Lecture 3:

RPC

Last Week:

- Causal Order & Traces
- Correctness Conditions Restrict What Traces Are Admissible (can be produced By A correct Protocol)
  - → Safetey: Bad State Never Occurs
  - → Liveness: There Is A Path To Good State
- Next Week: Return To Talking About Traces & Correctness

This Week: Remote Procedure Calls

→ WHY?

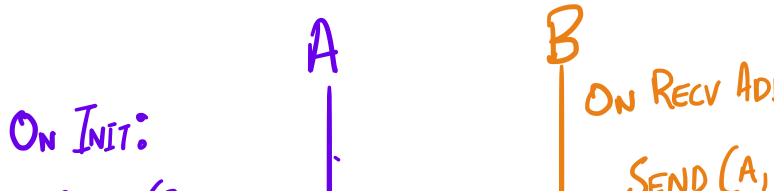
→ COMPARISONS TO HOW RPC WORKS TODAY

→ CONCERNS FOR SYSTEM DESIGN

WHY?

- THE WAY MOST PRACTICAL DISTRIBUTED SYSTEMS ARE IMPLEMENTED TODAY
  - ↳ GRPC, THRIFT, ...
- FOUND THEIR WAY INTO HOW MANY OF THE PROTOCOLS WE WILL TALK ABOUT ARE DESCRIBED

HOW DO RPCS AFFECT PROTOCOL DESCRIPTIONS



SEND(B, ADD(1,2))

ON RECV SUM(x):

...

...

A

ON INIT:

x = B.ADD(1,2)

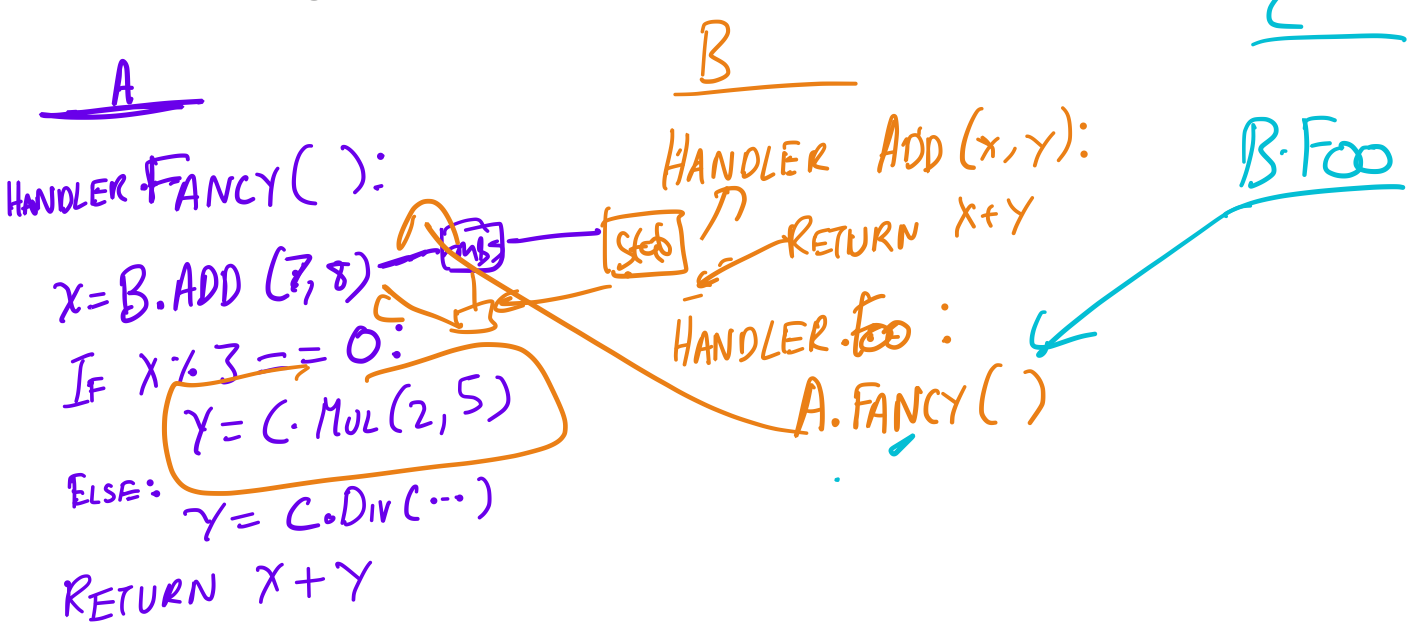
B

DEF HANDLER ADD(x,y):

RETURN x+y

...

- CALLS HAVE RETURNS
  - ↳ MIGHT BE NULL, BUT SIGNALS COMPLETION
    - ↳ GOOD PRACTICE. WHY?
- ALGORITHM & PROGRAM STRUCTURE REQUIRES BEING AWARE OF CONCURRENCY



SOURCES OF CONCURRENCY

⇒ RPC calls from other nodes

→ FAILURES

→ ...

CONSIDERATIONS WHEN READING & IMPLEMENTING ALGORITHMS SPECIFIED USING RPC

- CONCURRENCY

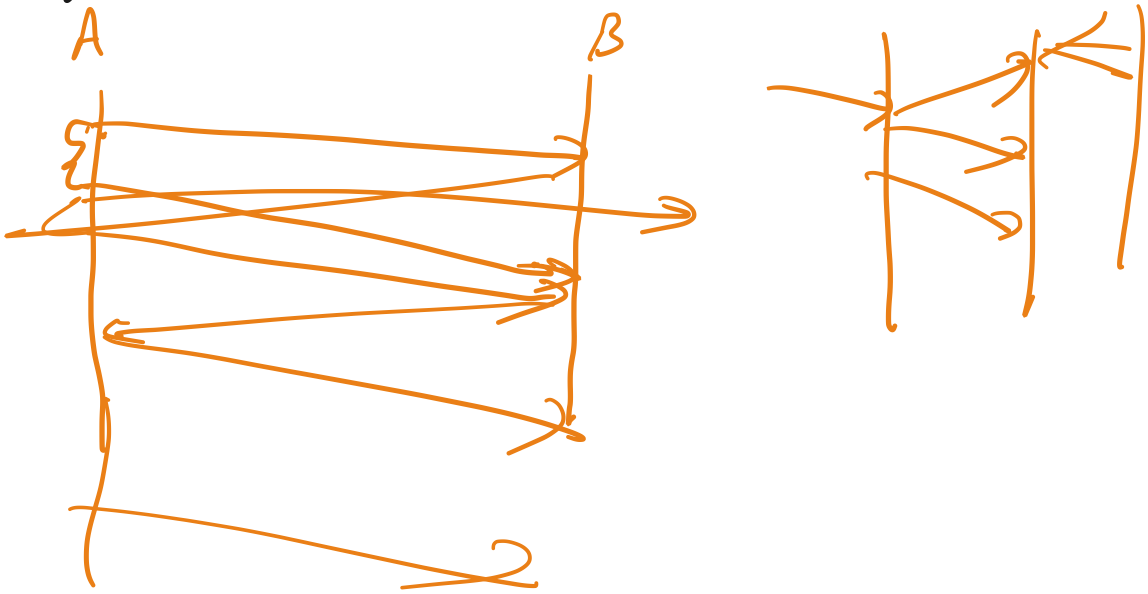
```
for p ∈ P {
    p.IAmAlive()
}
```

COMPARISON TO PRACTICE TODAY

BIRREL & NELSON Vs GRPC/Thrift

System Design Considerations When Using RPC

- Concurrency & Overload



- Tracing, Debugging, Reasoning About Performance

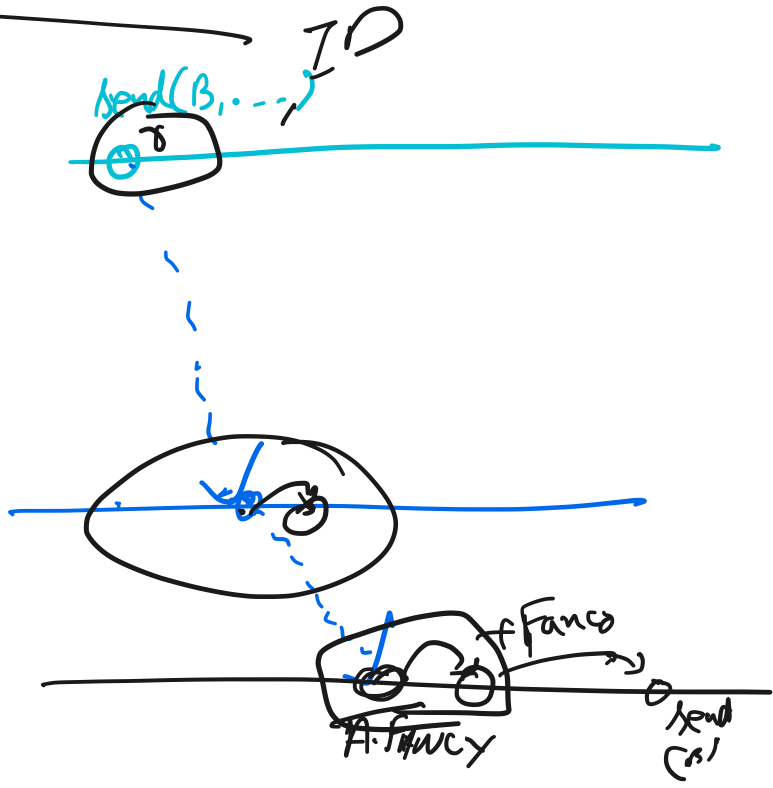
C

B.FOO{

SEND(B, FOO, ..., C)

B

A.FANCY



- FAILURES & FAILURE HANDLING

A QUICK PRIMER FOR NEXT WEEK

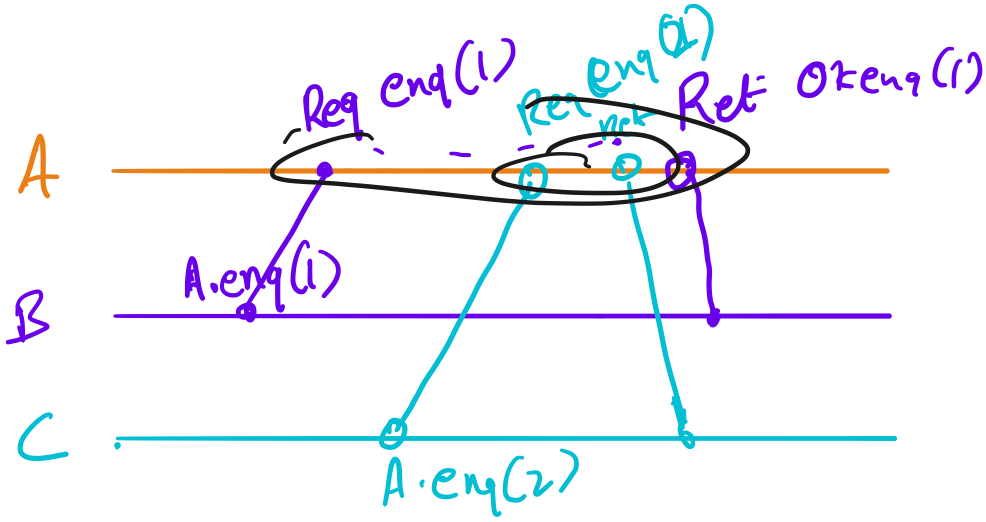
- DEFINING CONCURRENT REQUEST

Queue ← A

eng

Req X

Ret X



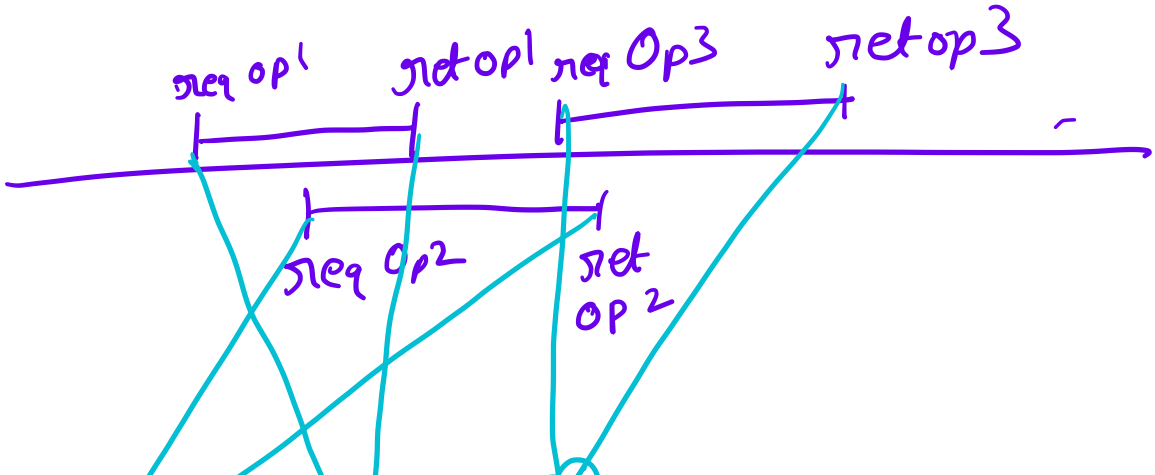
Op 1 ← RPCall , Op2

π_{req} Op1 π_{ret} Op1

Op1 and op2 are concurrent if

$(\pi_{req}\ Op1, \pi_{ret} op1) \cap$

$(\pi_{req}\ Op2, \pi_{ret} op2) \neq \emptyset$



Linearizable Queue

- op1 ← enq(5)
- op2 ← enq(3)
- op3 ← deq()