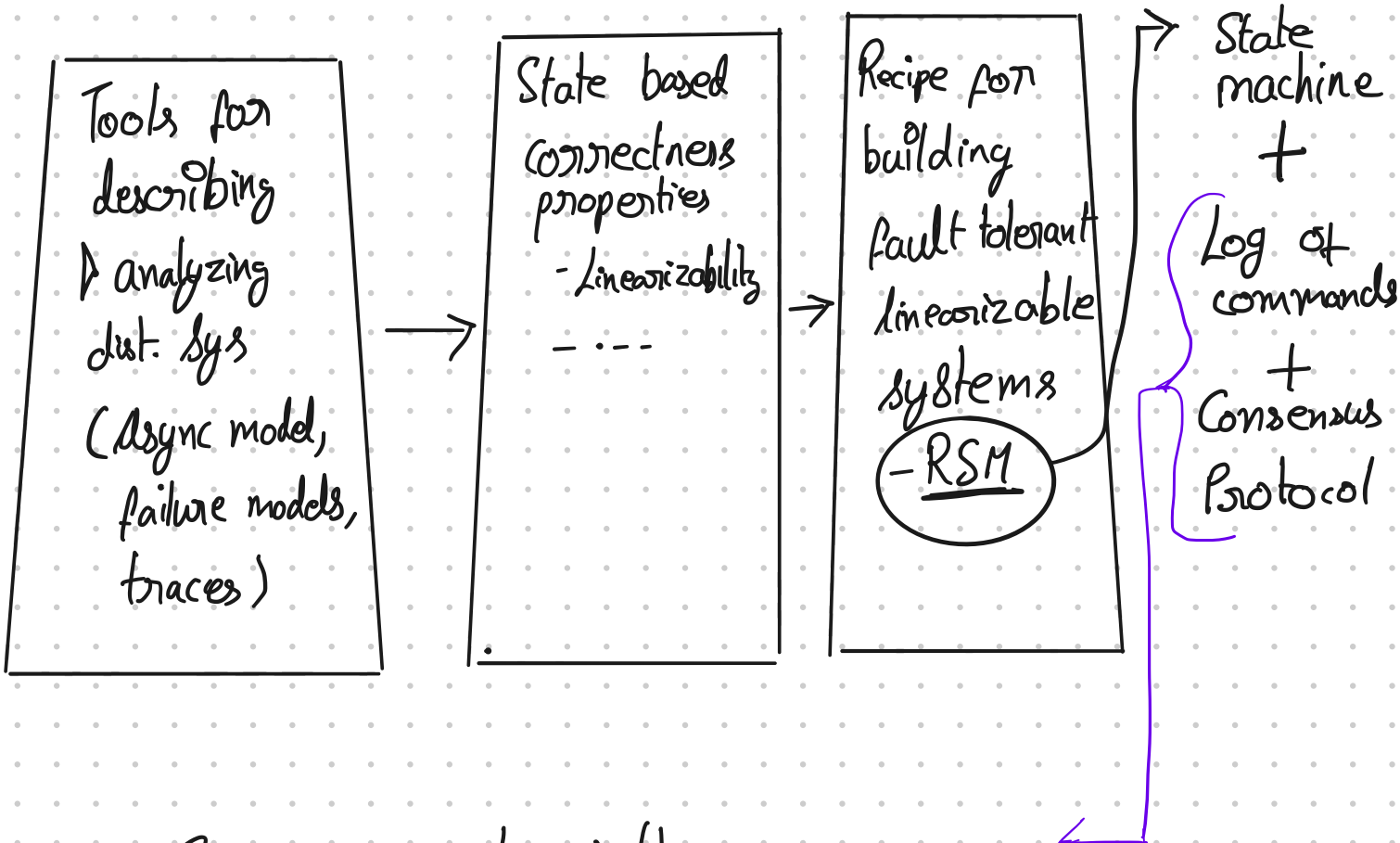
Distributed Systems:

FLP + Partial Synchrony

Rough Outline of Class so far



Seen a couple of these

- Raft
- Multi Paxos

} One of the most common class of Dist Protocols.

Today: Two questions

(1) Can we prove consensus protocols correct assuming only the asynchronous model?

Answer is No 😕

Why does this matter?

- Might be building fault tolerance on rotten foundations
  ↳ Practical: Fault Tolerant System Might Fail Unexpectedly, When Least Desired
- Unsatisfying: We don't understand our own construction

Aside: PODC'01 Award Speech FLP

- WORK INITIATED BECAUSE LYNCH WANTED TO PROVE CORRECTNESS OF AN IN-USE CONSENSUS PROTOCOL DESIGNE BY BUTLER LAMPSON.
- PAPER WAS DUE TO INABILITY TO DO THIS

Today: Two questions

(1) Can we prove consensus protocols correct assuming only the asynchronous model? No FLP'85

(2) WHAT ADDITIONAL ASSUMPTIONS ARE REQUIRED TO BUILD CORRECT CONSENSUS PROTOCOLS?

Intuition: Consensus seems to work in practice
Likely that our algorithms are assuming additional things about the Environment.

Generally characterized as either
(a) additional capabilities provided by ENVIRONMENT
(b) additional information available to protocol

Goals:
- Identify Env. assumptions

[Today,]

- See what is necessary & sufficient

Equivalently: what "minimal assumptions" suffice

Usually easier to think about this as what is the minimal amount of additional information required.

Preview: CHT'96

Ω { Eventually all processes know identity of one process p that will not fail

↳ Weakest FD for consensus

↳ (a) Ω sufficient to solve consensus

(b) If X sufficient to solve consensus then X sufficient to implement Ω

$\Rightarrow \Omega$ necessary.

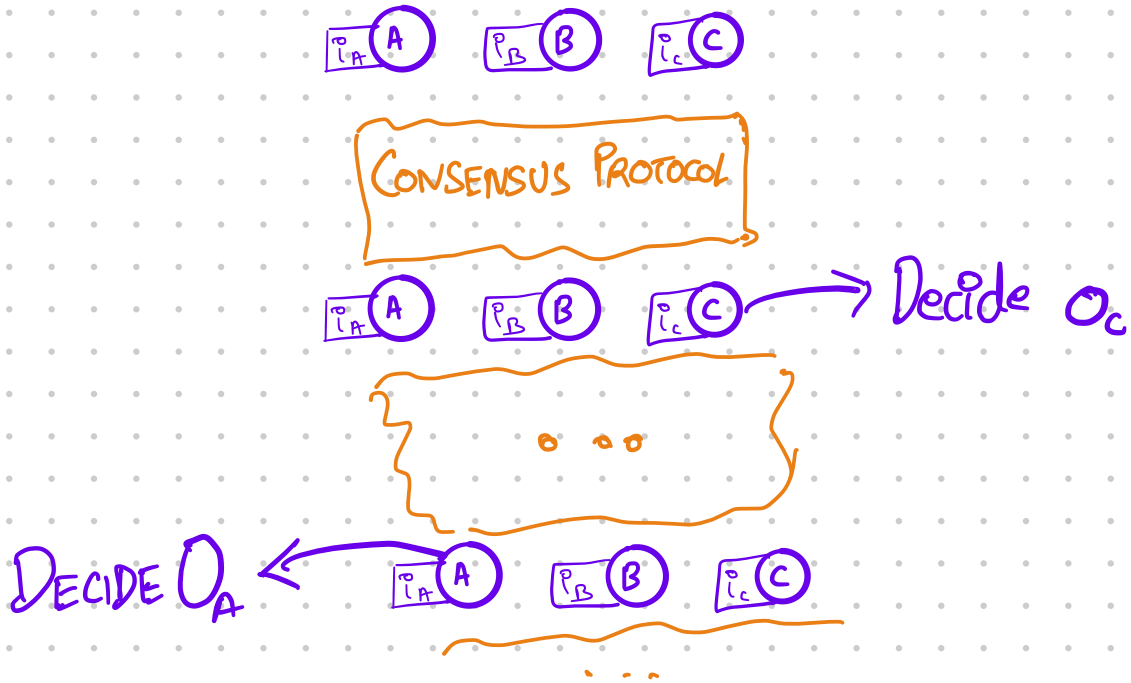
---

(1) Can we prove consensus protocols correct assuming only the asynchronous model? No

- Goal in class
  - Reiterate notion of correctness
  - Walk through some interesting bits of the proof
  - PLEASE WORK THROUGH THE PROOF YOURSELF

Why? - Simple & elegant argument
- Developed many of the techniques used to reason about dist. systems, e.g., IO automata

- From last week
  - Setting: BINARY CONSENSUS



$i_A, i_B, i_C, O_A, O_B, O_C \in \{\emptyset, 1\}$

CORRECTNESS Requirements
- Agreement: If O_x, O_y output by processes x, y then $O_x = O_y$
- Validity: O_x output by process x then $O_x = i_y$ for some process y.

- Termination: Eventually some process x decides.

Goal: Show that there does not exist a correct (a), deterministic (b), fault tolerant (c) consensus protocol in the async. model.

(a) Correct: Meets all three requirements above.

(b) Deterministic: Each processes behavior only depends on state + recvd message/timeout

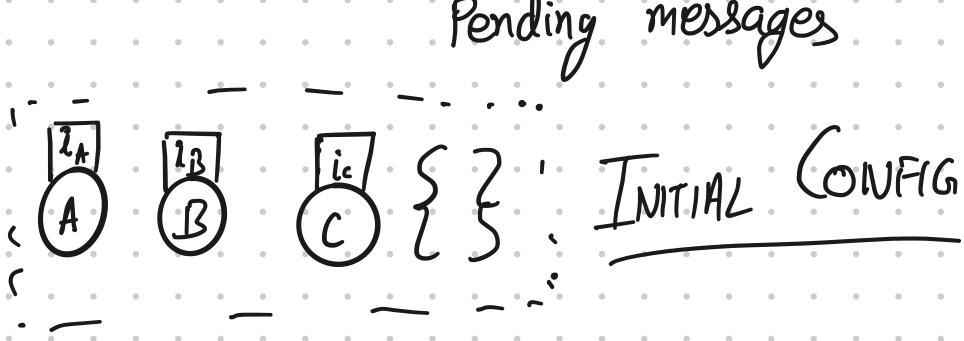
(c) Fault tolerant: Remains correct even if 1 process fails.

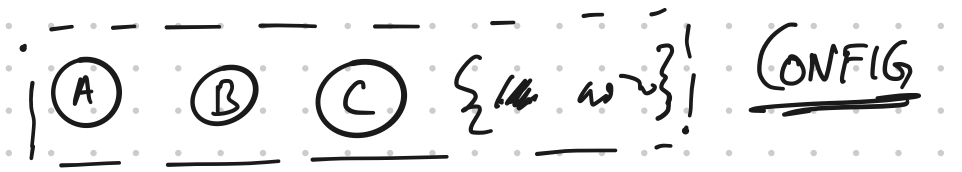
↳ Pick 1 for generality.
If protocol can work with f-faults, should also work with 1.

Core Approach: Show that for any protocol ∃ a trace (sequence of processing steps) that is of unbounded length

- Core idea: Configuration valence

Reminder: Configuration: State of all processes +





$\emptyset$-VALENT CONFIGURATION (C): All traces from C lead to a process deciding $\emptyset$

1-valent: —— Same with decisions of 1 ——

Bivalent: Some traces lead to a decision of $\emptyset$ & others lead to a decision of 1.

Observe: Configuration where a process decides is $\emptyset$ or 1 valent!

Claim 1: $\exists$ Initial Bivalent Configuration

i_A	i_B	i_C		
$\emptyset$	$\emptyset$	$\emptyset$	$\rightarrow$	$\emptyset$ ← Validity
$\emptyset$	$\emptyset$	1	$\rightarrow$	$\emptyset$
$\emptyset$	1	1	$\rightarrow$	1

Bivalent

Reminder: Any protocol

$\emptyset$ 1 1 $\rightarrow$
1 1 1 $\rightarrow$ 1 ← Validity

we consider must tolerate 1-fault & terminate

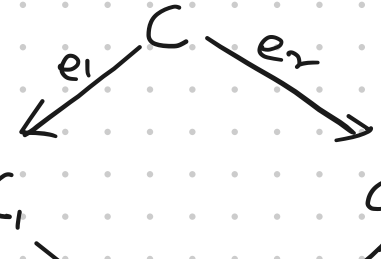
Claim 2: Given bivalent configuration C and enabled event e, ∃ enabled trace σ

C_i $\quad C \xrightarrow{\sigma} \xrightarrow{e} C'$ ← Bivalent

- Why the focus on applying a selected event e?

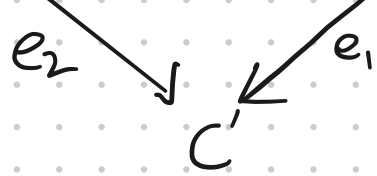
- Note, σ only consists of message delivery & timeouts: NO PROCESSES FAILED.

This is what most of the paper's proof focuses on. The proof is by contradiction, builds on a simple construct: THE DIAMOND LEMMA



e1, e2 affect different process

ORDER IN WHICH



e_1, e_2 are applied does not affect outcome.

See paper for full proof

Note on proof by contradiction

- Concern: Shows existence, not how to find it

Later work (Volzer '04) shows constructive proof

Net result:

Claim 1: ∃ Bivalent initial config C_I

Apply Claim 2: Pick event e enabled in C_I,

find σ

$$C_I \xrightarrow{\sigma} \xrightarrow{e} C_1$$

Apply claim 2 to C_1

...

Unbounded trace ⇒ No termination

↳ Does not violate failure model. Why?

- Where did we use assumption about fault tolerance?

(1) Can we prove consensus protocols correct assuming only the asynchronous model? No FLP'85

(2) WHAT ADDITIONAL ASSUMPTIONS ARE REQUIRED TO BUILD CORRECT CONSENSUS PROTOCOLS?

Remember: Async ⇒

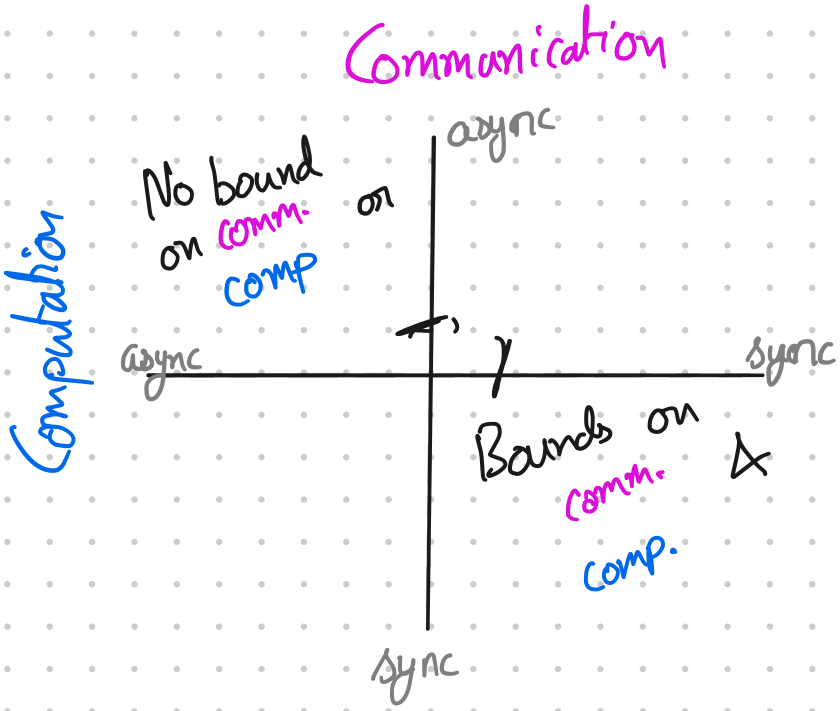
- Cannot distinguish b/w failure & delay
  ↳ Times are unpredictable, etc.
  Should not be able to use timeout, etc. to avoid problems.

But use of timeouts is common
↳ Standard approach to detecting failure, etc.

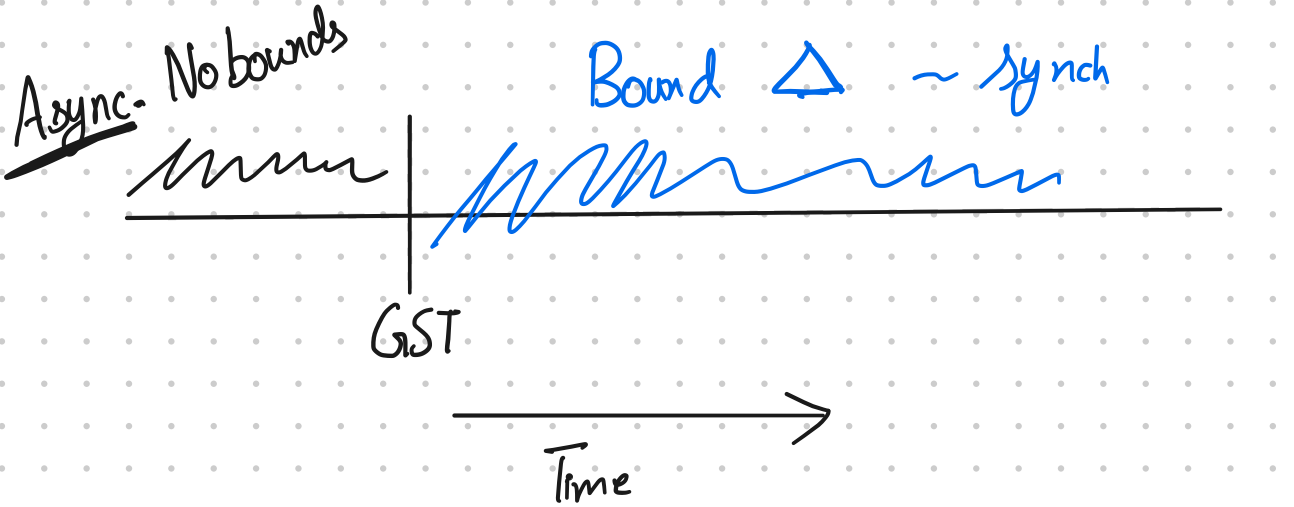
Hypothesis: Timeouts help

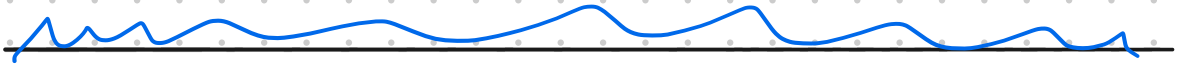
↳ Design a stronger model

↳ Partial synchrony



Partial Synchrony : Two definitions





Bound ~ sync
(Bound is not known)

Equivalent?

Relation to reality

- Communication: Engineered systems
  In steady state max bound exists
  - Latency - Transmission time (speed of light)
    Buffers
  - Reasonable bounds on loss
  - ...
  But bounds might not hold
  - During initialization
  - When recovering from failure
  - .
- Computation
  - Bounded response time except
    - Sched } Lots of reasons!!
    - GC
    - ...

Observe: Requires more assumptions about
- How everything was built

- Who else is using shared resources
- How they are shared

But can it get us fault-tolerant consensus?

Yes — use quorum intersection for agreement + validity; bounded latency to ensure termination

For fail-stop faults: $N \geq 2f+1$

Run in rounds $(0..\infty)$. Process i ($i = r \bmod N$, round r) is the designated proposer

r Step 0: All processes broadcast the set of values that should be considered (PROPER)

- Round 0: Process p broadcasts its input

- Round $k > 0$: Process p broadcasts
  (a) Values v on which it has a lock
  (b) If no v is locked, its input + past proposals.

i waits for $N-f$ messages

r Step 1: If i sees $N-f$ messages proposing

value v

Send (lock v, π)

else if i sees $N-f$ messages proposing

set S, pick $v \in S$

Send (lock v, π)

Process p on recv (lock v, π)

- Records v locked in π
- Sends (ack v, π) to i

i waits

Step 2

Process p releases locks on all values w, $w \neq v$ & w locked before π

If i receives $N-f$ ack v, π

Decide v

- Agreement
  - ↳ Core argument: If v is locked by $N-f$ process, all future proposals are for v
    - ↳ Paxos P1

- Termination — After GST
  - ↳ Timeout is long enough to collect responses from live

nodes + prevent failed nodes from blocking progress.

This round based structure to reason about termination, etc. is going to also show up in Failure Detectors

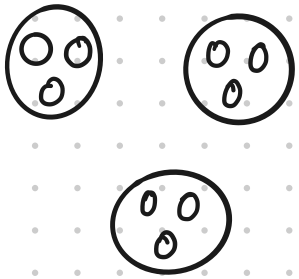
Observations from the midterm

- Failure models

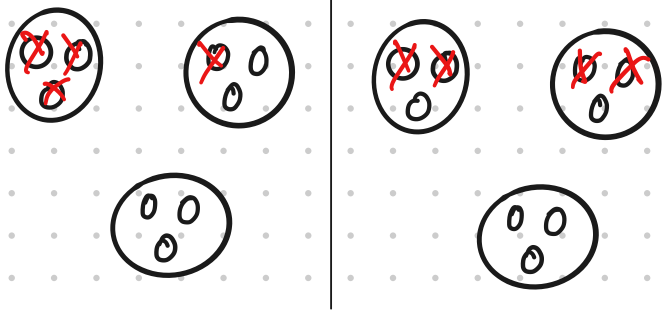
# of nodes that can fail

↳  Gets to pick what nodes to fail

→ Your analysis should consider worse case, not best case.



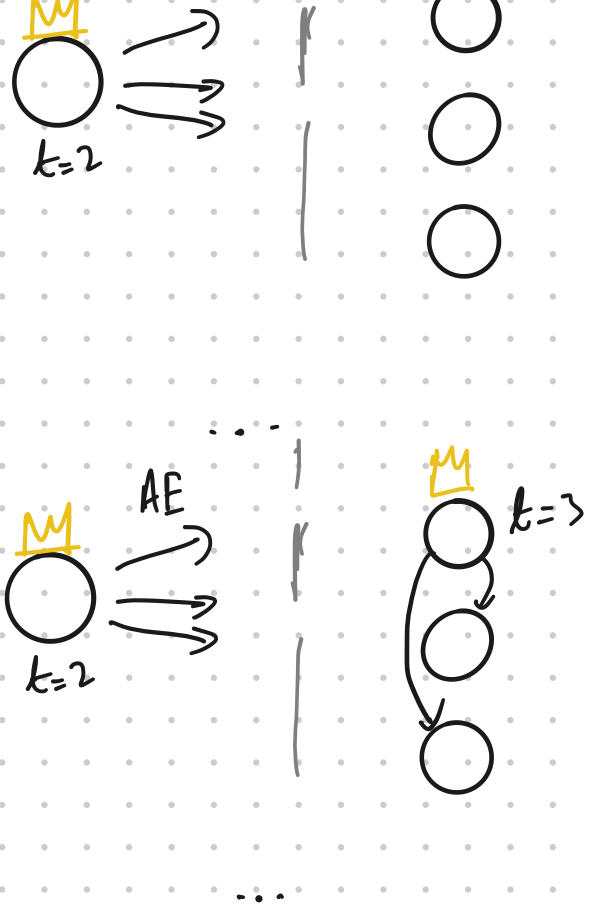
4 failures — 2 ways



- Leader Leases

- In Raft leaders do not STEP DOWN if th do not see enough responses to AE/HB

AE


k=2
AE
k=2
k=3