Remote Procedure Calls (RPC)

Ashvin Goel

Electrical and Computer Engineering
University of Toronto

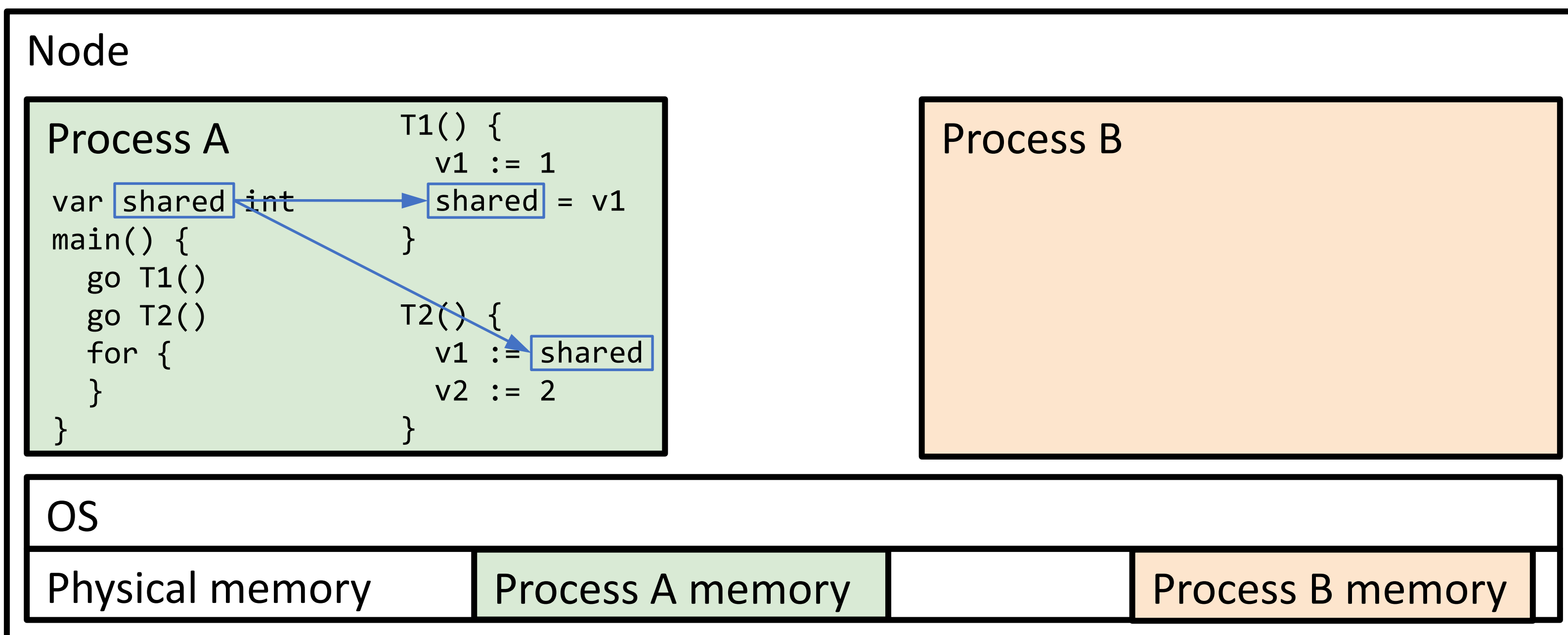
Distributed Systems
ECE419

Overview

- Shared memory versus message passing communication
- RPC
- RPC failures

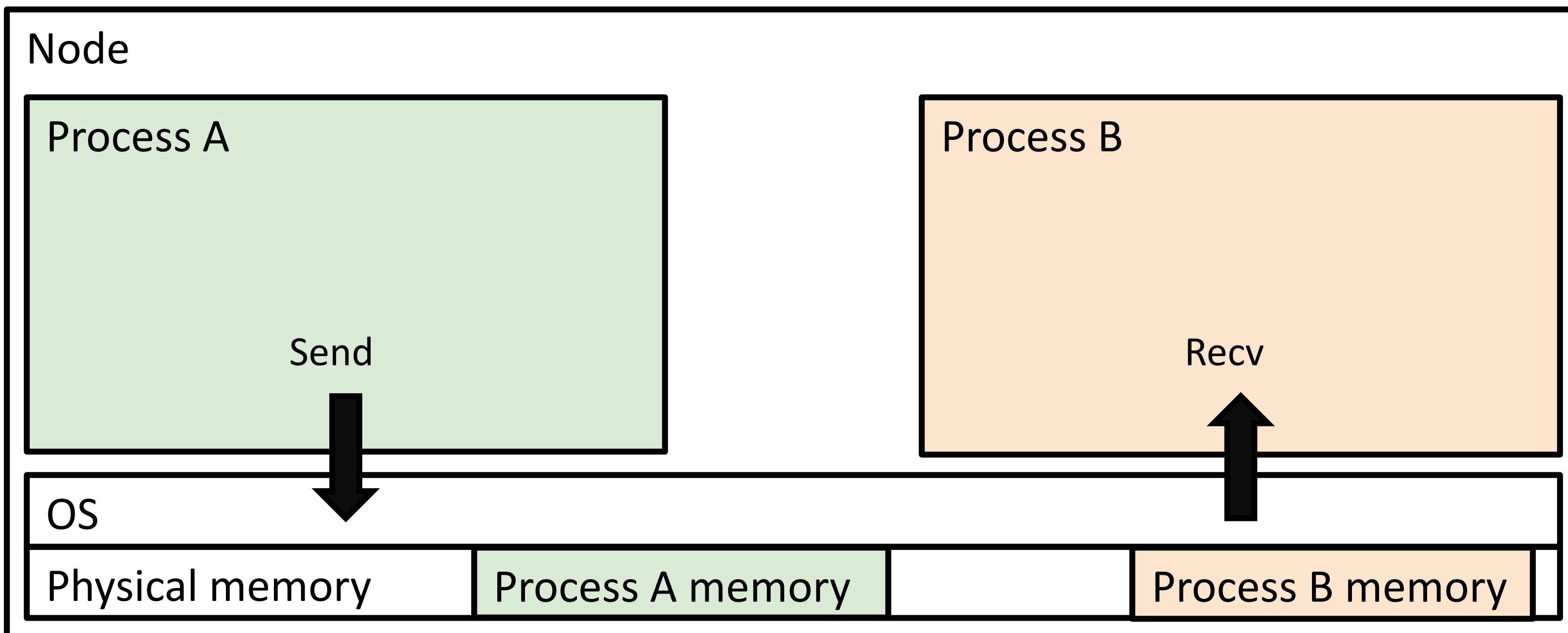
Shared memory communication

- Threads within a process share memory
- We have seen that they communicate with each other by reading and writing to shared memory
- This is called shared memory communication



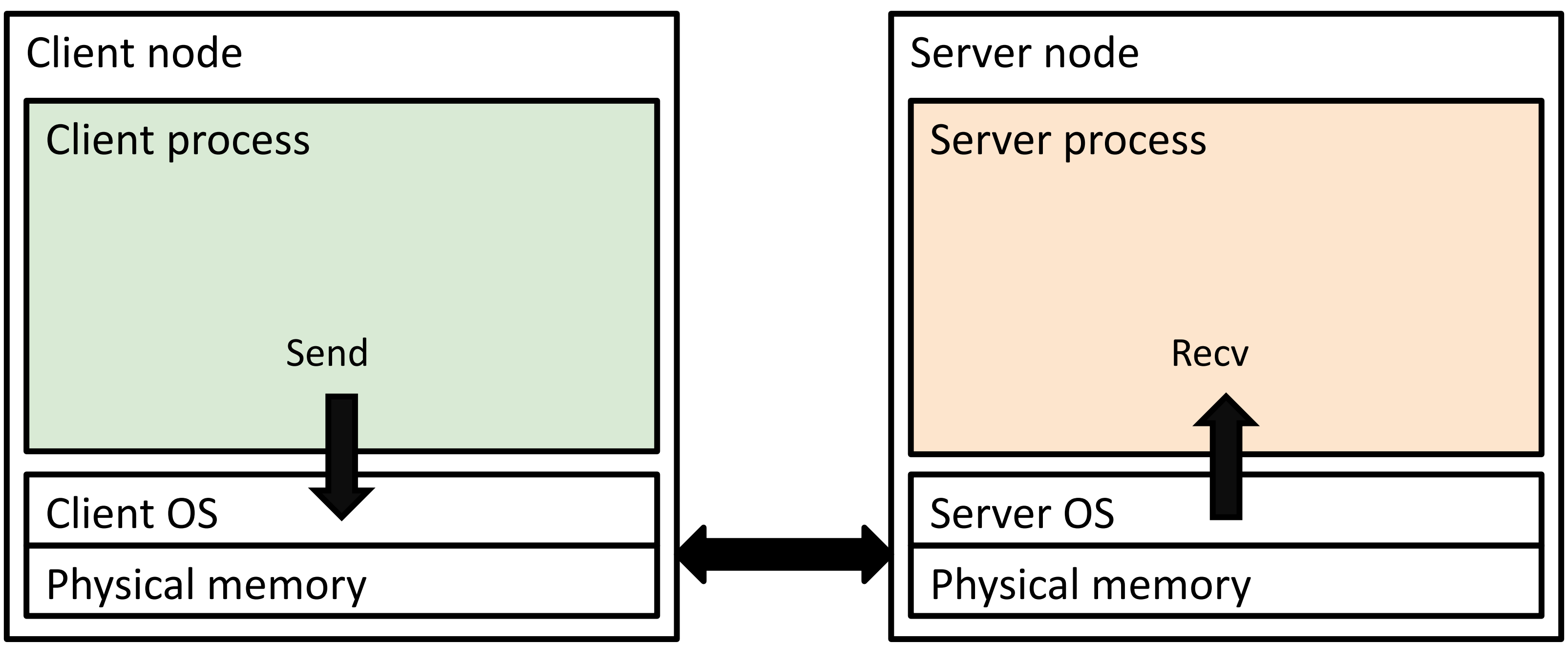
Inter-process communication

- Processes do not share memory
  - How can processes communication with each other?
- Processes send and receive messages
  - OS provides IPC interface: send(), recv()



Socket-based communication

- Nodes do not share memory
  - How can processes communicate across nodes?
- Processes send and receive messages across nodes
  - OS provides socket interface: send(), recv() ← Similar to IPC



Challenges with Socket API

- Programming interface is low level
  - Sender: needs to convert data structures to bytes,
    package the bytes to packet headers and body, send packets
  - Receiver: needs to wait to receive packets,
    parse packet headers, convert packet body into data structures
- What happens with multiple concurrent requests?
- How to match requests and responses?
- What happens on packet drops, node failures?
- Need higher-level API for communicating across nodes

RPC

Why RPC?

- Programmers are used to procedure call interface
  - A() calls B()
  - B() does its job, returns value to A()
  - A() continues

```
A() {
  val = B(args);
  …
}
```

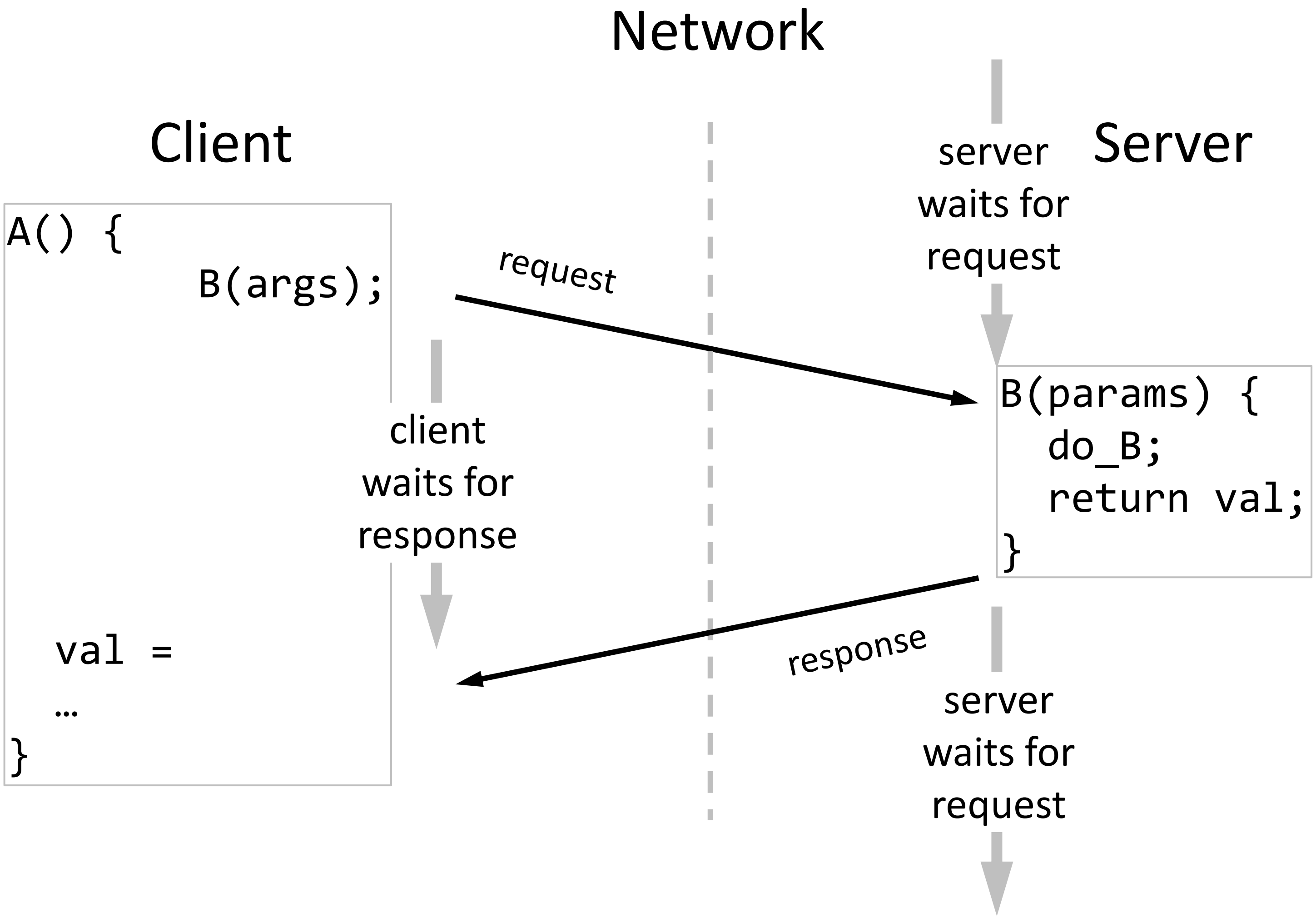
```
B(params) {
  do_B;
  return val;
}
```

- Same interface is used by apps to call kernel code
  - A() is a user function, B() is a system call
- Remote procedure calls: use the procedure call interface for communicating across nodes
  - A() runs on a node (e.g., client)
  - B() runs on another node (e.g., server)

Benefits of RPC

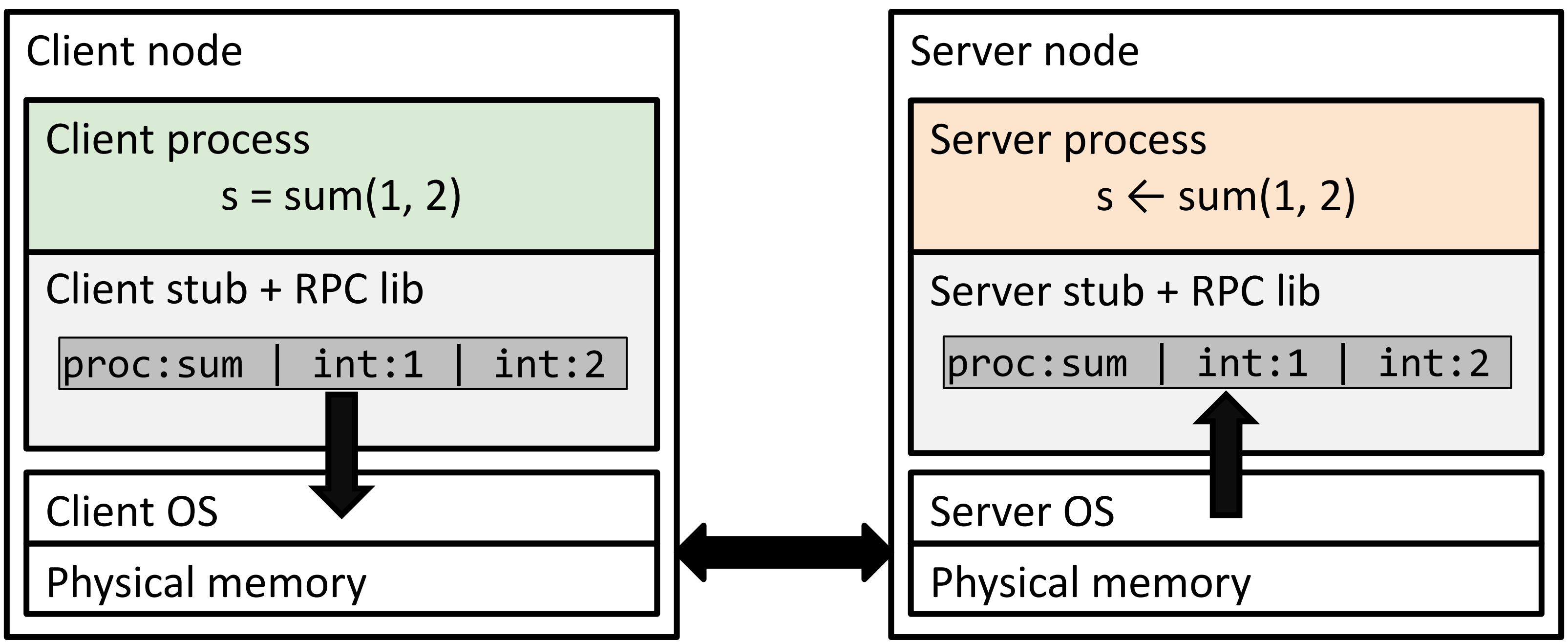
- Simplifies client-server communication
  - Hides details of network protocols
    - Converts data structures (e.g., strings, arrays, maps, etc.) to/from packet format
  - Provides portability / interoperability
    - Sender and receiver side can
      - Have different endianness
      - Use data types of different sizes, different alignment
      - Use different languages
- Today, RPCs are used extensively in distributed systems
  - Google gRPC, Facebook/Apache Thrift
  - REST with JSON, Ajax in browsers, build on RPCs

RPC messages



RPC request processing

- Client side: client stub marshalls (converts) call and arguments into network format, sends packet
- Server side: receives packet, server stub unmarshalls packet, calls sum() handler function



RPC example

- Let’s look at an RPC example in Go
- Use a trivial key-value storage server that supports
  - put(key, value)
  - value = get(key)

RPC details

- Server location: how does client know server’s location?
  - Go's RPC requires server name/port as an argument to Dial
  - Another option is to use a name service, e.g., DNS
- Marshalling: How to format complex data types?
  - Go's RPC library
    - Can pass strings, arrays, objects, maps
    - Passes pointers by copying the pointed-to data
    - Cannot pass channels or functions
    - Only marshals exported fields
- Multi-threading
  - Client can use multiple threads to send concurrent requests, RPC library matches requests with responses

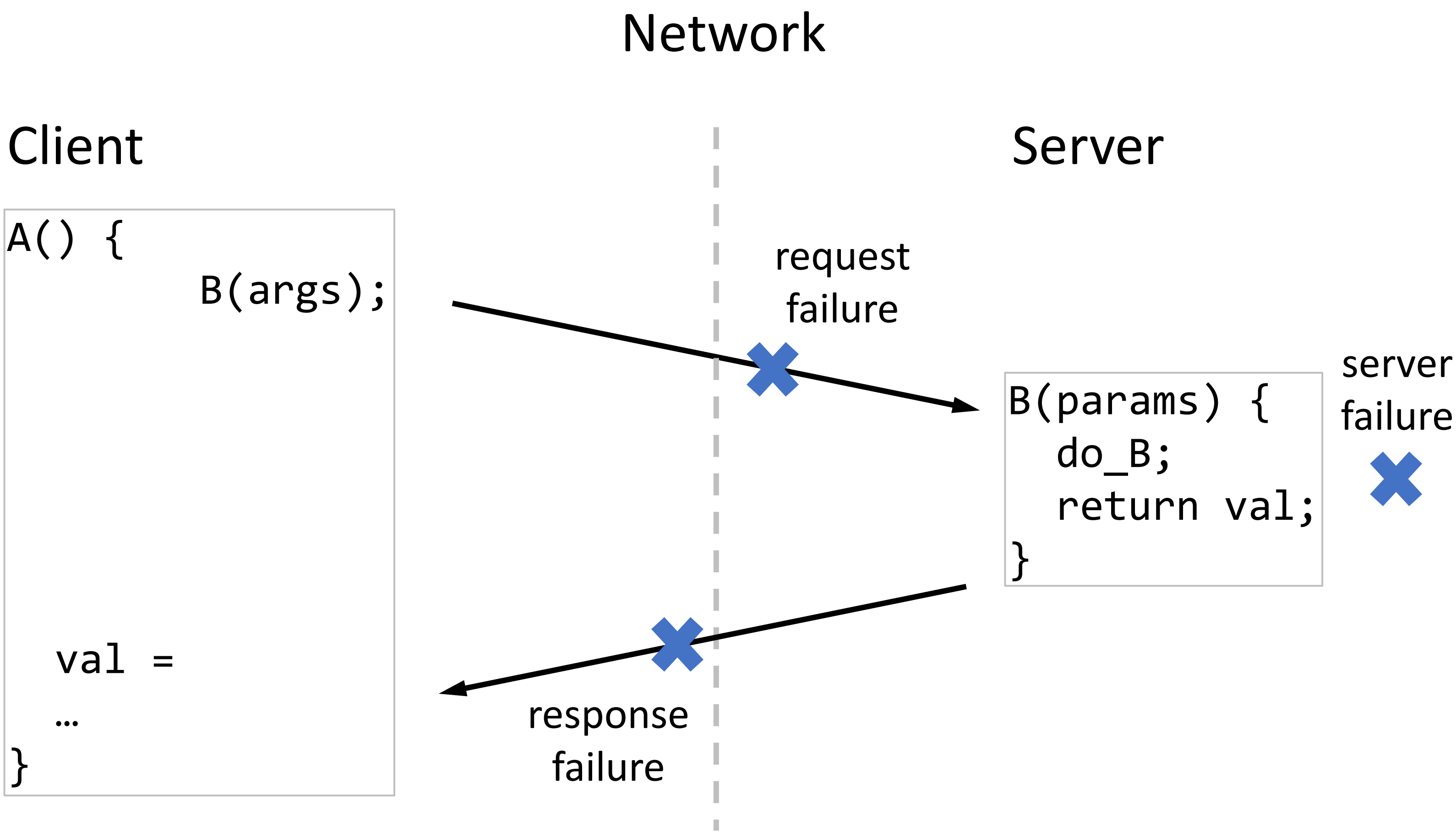
RPC Failures

Failures

- Until now, RPC appears to provide the same semantics as local procedure calls
  - If a client issues an RPC call, the server executes it once
- However, failures complicate RPC semantics
- Lots of failures possible in distributed systems
  - Packets may be dropped, reordered, duplicated
  - Network or server is slow
  - Client or server crashes (and reboots)
- Consider an RPC client
  - If a response doesn't arrive, the client does not know whether the server executed the request or not!

How is this different from local procedure calls?

Failures during RPC



Failures during RPC

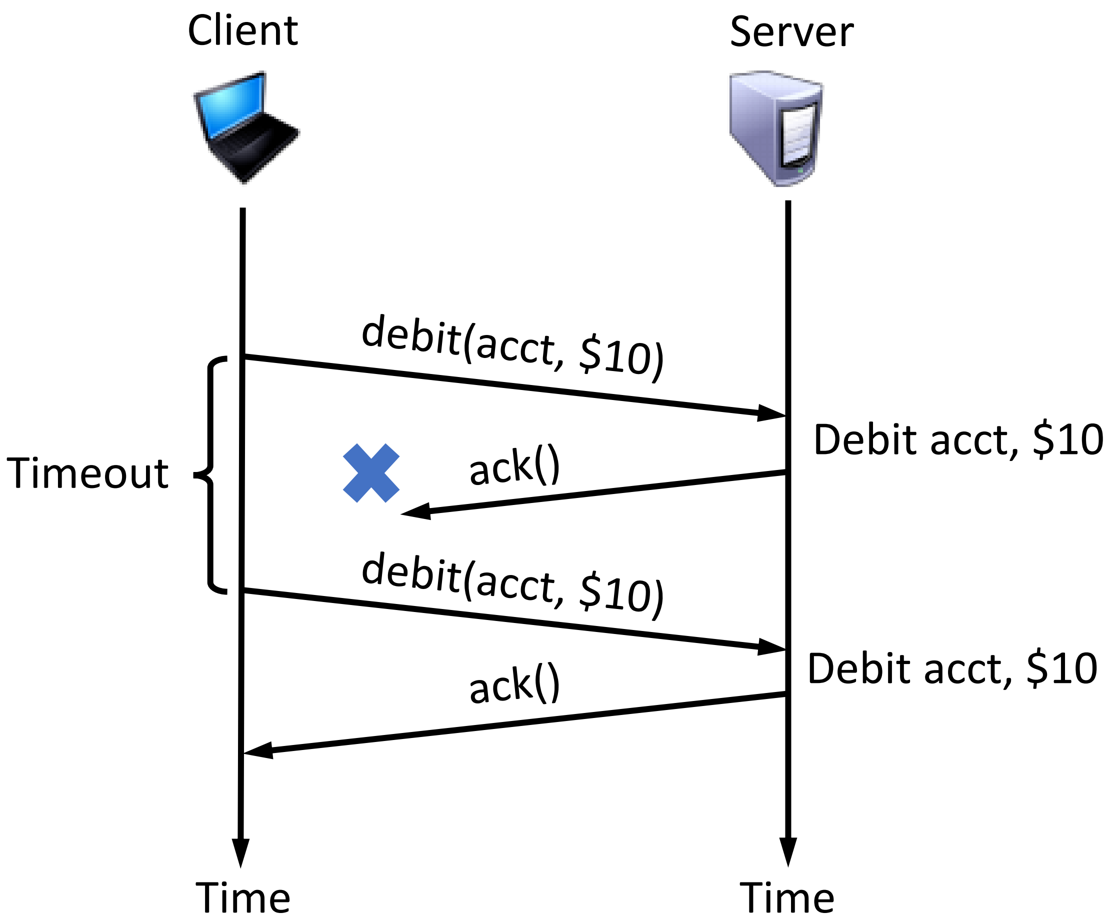
- Request failure: server didn't execute the request
- Response failure: server executed the request
- Server failure: server may or may not have executed the request (or partly executed the request ☹)

Best-effort RPC

- Wait for a response to a request for some time
- If no response arrives, re-send the request
- Do this a few times
- Then give up and return an error

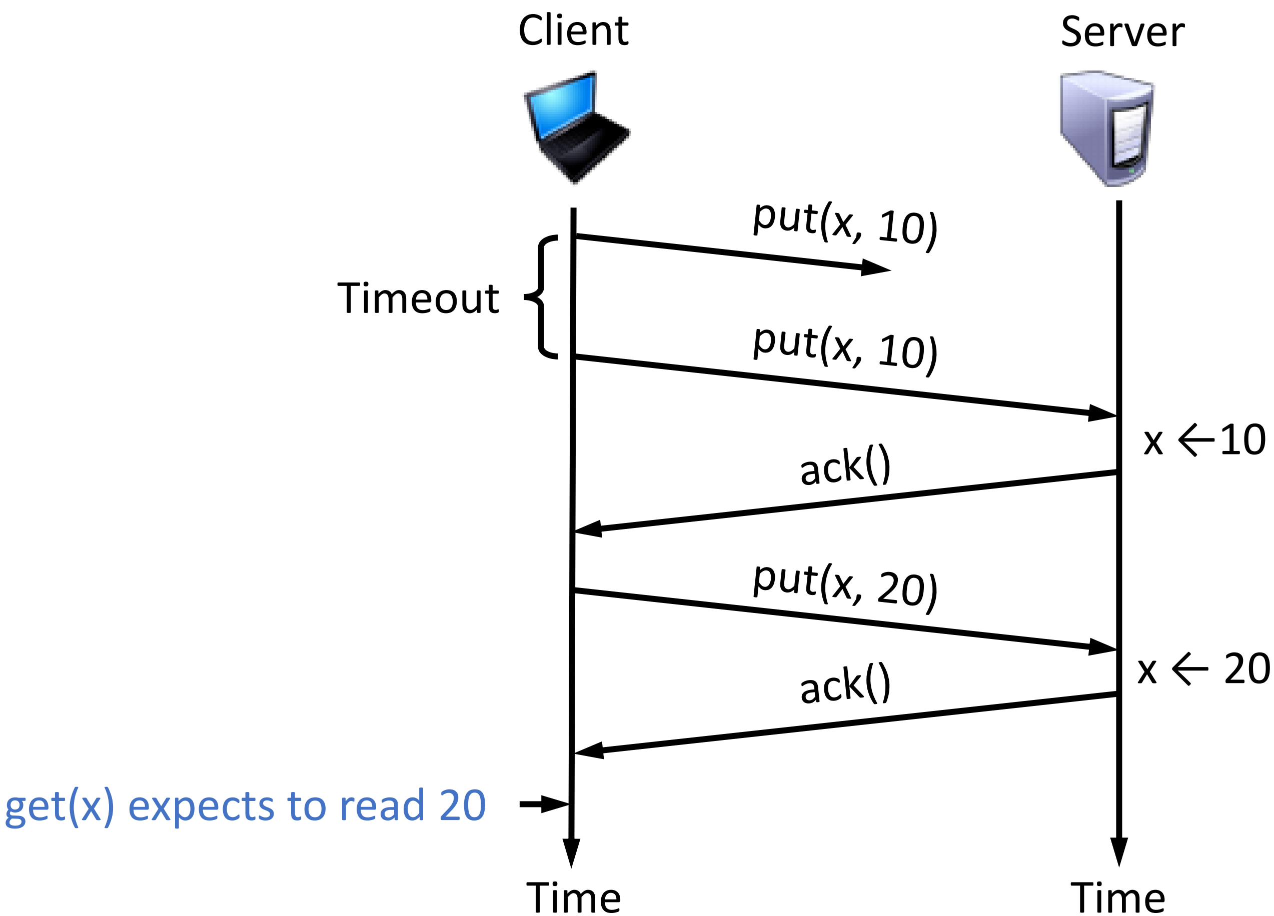
Side effects with best-effort RPC

- Client sends a "debit $10 from bank account" RPC
  - Re-send causes $20 debit!



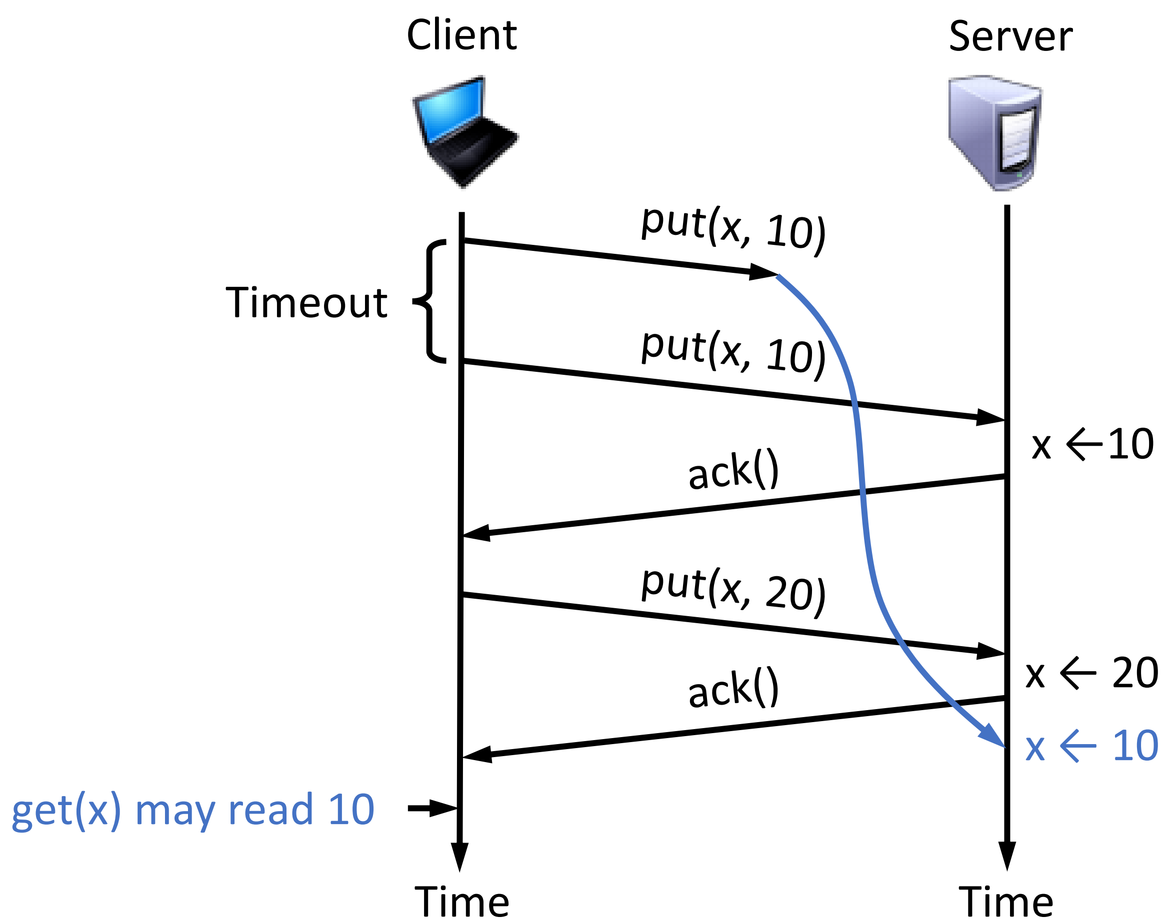
Inconsistency with best-effort RPC

- Consider our key-value server
    - get(x) expects to read last value of put(x, value)



Inconsistency with best-effort RPC

- get(x) expects to read last value of put(x, value)
  - get(x) may read 10 due to delayed original request!



When will best-effort RPC work?

- For read-only operations, with no side effects
- If application handles duplicate, reordered requests

Better RPC semantics

- Previous best-effort RPC is also called at-least-once RPC
  - When client receives a response:
    - Request has executed once or more times
  - When client doesn't receive a response:
    - Request may not have executed, or executed once or more times
- A better option is at-most-once RPC
  - When client receives a response:
    - Request has executed exactly once
  - When client doesn't receive a response:
    - Request may not have executed, or executed once

We have seen that executing a request multiple times caused problems

At-most-once RPC

- Client is unchanged: re-sends request when no answer
- Server RPC code
  - Detects (duplicate) requests that it has already executed
  - Returns previous reply instead of re-running handler
- How to detect a duplicate request?
  - Can a server look for the same function invocation,
    with same arguments?
  - No! A program may legitimately submit the same function
    with same augments, twice in a row

At-most-once RPC

- Solution: client includes unique ID (XID) with each request, uses same XID when resending request
- Server detects duplicate requests based on XID

```
// server code ensures that rpc_handler() executes once

if ret_value, ok := response[xid]; ok {
    // rpc_handler() already executed
} else {
    ret_value = rpc_handler()
    // save ret_value
    response[xid] = ret_value
}
// send ret_value to client
```

Seems simple, but it raises several issues

Generating unique IDs

- How to generate unique ids for at-most-once RPC?
  - Use a large random number
    - Only probabilistic guarantee
  - Use a client ID (e.g., IP address) and a sequence number
    - What happens if client crashes and restarts?

Getting rid of server state

- response[xid] array grows on the server
- After client gets response for xid,
  it could inform server to delete xid entry in the array
- Better method
  - Assume xid = (client, seq)
  - Client waits to get response for all requests <= seq
  - Client informs server to delete all entries for this client whose sequence number <= seq
  - Similar to TCP sequence numbers, acks
  - Server maintains state roughly proportional to # clients
- Server must still handle non-responsive clients, how?

Concurrent requests

- How to handle a duplicate request while the original is still executing?
  - Server doesn’t know reply yet, so can’t send “previous” reply
- Solution 1:
  - Queue the requests, execute them serially
- Solution 2:
  - Add a pending flag per executing RPC
  - Wait for RPC to be done, then respond to duplicate request

Server crashes, then restarts

- Until now, we have assumed that the requests or responses may fail but the server doesn’t fail
- What happens if the server crashes and restarts?
- Suppose response[xid] array is kept in memory
  - After server restarts, they are lost
  - Now, server may run duplicate requests more than once
- Let’s look at two options for solving this problem
  1. Keep array in memory, track number of server restarts
  2. Keep array on disk

1. Track number of server restarts

- Server uses an epoch number, stored on disk, incremented after each restart
  - Server adds its epoch numbers to all responses
- Client sends epoch number with each request
  - Allows server to distinguish requests that first arrived before crash or after restart
- Server serves requests with current epoch number, sends error otherwise
  - Why send error?
  - Why does this method ensure at-most-once RPC semantics?
  - Any issues with this approach?

2. Keep response[] array on disk

- Server stores response[xid] on disk before returning reply to the client
  - When is this data stored, before or after handler executes?
  - What if there is a server crash in between?
  - What if the handler executes partially, writes some data to disk, and then the server crashes?
- Need to ensure that all this data is written to disk correctly (atomically) and can be recovered on failure

Exactly-once RPC

- Client side:
  - When client doesn't receive response, it keeps retrying forever
    - Avoids the problem with at-most-once, where on a failed response, the request may or may not have executed
- Server side:
  - Perform duplicate detection (same as at-most-once)
  - Handle server crashes, or
  - Use a fault-tolerant service (server appears to never fail)

Exactly-once RPC

- Client side:
  - When client doesn't receive response, it keeps retrying forever
    - Avoids the problem with at-most-once, where on a failed response, the request may or may not have executed
- Server side:
  - Perform duplicate detection (same as at-most-once)
  - Handle server crashes, or
  - Use a fault-tolerant service (server appears to never fail)
- We will discuss these topics in detail later

There are only
two hard problems
in distributed systems

RPC semantics in Go

- Go RPC uses a simple form of at-most-once semantics
- Each request opens a TCP connection, writes a request
- Requests are never re-sent, so server doesn’t see duplicate requests
- Go RPC returns error when a response is not received
  - Could happen due to TCP timeout, network or server failure
  - In this case, a request may or may not have been processed

RPC performance

- A local procedure call takes a few nanoseconds
- RPC to a machine in the same data center can take about 100 microseconds (10^5x slower)
- RPC to a machine on other side of planet can take about 100 milliseconds (10^7x slower)
- Solutions:
  - Issue multiple requests in parallel
  - Batch requests and send them together
  - Cache results of requests

Conclusions

- Sockets are low level for programming distributed systems
- RPC provides a simple procedure call interface for a client to invoke server code
- RPC failures complicate RPC semantics
  - Requests need to be retried on failure,
    but retries may cause duplicate requests,
    which need to be ignored (which is non-trivial)