

CPSC 416 – Distributed Systems

Lecture 21: Adversarial Protocol Design

UBC Vancouver

2026-03-24

From Last Time

Articulate:

L20 gave us distributed transactions: 2PC guarantees atomicity across participants.

Turn to your neighbor. In one sentence: what is the fundamental limitation of 2PC?

(Pairs: 1 minute)

Failure-First Question

Every protocol we've studied has a failure model — the things that can go wrong.

What happens when the failure model is right, the protocol is correct, and things still break?

Sometimes the bug is in the protocol itself.

You understand a protocol when you can break it.

I'm going to give you three protocols.

Each one has a bug. The bug is *within* the stated failure model.

Your job: **construct an attack trace that makes it fail.**

Three rounds. Red team vs. blue team. Let's go.

The Rules

- **Red team:** Find the bug. Construct a concrete execution that violates the protocol's stated guarantee.
- **Blue team:** Propose a fix. Argue why your fix is correct.
- You stay within the *stated failure model*. No cheating.
- The adversary is honest: no message corruption, no Byzantine behavior — unless the failure model allows it.

Each round:

- 15 min — find the attack (groups)
- 5 min — present your attack (plenary)
- 5 min — discuss the fix (plenary)

Round 1

Gossip Membership Protocol

Difficulty: ★

Round 1: The Protocol

Goal: Every node maintains a consistent view of cluster membership.

Failure model: Crash-recovery. Network partitions can occur and heal.

Gossip Membership Protocol

```
State:
  neighbors: Set of (node_id, status)

On timer (every 5 seconds):
  peer = random_choice(neighbors)
  send GOSSIP(my_id, neighbors) to peer

On receive GOSSIP(sender_id, remote_neighbors):
  for each (node, status) in remote_neighbors:
    neighbors[node] = status // overwrite local
  neighbors[sender_id] = ALIVE // sender is alive
```

Stated guarantee: After a partition heals, all nodes eventually converge to the same membership view.

Round 1: Your Mission

Find an execution where two nodes permanently disagree about membership — even after the partition heals.

- You control when partitions occur and when they heal.
- You do **not** corrupt messages or violate crash-recovery.
- Construct a concrete trace: which node sends what, when.

(Groups of 3–4: 15 minutes. Write your attack on paper.)

Round 1: The Bug

The merge is a blind overwrite. There are no version vectors, no timestamps, no causal ordering.

Attack trace:

- 1 Node C joins the cluster. Nodes A and B learn about C.
- 2 **Network partitions:** {A, C} and {B}.
- 3 Node C crashes. A marks C as DEAD.
- 4 Partition heals. B gossips its *stale* view to A: "C is ALIVE."
- 5 A overwrites: C is ALIVE again. **But C is dead.**
- 6 Later, A gossips "C is ALIVE" back to B.
- 7 Both agree C is alive. Both are wrong.

Stale information overwrites fresh information. The protocol has no way to tell which is newer.

Round 1: The Fix

Version vectors (or logical timestamps per entry):

- Each membership entry carries a version number
- On merge: keep the entry with the *higher* version
- On status change: increment the version

This is what real gossip protocols do:

- SWIM (Scalable Weakly-consistent Infection-style Membership)
- Serf (HashiCorp)
- Cassandra's gossip protocol

The merge function must be monotonic: new information must always win over old.

Round 2

Heartbeat Leader Election

Difficulty: ★★

Round 2: The Protocol

Goal: Exactly one leader at any time. **Failure model:** Crash-stop. Partial synchrony: messages within 80ms after GST.

Heartbeat Leader Election

Constants:

```
T = 100ms // heartbeat timeout  
HEARTBEAT_INTERVAL = T / 3 // ~33ms
```

Leader behavior:

```
every HEARTBEAT_INTERVAL:  
    send HEARTBEAT(my_id, term) to all followers
```

Follower behavior:

```
on receive HEARTBEAT(leader_id, term):  
    reset election_timer to T  
on election_timer expires:  
    term = term + 1; become CANDIDATE  
    send VOTE_REQUEST(my_id, term) to all  
    collect votes; if majority: become LEADER
```

Stated guarantee: At most one leader per term.

Round 2: Your Mission

Construct an execution where two nodes both believe they are the leader — simultaneously.

- You control message delivery timing (within the 80ms bound after GST).
- You can choose when GST occurs.
- You do **not** crash any nodes or corrupt any messages.

(Groups of 3–4: 15 minutes. Draw a timeline.)

Round 2: The Bug

The timeout $T = 100\text{ms}$ is too close to the network delay bound of 80ms .

Attack trace (5 nodes: A is leader, B–E are followers):

- 1 A sends HEARTBEAT at $t = 0$.
- 2 **Network is slow.** Messages to B and C take 90ms . (Before GST, this is allowed.)
- 3 B's timer expires at $t = 100\text{ms}$. B becomes candidate (term 2).
- 4 B's VOTE_REQUEST reaches C, D, E. C and D vote for B.
- 5 B becomes leader at term 2 — **while A is still leader at term 1.**
- 6 A doesn't know it's been deposed. A's heartbeats are still in flight.

Two nodes act as leader simultaneously. Clients talking to A get stale responses.

Round 2: Why This Is Subtle

The “at most one leader per term” guarantee is not violated.

A is leader of term 1. B is leader of term 2.

The real problem:

- The guarantee says “one leader per term”
- It does **not** say “one leader at a time”
- A doesn't know it's been superseded
- There is no *fencing* mechanism

The timing bug (T too close to network delay) makes this happen *frequently* rather than rarely.

The failure model allows it. The protocol doesn't prevent it.

Round 2: The Fix

Multiple defenses:

- 1 **Increase timeout margin:** $T \gg$ network delay bound. Rule of thumb: $T \geq 10 \times \max$ network RTT.
- 2 **Leader leases:** Leader holds a time-bounded lease. Only serves reads while lease is valid. Followers don't start elections until lease expires.
- 3 **Fencing tokens:** Every leader operation carries its term. Followers reject operations from stale terms.

Raft uses: randomized election timeouts (150–300ms) with typical heartbeat intervals of 50ms. The margin is wide.

Timing parameters are not arbitrary. They encode assumptions about the failure model.

Round 3

Simplified Two-Phase Commit

Difficulty: ★ ★ ★

(Callback to L20)

Round 3: The Protocol

Goal: Atomic commit across 3 participants.

Failure model: Crash-recovery. Stable storage survives crashes.

Simplified 2PC

Coordinator:

```
send PREPARE to all participants
wait for votes from all participants
if all vote YES:
    log(COMMIT) to stable storage; send COMMIT to all
else:
    log(ABORT) to stable storage; send ABORT to all
```

Participant:

```
on PREPARE:
    if can_commit:
        log(YES) to stable storage; lock resources
        send YES to coordinator
    else: send NO to coordinator
on COMMIT: apply changes, release locks
on ABORT: release locks
```

Round 3: The Stated Guarantees

Safety:

- If any participant sees COMMIT, all eventually see COMMIT.
- If any participant sees ABORT, none see COMMIT.

Liveness:

- Every transaction eventually commits or aborts.

Failure model:

- Any node (including the coordinator) can crash and recover.
- On recovery, nodes read their stable storage log.

Round 3: Your Mission

Construct an execution where a participant is blocked forever — holding locks, unable to commit or abort.

- You can crash the coordinator at any point.
- You can crash and recover participants.
- Stable storage survives crashes (that's the model).
- Construct a concrete trace with 3 participants and 1 coordinator.

Bonus challenge: Design a recovery protocol that fixes this.

(Groups of 3–4: 15 minutes. This one is hard.)

Round 3: The Bug

No recovery protocol for coordinator crash between PREPARE and COMMIT/ABORT.

Attack trace (Coordinator C, Participants P1, P2, P3):

- ➊ C sends PREPARE to P1, P2, P3.
- ➋ P1 votes YES, logs it, locks resources.
- ➌ P2 votes YES, logs it, locks resources.
- ➍ P3 votes YES, logs it, locks resources.
- ➎ C receives all three YES votes.
- ➏ C crashes before logging COMMIT. (Or: C logs COMMIT, sends COMMIT to P1, then crashes.)

Result: P2 and P3 are stuck. They voted YES. They can't unilaterally abort (P1 might have received COMMIT). They can't commit (they never received COMMIT).

Locks held indefinitely. The system is blocked.

Round 3: Why Participants Can't Recover Alone

C: crashed

P1: voted YES

P2: voted YES

P3: voted YES

P2 asks P1 and P3: “Did anyone get a COMMIT?”

P1 says: “No.” P3 says: “No.”

Safe to abort? Not necessarily. C may have logged COMMIT before crashing. When C recovers, it will send COMMIT.

Round 3: The Fix Challenge

Option 1: Coordinator recovery (simple but slow)

- Wait for C to recover. C re-reads log: COMMIT or ABORT.
- **Problem:** What if C never recovers?

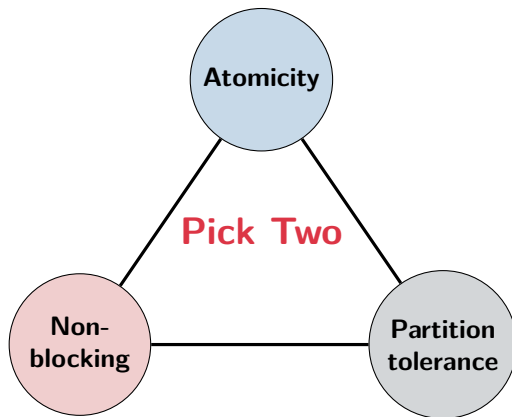
Option 2: Cooperative termination protocol

- On timeout, participants ask each other for status.
- Anyone received COMMIT → all commit. Anyone voted NO → all abort.
- **Problem:** All voted YES, none received COMMIT — still blocked.

Option 3: Three-Phase Commit (3PC)

- Add PRE-COMMIT phase between PREPARE and COMMIT.
- Eliminates the uncertainty window — but requires synchrony assumptions.

Round 3: The Fundamental Tension



2PC: Atomicity + Partition tolerance. Blocking.

3PC: Atomicity + Non-blocking. Requires synchrony.

Same structure as FLP (L12). Same escape patterns.

Debrief

Three Protocols, Three Bug Classes

Round	Protocol	Bug class	Fix
1	Gossip membership	Missing ordering	Version vectors
2	Leader election	Timing assumption	Wider margins / leases
3	2PC	Missing recovery	3PC / consensus

Pattern:

- Round 1: The protocol doesn't track *enough state*.
- Round 2: The protocol assumes *too much* about timing.
- Round 3: The protocol doesn't handle *all* failure cases.

The Adversary Is the Network

Recall L12:

“You never lie. You never corrupt. You just delay.”

In Round 1, you exploited *partition and heal timing*.

In Round 2, you exploited *message delay*.

In Round 3, you exploited *crash timing*.

None of these required malice. The network does all of this routinely.

The adversary doesn't exist. There's just a system where things take variable time and components fail.

Why Adversarial Thinking Matters

You will design protocols.

- In your capstone projects, right now
- In industry, at scale
- In research, at the frontier

The skill is not “know Paxos.” The skill is:

- 1 State the failure model.
- 2 State the guarantees.
- 3 **Try to break it yourself before the network does.**

If you can't construct an attack trace, you don't yet understand the protocol.

Connection: The Course Arc

- **L10:** Quorum intersection is the foundation.
- **L11–L13:** Walking and breaking Paxos.
- **L12:** “Be the Adversary” — we showed you how.
- **L14:** Flexible Paxos — same safety, different trade-offs.
- **L20:** 2PC and the blocking problem.
- **L21 (today):** You *are* the adversary. Three times.

Next (L22): The failure model itself is a choice. Who gets to make it? What are the consequences?

The failure model isn't physics. It's a political decision.

Evidence Question

Pick one of today's three protocols.

What would you log to detect the bug in production before a user notices?

What would the alert look like?

(Think about this for your self-assessment.)

- **Self-assessment:** Opens 13:00, due next class 11:30 (Pacific)
- **Reading for L22:** TBD (Canvas → Readings page)
- **Next lecture (L22):** Who decides the failure model?
- Capstone milestones (Canvas → Assignments)

Today you were the adversary. Next time we ask: who decides what the adversary is allowed to do?