Remote Procedure Call

Tom Anderson

Q&A During Lecture

- Verbal questions during lecture ok
  - Unmute to interrupt
  - Re-mute when done
- Chat questions also ok, if related to lecture topics
  - Send non-lecture Q&A to Ed
  - Please let the TA's or me answer lecture questions
- I will try to pause periodically for questions
- We will try to answer everyone's questions
  - If not live, then after class or on Ed
  - If we miss your question, please repost to Ed

Class Mechanics

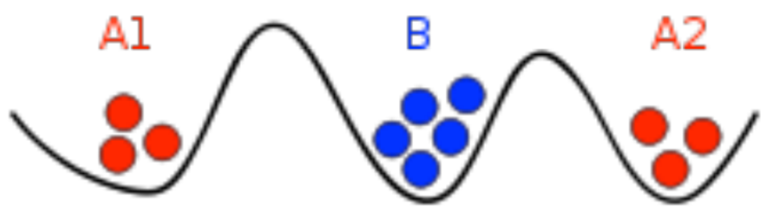
- Everyone will need (and should have):
  - Canvas access
    - Zoom lecture/section links (OH link under syllabus)
    - Recorded lectures/sections, with chats
    - Blog assignments (soon, Canvas Discussions)
  - Gitlab repo (uw netid)
  - Ed access
  - Gradescope (soon)

WiFi Carrier Sense

- Chat flood: example of synchronized behavior in a distributed system
- Another example: carrier sense
  - Multiple WiFi senders at the same time can interfere with each other -> no one gets through
  - Carrier sense: only send if no one else is sending
- What happens when previous sender finishes?
  - Everyone who is waiting tries to send, at same time!
  - Everyone collides, no one succeeds

The Two Generals Problem

- Two armies are encamped on two hills surrounding a city in a valley



- The generals succeed if they agree on the same time to attack, fail otherwise
- Their only way to communicate is by sending a messenger through the valley, but that messenger could be captured (and the message lost)

Two Generals Protocol

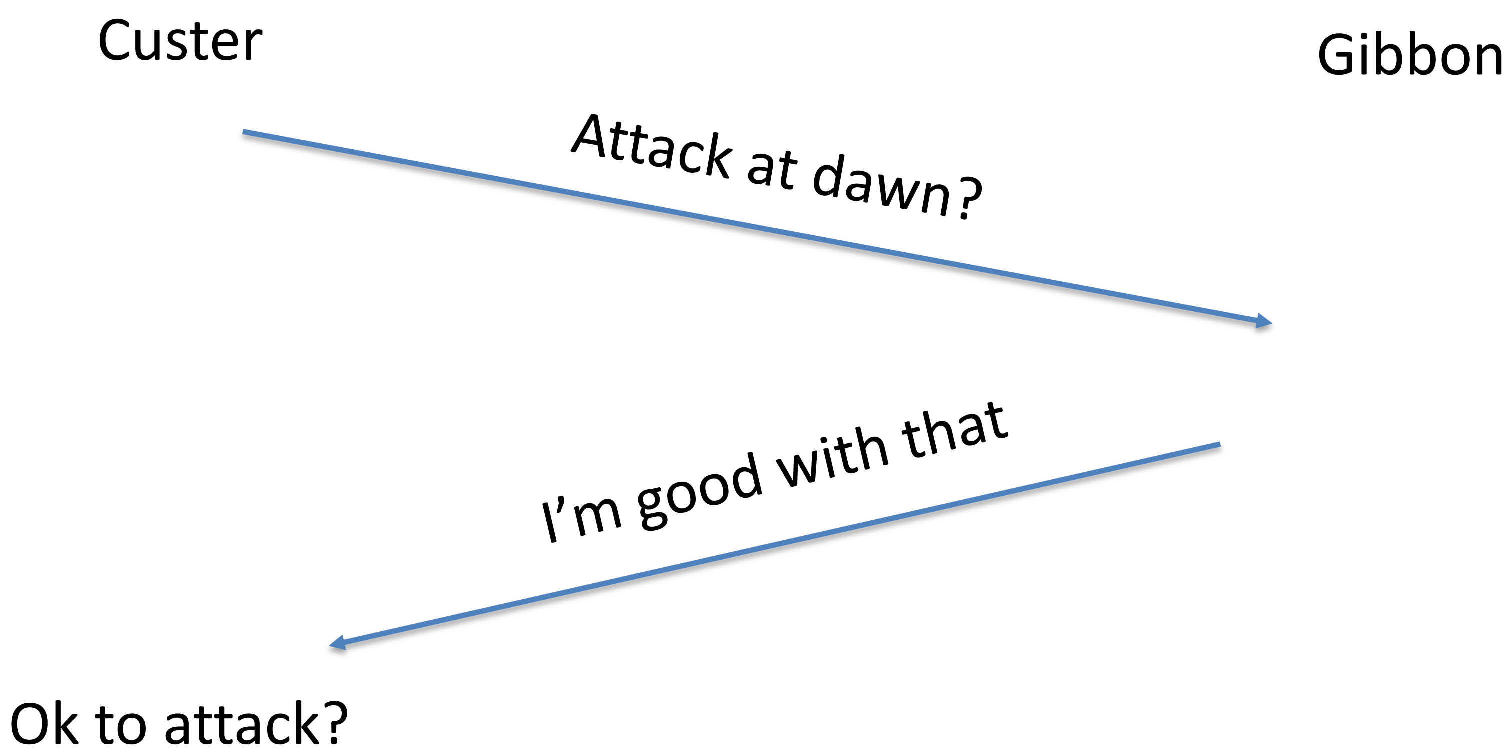
Custer

Gibbon

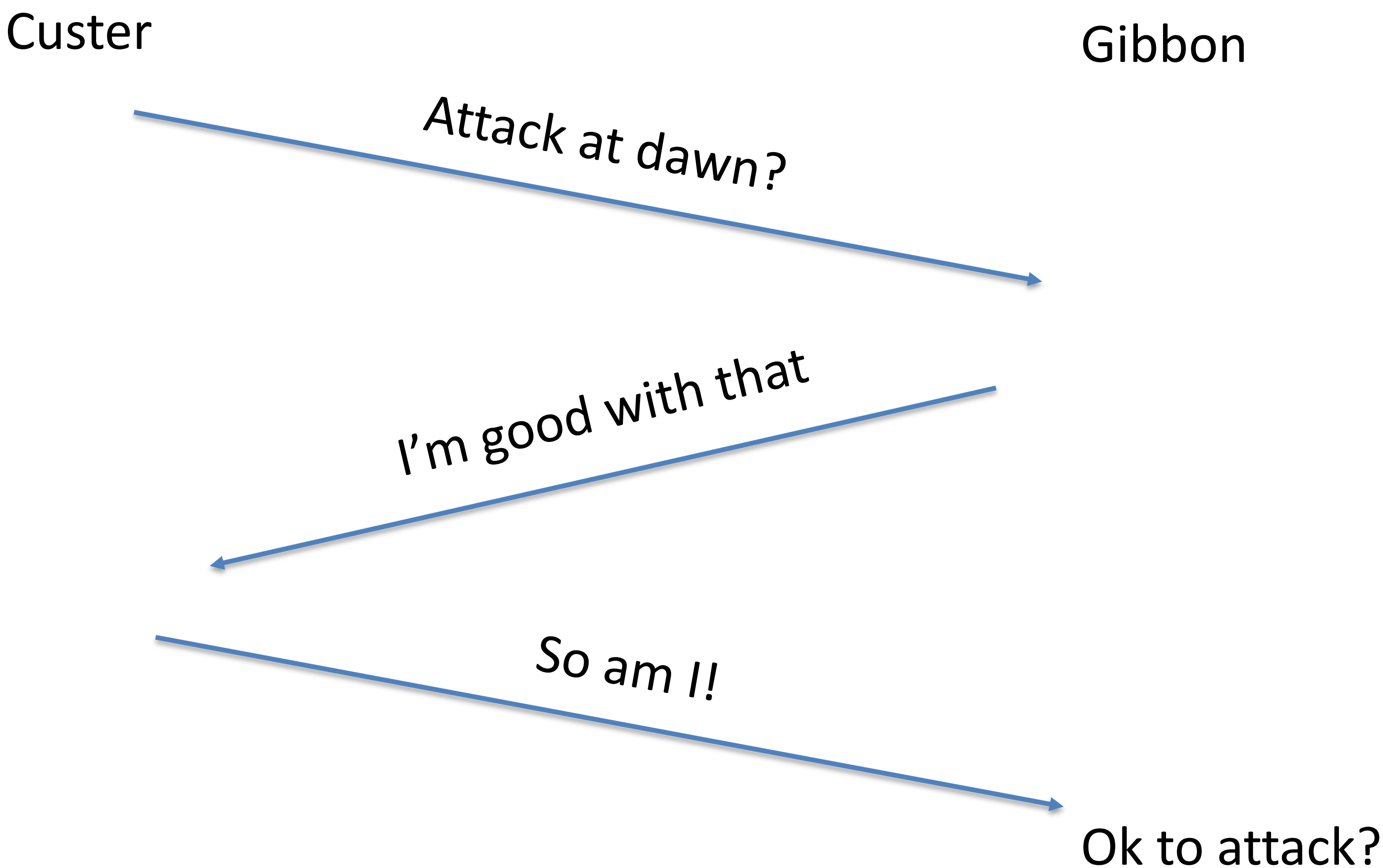
Attack at dawn?

Ok to attack?

Two Generals Protocol



Two Generals Protocol



The Two Generals Problem

- No solution is possible!
- If a solution were possible:
  - it must have involved sending some messages
  - but the last message could have been lost, so we must not have really needed it
  - so we can remove that message entirely
- We can apply this logic to any protocol, and remove all the messages — contradiction

Why Are Distributed Systems Hard?

- Asynchrony
  - Different nodes run at different speeds
  - Messages can be unpredictably, arbitrarily delayed
- Failures (partial and ambiguous)
  - Parts of the system can crash
  - Can’t tell crash from slowness
- Concurrency and consistency
  - Replicated state, cached on multiple nodes
  - How to keep many copies of data consistent?

Why Are Distributed Systems Hard?

- Performance
  - Have to efficiently coordinate many machines
  - Performance is variable and unpredictable
  - Tail latency: only as fast as slowest machine
- Testing and verification
  - Almost impossible to test all failure cases
  - Proofs (emerging field) are really hard
- Security
  - Need to assume adversarial nodes

Three-tier Web Architecture

- Scalable number of front-end web servers
  - Stateless (“RESTful”): if crash can reconnect the user to another server
- Scalable number of cache servers
  - Lower latency (better for front end)
  - Reduce load (better for database)
  - Q: how do we keep the cache layer consistent?
- Scalable number of back-end database servers
  - Run carefully designed distributed systems code

Three-Tier Web Architecture

Cache Server

Cache Server

Cache Server

Front End Server

Front End Server

Front End Server

Front End Server

Client

Storage Server

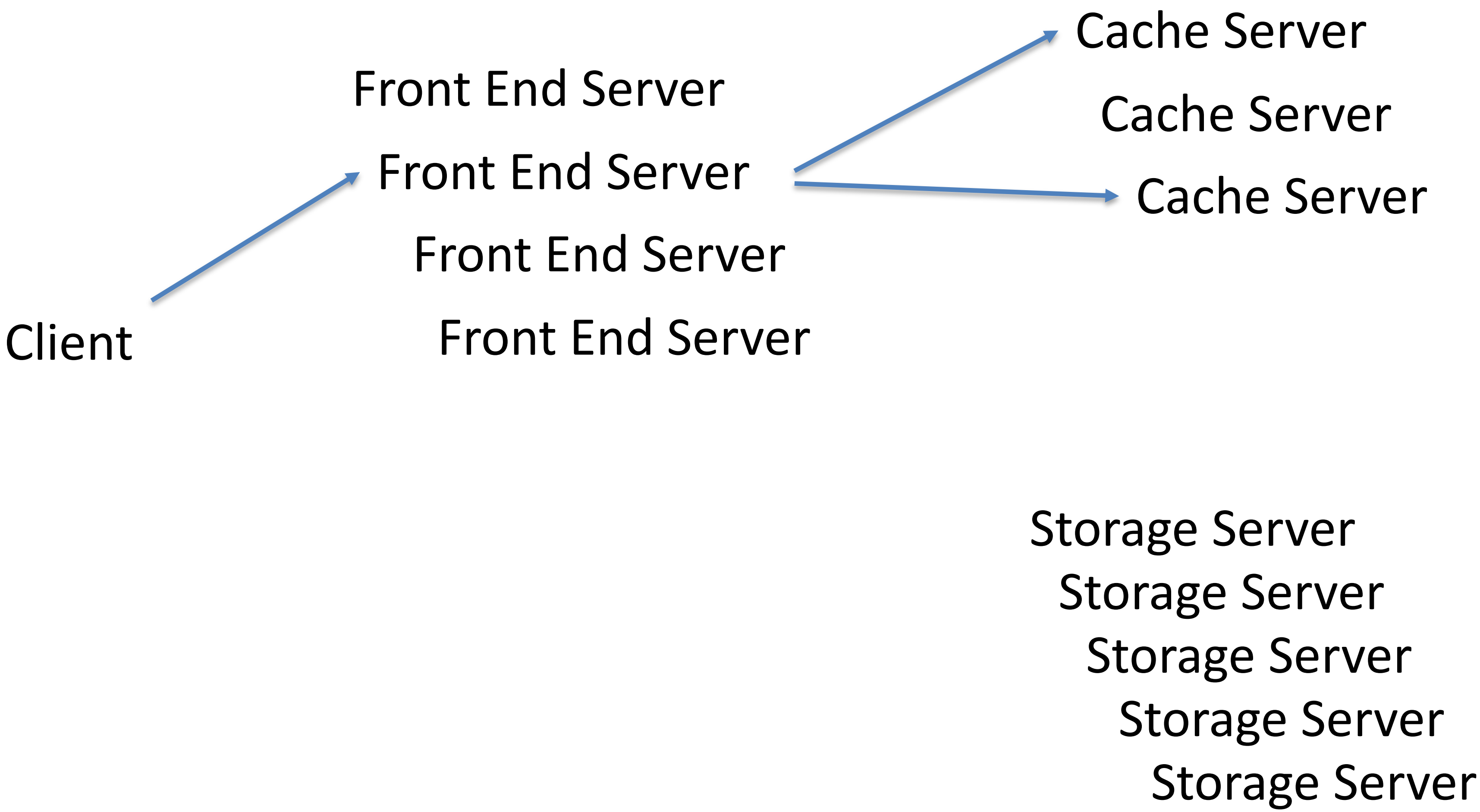
Storage Server

Storage Server

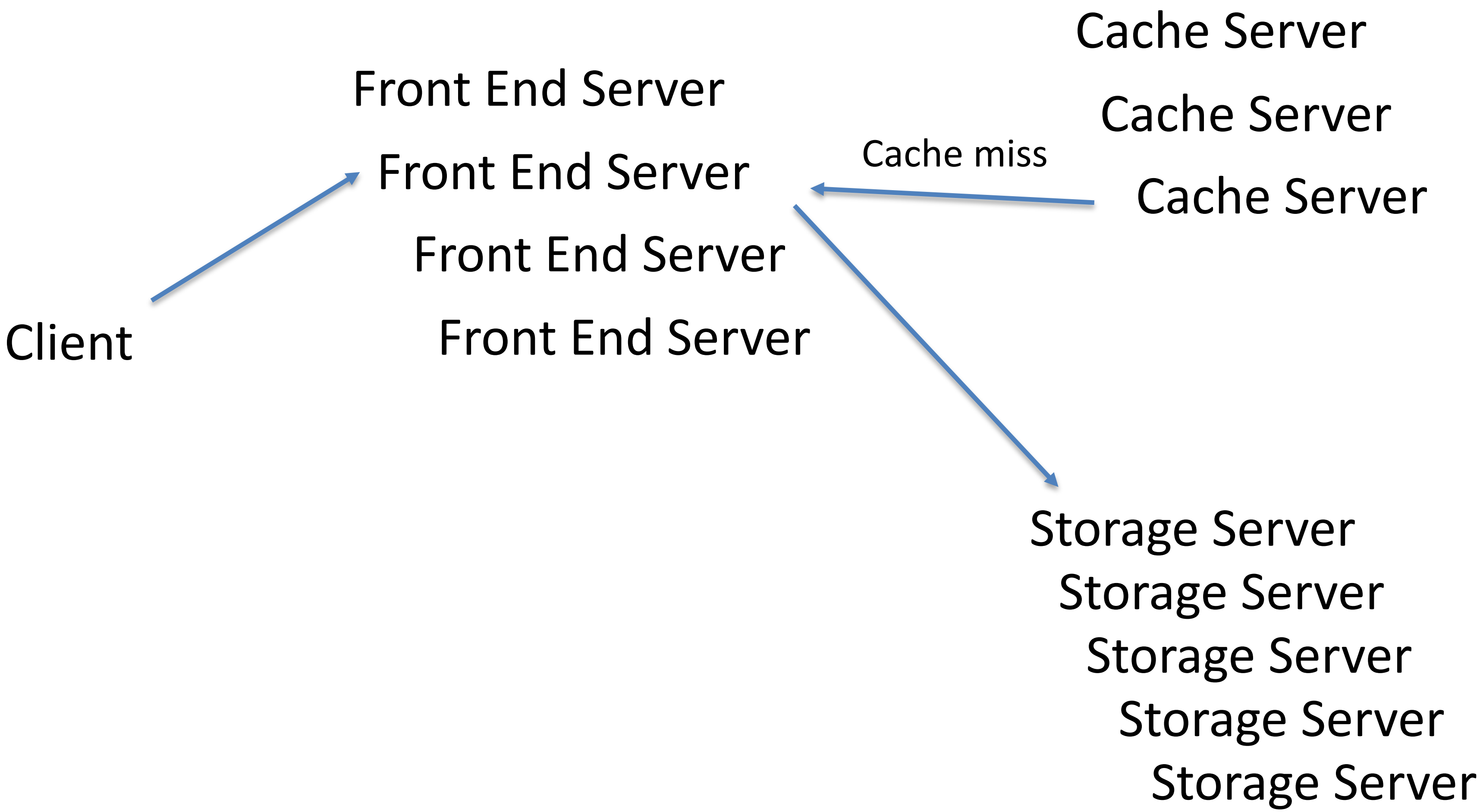
Storage Server

Storage Server

Three-Tier Web Architecture



Three-Tier Web Architecture



And Beyond

- Worldwide distribution of users
  - Cross continent Internet delay ~ half a second
  - Amazon: reduction in sales if latency > 100ms
- Many data centers
  - Near every user
  - Smaller data centers have web and cache layer
  - Larger data centers include storage layer as well
  - How do we coordinate updates across data centers?

Remote Procedure Call (RPC)

A request from a client to execute a function on a server.

- To the client, looks like a procedure call
- To the server, looks like an implementation of a procedure call

Thought Experiment

- Client sends a request to Amazon
- Network is flaky
  - Don’t hear back for a second
- Can you tell?
  - Request was lost
  - Server was down
  - Request got through, reply was lost
- Should the client resend?

Thought Experiment

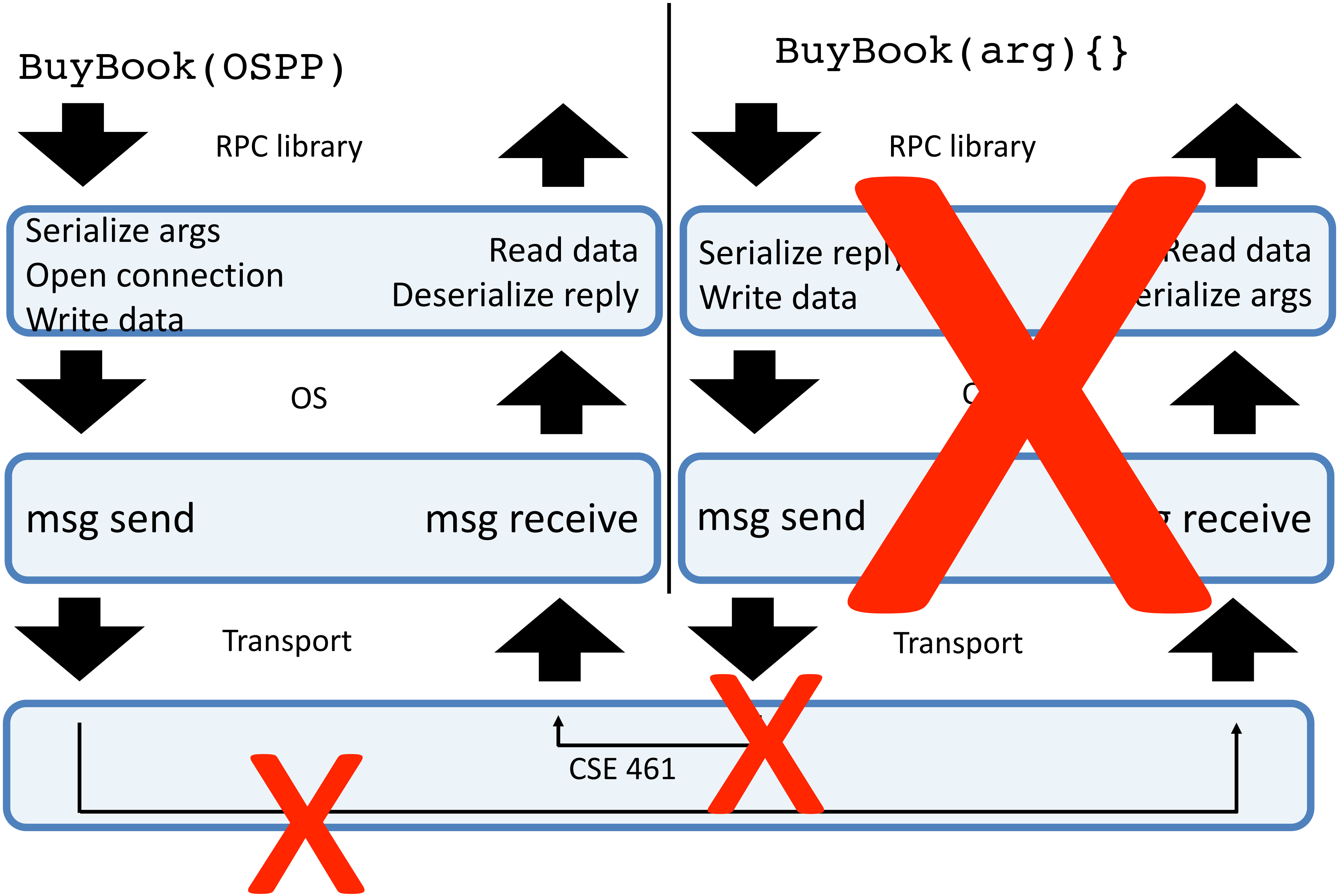
- The client resends
- But the original packet got through
- What should the server do?
  - Crash?
  - Do the operation twice?
  - Something else?

Remote Procedure Call (RPC)

Client request to execute a function on the server

- On client: result = BuyBook(OSPP)
  - Parameters marshalled into a message (arbitrary types)
  - Message sent to server (may be multiple pkts)
  - Wait for reply
- On server: implement BuyBook
  - message is parsed
  - Perform operation
  - Put result into a message (may be multiple pkts)
  - Result returned to client

RPC implementation



RPC vs. Procedure Call

- What is equivalent of:
  - The name of the procedure?
  - The calling convention?
  - The return value?
  - The return address?

RPC vs. Procedure Call

Binding

- Client needs a connection to server
- Server must implement the required function
- What if the server is running a different version of the code?

Performance

- procedure call: ~ 10 instructions = ~3 ns
- RPC in data center: 100 usec => 10K x slower
- RPC in the wide area: 100+ msec => 10M x slower

RPC vs. Procedure Call

Failures

- What happens if messages get dropped?
- What if client crashes?
- What if server crashes?
- What if server crashes after performing op but before replying?
- What if server appears to crash but is slow?
- What if network partitions?

Message Ordering

- Client sends a sequence of messages to server
  - a, b, c, d …
- Some can get dropped
  - Let’s say c
  - Receiver acks correctly received messages
  - Client retransmits anything missing (after timeout)
- Server gets sequence
  - a, b, d, e, c …
- Fix?

Message Ordering

- Client sends a sequence of messages to server
  - a, b, c, d …
- Some can get dropped
  - Receiver acks correctly received messages
  - Client retransmits anything missing (after timeout)
- Server gets sequence (why?)
  - a, b, c, d, e, c, …
- Fix?

Message Ordering

- Message ordering
  - Label messages with sequence number
  - Detect missing messages
  - Detect unneeded retransmissions
- Labs assume each client sends only one RPC at a time
  - Still need to worry about lost and duplicate RPCs

RPC vs. Procedure Call

Failures

- What happens if messages get dropped?
- What if client crashes?
- What if server crashes?
- What if server crashes after performing op but before replying?
- What if server appears to crash but is slow?
- What if network partitions?

RPC Semantics

- Semantics = meaning

- reply == ok => ???
- reply != ok => ???

Semantics

- At least once (NFS, DNS, lab 1b)
  - true: executed at least once
  - false: maybe executed, maybe multiple times
- At most once (lab 1c)
  - true: executed once
  - false: maybe executed, but never more than once
- Exactly once
  - true: executed once
  - false: never returns false

At Least Once

RPC library waits for response for a while

If none arrives, re-send the request

Do this a few times

Still no response -- return an error to the application

Non-replicated key/value server

Client sends Put k v

Server gets request, but network drops reply

Client sends Put k v again

- should server respond "yes"?
- or "no"?

What if op is “append”?

Does TCP Fix This?

- TCP: reliable bi-directional byte stream between two endpoints
  - Retransmission of lost packets
  - Duplicate detection
  - Useful: most RPCs sent over TCP!
- But what if TCP times out and client reconnects?
  - Browser connects to Amazon
  - RPC to purchase book
  - Wifi times out during RPC
  - Browser reconnects

When does at-least-once work?

- If no side effects
  - read-only operations (or idempotent ops)
- Example: MapReduce
  - doMapJob(i) – ok to do more than once
- Example: NFS
  - readFileBlock
  - writeFileBlock
  - What about delete file? Append to a file?

At Most Once

Client includes unique ID (UID) with each request

- use same UID for re-send

Server RPC code detects duplicate requests

- return previous reply instead of re-running handler

```
if seen[uid] {
    r = old[uid]
} else {
    r = handler()
    old[uid] = r
    seen[uid] = true
}
```

Some At-Most-Once Issues

How do we ensure UID is unique?

- Big random number?
- Combine unique client ID (IP address?) with seq #?
- What if client crashes and restarts? Can it reuse the same UID?
- In labs, nodes never restart
- Equivalent to: every node gets new ID on start

When Can Server to Discard Old RPCs?

Option 1:

    Never?

Option 2:

    unique client IDs

    per-client RPC sequence numbers

    client includes "seen all replies <= X" with every RPC

Option 3: only allow client one outstanding RPC at a time

    arrival of seq+1 allows server to discard all <= seq

Labs use Option 3

What if Server Crashes?

If at-most-once list of recent RPC results is stored in memory, server will forget and accept duplicate requests when it reboots

- Does server need to write the recent RPC results to disk?
- If replicated, does replica also need to store recent RPC results?

In Labs, server gets new address on restart

- Client messages aren’t delivered to restarted server

backup

MapReduce Computational Model

For each key k with value v, compute a new set of key-value pairs:

map (k,v) → list(k',v')

For each key k' and list of values v', compute a new (hopefully smaller) list of values:

reduce (k',list(v')) → list(v'')

User writes map and reduce functions.

Framework takes care of parallelism, distribution, and fault tolerance.

MapReduce Example: grep find lines that match text pattern

1. Master splits file into M almost equal chunks at line boundaries
2. Master hands each partition to mapper
3. map phase: for each partition, call map on each line of text
   - search line for word
   - output line number, line of text if word shows up, nil if not
4. Partition results among R reducers
   - map writes each output record into a file, hashed on key

Example: grep

5. Reduce phase: each reduce job collects 1/R output from each Map job
   - all map jobs have completed!
   - Reduce function is identity: v1 in, v1 out

6. merge phase: master merges R outputs

MapReduce (or ML or …) Architecture

- Scheduler accepts MapReduce jobs
  - finds a MapReduce master and set of avail workers
- For each job, MapReduce master <array>
  - farms tasks to workers; restarts failed jobs; syncs task completion
- Worker <array>
  - executes Map and Reduce tasks
- Storage <array>
  - stores initial data set, intermediate files, end results