

CPSC 416 – Distributed Systems

Lecture 23: CRDTs + Consensus in Different Clothes

UBC Vancouver

2026-03-31

From Last Time

Turn to your neighbor. 2 minutes.

L22 asked: Every coordination mechanism embeds a political choice—who waits, who decides, who gets to speak.

Name one mechanism we've studied and the political choice it makes.

(Pairs: 2 minutes)

Failure-first question

Coordination is expensive. It costs latency, availability, and complexity.

What if you didn't need it at all?

Is there a class of operations where replicas can act independently and still converge?

The Cost of Consensus

Every consensus protocol we've studied has the same shape:

- Contact a quorum *before* committing
- Wait for responses
- If the leader fails, recover before making progress

Cost:

- Latency: at least one round-trip per write
- Availability: need a quorum to be reachable
- Complexity: leader election, ballot ordering, log repair

What if the data type itself guaranteed convergence?

The Idea

Instead of coordinating *replicas*,
design the *data type* so that
any merge of any states converges.

No leader. No quorum. No ballots.

Every replica accepts writes locally. Replicas merge when they communicate.

Conflict-free Replicated Data Types (CRDTs).

Shapiro, Preguiça, Baquero, Zawirski (2011)

Mathematical Foundation: Join Semilattice

Join Semilattice

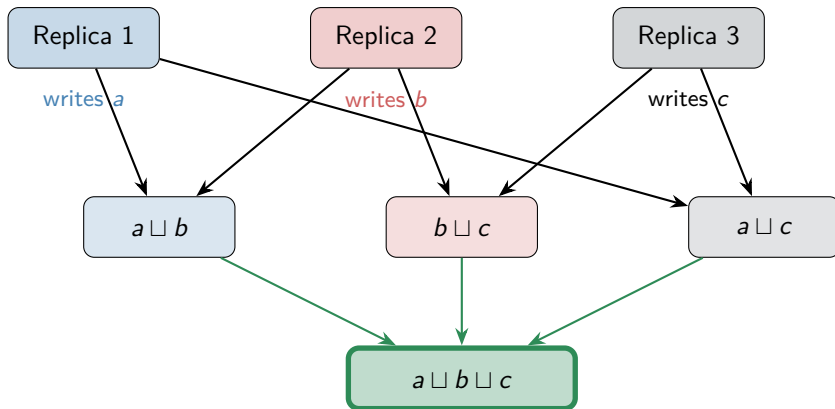
A set S with a binary operation $\sqcup$ (join) that is:

- **Commutative:** $a \sqcup b = b \sqcup a$
- **Associative:** $(a \sqcup b) \sqcup c = a \sqcup (b \sqcup c)$
- **Idempotent:** $a \sqcup a = a$

Translation: Merge order doesn't matter. Merge grouping doesn't matter. Merging with yourself changes nothing.

If your merge operation forms a join semilattice, replicas converge regardless of communication order.

Why the Lattice Matters



Different merge orders, same result.
Commutativity + associativity guarantee this.

Checkpoint 1

Articulate:

In L10, we said the quorum intersection property is what makes consensus work. CRDTs don't use quorums at all. What property replaces intersection?

(Pairs: 2 minutes)

Example 1: G-Counter (Grow-Only Counter)

Problem: Three replicas need to count events independently.

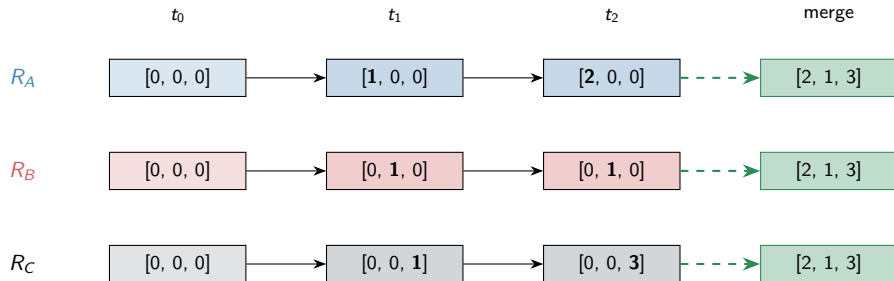
Structure: Each replica maintains a vector of counts, one entry per replica.

- **Increment at replica i :** $V[i] += 1$
- **Value:** $\sum_i V[i]$
- **Merge:** $V_{\text{merged}}[i] = \max(V_A[i], V_B[i])$ for each i

max is commutative, associative, and idempotent.

The counter can only grow. That's the constraint that makes it work.

G-Counter: Walkthrough



Value at merge: $\max(2, 0, 0) + \max(0, 1, 0) + \max(0, 0, 3) = 2 + 1 + 3 = 6$

No coordination. No leader. No lost increments.

Why G-Counter Works

The merge is max per entry.

- Each replica only increments its own entry
- max never loses an increment (monotonically non-decreasing)
- Re-merging with the same state doesn't double-count (idempotent)

The safety guarantee is embedded in the data type.

No protocol can violate it because there is no protocol. There's just a merge function with algebraic properties.

Contrast with Paxos: safety comes from quorum intersection + ballot ordering.
Here: safety comes from the lattice structure of the data.

Example 2: OR-Set (Observed-Remove Set)

Problem: Concurrent add and remove of the same element.

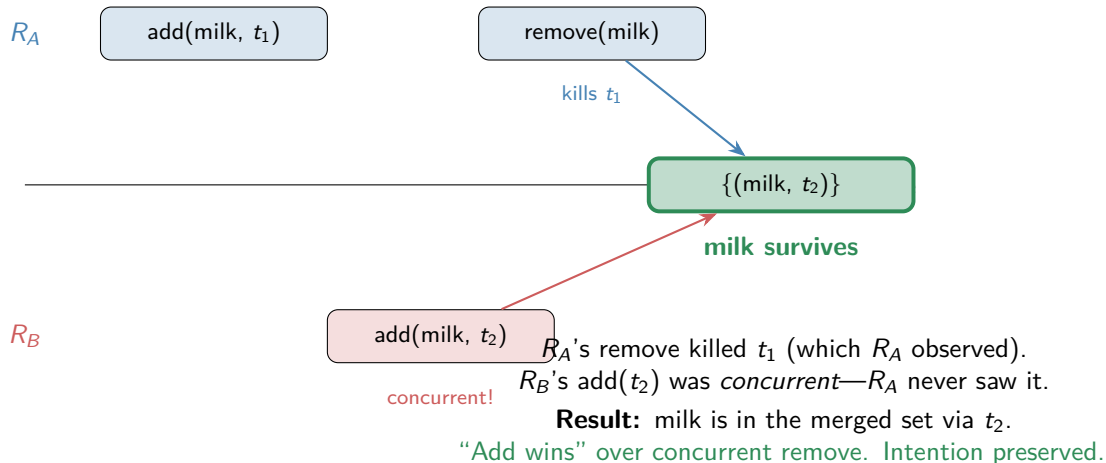
Naïve approach: “Last write wins.” But without a global clock, which write was “last”?

OR-Set approach: Tag each add with a unique identifier.

- **Add(e):** Insert $(e, \text{unique_tag})$
- **Remove(e):** Remove all (e, tag) pairs *currently observed*
- **Merge:** Union of all (e, tag) pairs, minus tags both sides removed

Semantics: A remove only kills adds it has *seen*. Concurrent adds survive.

OR-Set: Concurrent Add/Remove



Checkpoint 2

Articulate:

Why do the unique tags matter in the OR-Set?

What goes wrong if you just track {element} without tags?

(Pairs: 2 minutes)

What CRDTs Buy You

- **Availability:** Every replica can accept writes, always
- **Latency:** Local writes, no round-trips
- **Partition tolerance:** Replicas can diverge and reconverge
- **Simplicity:** No leader election, no ballot ordering, no log repair

This is **Strong Eventual Consistency (SEC)**:

Replicas that have received the same set of updates have the same state—regardless of order.

What CRDTs Cost You

- **Expressiveness:** Not every operation is CRDT-friendly
- **Metadata overhead:** Tags, vectors, tombstones grow over time
- **No total ordering:** You get convergence, not sequencing
- **No “strong” consistency:** Reads can see different states at different replicas

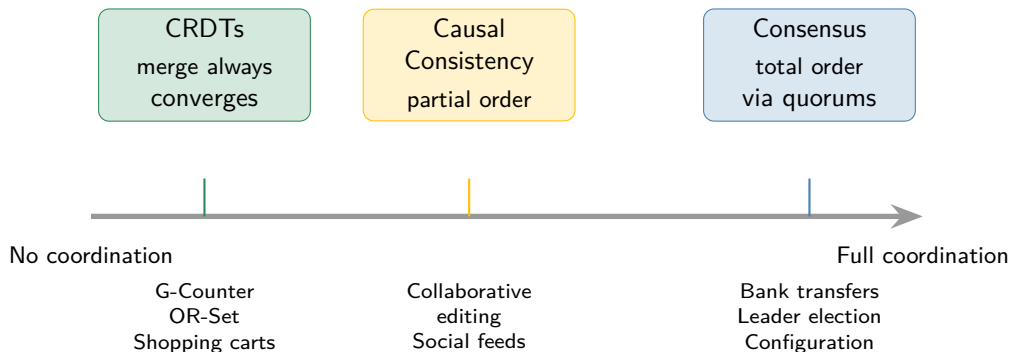
What you can't build with CRDTs:

- A bank account with a non-negative balance invariant
- A unique username registry
- Anything that requires “check-then-act” atomically

If your safety property requires global knowledge, you need consensus.

If your safety property is local and monotonic, CRDTs may suffice.

The Design Spectrum



In-Class Activity: Design a CRDT Shopping List

Setup: Multiple users share a shopping list. They can add and remove items concurrently from different devices.

Questions:

- 1 What merge operation makes the list converge?
- 2 User A adds “eggs.” User B removes “eggs” concurrently. What should happen?
- 3 User A adds “milk (2L).” User B adds “milk (1L).” How do you handle this?
- 4 Can you support “buy exactly 3 of item X”? Why or why not?

(Groups of 3: 10 minutes. Then present.)

Activity Debrief

- (1) **Merge:** OR-Set semantics. Each add tagged uniquely. Remove kills observed tags.
- (2) **Concurrent add/remove:** “Add wins”—the concurrent add survives because the remover never saw it. Same as OR-Set.
- (3) **Variant items:** Treat “milk (2L)” and “milk (1L)” as different elements? Or merge quantities? **Design choice—no single right answer.**
- (4) **Exact quantities:** **No.** “Buy exactly 3” requires a check (“how many so far?”) before acting. That’s a global invariant. CRDTs can’t enforce it without coordination.

The boundary of CRDT expressiveness is exactly the boundary where you need consensus.

Part 2: Consensus in Different Clothes

After five lectures of Paxos (L10–L14),
you already understand Raft.

You just don't know it yet.

The Same State Machine

Every practical consensus system has the same core:

- ① **Leader election:** One node gets to propose
- ② **Log replication:** Leader sends entries to followers
- ③ **View/term changes:** When the leader fails, pick a new one

Paxos, Raft, Viewstamped Replication (VR), Zab (ZooKeeper)—all implement this.

The differences are real but secondary:

naming, leader constraints, log completeness invariants, membership changes.

The Rosetta Stone

Concept	Paxos	Raft	VR	Zab
Epoch ID	Ballot	Term	View #	Epoch
Who proposes	Proposer	Leader	Primary	Leader
Who stores	Acceptor	Server	Replica	Follower
Leader election	Phase 1	RequestVote	View change	Recovery
Normal operation	Phase 2	AppendEntries	Normal op	Broadcast
Safety invariant	Chosen values are never overwritten			
Quorum	Majority (or flexible, per L14)			

Different names. Same structural roles.

Paxos Phase 1 = Raft Election

Paxos Phase 1:

- Proposer picks ballot b , sends $\text{PREPARE}(b)$
- Acceptors promise not to accept lower ballots
- Proposer learns what was previously accepted

Raft Election:

- Candidate picks term t , sends $\text{RequestVote}(t)$
- Servers grant vote if $t >$ their current term
- Candidate wins if it gets a majority of votes

The structural difference:

Raft adds a constraint: servers only vote for candidates whose log is at least as up-to-date as their own.

This eliminates the need for Paxos's “adopt the highest-ballot value” rule. The leader already has it.

Paxos Phase 2 = Raft AppendEntries

Paxos Phase 2:

- Proposer sends $\text{ACCEPT}(b, v)$ to a quorum
- Acceptors accept if $b \geq$ their highest promise
- Value is chosen when a quorum accepts

Raft AppendEntries:

- Leader sends $\text{AppendEntries}(\text{term}, \text{entries})$ to followers
- Followers append if $\text{term} \geq$ their current term
- Entry is committed when replicated to a majority

Same mechanism: Leader broadcasts, quorum confirms, entry is durable.

The ballot IS the term. The accept IS the append. The quorum IS the majority.

Checkpoint 3

Articulate:

In Paxos, a new proposer must adopt the highest-ballot value it discovers in Phase 1.

In Raft, a new leader doesn't need this step. Why not?

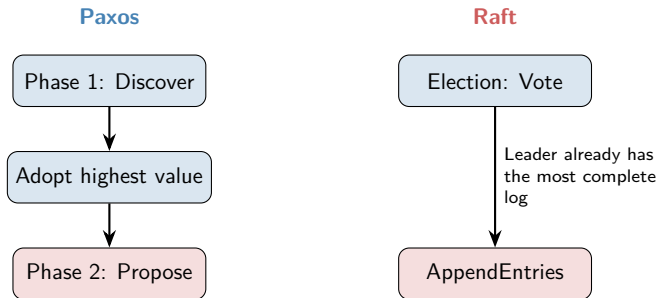
Hint: think about the election constraint.

(Pairs: 2 minutes)

The Raft Simplification

Paxos says: Any proposer can propose. Phase 1 discovers prior state.

Raft says: Only a candidate with the most complete log can win.



Same safety. Different mechanism. Raft front-loads the work into election.

Viewstamped Replication

Oki & Liskov (1988) — predates Paxos by a year!

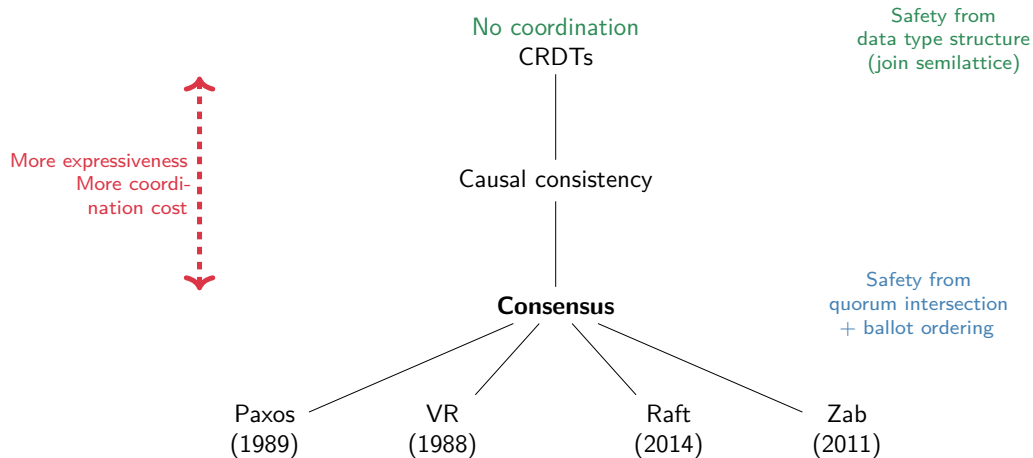
- **View:** An epoch with a designated primary
- **Normal operation:** Primary broadcasts operations, replicas execute in order
- **View change:** When primary is suspected failed, replicas elect a new primary and reconcile logs

The view change protocol is structurally Phase 1: the new primary discovers what the old primary might have committed.

VR was designed for state machine replication from the start. Paxos was designed as single-decree and extended. Raft was designed to be understandable.

Same problem. Different starting assumptions. Isomorphic solutions.

The Design Lattice (Revisited from L10)



Where CRDTs and Consensus Meet

Real systems use both.

Example: a collaborative editor

- Text editing: CRDT (character-by-character merge)
- File permissions: consensus (who can edit?)
- User presence: CRDT (heartbeat counters)
- Account management: consensus (unique usernames)

Example: a distributed database

- Session data, shopping carts: CRDTs (availability wins)
- Account balances, inventory: consensus (correctness wins)

The question is never “CRDTs vs consensus.” It’s “which operations need which guarantees?”

Evidence Question

You're building a system with both CRDTs and consensus-replicated state.

A developer proposes using a CRDT for a “likes” counter on posts, and consensus for the post content itself.

Is this the right split?

What failure would reveal a problem with this design?

(Consider: what happens if a post is deleted via consensus, but like-count updates are still merging via CRDT?)

What We Learned

- **CRDTs** embed safety in the data type via join semilattice properties
- **Commutativity + associativity + idempotency** = convergence without coordination
- G-Counter: max per entry. OR-Set: unique tags resolve concurrent add/remove.
- CRDTs trade expressiveness for availability—no global invariants

- **Raft, VR, Zab** are consensus in different clothes
- Paxos ballot = Raft term = VR view number
- Paxos Phase 1 = leader election. Phase 2 = normal operation.
- Raft front-loads log completeness into the election step

The lattice from L10:

- CRDTs use the lattice (join semilattice) *directly* as the safety mechanism
- Consensus protocols use quorum intersection as the safety mechanism
- Both are paths through the same design space

- **Reading:** Shapiro et al., “A comprehensive study of CRDTs” (2011)—Sections 1–3
- **Optional:** Ongaro & Ousterhout, “In Search of an Understandable Consensus Algorithm” (Raft, 2014)—skim
- **Self-assessment:** Opens 13:00, due next class 11:30 (Pacific)
- **Next lecture (L24):** Course synthesis
- Capstone milestone details on Canvas