

CPSC 416 – Distributed Systems

Lecture 08 — Coordinator Failure and Blocking

UBC Vancouver

2026-01-29

Failure-first question

*The coordinator crashes after collecting votes. You're a participant who voted YES.
What can you do?*

Recap: 2PC works beautifully...

The 2PC protocol:

- ① Coordinator sends PREPARE
- ② Participants vote YES/NO
- ③ Coordinator writes decision to log
- ④ Coordinator broadcasts decision
- ⑤ Everyone commits or aborts together

Key insight from last time: The coordinator's log entry is the “last vote” — the moment consensus crystallizes.

But we assumed the coordinator is *available*...

...until the coordinator crashes

Scenario: Three participants (P1, P2, P3) and a coordinator (C).

- 1 C sends PREPARE to all
- 2 P1, P2, P3 all vote YES
- 3 C receives all YES votes
- 4 **C crashes before writing decision**

Now what?

- P1, P2, P3 all voted YES
- They promised to commit if asked
- But no one knows the decision
- They can't reach C

Coordinator failure scenarios

Before sending PREPARE:

- No problem — transaction never started
- Participants didn't prepare anything

After PREPARE, before decision:

- Participants have voted YES
- They're holding locks
- Waiting for a decision that may never come

After decision, before full broadcast:

- Some participants know the outcome
- Others are still waiting
- Inconsistent knowledge

The UNCERTAIN state

A participant who voted YES is in the **UNCERTAIN** state:

- **Can't commit unilaterally:** Coordinator might have aborted (other participant voted NO, or timeout)
- **Can't abort unilaterally:** Coordinator might have committed (and others already committed)
- **Can't wait forever:** Need to release locks, make progress

The participant has surrendered autonomy.

By voting YES, it gave up the right to decide on its own.

The blocking problem

Blocking: Participant is stuck waiting, holding locks.

Why is this bad?

- Locks block other transactions
- Other transactions wait for locks
- They timeout or block further
- System throughput drops to zero

2PC is a **blocking protocol**.

If the coordinator is unavailable, participants may block indefinitely.

The safety/liveness tension

Safety: All participants must reach the same decision.

Liveness: The protocol must eventually terminate.

In UNCERTAIN state:

- If we commit: might violate safety (coordinator aborted)
- If we abort: might violate safety (coordinator committed)
- If we wait: violates liveness

2PC chooses safety over liveness.

Participants wait (potentially forever) rather than risk inconsistency.

Quick discussion: Escaping UNCERTAIN (2 min)

Turn to your neighbor:

You're a participant. You voted YES. The coordinator is unreachable.

What could you try? What would go wrong?

Every approach has a problem

Ask other participants: “Did anyone get a decision?”

- Problem: What if everyone is UNCERTAIN?

Elect a new coordinator:

- Problem: How do you agree on who? That requires coordination!

Timeout and abort:

- Problem: Coordinator might have already committed!

Timeout and commit:

- Problem: Coordinator might have aborted!

There's no safe unilateral action. This isn't a bug — it's fundamental.

The “obvious” fix: 3PC

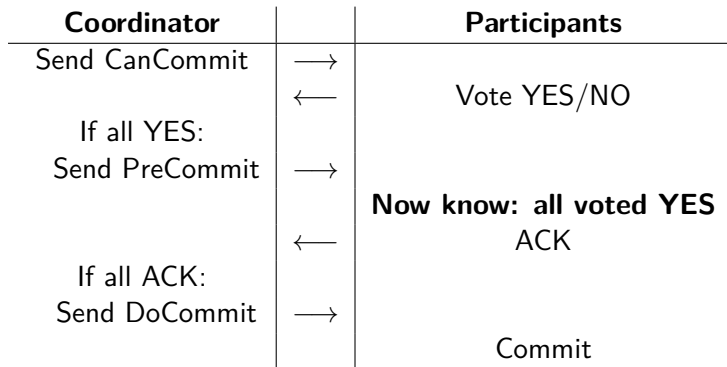
Key insight: The problem is that participants go directly from UNCERTAIN to COMMIT. What if we added an intermediate state?

Three-Phase Commit (3PC):

- 1 CanCommit: “Can you commit?” → votes
- 2 PreCommit: “Prepare to commit” → acknowledgments
- 3 DoCommit: “Actually commit” → commit

The PreCommit phase means: everyone knows everyone voted YES.
If coordinator dies, participants can coordinate to finish.

3PC: The protocol



Key: After receiving PreCommit, participant knows everyone voted YES.

See also: Bernstein & Newcomer Ch. 7, Skeen “Nonblocking Commit Protocols” (1981)

3PC: Recovery from coordinator failure

Coordinator dies after PreCommit:

- Participants are in “pre-committed” state
- They know everyone voted YES
- Safe to commit without coordinator!
- One participant can become new coordinator

Coordinator dies before PreCommit:

- Some participants might have voted YES
- But no one is pre-committed
- Safe to abort (no one has promised to commit)

3PC: The catch

3PC works... **under certain assumptions.**

The synchrony assumption:

- Messages arrive within a known time bound
- Timeouts mean “actually dead” not “slow”
- Processes can distinguish crash from delay

What happens without synchrony?

- Coordinator sends PreCommit to P1, then crashes
- Network partition: P1 is isolated, P2/P3 can't reach it
- P2/P3 timeout, elect new coordinator, abort
- P1 commits (it received PreCommit)
- **Inconsistency!**

Network partitions break 3PC

The fundamental problem:

In an asynchronous network, you can't distinguish:

Crashed node	Slow/partitioned node
Will never respond	Will eventually respond

If you guess wrong:

- Guess crashed, it's actually partitioned → inconsistency
- Guess partitioned, it's actually crashed → blocking

FLP: The impossibility result

Fischer, Lynch, Paterson (1985):

In an asynchronous system where even one process can crash, there is no protocol that guarantees consensus will be reached.

What this means:

- Safety (no disagreement) — achievable
- Liveness (eventual termination) — achievable
- **Both together with crash failures** — *impossible to guarantee*

This isn't an engineering problem. It's a **mathematical impossibility**.

Paper: "Impossibility of Distributed Consensus with One Faulty Process" (JACM 1985)

FLP intuition: The “bad run”

Why is consensus impossible?

The proof constructs an adversarial execution:

- ① Start in a state where the decision could go either way
- ② For any protocol step that would decide, delay that message
- ③ The adversary can always find *one more step* to delay
- ④ The protocol never terminates (liveness violated)

Key insight: In an asynchronous system, you can't tell if a message is delayed or if the sender crashed.

The adversary exploits this: it delays the “deciding” message just long enough to keep the outcome uncertain.

FLP: What it does and doesn't say

FLP does NOT say:

- Consensus is impossible (it's possible in practice)
- Distributed systems are hopeless

FLP DOES say:

- No protocol can *guarantee* termination in all runs
- There's always a “bad run” where progress stalls
- You must weaken assumptions or accept probabilistic guarantees

Practical implication: Paxos/Raft don't violate FLP — they give up guaranteed termination in favor of *probabilistic* termination (which works in practice).

The turtle problem

Our recovery pattern:

- Single node crashes → add a log
- Nodes need to agree → add a coordinator
- Coordinator crashes → add a backup coordinator?
- Backup crashes → add a backup backup?

Turtles all the way down.

At some point, we need a fundamentally different approach.
Adding more turtles doesn't solve the problem.

The cliffhanger

What if we **replicated** the coordinator's state across multiple nodes?

- 3 nodes hold the coordinator's state
- If 1 crashes, 2 remain
- Can make progress with a majority

But wait:

- The 3 nodes need to agree on the coordinator's state
- How do they agree? They need... coordination
- We need **consensus to build consensus**

Coming soon: Paxos — consensus that actually works (mostly).

What systems actually do

Option 1: Accept blocking (2PC)

- Used in databases, distributed transactions
- Coordinator failure is rare, usually recovers quickly
- Blocking is acceptable if coordinator is reliable

Option 2: Avoid synchronous coordination

- Use asynchronous replication (faster, but weaker guarantees)
- Design around eventual consistency
- Most large-scale systems choose this

Option 3: Use Paxos/Raft for coordinator

- Replicate coordinator state with consensus
- More complex, higher latency, but non-blocking
- We'll cover this soon

Real-world: Azure September 2018

What happened:

- Lightning storm in Texas datacenter
- Cooling failed → servers overheated → hardware damaged

The trade-off they'd made:

- Geo-replication was *asynchronous* (Option 2)
- Fast in normal operation, but...
- Failover would lose recent writes

Their choice: Wait and recover data (slow) rather than fail over (fast but data loss).

Lesson: There's no free lunch. Async replication avoids 2PC blocking, but you pay during failures.

Real-world: Google Spanner

Spanner's approach:

- Uses 2PC for distributed transactions
- Coordinator is replicated using Paxos
- TrueTime provides bounded clock uncertainty

Trade-offs:

- Higher latency (Paxos rounds for every coordinator action)
- Requires special hardware (atomic clocks, GPS)
- But: globally consistent, non-blocking (under Paxos assumptions)

Key insight: Spanner doesn't avoid the turtle problem — it uses consensus (Paxos) as the bottom turtle.

Summary: The limits of coordination

- **2PC is blocking:** Coordinator failure → participants stuck
- **UNCERTAIN state:** Can't commit, can't abort, can't wait
- **3PC helps:** But requires synchrony (breaks with partitions)
- **FLP:** Consensus is impossible with crashes in async systems
- **The turtle problem:** Can't just keep adding layers
- **The solution:** Replicate the coordinator using consensus

Next: Paxos — how to actually achieve consensus.

The three-lecture arc: Complete

L06: Single-node failure → logs → recovery

L07: Multi-node coordination → 2PC → coordinator as turtle

L08: Coordinator failure → blocking → limits of turtles

Key takeaway:

The same pattern (log → recovery → consistency) applies at each layer.
But there are fundamental limits to what coordination can achieve.

For DP1:

- View Server is your coordinator
- What happens if it's unavailable?
- How does Primary/Backup handle that?

- Readings + quiz (Canvas → Readings page)
- Self-assessment opens at 13:00, due next class 11:30 (Pacific)
- DP1 deadline — after these lectures, you have the mental model

Key concepts from today:

- Coordinator failure and blocking in 2PC
- The UNCERTAIN state and why it's problematic
- 3PC as a partial solution (requires synchrony)
- FLP impossibility and the limits of turtles