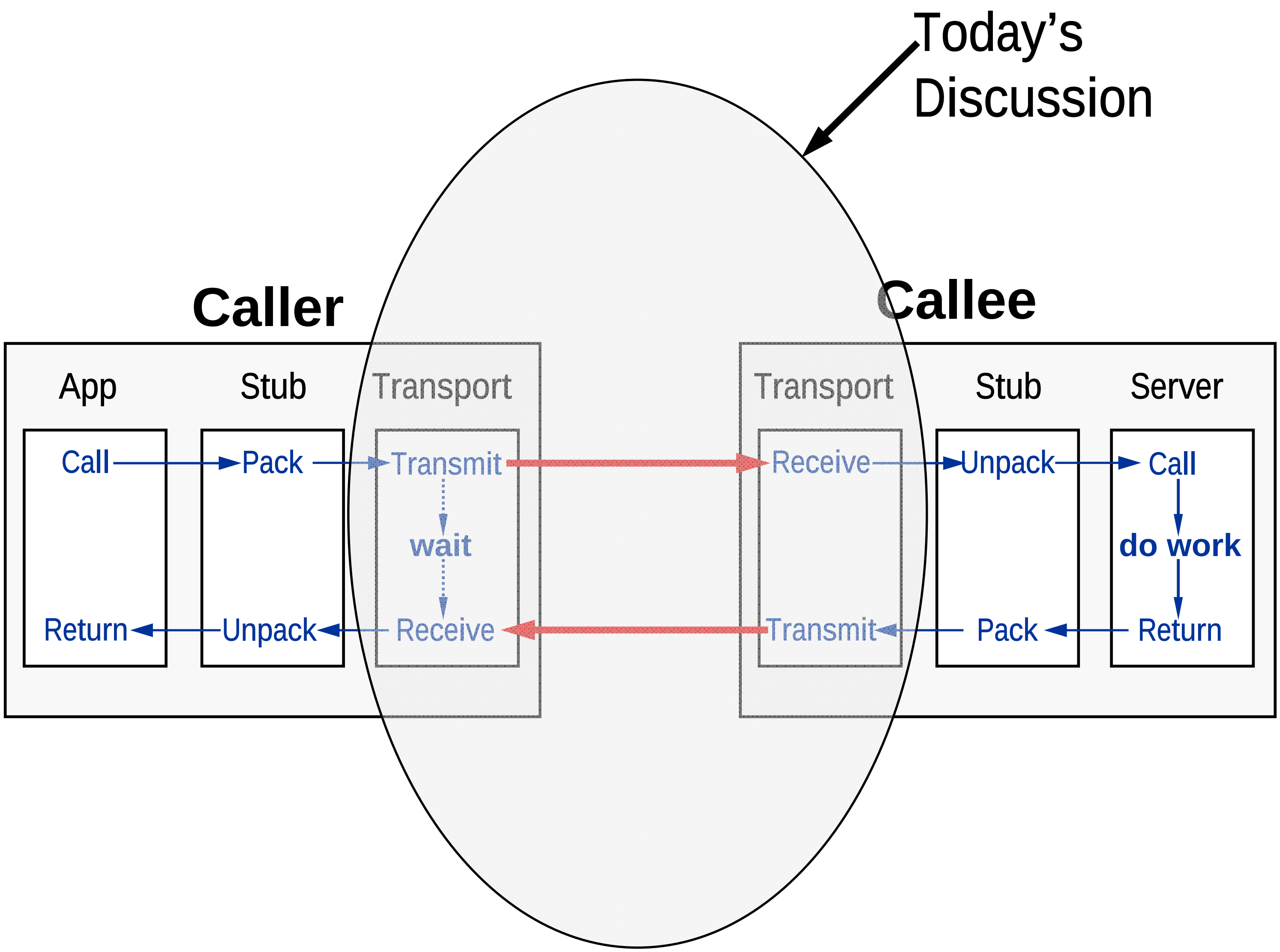

Today's Discussion
Caller
Callee
App
Stub
Transport
Transport
Stub
Server
Call
Pack
Transmit
wait
Receive
Unpack
Return
Receive
Transmit
Unpack
Pack
Call
do work
Return

RPC Transport

Local procedure semantics

- ***1 invocation by caller → 1 execution of callee***
- **ideal distributed environment: no failures**
- **trivial to emulate this aspect of local calls**

But real distributed systems are flaky

- **server hardware failures, server software failures, …**
- **lost packets, mutilated packets, hard network failures, …**
- **1 invocation by caller may or may not succeed**
- **even more complexity in store, as discussed below**

Timeouts in Distributed Systems

One approach

- **send request packet**
- **start timer**
- **if reply not in when timer goes off, declare failure**

Two problems with this approach

1. **lost packets for transient reasons common**
   giving up too soon is pessimistic (maybe server never received your request)
2. **perhaps server is still computing or perhaps it is overloaded**
   or perhaps it sent a reply, and this was lost

How do you pick a perfect timeout value?

- **in the worst case, no perfect value exists**
- **at best, using known statistics, one can pick a “reasonable” value**
  can be wrong, sometimes giving up too soon
- **no matter what value is picked, it could be “too soon”**
  reply could arrive just after you give up

Ideal vs Real Distributed Systems

Ideal

- Perfect in-order delivery of uncorrupted packets with predictable delays
- Perfect server hardware and bug-free server software
- Perfect operating conditions at server's data center

Real

- Packet loss, corruption, re-ordering, unpredictable delays
- Flaky server hardware and buggy server software
- Power failures, air-conditioning failures, hurricanes, tornados, tsunamis, floods, …

Failure independence **of clients and servers adds complexity**

Cope with transient flakiness by *retransmission*

- strategy assumes cause was transient condition
- lost request → perhaps this retry will get through

RPC can be layered over TCP or over UDP

- over TCP: retransmission handled by TCP implementation
  possibly reconnect if TCP timeout too small
- over UDP: retransmission implemented by RPC layer

But what if cause was sluggish server or lost reply?

- executing second request packet
  → one invocation causes multiple computations
- *violates local emulation semantics*

Hence server must perform *duplicate elimination*

- server must be able to identify retransmissions
- use per-connection *sequence numbers*
- sequence number increases monotonically
- RPC: only one call at a time
  → only most recent sequence number needed

What should server do when it sees a duplicate?

- **may mean reply lost**
- **may mean reply crossed retransmitted request**
- **the best server can do is to retransmit reply**

Replies must be preserved

- **only 1 reply saved per connection**
- ***cannot re-compute reply***
  would result in multiple computations per invocation

Exactly-once Semantics

(theoretical ideal)

How long to keep old replies and sequence numbers?

- **rigorous interpretation of “RPC” $\rightarrow$ forever!**
- **across server crashes too**
  - **they have to be saved in non-volatile memory**
  - **server response has to be after non-volatile write**
  - **disk (or flash) latency on every RPC**
- ***clean undo of partial computations*** **before crash**

Such an RPC would have *exactly-once* semantics

- **success return from RPC call $\rightarrow$ call executed exactly once**
- **call blocks indefinitely, no failure return**

Not appropriate for many real applications

- **too slow because of synchronous disk writes**
- ***indefinite blocking*** **unacceptable in many cases**
- ***application-level recovery*** **precluded**
- **requires *transactional semantics* for server actions**

At-most-once Semantics

(practically achievable)

How to avoid indefinite blocking?

- **place upper bound on call duration**
- **declare timeout if it takes longer**

Such an RPC has *at-most-once* semantics

- **refers to what can be inferred in the worst case**
- **successful RPC $\rightarrow$ call executed exactly once**
- **failed RPC $\rightarrow$ call executed once or not at all**
- **most commonly used semantics**

Many possible reasons for RPC timeout

- **request and retries never got to server**
- **server died while working on request**
- **network broke while server working on request**
- **server replied, but replies and retries lost**

Servers may be sluggish or unreachable

- **complicates setting of timeout value**
- **probes to check server health during long calls**
- **server responds with *busy* if still working**
- **essentially a *keepalive* mechanism**

Choice of Semantics

Achieving exactly-once semantics

- **not provided by any real RPC package**
- **requires application-level duplicate elimination**
- **built on top of at-most-once RPC**

Most RPC packages provide at-most-once RPC

All at-most-once packages promise

- **at-most-once execution in case of timeout**
- **exactly-once execution if client receives reply**

At-most-once semantics avoids

- **transactional storage**
- **non-volatile storage of replies and sequence #s**
- **indefinite storage of replies**

At-least-once **semantics even simpler to implement**

- **requires *operation idempotency***
  no duplicate elimination necessary
  no server state needed
- **example:** read( ) **request on locked object**

Orphaned Computations

Danger with at-most-once semantics

- client sends request
- server starts computing
- network partition occurs
- server continues, unaware its work is useless
- server may hold resources (e.g. locks)

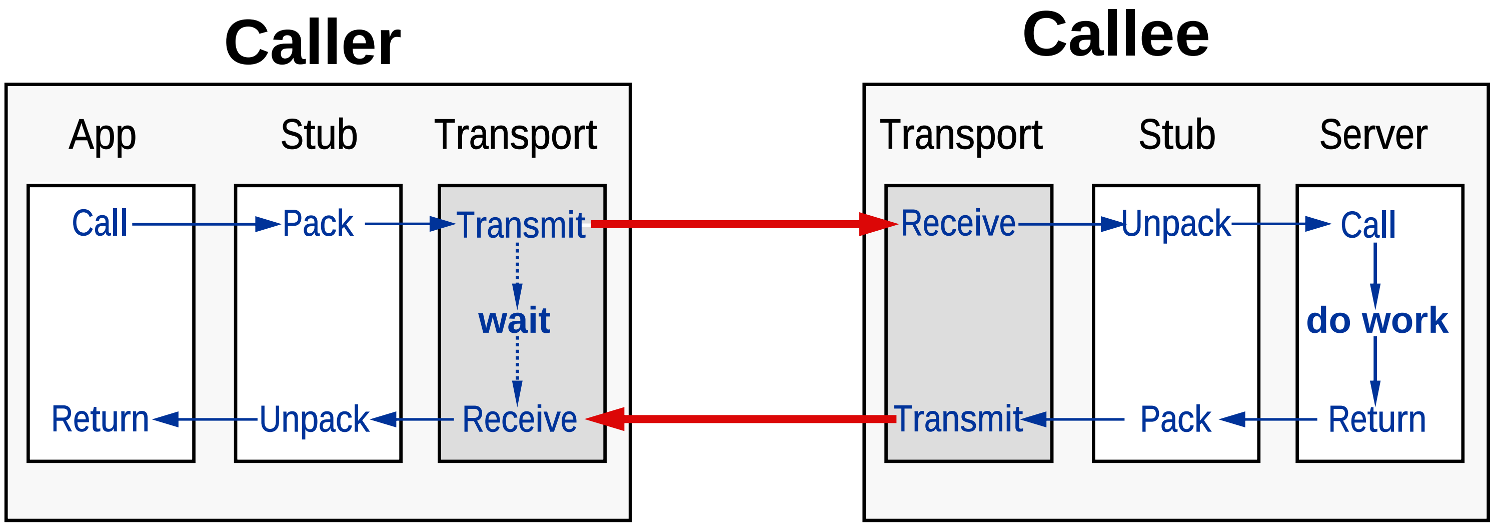
Orphan detection and *extermination* are difficult

- typically require application-specific recovery

“Failure” closely related to “timeout value”

- fundamental limitation in a distributed system
- due to absence of *out-of-band error detection*
- can't tell server death from network failure

Protocol Layering



Why can't we just use TCP retransmission mechanism?

- **btw, this is exactly what you are doing in Project 1**
- **yes, we can, but there is a price …**
- **and less benefit than you think …**

What Can TCP Do For RPC?

TCP timeout → we still have to reconnect

- **new TCP connection unaware of old**
  → server must keep higher level sequence #s
- **server must do duplicate elimination**
- **orphans still possible**
- ***exactly-once RPC no easier with TCP***

TCP can simplify at-most-once RPC

- **use two TCP connections**
  avoids need to implement retransmission and duplicate elimination
- **absence of TCP failure ⇒ exactly once RPC**
- **on TCP failure, declare RPC failure (at most once RPC)**

Use of TCP hurts best-case performance

- **two TCP connections unaware of each other**
- **TCP uses independent acks on each connection**

  client → server : request; server → client: ack

  server → client: reply; client → server: ack

- **4 packets for best case**
- **but RPC on UDP only uses 2 packets!**
- ***reply is an implicit ack***