CS 582: Distributed Systems

Remote Procedure Calls

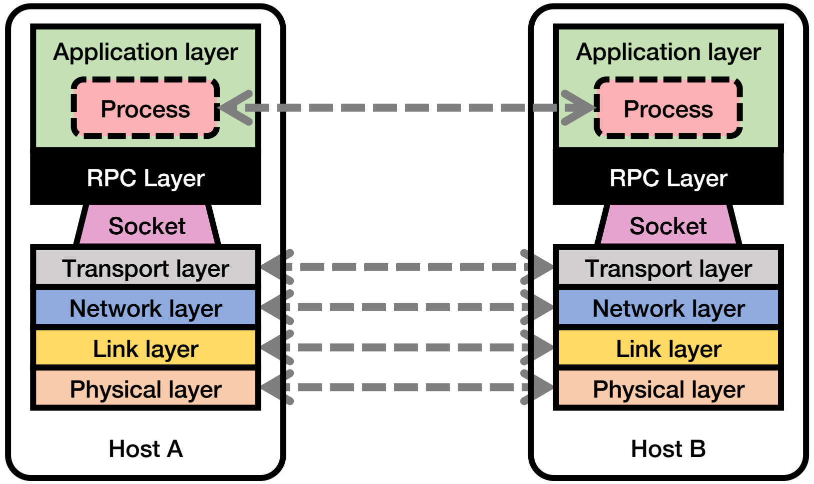
Dr. Zafar Ayyub Qazi

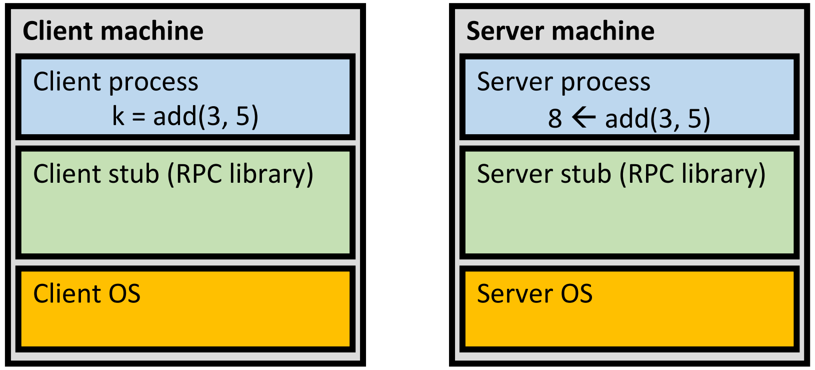
Fall 2024

Acknowledgements: Slides use some material from Wyatt Lloyd (Princeton University)

Remote Procedure Call (RPC)

- Why RPC?
- What are RPCs?
- Issues in implementing RPCs
- How do RPCs work?





Learning Outcomes: RPCs

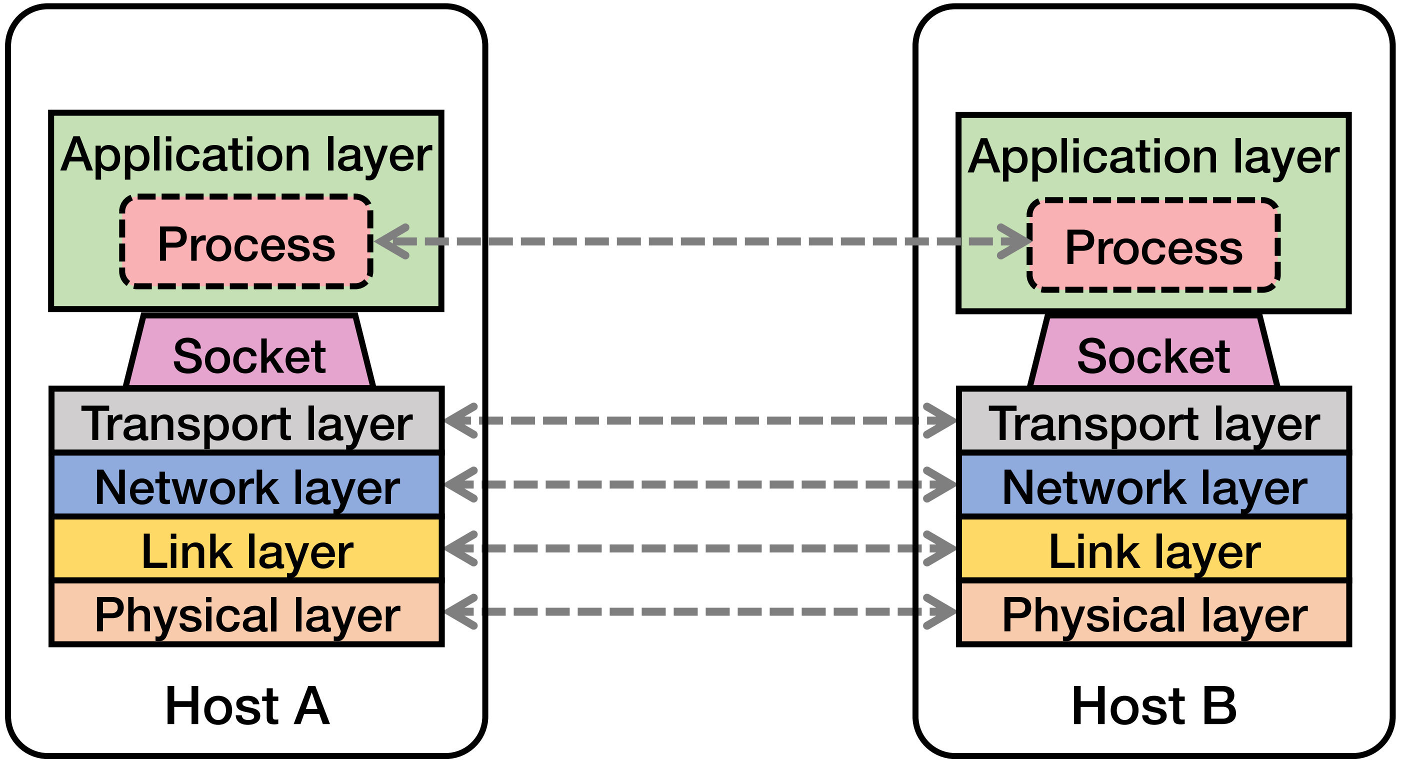
You should be able to:

- ❑ Define Remote Procedure Calls (RPCs) and explain their purpose in distributed systems
- ❑ Describe the challenges associated with implementing remote procedure calls
- ❑ Compare and contrast local and remote procedure calls, evaluating their respective advantages and disadvantages
- ❑ Analyze how the choice of transport protocol impacts RPC performance and delays
- ❑ Examine how failures affect RPC semantics and predict potential outcomes in failure scenarios
- ❑ Evaluate different strategies for handling failures in RPCs and propose appropriate solutions for given scenarios

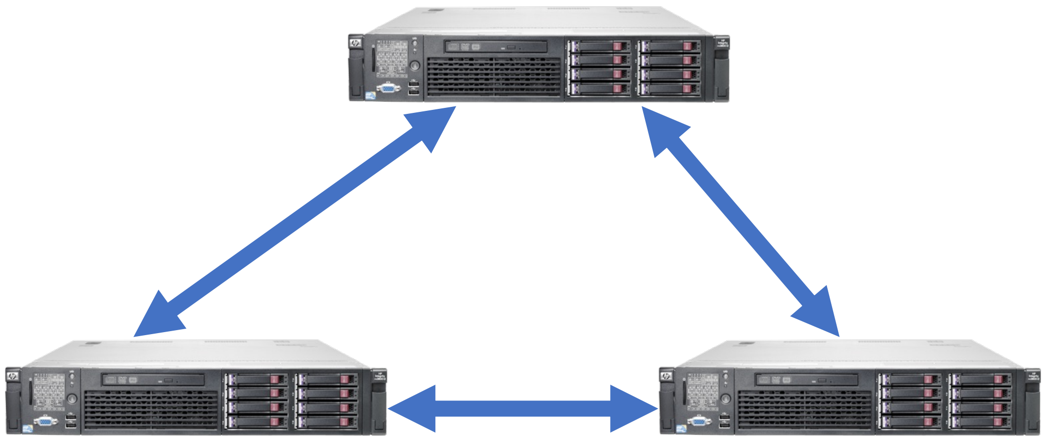
Everyone uses RPCs

- Google gRPC
- Facebook/Apache Thrift
- Twitter Finagle
- CS 582 assignments
- …

Why RPCs? Or whats wrong with sockets?



When designing a distributed system often there is a desire to make it "work like a single system"



Principle of Transparency

- Principle of transparency: Hide the fact that resources are physically distributed across multiple computers
  - Access resources the same way as we would do locally
  - Users can't tell where resources are physically located

Simple socket-based program

```c
if ((sockfd = socket (AF_INET, SOCK_STREAM, 0)) < 0) {
  perror(”Socket creation");
  exit(2);
}

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

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

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);
```

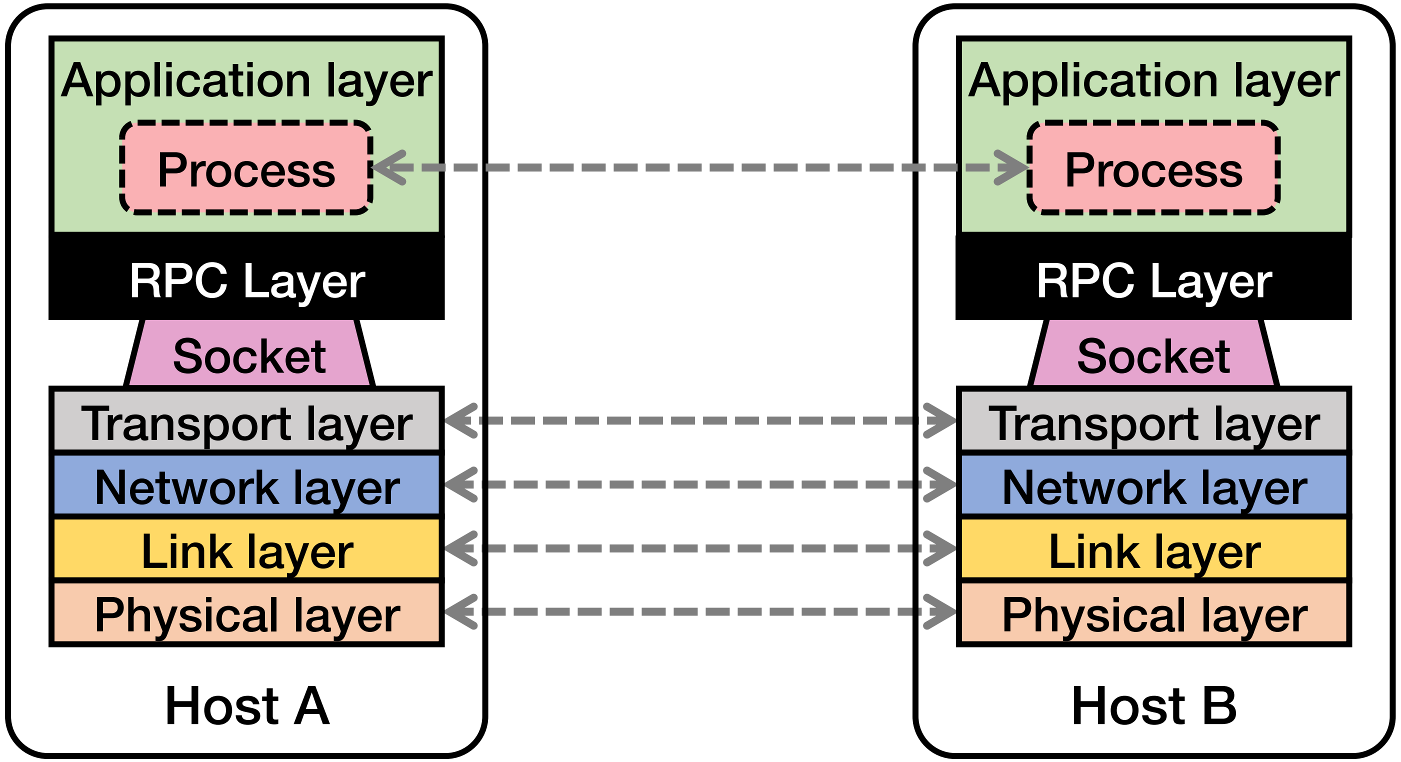
Network sockets: Summary

- Network sockets provide apps with point-to-point communication between processes
  - Only interface to the network provided by the OS
- However, the socket interface forces us to design distributed applications using a read/write interface
- put(key, value) → message with sockets

"All problems in computer science can be solved by another level of indirection."

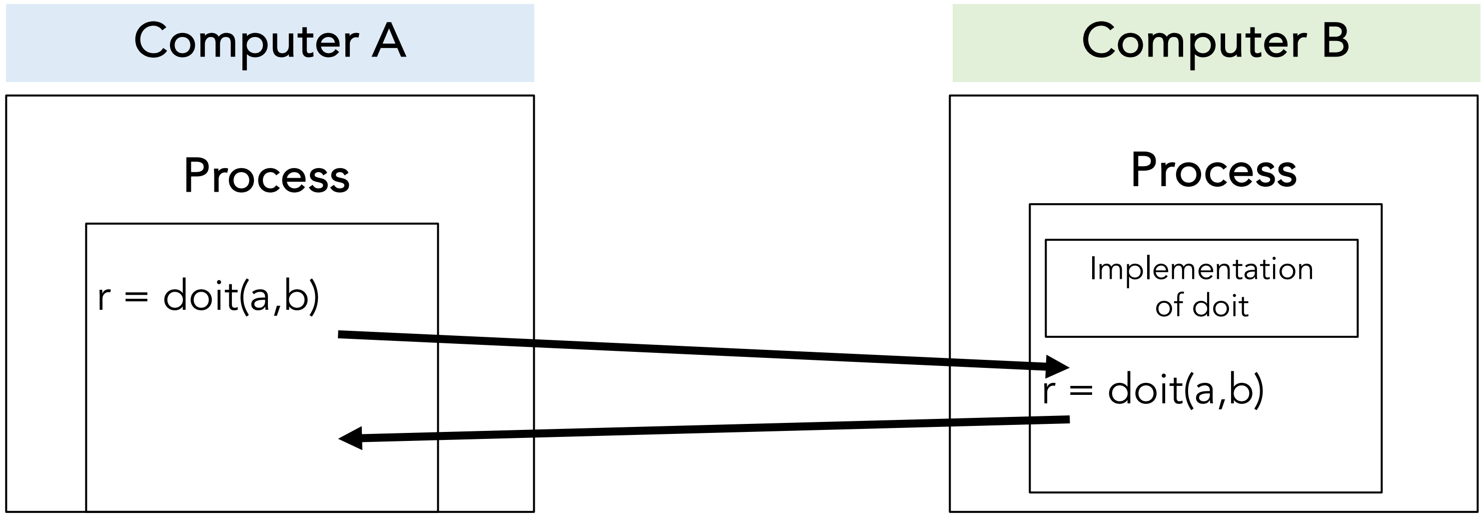
— **David Wheeler**

Solution: Another layer!



RPC Goal

- To allow programs to call procedures located on other machines *as if they were local*



RPC Goal: To make network communication appear like a local procedure call—transparency for procedure calls

Historical note: Seems obvious in retrospect, but RPC was only invented in the '80s.
See Birrell & Nelson, "Implementing Remote Procedure Call" … Or Bruce Nelson, Ph.D. Thesis, Carnegie Mellon University: Remote Procedure Call, 1981

"All problems in computer science can be solved by another level of indirection"

"But that usually will create another problem."

— David Wheeler

RPC issues

1. Heterogeneity
   - Dealing with differences in data representation on different machines
2. Failure
   - What if messages get dropped?
   - What if client, server, or network fails?
3. Performance
   - Local procedure call takes ≈ 10 cycles ≈ 3 ns
   - RPC in a data center takes ≈ 10 μs (10^3X slower)
     - In the wide area, typically 10^6X slower

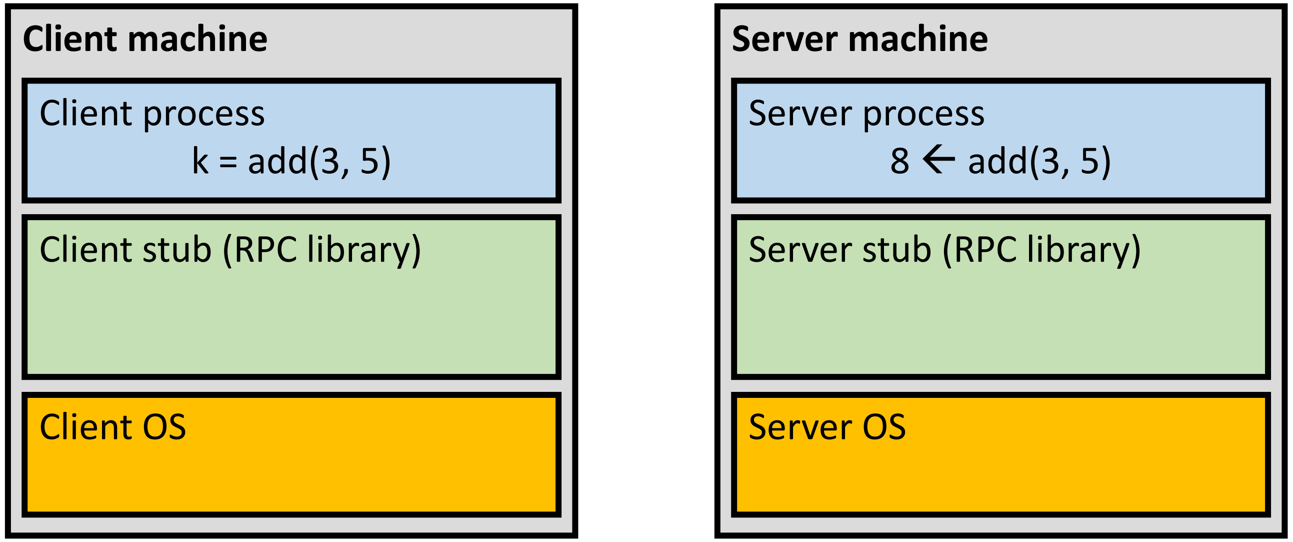
Problem: Differences in data representation

- Not an issue for local procedure calls
- For a remote procedure call, a remote machine may:
  - Use a different byte ordering (endianness)
  - Run process written in a different language
  - Represent data types using different sizes
  - Represent floating point numbers differently

Solution: Interface Description Language (IDL)

- Mechanism to pass procedure parameters and return values in a machine-independent way
  - E.g., Protocol Buffers, Apache Thrift
- A 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

How do RPCs work?

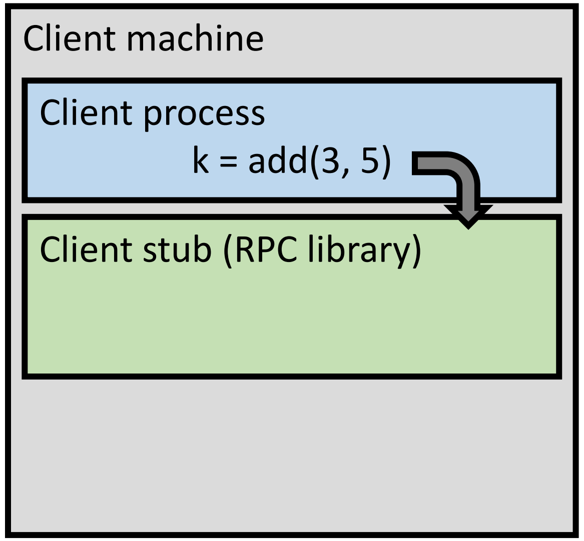


- Client stub: Forwards local procedure call as a request to the server
- Server stub: Dispatches RPC to its implementation

A day in the life of an RPC

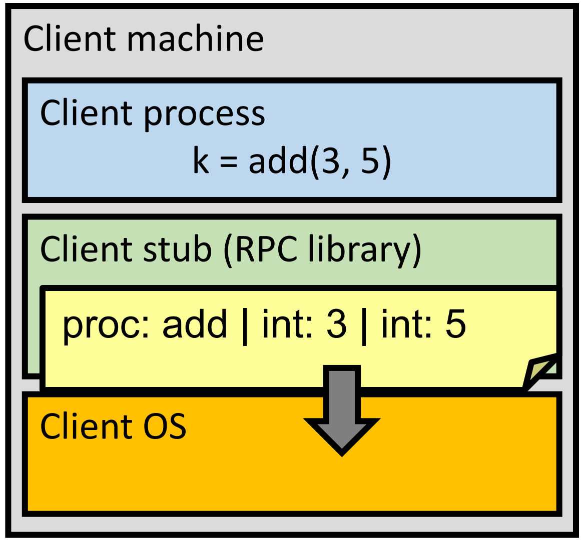
A day in the life of an RPC

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



A day in the life of an RPC

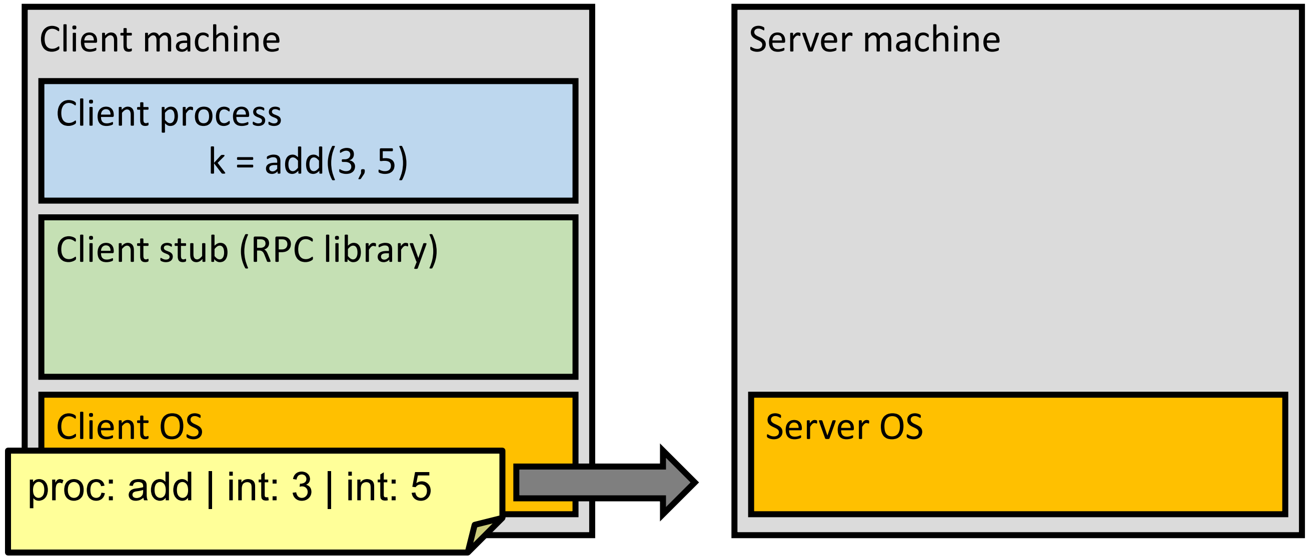
1. Client calls stub function (pushes parameters onto stack)
2. Stub marshals parameters to a network message



A day in the life of an RPC

2. Stub marshals parameters to a network message

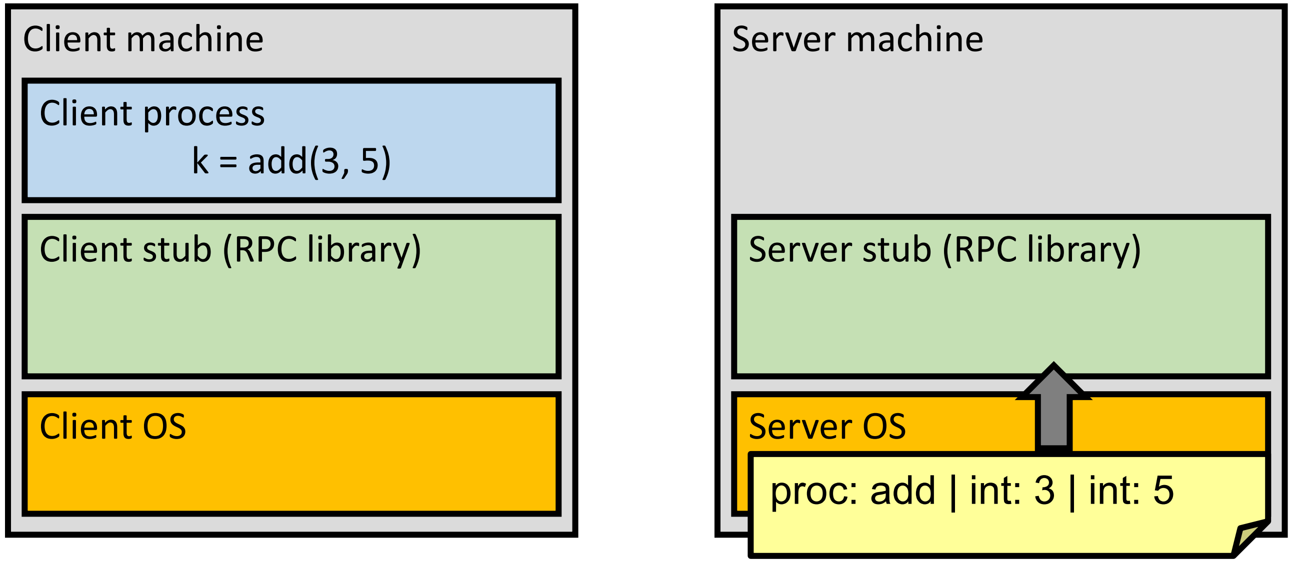
3. OS sends a network message to the server



A day in the life of an RPC

3. OS sends a network message to the server

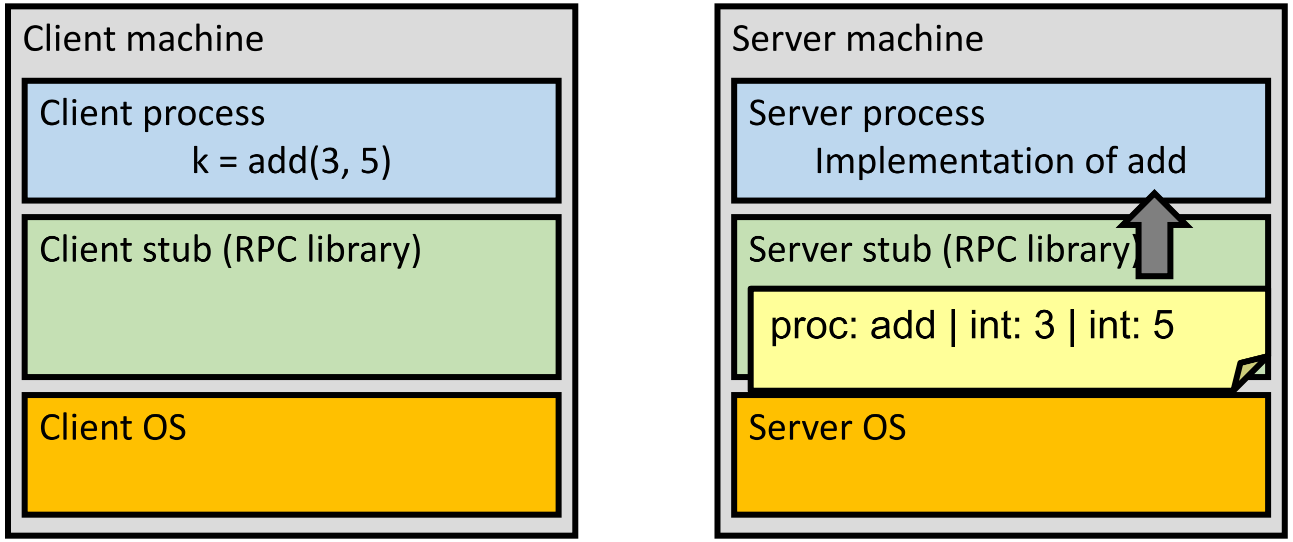
4. Server OS receives message, sends it up to stub



A day in the life of an RPC

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

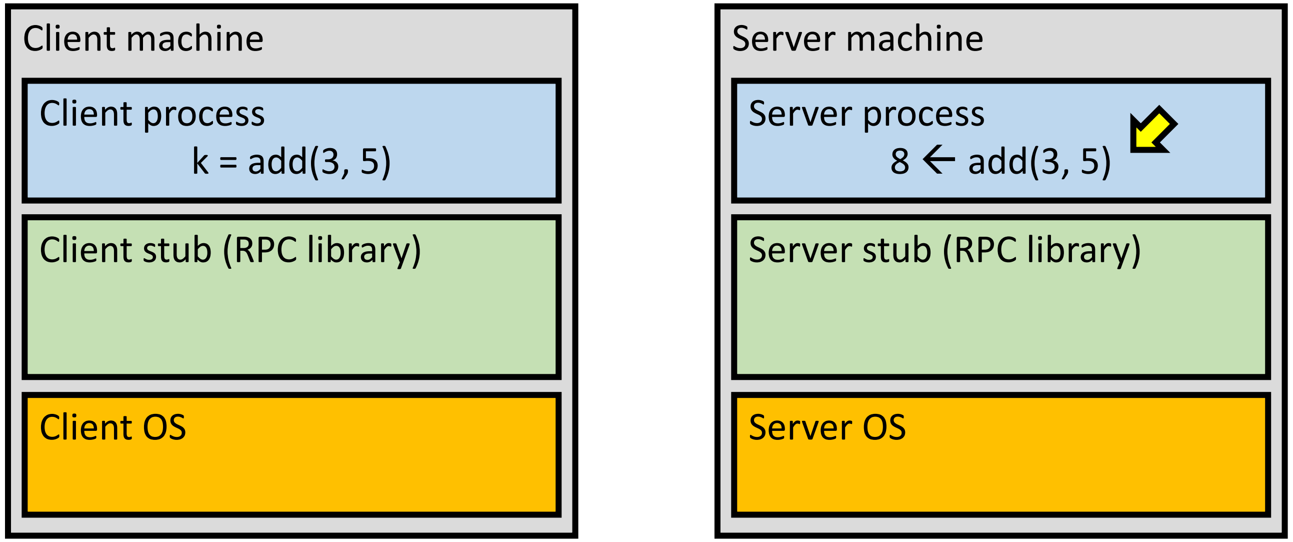
5. Server stub unmarshals params, calls server function



A day in the life of an RPC

5. Server stub unmarshals params, calls server function

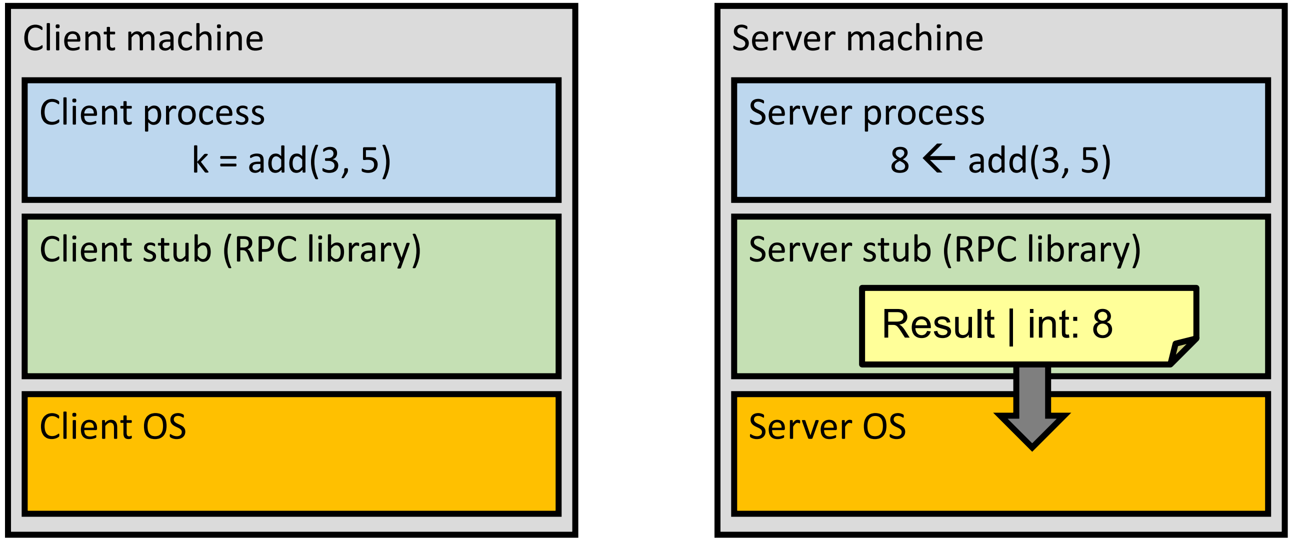
6. Server function runs, returns a value



A day in the life of an RPC

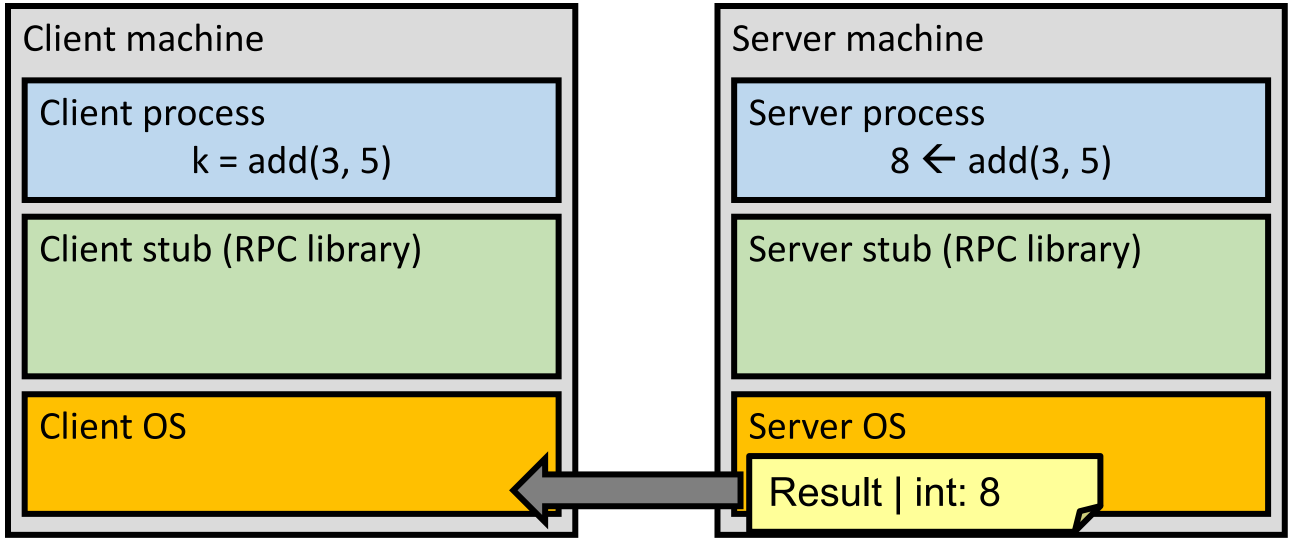
6. Server function runs, returns a value

7. Server stub marshals the return value, sends message



A day in the life of an RPC

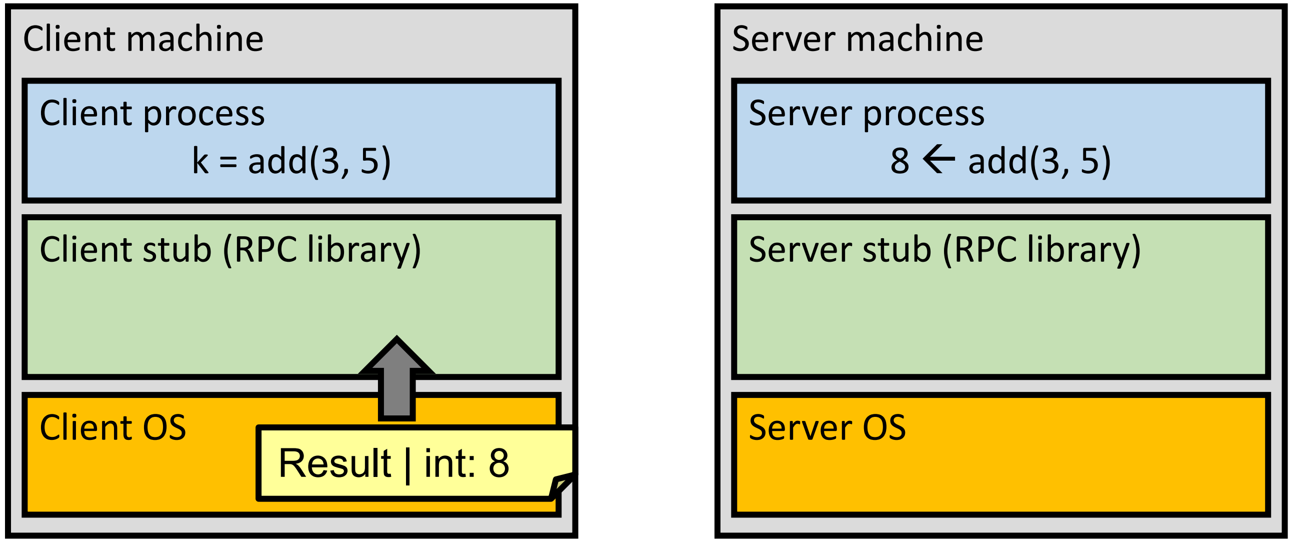
7. Server stub marshals the return value, sends message
8. Server OS sends the reply back across the network



A day in the life of an RPC

8. Server OS sends the reply back across the network

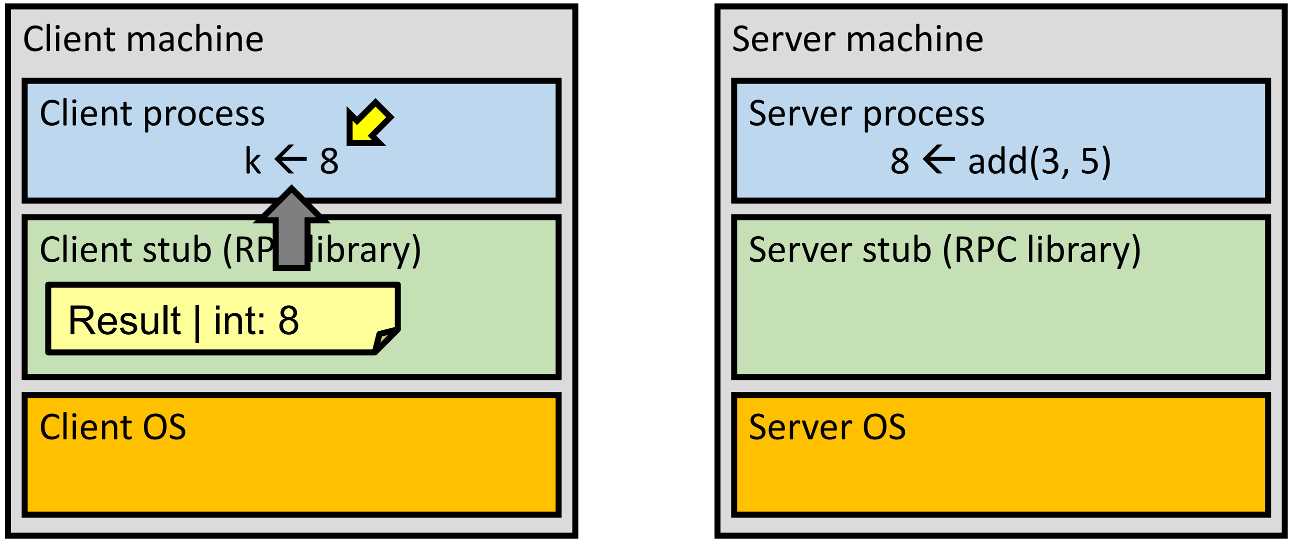
9. Client OS receives the reply and passes up to stub



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



Passing Value Parameters

- Copy-by-value

What about passing parameters by reference?

RPC: Passing Reference Parameters

- Replace copy-by-reference by call-by-copy/restore

Question: Would this always give us the same results as passing by reference?

What to bind to?

- Need to locate a remote host and the proper process (port)
- Solution1:
    - Maintain a centralized database that can locate a host that provides a type of service (proposed by Birrell and Nelson in 1984)
    - A server sends a message to a central authority stating its willingness to accept certain remote procedure calls
    - Clients contact this central authority when they need to locate a service
- Solution 2:
    - Require the clients to know which hosts they needs to contact

Rest of the lecture

- Failure Handling in RPCs

Failures during RPCs

What do we want in RPC systems?

- Exactly-once semantics, like local procedure calls

RPC Semantics

- Exactly-Once
  - Remote procedure will be executed exactly once on the server
- At-Least-Once
  - Remote procedure may execute more than once
  - Possible Side effects
- At-Most-Once:
  - Remote procedure will not execute more than once

At-Most-Once RPC semantics

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

Detecting duplicate requests

- 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 = remote_proc()
       old[xid] = retval
       seen[xid] = true
return retval
```

At-Most-Once: Providing unique XIDs

- Combine a unique client ID with a sequence number
  - MAC address + a strictly increasing number
  - Static IP address + current time of day
- What might not work?
  - Big random number (probabilistic, no guarantees that it will be unique)
  - DHCP assigned IP + seq. number (IP might change across reboots)
    - If the client crashes and restarts, it will have a different client ID

At-Most-Once: Issues?

At-Most-Once: Discarding server state

- Problem: seen and old tables 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: server state 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: Concurrent Updates

- 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 state to disk (persistent memory)?
- Yes! On server crash and restart:
  - If state is only in volatile memory (e.g., RAM):
    - 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)
- However: Go RPC returns an error if it doesn't get a reply
  - After a TCP timeout
  - Perhaps server didn’t see request
  - Perhaps server processed request but server/net failed before reply came back

Exactly-Once RPC semantics?

- Need retransmission
- Plus the duplicate filtering of at most once scheme
  - To survive client crashes, client also needs to make sure it has the same unique id after restart OR
  - The client should 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, the system needs to continue with full state
  - To survive server crashes, server should log to disk results of completed RPCs (to suppress duplicates)

Synchronous and asynchronus RPCs

- Synchronous remote procedure call is a blocking call
  - I.e., when a client has made a request to the server, the client will wait until it receives a response from the server

- Asynchronous RPC: Client does not wait for a response
  - This is useful when the RPC call is a long-running computation on the server, meanwhile, client can continue execution.

Summary of RPCs

- RPCs key building block, used commonly
- Make network comm. appear like a local procedure call
- Issues surrounding machine heterogeneity
- Subtle issues around failures
  - Need retransmissions to deal with failures
  - At-most-once w/ duplicate filtering
    - Discard server state w/ cumulative acks
  - Exactly-once with:
    - retransmissions + at-most-once + fault tolerance (of servers)