REMOTE PROCEDURE CALLS

George Porter
Module 3
Fall 2020

UC San Diego

Outline

1. RPC fundamentals
2. RPC Implementations
3. Handling failures in RPCs

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

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 EXAMPLE (PSEUDOCODE)

Local computing

```
X = 3 * 10;

print(X)

> 30
```

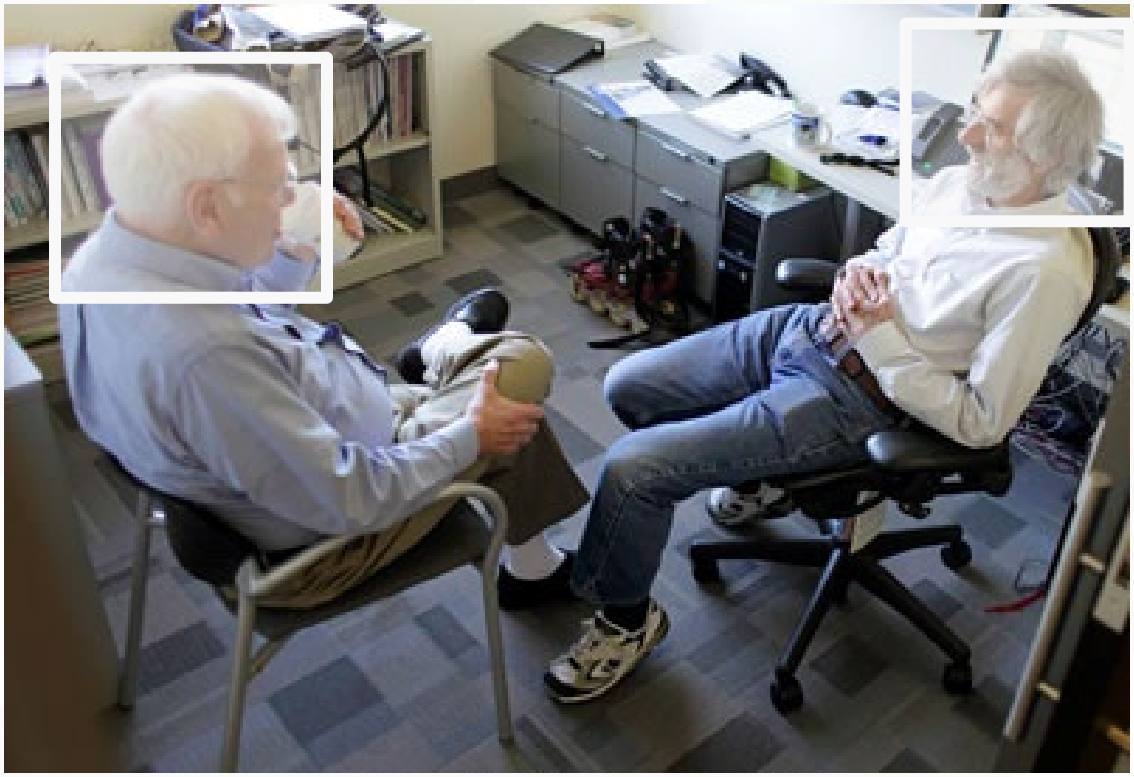
Remote computing

```
server = connectToServer(S);

Try:
   X = server.mult(3,10);
   print(X)
Except e:
   print “Error!”
> 30

or

> Error
```

RPC ISSUES

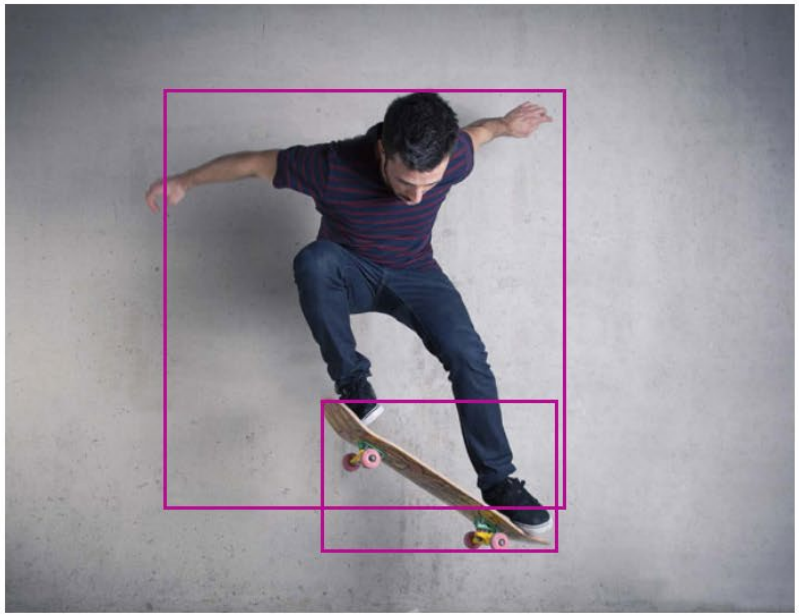
- Heterogeneity
  - Client needs to **rendezvous** with the server
  - Server must **dispatch** to the required function
    - What if server is **different** type of machine?
- Failure
  - What if messages get dropped?
  - What if client, server, or network fails?
- Performance
  - Procedure call takes ≈ 10 cycles ≈ 3 ns
  - RPC in a data center takes ≈ 10 µs (10^3× slower)
    - In the wide area, typically 10^6× slower

CLOUD-HOSTED RPC



- Functions that just can’t run locally
  - tags = Facebook.MatchFacesInPhoto(photo)
  - Print tags: [“Chuck Thacker”, “Leslie Lamport”]

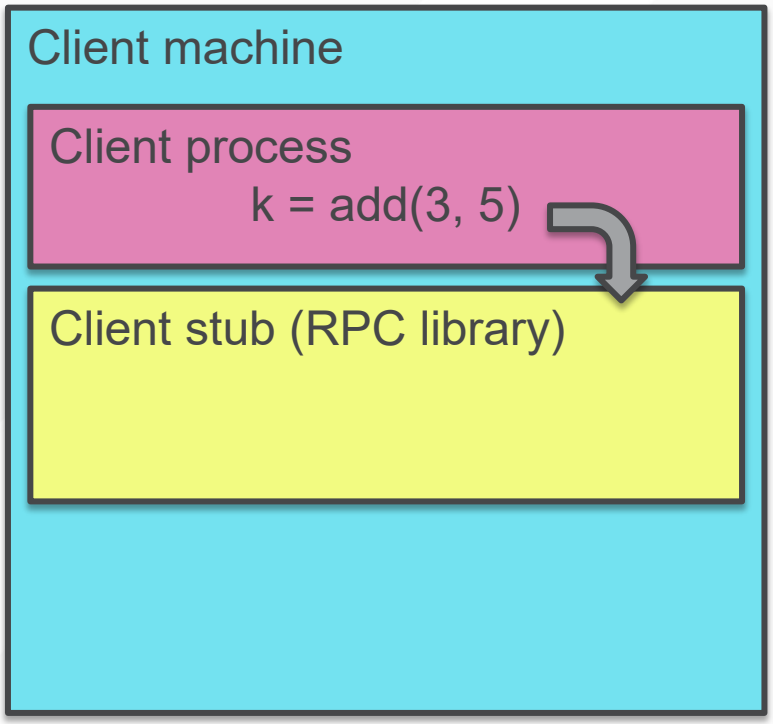
MICROSOFT AZURE VISION API



Objects	[{ "rectangle": { "x": 238, "y": 299, "w": 177, "h": 117 }, "object": "Skateboard", "confidence": 0.903 }, { "rectangle": { "x": 118, "y": 63, "w": 305, "h": 321 }, "object": "person", "confidence": 0.955 }]
Tags	[{ "name": "skating", "confidence": 0.999951541 }, { "name": "snowboarding", "confidence": 0.990067363 }, { "name": "sports equipment", "confidence": 0.9774853 }, { "name": "person", "confidence": 0.9605776 }, { "name": "roller skating", "confidence": 0.945730746 }, { "name": "boardsport", "confidence": 0.9242261 }, { "name": "man", "confidence": 0.9188208 }, { "name": "outdoor", "confidence": 0.9107821 }, { "name": "riding", "confidence": 0.900007248 }, { "name": "skiing", "confidence": 0.894337356 }, { "name": "footwear", "confidence": 0.8788208 }, { "name": "sport", "confidence": 0.86974 }, { "name": "skateboarder", "confidence": 0.840728462 }, { "name": "snowboard", "confidence": 0.834259868 }, { "name": "skateboarding equipment", "confidence": 0.831454 }, { "name": "individual sports", "confidence": 0.824958563 }, { "name":

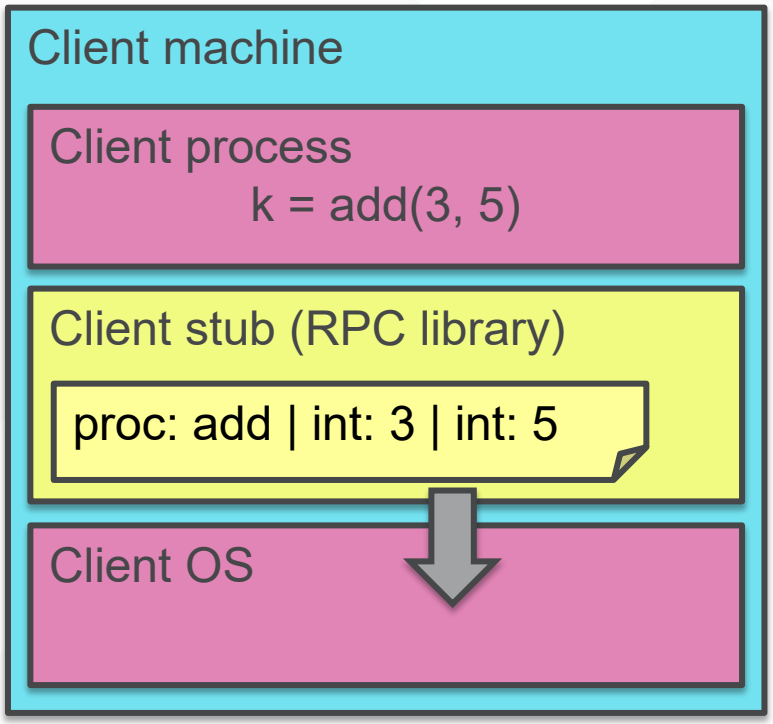
A DAY IN THE LIFE OF AN RPC

1. **Client calls stub function (pushes params onto stack)**



A DAY IN THE LIFE OF AN RPC

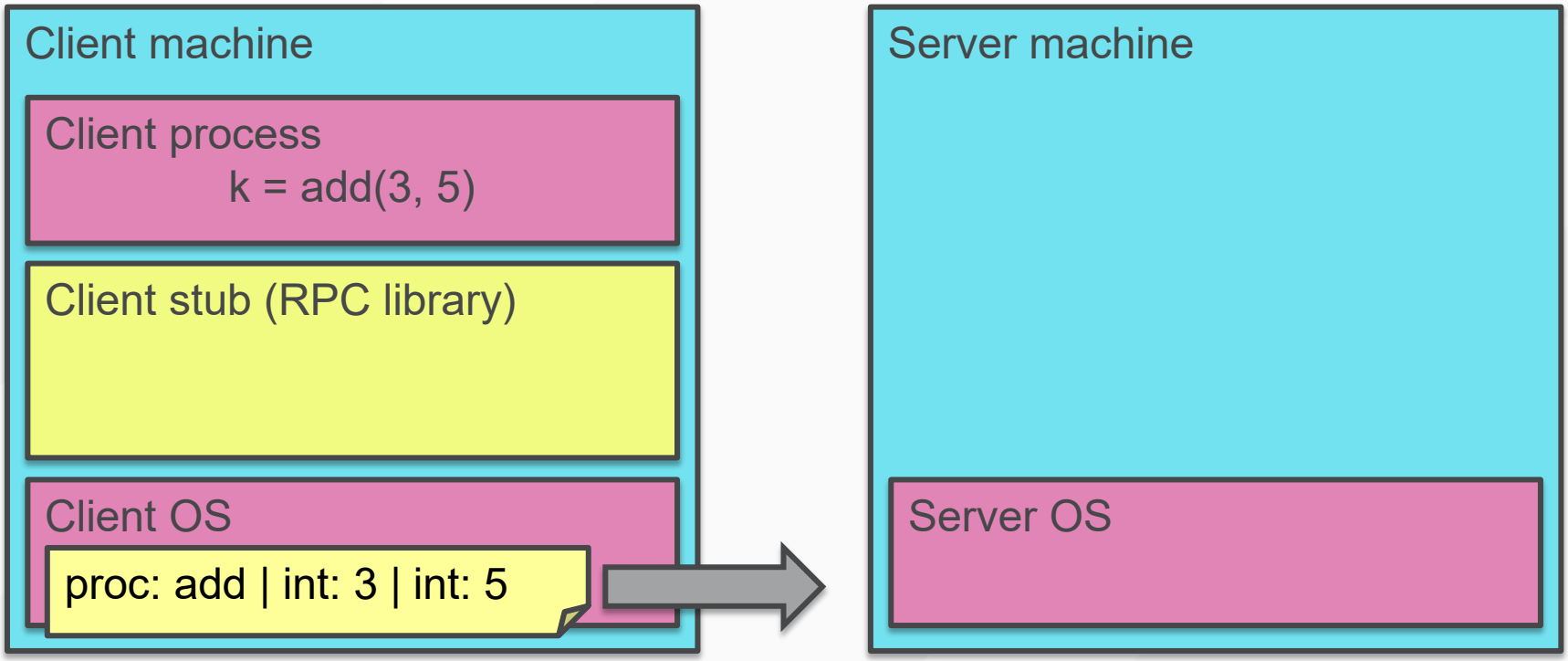
1. Client calls stub function (pushes params onto stack)
2. **Stub marshals parameters to a network message**



A DAY IN THE LIFE OF AN RPC

2. Stub marshals parameters to a network message

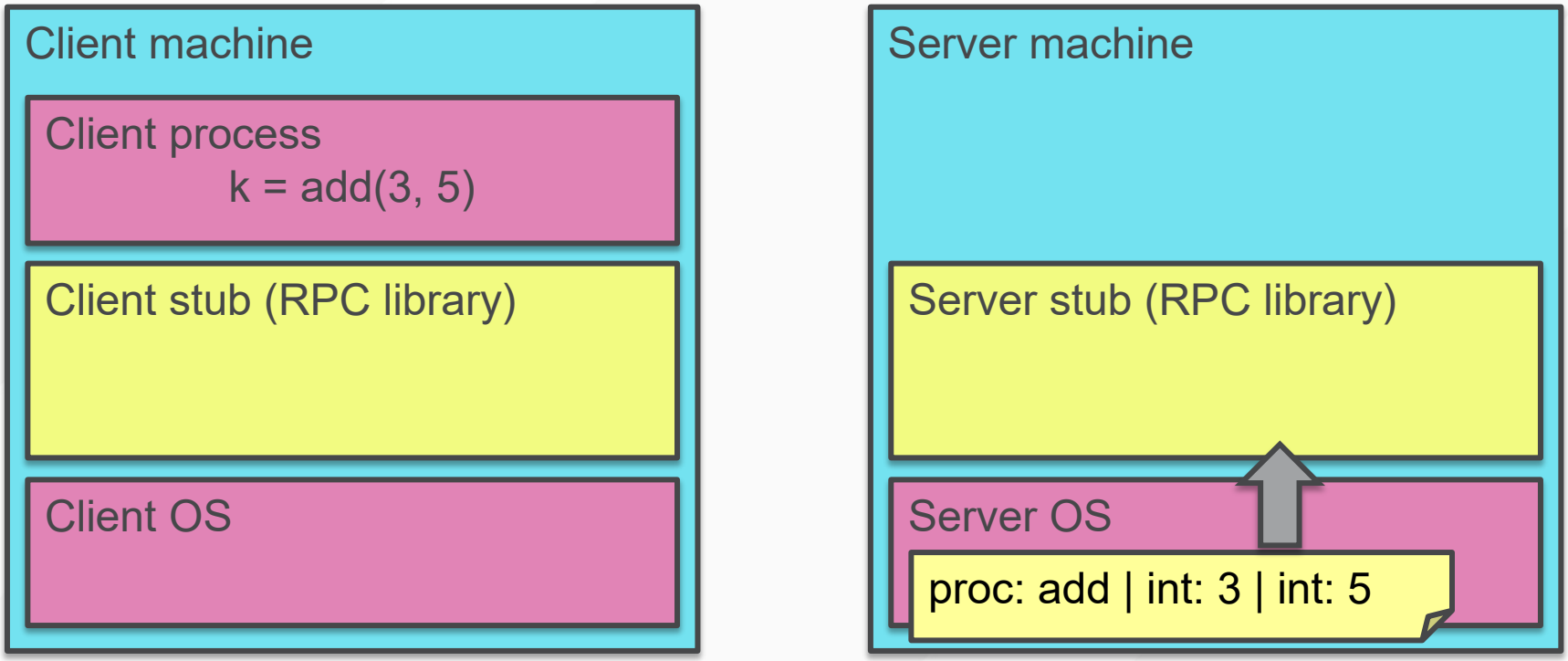
3. **OS sends a network message to the server**



A DAY IN THE LIFE OF AN RPC

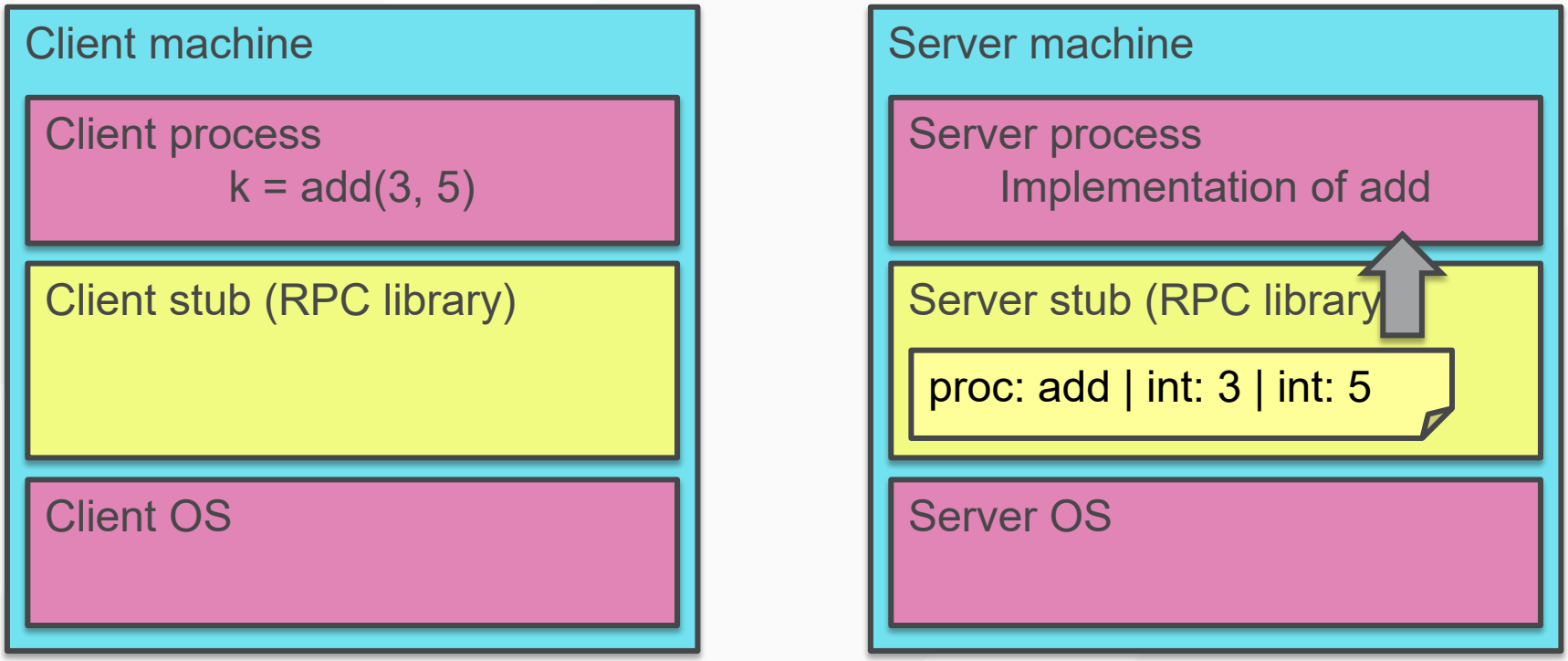
3. OS sends a network message to the server

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



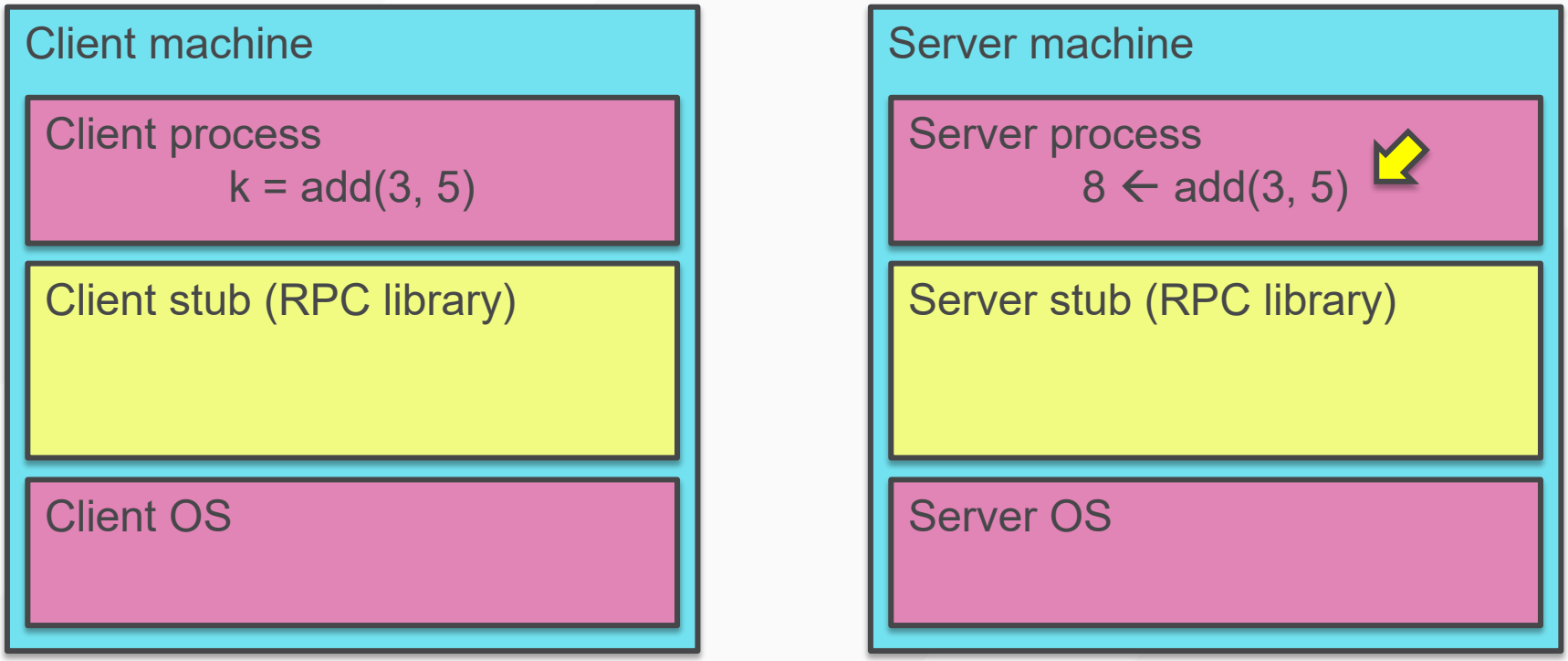
A DAY IN THE LIFE OF AN RPC

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



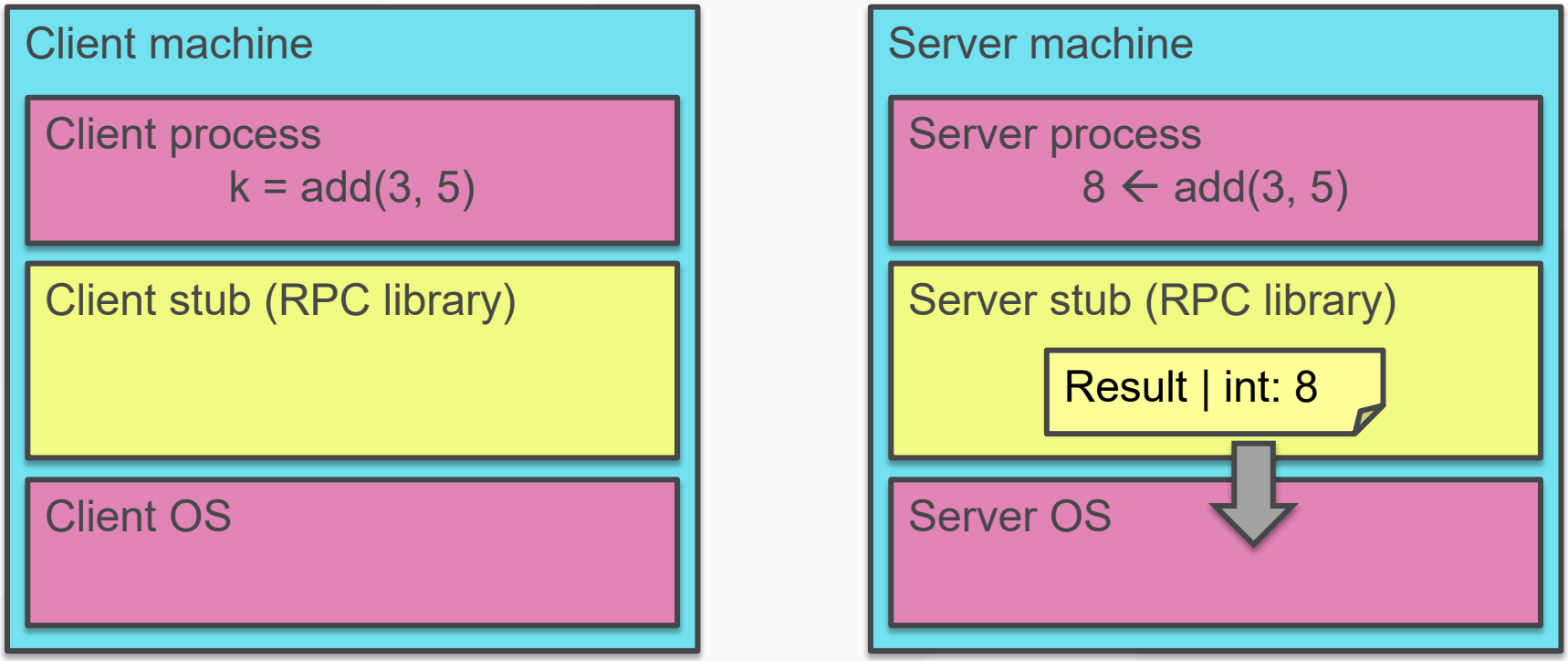
A DAY IN THE LIFE OF AN RPC

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



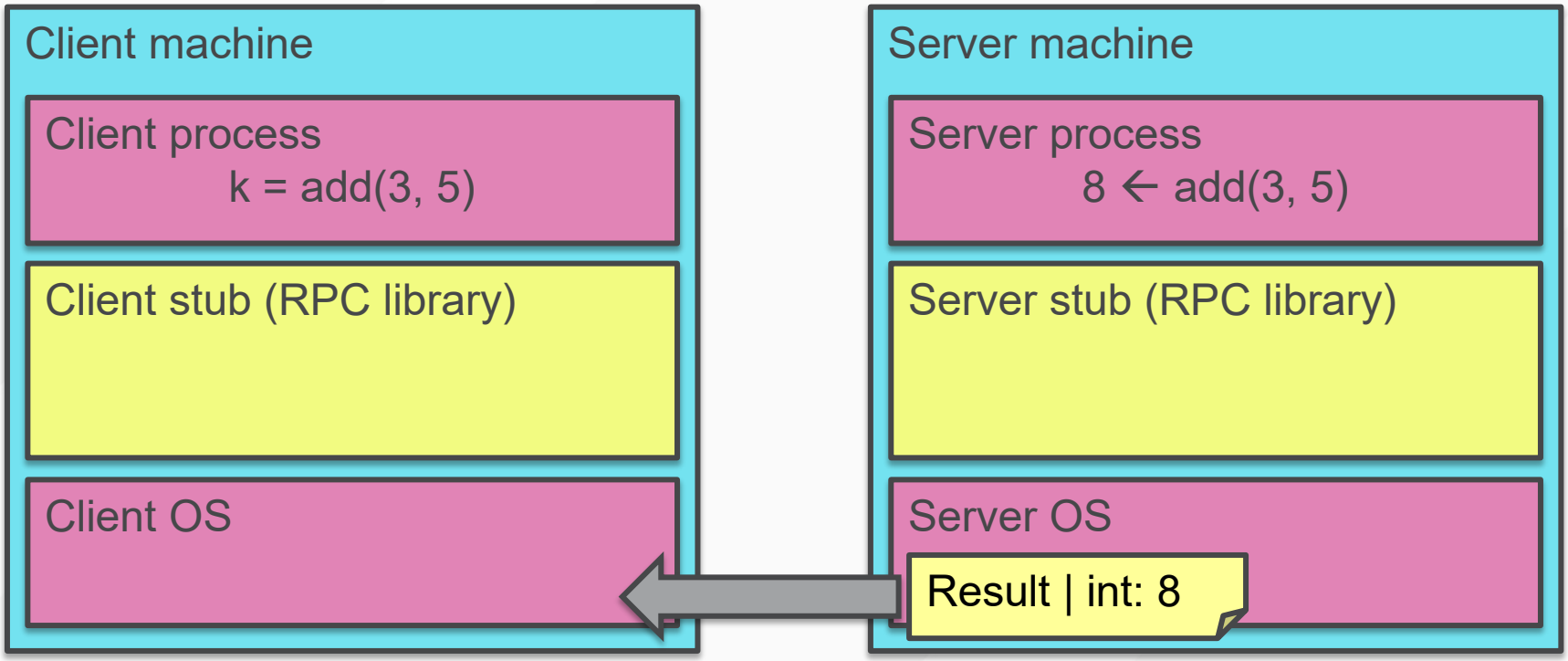
A DAY IN THE LIFE OF AN RPC

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



A DAY IN THE LIFE OF AN RPC

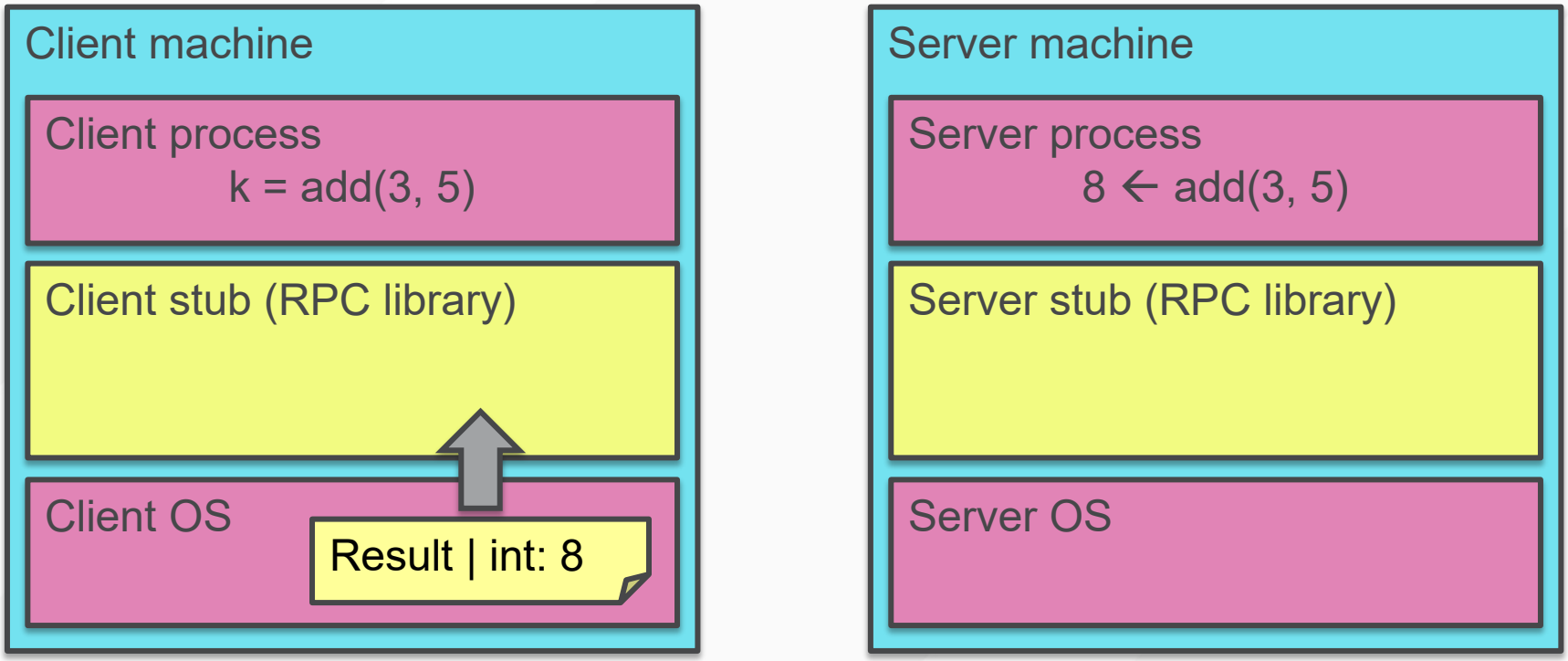
7. Server stub marshals the return value, sends msg
8. **Server OS sends the reply back across the network**



A DAY IN THE LIFE OF AN RPC

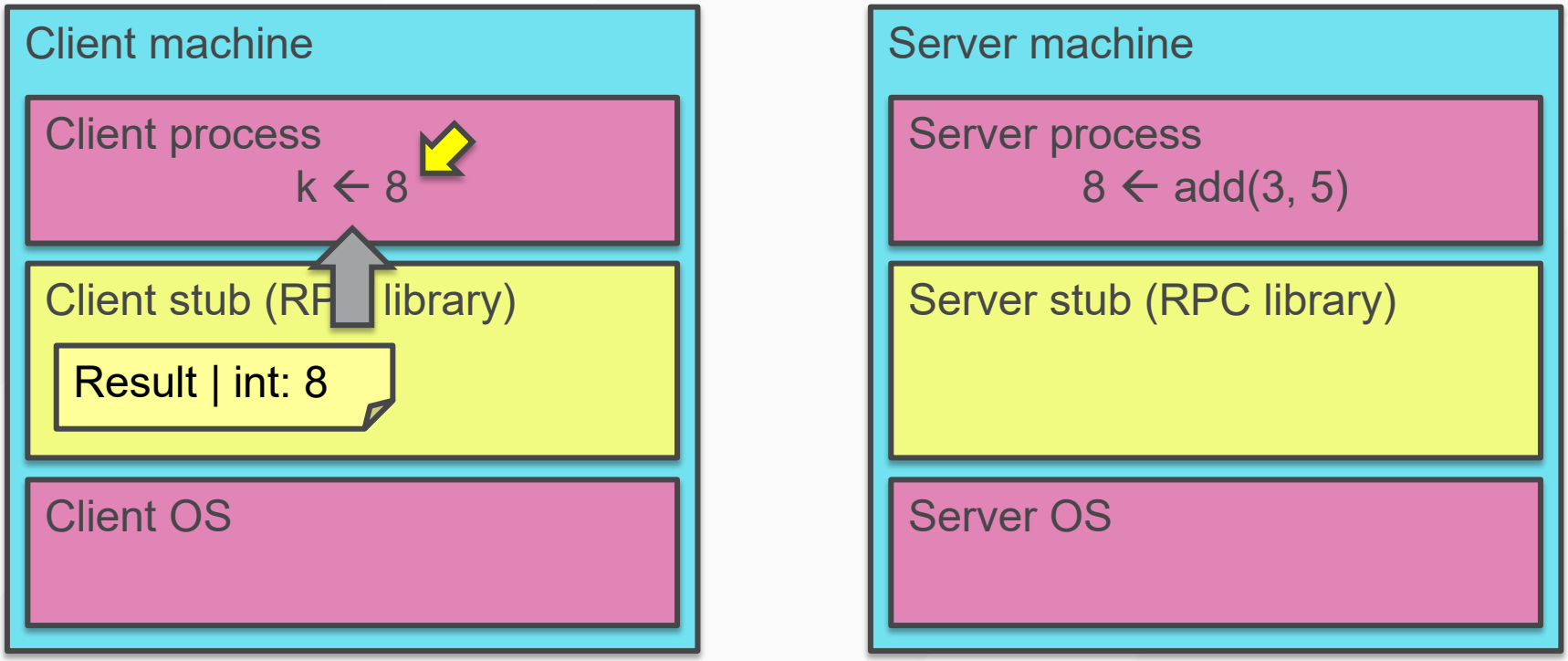
8. Server OS sends the reply back across the network

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



A DAY IN THE LIFE OF AN RPC

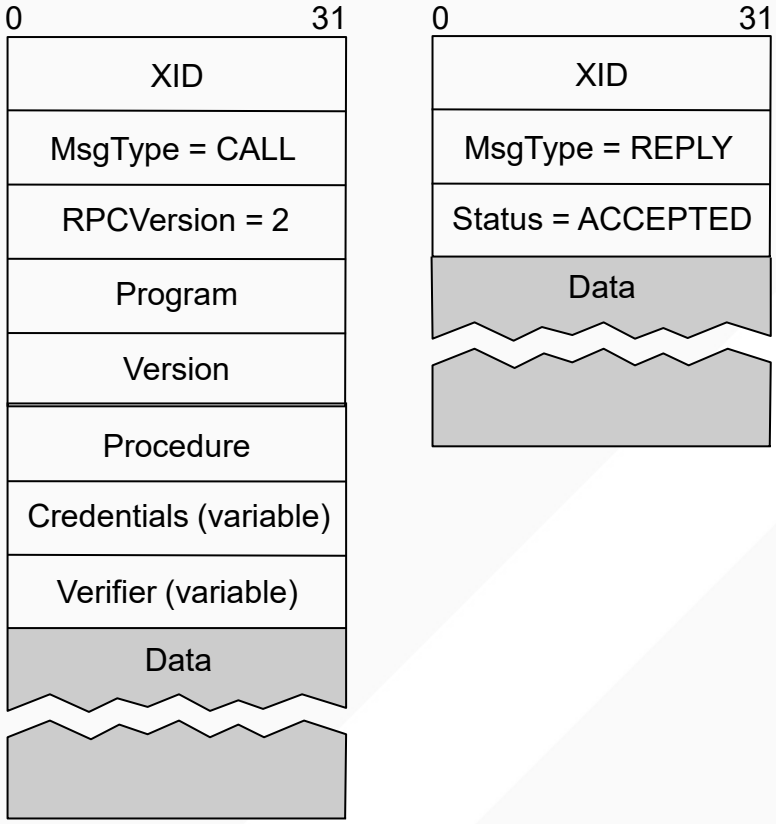
9. Client OS receives the reply and passes up to stub
10. **Client stub unmarshals return value, returns to client**



Outline

1. RPC fundamentals
2. RPC Implementations
3. Handling failures in RPCs

SUN RPC (RUNS OVER UDP)



XML-RPC

- XML is a standard for describing structured documents
  - Uses tags to define structure: <**tag**> … </**tag**> demarcates an element
    - Tags have no predefined semantics …
    - … except when document refers to a specific namespace
  - Elements can have attributes, which are encoded as name-value pairs
    - A well-formed XML document corresponds to an element tree
- **<?xml version="1.0"?>**
  <methodCall>
  <methodName>SumAndDifference</methodName>
  <params>
  <param><value><i4>40</i4></value></param>
  <param><value><i4>10</i4></value></param>
  </params>
  </methodCall>

XML-RPC WIRE FORMAT

- Scalar values
  - Represented by a **<value><type> … </type></value>** block
- Integer
  - <i4>12</i4>
- Boolean
  - <boolean>0</boolean>
- String
  - <string>Hello world</string>
- Double
  - <double>11.4368</double>
- Also Base64 (binary), DateTime, etc.

WIRE FORMAT (STRUCT)

- Structures
  - Represented as a set of <**member**>s
  - Each member contains a <**name**> and a <**value**>
- **<struct>**
  <member>
  <name>lowerBound</name>
  <value><i4>18</i4></value>
  </member>
  <member>
  <name>upperBound</name>
  <value><i4>139</i4></value>
  </member>
  </struct>

WIRE FORMAT (ARRAYS)

- Arrays
  - A single <**data**> element, which
  - contains any number of <**value**> elements
- **<array>**
  <data>
  <value><i4>12</i4></value>
  <value><string>Egypt</string></value>
  <value><boolean>0</boolean></value>
  <value><i4>-31</i4></value>
  </data>
  </array>

XML-RPC REQUEST

- HTTP ***POST*** message
  - URI interpreted in an implementation-specific fashion
  - Method name passed to the server program

```
POST /RPC2 HTTP/1.1
Content-Type: text/xml
User-Agent: XML-RPC.NET
Content-Length: 278
Expect: 100-continue
Connection: Keep-Alive
Host: localhost:8080


<?xml version="1.0"?>
<methodCall>
<methodName>SumAndDifference</methodName>
<params>
<param><value><i4>40</i4></value></param>
<param><value><i4>10</i4></value></param>
</params>
</methodCall>
```

XML-RPC RESPONSE

- HTTP Response
  - Lower-level error returned as an HTTP error code
  - Application-level errors returned as a <**fault**> element (next slide)

```
HTTP/1.1 200 OK
Date: Mon, 22 Sep 2003 21:52:34 GMT
Server: Microsoft-IIS/6.0
Content-Type: text/xml
Content-Length: 467


<?xml version="1.0"?>
<methodResponse>
<params><param>
<value><struct>
<member><name>sum</name><value><i4>50</i4></value></member>
<member><name>diff</name><value><i4>30</i4></value></member>
</struct></value>
</param></params>
</methodResponse>
```

XML-RPC FAULT HANDLING

- Another kind of a MethodResponse

```
<?xml version="1.0"?>
<methodResponse>
<fault>
<value><struct>
<member>
<name>faultCode</name>
<value><i4>500</i4></value>
</member>
<member>
<name>faultString</name>
<value><string>Arg `a' out of
range</string></value>
</member>
</struct></value>
</fault>
</methodResponse>
```

Outline

1. RPC fundamentals
2. RPC Implementations
3. Handling failures in RPCs

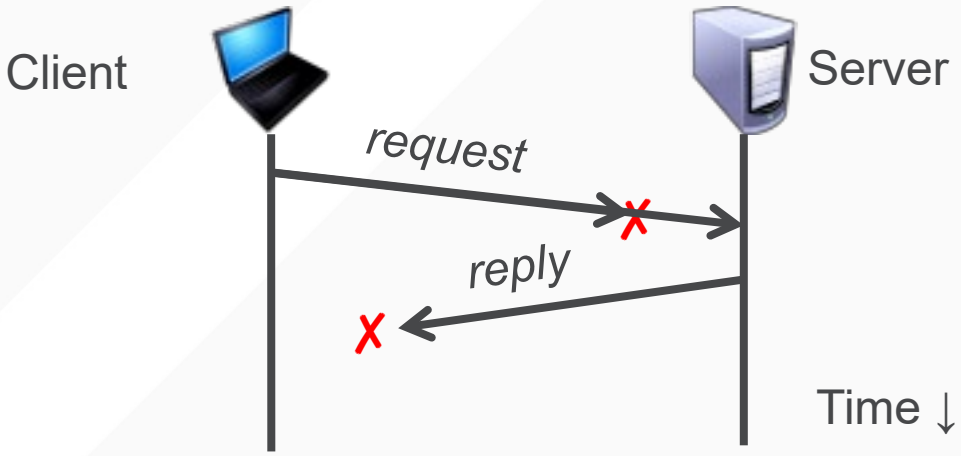
HANDLING FAILURES IN REMOTE PROCEDURE CALLS

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 these may look the same to the client…

FAILURES, FROM CLIENT'S PERSPECTIVE



The cause of the failure is hidden from the client!

AT-LEAST-ONCE SCHEME

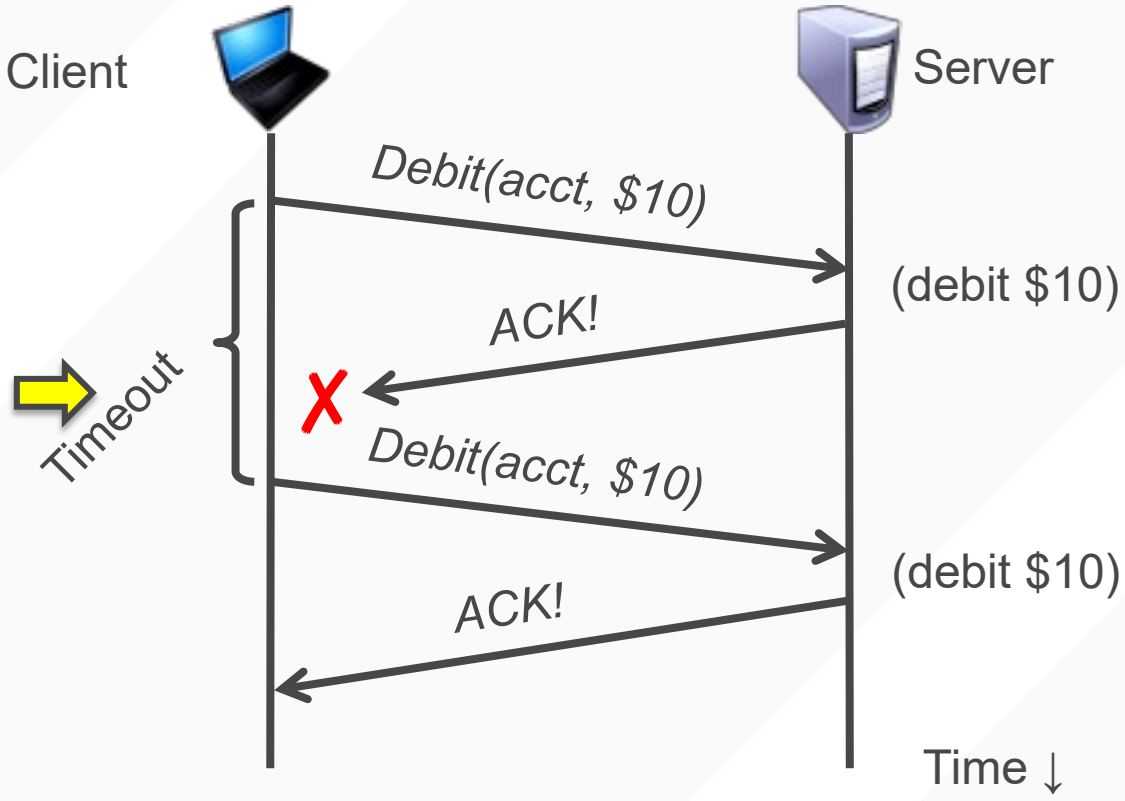
- **Simplest** scheme for handling failures

1. Client stub **waits for a response**, for a while
   - Response takes the form of 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

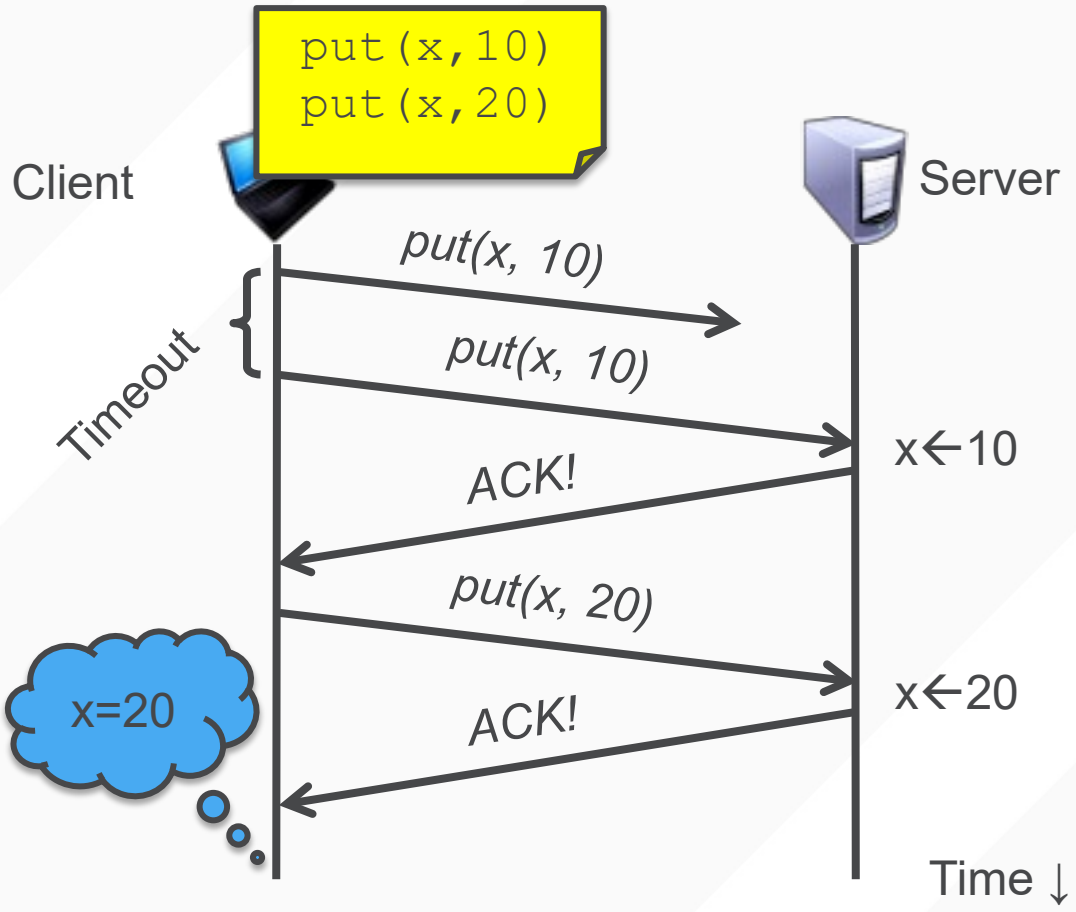
AT-LEAST-ONCE AND SIDE EFFECTS

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



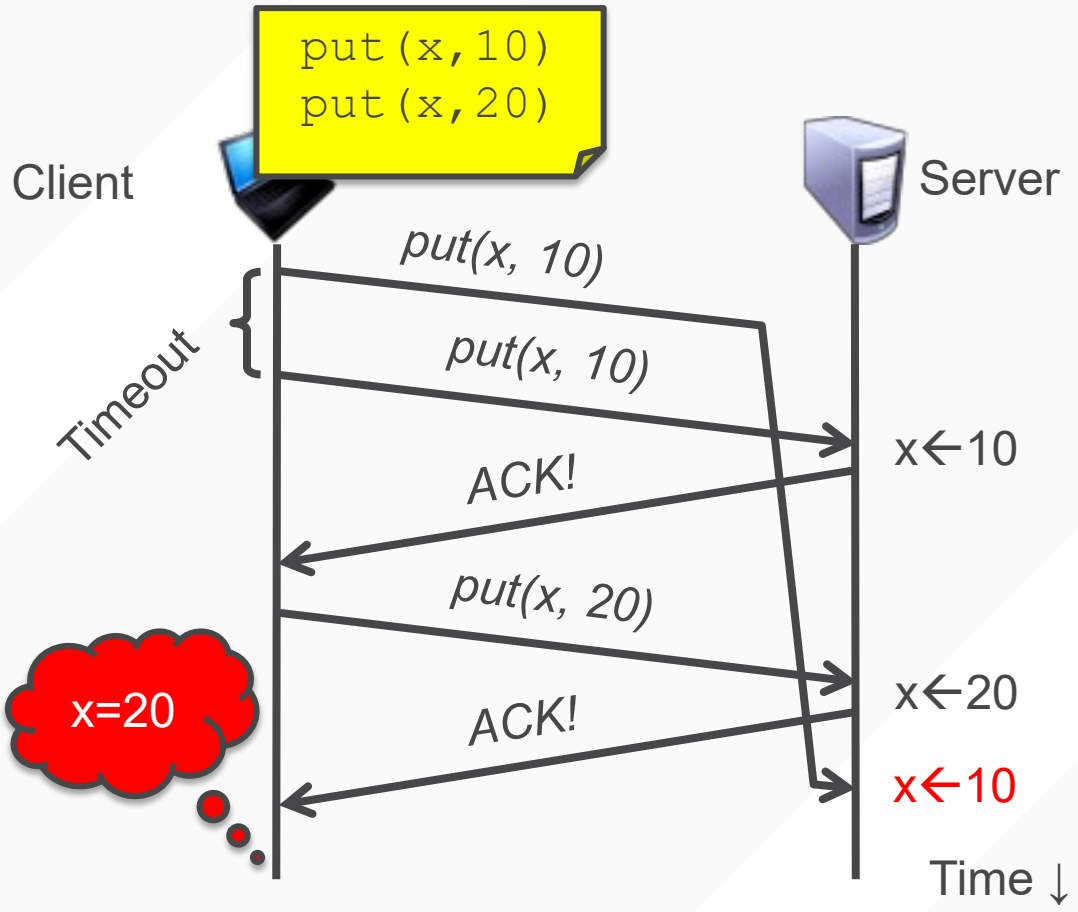
AT-LEAST-ONCE AND WRITES

- 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
  - You will implement this in a later project

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 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 to ensure that the xid is unique?***

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 the client crashes and restarts. *Can it reuse the same client ID?*
3. Big random number

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

RPC SEMANTICS

Delivery Guarantees			RPC Call Semantics
Retry Request	**Duplicate Filtering**	**Retransmit Response**	
No	NA	NA	*Maybe*
Yes	No	Re-execute Procedure	*At-least once*
Yes	Yes	Retransmit reply	*At-most once*

UC San Diego