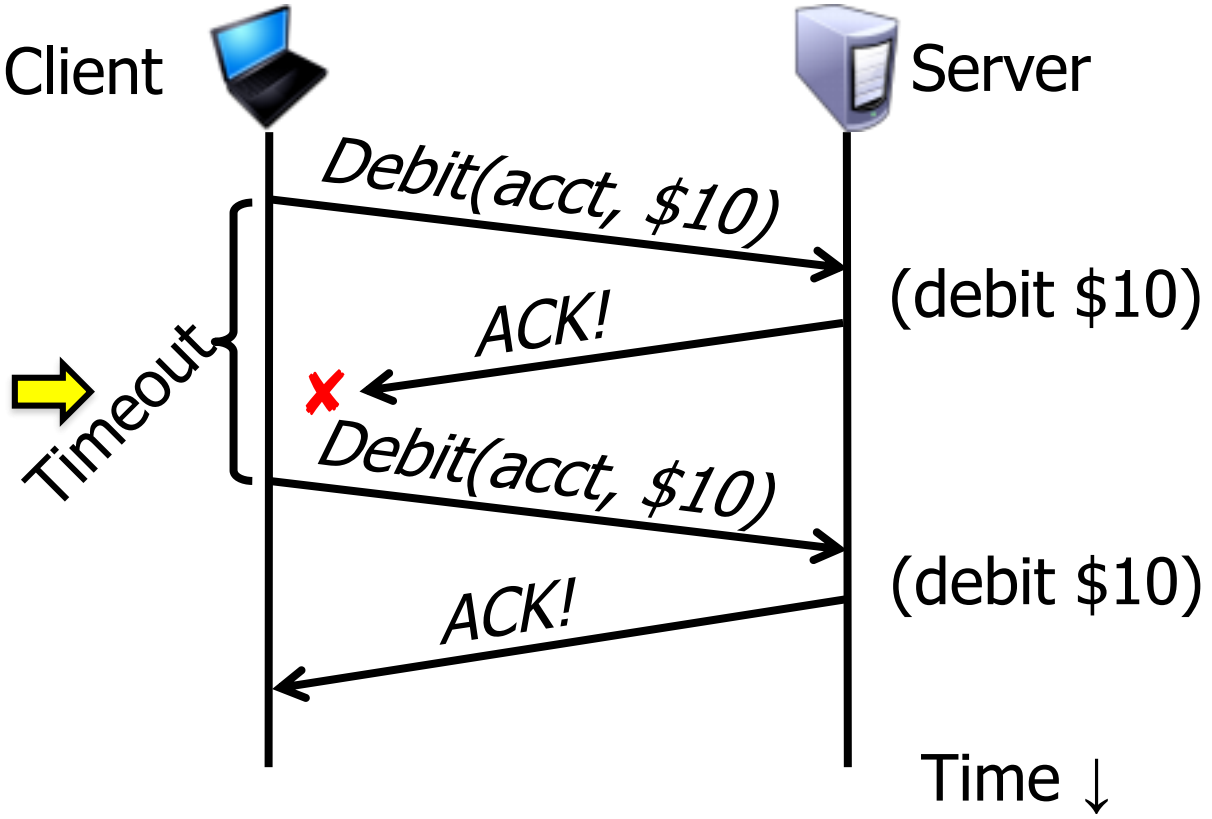
- Exactly-once
  - Impossible in practice
- At-least-once
  - Only for idempotent operations
- At-most-once
  - Zero, don't know, or once
- Zero-or-once
  - Transactional semantics

At-Least-Once

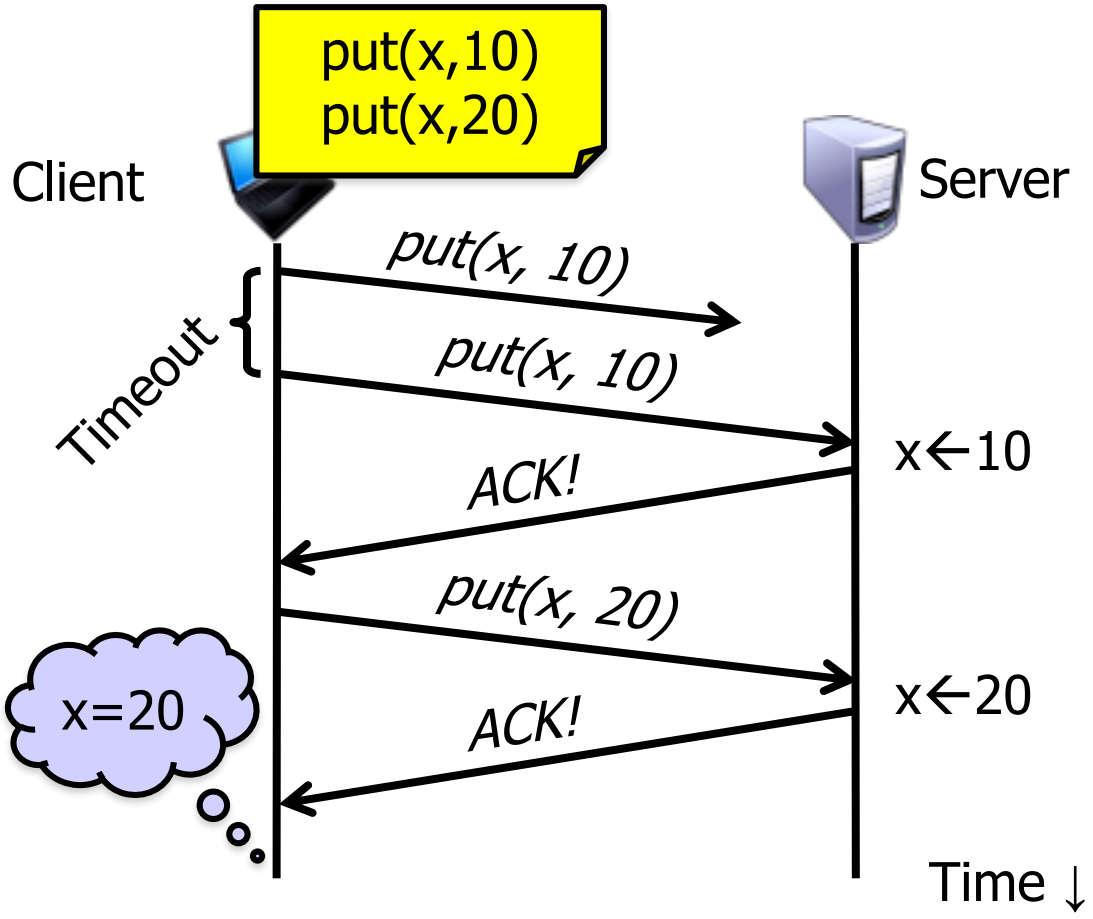
Keep retrying on client side until you get a response

Server just processes requests as normal, doesn't remember anything. Simple!

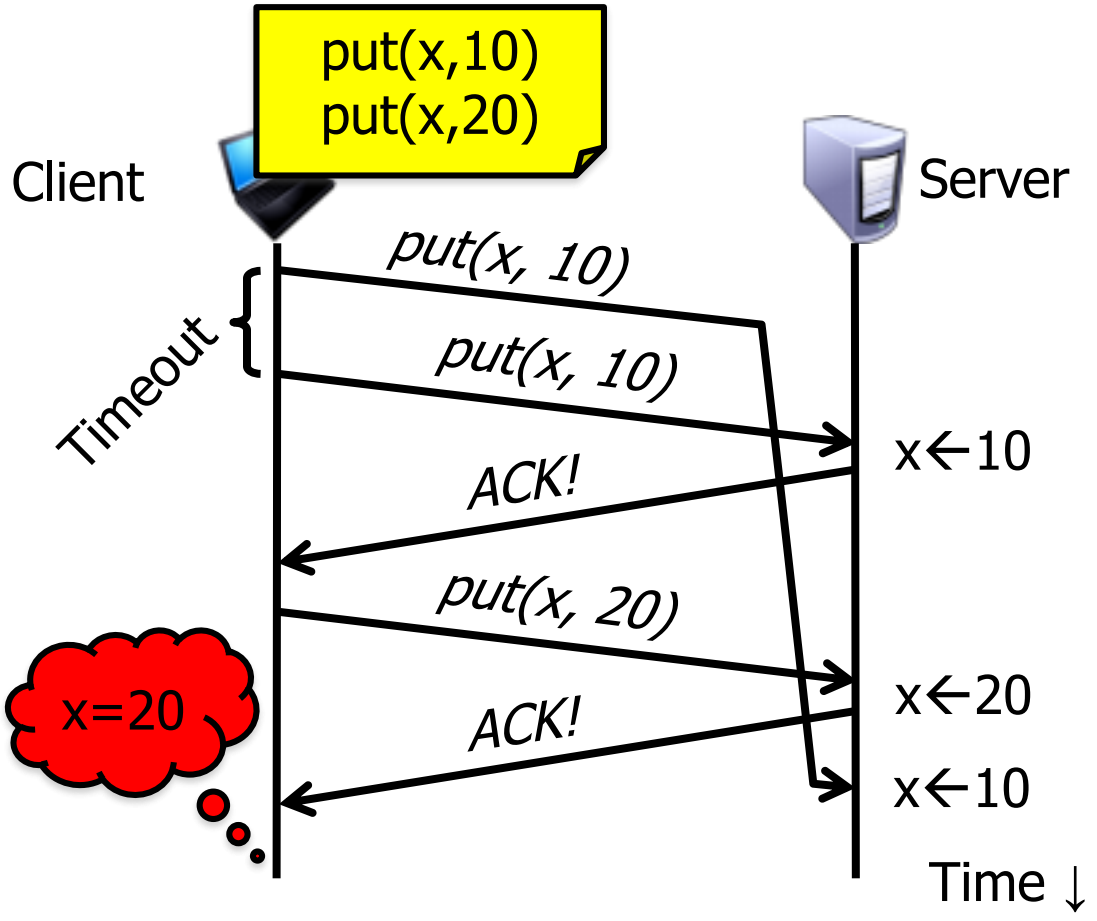
At-Least-Once and Side Effects



At-Least-Once and Writes



At-Least-Once and Writes



At-Least-Once is Ok ...

- ... for read-only operations with no side effects
  - Example: read a key's value in a database
- ... if the application has its own functionality to cope with duplication and reordering

At-Most-Once

Server might get same request twice...

- Must re-send previous reply, not process request
  - Implies: keep cache of handled requests/responses
  - Discard replies after client confirmed receipt (how?)
- Must be able to identify requests
  - ~~Same name, same arguments = same request~~
  - ~~Give each RPC an ID, remember all RPC IDs handled~~
  - Have client number RPC IDs sequentially, keep sliding window of valid RPC IDs
    - Never re-use IDs! Store on disk, or use boot time, or use big random numbers.

What Does the Client Know?

If the server receives request with invalid ID (or server has crashed), responds with exception...

What does the client know if they receive an exception from an at-most-once server?

Maybe the money was transferred, maybe it was not – but it was not transferred twice!

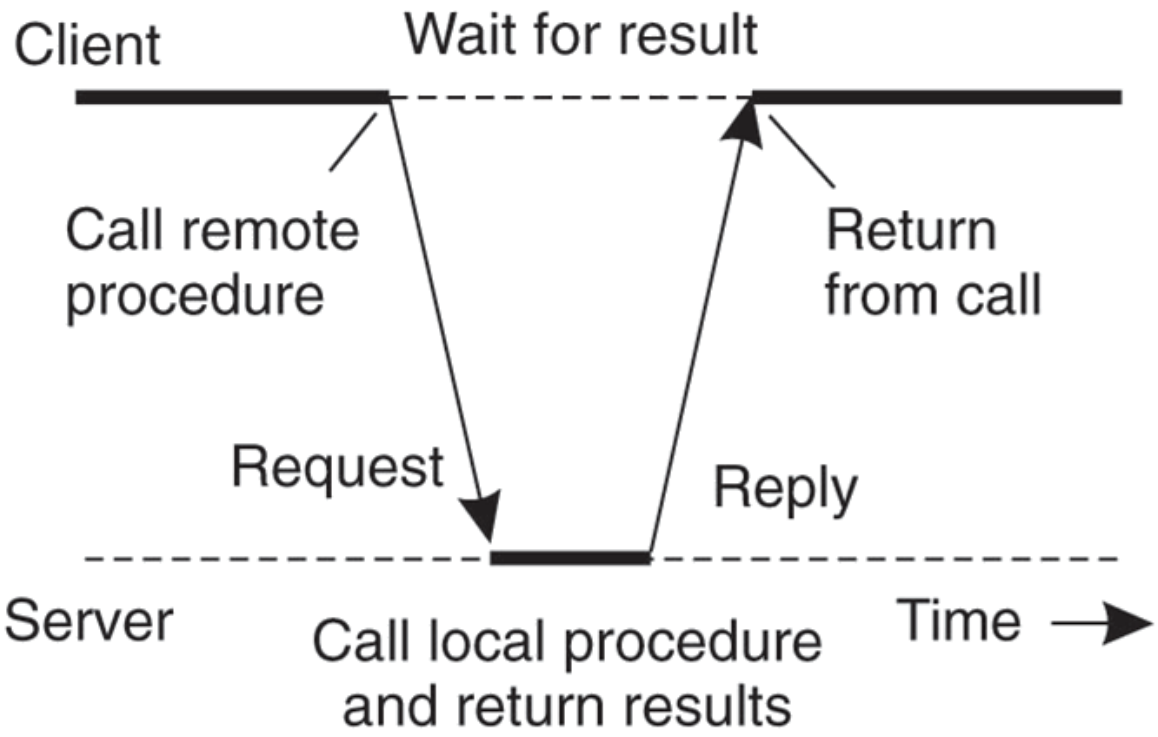
Exactly-Once

- Impossible in general
- Imagine that message triggers an external physical thing
  - Example: medical device injects medicine into patient
- The device could crash immediately before or after firing and lose its state - don't know which one happened
  - Can, however, make this window very small

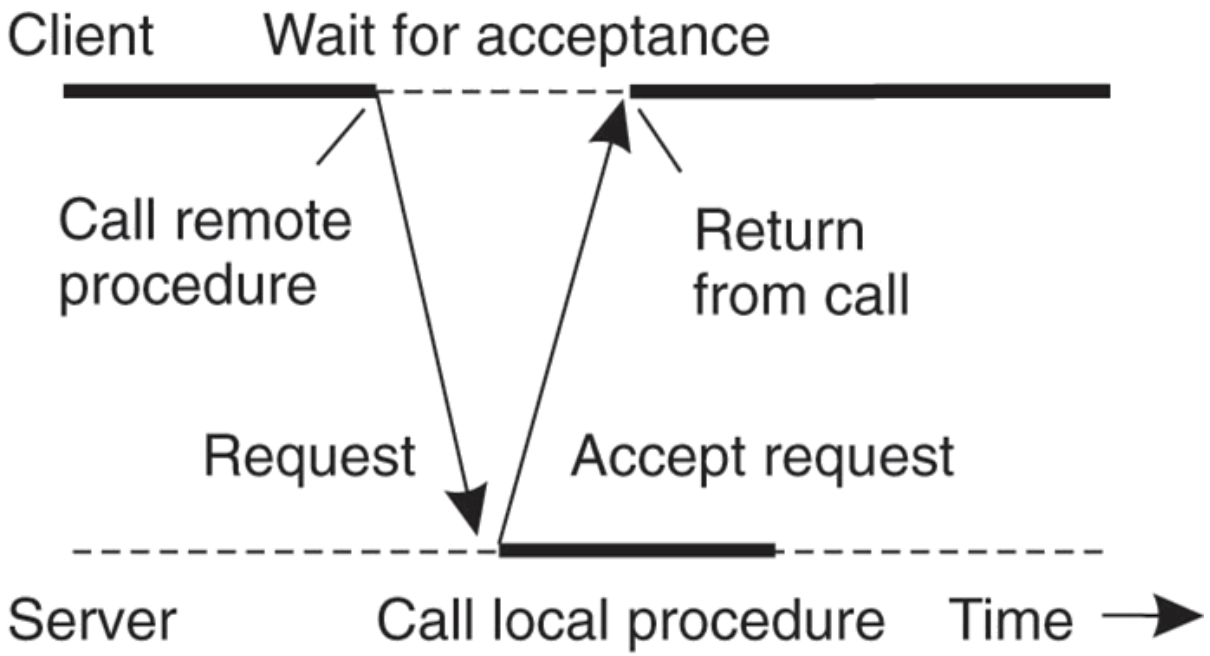
Implementation Concerns

- As a general library, performance is often a big concern for RPC systems
- Major source of overhead: copies and marshaling/unmarshaling overhead
- Zero-copy tricks
  - Representation: send on the wire in native format and indicate that format with a bit/byte beforehand. What does this do? Think about sending uint32 between two little-endian machines
- Scatter-gather writes - writev() and friends

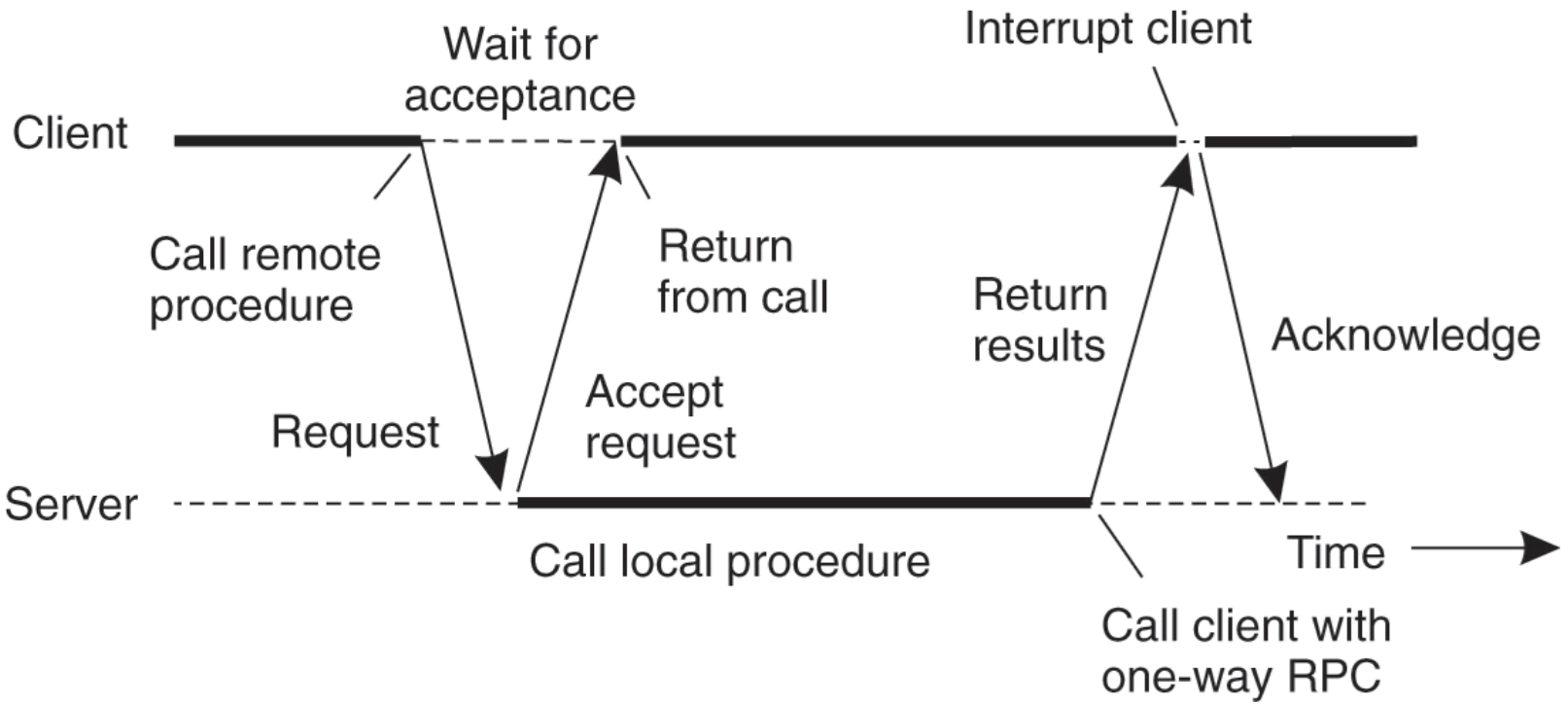
Traditional RPC



Asynchronous RPC (1)



Asynchronous RPC (2)



Example: Distributed Bitcoin Miner

- Request client
  - Submits cracking request to server, waits until response
- Worker
  - Initially a client, sends join request to server, then reverses role and becomes a server, receives requests from main server to attempt cracking over limited range
- Server (orchestrates the whole thing)
  - When receives request from client, splits it into smaller jobs over limited ranges, farms these out to workers. When finds bitcoin or exhausts complete range, responds to request client.

Using RPC

- Request→Server→Response
  - Classic synchronous RPC
- Worker→Server
  - Synchronous RPC, but no return value ("I'm a worker and I'm listening for you on host XXX, port YYY")
- Server→Worker
  - Synchronous RPC would be a bad idea: have to block while worker does its work, which misses the whole point of having many workers
  - Need asynchronous RPC

RPC Systems

- ONC RPC (a.k.a. Sun RPC)
  - Fairly basic, includes encoding standard XDR + language for describing data formats
- Java RMI (remote method invocation)
  - Very elaborate, tries to make it look like can perform arbitrary methods on remote objects
- Thrift
  - Developed at Facebook, open-sourced at Apache. Multiple data encodings and transport mechanisms
- Avro
  - From Hadoop, now part of Apache

Important Lessons

- Procedure calls
  - Simple way to pass control and data
  - Elegant, transparent way to distribute application
- Hard to provide true transparency
  - Failures, performance, memory access, etc.
- Practical solution: expose RPC properties to client since can't hide them
  - Application developer must decide how to deal with partial failures (example: e-commerce app vs. game)
  - "Worse is better"