

17-350 / 17-650

Lecture 10: Eventual Consistency and CRDTs

Jonathan Aldrich

Sep 29, 2026

Lecture Plan

- First, finishing Lecture 09: `select!`, cancellation, and actors
- When replicas **disagree**: the consistency spectrum
- What consensus buys, and what it costs
- CRDTs: merges that **always converge**
- Counters, registers (and Lamport clocks), version vectors, sets
- In Rust: `trait Lattice`, generic composition, and HW4 Part 1

HW3 is due today. HW4 is released today and due Oct 20. Today's material is HW4 Part 1.

Finishing Lecture 09: Tokio's Channels

Channel	Shape	Role in HW4
<code>mpsc</code>	many senders → one receiver, a queue	an actor's mailbox ; local edits arrive this way
<code>oneshot</code>	exactly one value, once	the reply to a request
<code>watch</code>	many readers see the latest value only	publishing replica snapshots
<code>broadcast</code>	every message to every receiver	(not needed)

- `send(...).await` on a bounded `mpsc` waits when the channel is full (backpressure); `try_send` never waits
- A `recv()` returning `None` means every sender is gone, so shut down
- `watch` : the writer calls `send_replace(v)` ; readers call `borrow()` , or `changed().await` to wait for a new value
 - `borrow()` holds a lock, like a `MutexGuard` . Clone what you need and **don't hold it across an `.await`**
 - In a spawned task the compiler rejects it (not `Send`); anywhere else (`main` , the HW4 simulator's hosts) it compiles and **can deadlock**

Timers

```
// Pause this task (not the thread) for 50 ms
tokio::time::sleep(Duration::from_millis(50)).await;

// A periodic tick, e.g., "gossip every 100 ms"
let mut gossip = tokio::time::interval(Duration::from_millis(100));
gossip.tick().await; // the first tick completes immediately

// Give up on a slow peer
match tokio::time::timeout(Duration::from_millis(500), TcpStream::connect(addr)).await {
    Ok(Ok(stream)) => { /* connected */ }
    Ok(Err(e))      => { /* connection failed */ }
    Err(_elapsed)  => { /* took longer than 500 ms */ }
}
```

- `timeout` is a future holding **two** futures, yours and a `sleep`, and it polls **both** each time it's polled: whichever finishes first wins, and the other is dropped. (`select!`, `next`, generalizes this)
- To bound a whole multi-step exchange (connect, send, await the reply), wrap it in one async block: `timeout(d, async { ... }).await`

`select!`: Wait for Whichever Comes First

The main loop of a replica (from the `gossip-toy` demo, abbreviated):

```
let mut sensor = interval(Duration::from_millis(300));
let mut gossip = interval(Duration::from_millis(200));
loop {
    tokio::select! {
        _ = sensor.tick(), if readings_left > 0 => { /* take a local reading */ }
        Some(msg) = inbox.recv() => { /* merge a peer's value */ }
        _ = gossip.tick() => { /* send my value to a peer */ }
        _ = &mut shutdown => break,
    }
}
```

- Polls all branch futures; the **first to complete** wins, and its body runs
- Each pass makes **new** branch futures. The state lives in `inbox`, `sensor`, and `gossip`, outside the loop
- This is the shape of HW4 Part 3's node: local edits, peer messages, a gossip timer

`select!` Details, and Cancellation Is `drop`

- `, if cond` is a **precondition**: when false, that branch is disabled
- **Pattern mismatch disables the branch**: `Some(msg) = inbox.recv()` ignores `None` (the channel closed)
- If several branches are ready, tokio picks one **at random**; `biased`; checks them in order instead (remember "random" on Thursday)
 - The other ready branch isn't lost: `tick()` and `recv()` are cancel-safe, so it wins on the next pass
- Bodies run **one at a time**: keep them short, and don't await slow things there
- The losing branch futures are **dropped**, and that *is* cancellation: the future stops at its last `.await`, and destructors run
- The same happens when a `timeout` expires. Careful: dropping a `JoinHandle` does **not** cancel a task (use `abort()`)

Cancel Safety

A future is **cancel-safe** if dropping it mid-way loses nothing.

```
// NOT cancel-safe as a select! branch:
async fn read_msg(s: &mut TcpStream) -> io::Result<Vec<u8>> {
    let len = s.read_u32().await?;           // if cancelled after this...
    let mut buf = vec![0; len as usize];
    s.read_exact(&mut buf).await?;          // ...the length prefix is gone, and the
    Ok(buf)                                 // stream is in the wrong position to restart
}
```

- `mpsc::Receiver::recv` is cancel-safe: either you got the message, or it's still in the channel
- Rule: before putting a future in a `select!` branch, check the "Cancel safety" section of its tokio docs
- For network messages, use a **framed stream**, whose `next()` is cancel-safe

The Actor Pattern - Another Counter Implementation

```
enum Command {
    Increment { key: String },
    Get { key: String, reply: oneshot::Sender<u64> },
}

fn spawn_counter_actor() -> mpsc::Sender<Command> {
    let (tx, mut rx) = mpsc::channel::<Command>(32);
    tokio::spawn(async move {
        let mut counts = Counts::new(); // owned by this task alone
        while let Some(cmd) = rx.recv().await {
            match cmd {
                Command::Increment { key } => *counts.entry(key).or_default() += 1,
                Command::Get { key, reply } =>
                    { let _ = reply.send(counts.get(&key).copied().unwrap_or(0)); }
            }
        }
        // all senders gone: exit
    });
    tx
}
```

- One task **owns** the state; everyone else sends it messages
- A request that needs an answer carries a `oneshot::Sender` for the reply

Actor vs. Shared State

	Actor (one owner task)	<code>Arc<Mutex<State>></code>
Locks	none	every access
Held across <code>.await</code> ?	impossible	must be scoped carefully
Where invariants are maintained	one place: the actor loop	every task that locks
Cost	a message per operation	contention on the lock

- Lecture 08 made the module the **trusted computing base** for an invariant; here the actor task plays the same role
- HW4's replica has an invariant about its operation log. With an actor, it's maintained in exactly one loop
- HW4 lets you choose, and asks you to justify the choice

Networking: Listen, Accept, Forward

```
async fn accept_loop(listener: TcpListener, mailbox: mpsc::Sender<Gossip>) {
    loop {
        let Ok((stream, _peer)) = listener.accept().await else { continue };
        let mailbox = mailbox.clone();
        tokio::spawn(async move { // one task per connection
            let mut frames = FramedRead::new(stream, LengthDelimitedCodec::new());
            while let Some(Ok(frame)) = frames.next().await {
                match serde_json::from_slice::<Gossip>(&frame) {
                    Ok(msg) => if mailbox.send(msg).await.is_err() { break },
                    Err(e)  => eprintln!("dropping malformed frame: {e}"),
                }
            }
        });
    }
}
```

- The actor **never touches a socket**: handler tasks decode messages and forward them into its mailbox
- Sending: `TcpStream::connect(addr)`, done in its own spawned task, never inside the actor loop

TCP Is a Byte Stream: Frame Your Messages

- TCP delivers **bytes**, not **messages**: one write may arrive as two reads, and two writes may arrive as one read
- Fix: **length-delimited framing**, a 4-byte length followed by that many bytes

```
#[derive(Serialize, Deserialize)]
struct Gossip { from: usize, max_seen: u64 }

let mut frames = FramedWrite::new(stream, LengthDelimitedCodec::new());
frames.send(Bytes::from(serde_json::to_vec(&msg)?)).await?;
```

```
wire format: [u32 length = 25] {"from":0,"max_seen":123}
```

- `serde` generates the (de)serialization; `serde_json` because you can **read** the messages when debugging
- HW4 provides these helpers, so you won't write marshalling code by hand

Demo: gossip-toy

Three replicas, each taking 5 sensor readings and gossiping the **largest value seen**; the merge is `max`.

```
node 2: local reading 955 (new max)
node 1: learned max 955 from node 2
node 0: learned max 955 from node 1

final values: [955, 955, 955]
all 3 replicas converged to 955
```

- The same actor code runs over in-memory channels and over TCP
- Duplicated gossip is harmless: `max(a, a) = a`. Lost gossip just costs a round: every message carries the full state, so the next one makes up for it
- **Question:** what if we wanted the **total number** of readings instead? Would `+` work as the merge?

[Live demo] `cargo run -p gossip-toy`, then `cargo run -p gossip-toy --bin tcp`

Replicas That Disagree

- Why keep several copies of the same data?
 - **Offline devices:** a phone on a plane, a laptop on bad Wi-Fi
("local-first" software: your data lives on your device and syncs when it can)
 - **Latency:** a replica near every user, in every region
 - **Availability:** keep working when a machine or a link fails
- Alice (on a plane) and Bob both edit the same shopping list. When Alice lands, what should the list contain?

Strategy	How	Today
Prevent disagreement	coordinate <i>before</i> every write	consensus (briefly)
Allow it, then merge	accept writes locally; merge deterministically <i>afterward</i>	CRDTs (most of today)

The Consistency Spectrum

Model	Guarantee	Example use	Coordination per write?
Linearizable	every operation takes effect at one instant; everyone sees one up-to-date copy	a lock service, "is this username free?"	yes: a majority round trip
Causal	if you saw X before writing Y, everyone sees X before Y	a reply never appears before its post	no, but track dependencies
Eventual	if writes stop, replicas <i>eventually</i> agree	DNS caches (stale until the TTL expires); git (clones converge after someone merges)	no
Strong eventual	replicas that have seen the same updates are in the same state	CRDTs, HW4	no

- Up the table: easier to program against, slower, and less available
- Down the table: fast and always available, but the application must tolerate disagreement

CAP and PACELC, in Brief

- **CAP** (Brewer 2000; proved by Gilbert and Lynch 2002): during a network Partition, a replicated system must give up either Consistency (linearizability) or **A**ailability (every replica that's up still answers)
 - Partitions happen whether you like it or not, so it's really "C or A, *when P*"
- **PACELC** (Abadi 2012): if **P**artitioned, choose **A** or **C**; Else, choose **L**atency or **C**onsistency
 - Even on a healthy network, strong consistency costs round trips on every write
- CRDTs choose **A** and **L**: every replica accepts writes locally, immediately, even offline. The price is a weaker guarantee

What Consensus Buys

- **Consensus:** replicas agree on a value; in practice, on **one total order** of operations (a replicated log)
 - Every replica applies the same operations in the same order: a *replicated state machine*
- That gives linearizability, and with it operations that **inspect, then act**:
 - compare-and-swap: "set `k` to `v` only if it's still `u` "
 - uniqueness: "claim the username `alice` only if no one has"
 - invariants over the whole state: "withdraw only if the balance covers it"
- Used for configuration and coordination (etcd, ZooKeeper), and for replication in databases such as Spanner (Paxos), CockroachDB, and TiKV (Raft)

If you've taken 15-440, this is a quick recap of Paxos.

Raft in One Slide

- Replicas elect a **leader** for a numbered **term** (randomized election timeouts make split votes rare)
- Clients send writes to the leader, which appends them to its **log** and replicates them to the followers
- An entry is **committed** once a **majority** has stored it; committed entries are applied, in log order, on every replica
- Any two majorities **intersect**, so any new leader has every committed entry
- 5 replicas tolerate 2 failures, as long as 3 can talk to each other

The whole protocol exists to make every replica apply the same operations in the same order.

What Consensus Costs

- Every write waits until a **majority has stored it**: a round trip from the leader (across regions: tens to hundreds of ms)
- A replica that can't reach a majority **can't accept writes**: the phone on the plane, the minority side of a partition (CAP's C over A)
- The leader is a throughput bottleneck, and failing over takes time
- **FLP** (Fischer, Lynch, Paterson 1985): in a fully asynchronous system where even one process may crash, no deterministic protocol can guarantee consensus terminates
 - Real protocols use timeouts: always **safe**, but **live** only when the network behaves

What if replicas never had to agree *before* acting?

Strong Eventual Consistency

- Each replica applies local updates **immediately**, and exchanges state with peers whenever it can: in any order, any number of times
- **Strong eventual consistency (SEC)** guarantees:
 - **Eventual delivery**: every update eventually reaches every replica
 - **Strong convergence**: replicas that have received the **same set of updates** are in the **same state**
- No coordination, no rollback, no "conflict resolution" round
- A **CRDT** (Conflict-free Replicated Data Type) is a data type designed to give SEC
- "The same set of updates, received in any order, possibly more than once": what properties does a merge operator need to deal with this?

Shapiro, Preguiça, Baquero, and Zawirski, "Conflict-free Replicated Data Types," 2011.

Back to the Question: Why Not $+$?

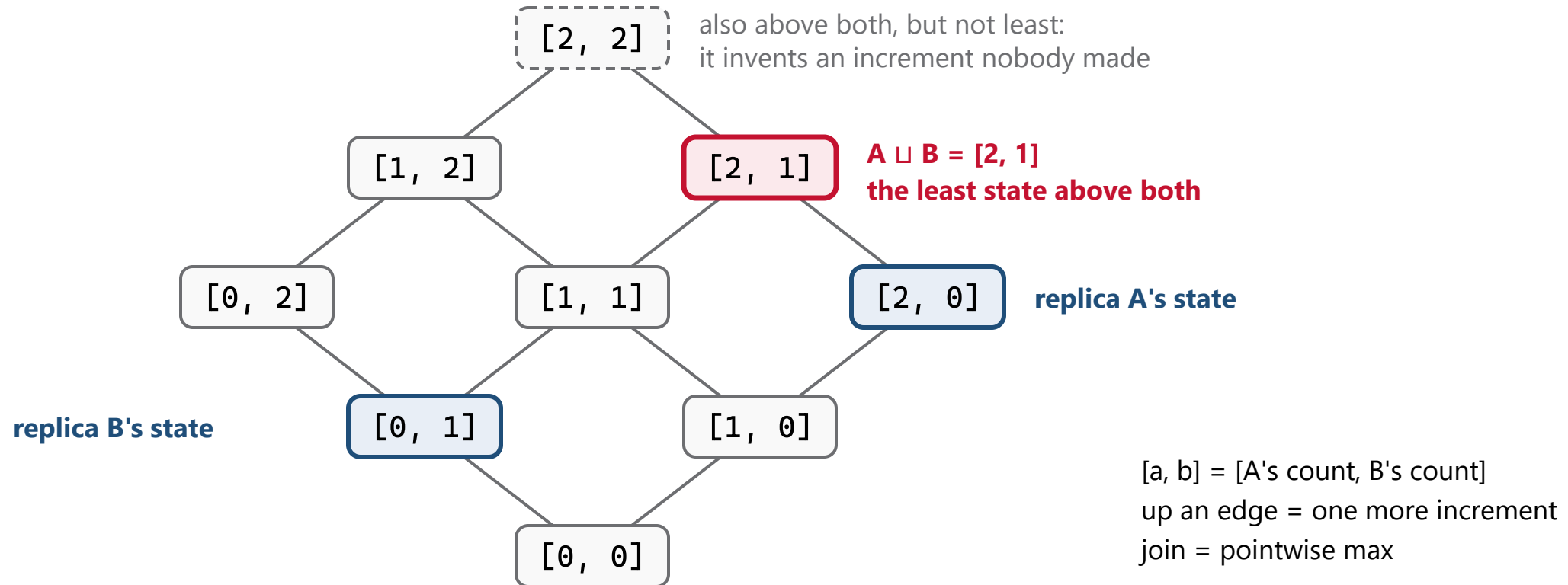
The network may...	So the merge $\sqcup$ must be...	Law
deliver states in a different order	order-insensitive	commutative: $a \sqcup b = b \sqcup a$
combine them in a different grouping	grouping-insensitive	associative: $(a \sqcup b) \sqcup c = a \sqcup (b \sqcup c)$
deliver a duplicate, or send your own state back	duplicate-insensitive	idempotent: $a \sqcup a = a$

- $\max$ satisfies all three
- $+$ is commutative and associative, but $1 + 1 \neq 1$
 - Every redundant gossip message counts the same readings again

Join-Semilattices

- A set of states with a merge $\sqcup$ (**join**) that is commutative, associative, and idempotent is a **join-semilattice**
- The join defines an **order**: $a \leq b$ iff $a \sqcup b = b$
 - read it as " b already knows everything a knows"
- Then $a \sqcup b$ is the **least upper bound**: the smallest state that knows everything either input knows
- Local updates must be **inflations**: they move the state up
($s \leq \text{update}(s)$)
 - Information is never lost; merging only moves up; replicas that have seen the same updates meet at the same point
- Examples: numbers with $\max$, sets with $\cup$, booleans with $\vee$, and pointwise combinations of these

Picture: A Two-Replica G-Counter



The G-Counter (Grow-Only Counter)

```
/// One count per replica; each replica increments only its own entry.
pub struct GCounter(BTreeMap<ReplicaId, u64>);

impl GCounter {
    pub fn increment(&mut self, me: ReplicaId) { *self.0.entry(me).or_insert(0) += 1; }
    pub fn value(&self) -> u64 { self.0.values().sum() }
}

impl Lattice for GCounter {
    fn join(&mut self, other: &Self) {
        for (&r, &n) in &other.0 {
            let mine = self.0.entry(r).or_insert(0);
            *mine = (*mine).max(n); // pointwise max
        }
    }
}
```

- Only replica `r` writes entry `r`, and it only grows. So the max of entry `r` is simply the most recent count from `r` that anyone has seen
- The `+` happens once, at read time (`value`), never in the merge

Demo: merge-toy, Counting

A takes 3 readings, B takes 2, C takes 1: the true total is 6. Same readings, same gossip schedule ($A \rightarrow B$, $B \rightarrow C$, $C \rightarrow A$, $A \rightarrow B$, $B \rightarrow A$, $A \rightarrow C$), three merges:

merge	A	B	C	
+ on totals	23	14	29	replicas disagree
max on totals	3	3	3	agree, but wrong
G-Counter	6	6	6	agree, and right

- `+` isn't idempotent: the answer depends on how much gossip happened
- `max` on totals is a lattice, so it converges, but to the wrong answer: it keeps only the biggest contribution
- Converging is necessary, **not** sufficient: the lattice must also mean what you want

[Live demo] in 10-crdts/demos/: `cargo run -p merge-toy`

The PN-Counter (Positive-Negative Counter): Adding Decrement

- Why not let a replica decrement its own G-Counter entry?
 - Then an entry can go **down**, and `max` with an older state brings the old value back: the decrement is silently undone
 - A decrement is not an **inflation**
- Fix: **two** G-Counters, one counting increments and one counting decrements

```
pub struct PNCounter { incs: GCounter, decs: GCounter }  
// value = incs.value() - decs.value(); join = join each half
```

- Both halves only grow, and the value is computed at read time
- A **pair of lattices**, joined componentwise, is a lattice: our first example of **composition**

Registers: Who Wrote Last?

- A **register** holds one value. When two writes compete, one must win
- **Last-writer-wins (LWW)**: stamp every write; the join keeps the value with the **larger stamp**
 - It's a lattice (it's `max` on stamps), as long as stamps are totally ordered and unique
- The obvious stamp is the **wall clock**. Alice writes `"draft"` ; Bob sees it, then writes `"final"` . Bob's clock is 5 s slow:

```
wall-clock stamps: draft = (100000, 0), final = (97000, 1)
after sync: alice = "draft", bob = "draft"    <- Bob's later write is lost
```

- Clocks drift, NTP sometimes corrects them with a jump (even backward), VMs pause, and people set them wrong. Time on different machines isn't comparable enough to order events

Lamport Clocks

- **Happened-before** (Lamport 1978): $a \rightarrow b$ if a came before b on the same replica, or a was a send and b its receipt, or by transitivity. Otherwise a and b are **concurrent**
- **The rule:** each replica keeps a counter c
 - when it creates an event (a write): $c = c + 1$, and stamps it $(c, \text{my replica id})$
 - when it receives a stamp $(t, _)$: $c = \max(c, t)$
 - in short: **stamp = (max seen + 1, replica id)**
- **Guarantee:** $a \rightarrow b$ implies $\text{stamp}(a) < \text{stamp}(b)$
 - Not the converse: a smaller stamp might belong to a *concurrent* event
- Ties can be broken by replica id, which makes stamps unique and **totally ordered**, consistently with causality
- In HW4: `LwwRegister`'s stamps, and on Thursday, each character in a shared text document gets a Lamport stamp as its unique id

LWW with Lamport Stamps

A stamp is a pair `(counter, replica id)`. Pairs compare **counter first**; the replica id matters only when the counters are equal.

Scenario	Alice's write	Bob's write	Decided by	Both replicas end with
Bob saw "draft", then wrote	"draft" at <code>(1, 0)</code>	"final" at <code>(2, 1)</code>	counter: $2 > 1$	"final"
Neither saw the other	"red" at <code>(1, 0)</code>	"blue" at <code>(1, 1)</code>	counters tie at 1, so replica id: $1 > 0$	"blue"

- A write that **causally follows** another has a larger counter, so it always wins (row 1)
- **Concurrent** writes can have equal counters. Then the replica id picks a winner: arbitrary, but the same everywhere, so replicas converge (row 2)
 - The loser, "red", is **silently discarded**. Fine for a document title, not always though!
 - The alternative is a data type whose concurrent updates compose: counters, sets, text
- HW4's `LwwRegister` keeps its Lamport counter **per register**: a write's stamp is one more than the register's current stamp. That's enough, since only writes to the same register compete

Dots and Version Vectors

- A **dot** (r, n) : the n -th event originated by replica r
 - a unique name for an event, created with **no coordination**
- A **version vector**: for each replica, the highest sequence number seen from it, e.g., $\{A: 3, B: 1\}$
 - $\text{contains}(r, n)$ iff $n \leq vv[r]$
 - join: **pointwise max** (structurally, a G-Counter)
 - order: $v \leq w$ iff every entry of v is $\leq$. If neither $v \leq w$ nor $w \leq v$, they are **concurrent**: $\{A: 2, B: 1\}$ vs. $\{A: 1, B: 2\}$
- One number per replica summarizes **everything I've seen**, provided each replica's events are seen **in order** (Thursday: HW4's log guarantees this)
- In HW4: Part 1 implements `VersionVector`; the OR-Set uses dots as tags; Part 3's sync protocol exchanges version vectors

Sets: Adding Is Easy, Removing Is Hard

- **Grow-only set:** join is $\cup$. A lattice, but there's no remove
- Remove by deleting the element? Merging with an older state that still has it brings it back ($\cup$ resurrects it). Deletion isn't an inflation
- **Two-phase set:** a grow-only "added" set and a grow-only "removed" set (tombstones). It works, but a removed element can **never** be added back
- **An LWW flag per element, "present?".** Alice removes milk while Bob concurrently adds it again:

```
LWW flag per element, Alice = replica 0, Bob = replica 1: milk present = true  
LWW flag per element, Alice = replica 1, Bob = replica 0: milk present = false
```

- In this example, the outcome depends on who has the larger replica id, not on what anyone meant

The OR-Set (Observed-Remove, Add-Wins)

- Every `add` creates a fresh, unique **tag**: a dot `(replica, seq)`
- `remove(x)` **tombstones** the tags of `x` this replica has observed
- `x` is present iff it has at least one **live** (untombstoned) tag
- Join: union the tags, union the tombstones

Step	Alice	Bob
Alice adds milk	milk@A1	
sync	milk@A1	milk@A1
concurrently	remove milk: tombstone A1	add milk again: milk@B1
sync	live: milk@B1; tombstones: A1	the same

- A remove can only kill adds it has **seen**. A concurrent add made a tag it couldn't have seen, so **add wins**, whatever the replica ids

merge-toy prints this state, and checks that a remove that did observe the add sticks.

Composition: A Map of Lattices Is a Lattice

- `Map<K, V: Lattice>` : the join takes the union of the keys, and joins the values of keys present on both sides
 - the laws hold for the map because they hold for each value
- HW4's key-value store is one line:

```
pub type KvStore = Map<String, LwwRegister<String>>;
```

- More examples: `Map<String, PNCounter>` (likes per post), `Map<String, OrSet<String>>` (tags per document), maps of maps, ...
- A version vector is `Map<ReplicaId, max-of-u64>`; a PN-Counter is a pair of G-Counters. The big CRDTs are built from small ones
- **No key deletion:** removing a key has the set-removal problem (it needs "OR-Map" semantics; out of scope)

State-Based, Op-Based, and Delta-State

	State-based	Op-based	Delta-state
Sends	the whole state	each operation	a small piece of state
Applied by	<code>join</code>	<code>apply(op)</code>	<code>join</code>
Network must provide	nothing: any order, duplicates OK	exactly-once, causal-order delivery	nothing for convergence
Message size	grows with the state	small	small
G-Counter example	<code>{A: 4, B: 7, C: 2}</code>	"A incremented"	<code>{A: 4}</code>

- **Op vs. delta:** an op describes the *change* ("A incremented"); a delta is the *new value* of the part that changed (`{A: 4}`)
- HW4 uses all three: Part 1's lattices are state-based, the key-value store sends deltas, and the text CRDT (Part 2) is op-based

Delta-State CRDTs in Detail

- Same lattice and same `join` as state-based. Each update `m` gets a **delta-mutator** `mδ` returning a small piece of state, chosen so that $m(X) = X \sqcup m\delta(X)$
- The (per-replica) G-Counter `{A: 3, B: 7, C: 2}`, and A increments:
 - full state: `{A: 4, B: 7, C: 2}`
 - op "A: +1": applied twice gives A = 5, so it must arrive **exactly once**
 - delta `{A: 4}`: only the entry that changed (missing entries mean 0). Joined twice, or out of order, it still gives A = 4, so **duplicates and reordering are harmless**

```
impl DeltaSource for KvStore {           // your one delta-mutator in Part 1
    type Edit = KvEdit;                  // { key, value }
    fn delta_for(&self, edit: KvEdit, replica: ReplicaId) -> Self {
        // a one-entry map: edit.key -> its register, re-stamped newer than now
    }
}
```

- **The rule:** `self` $\sqcup$ `self.delta_for(e, r)` must equal the state after performing `e`, and the delta should hold only what changed
- In HW4, deltas ride the causal log as ops (the `Delta<L>` adapter), so one `Replica` carries both key-value deltas and text operations

In Rust: `trait Lattice`

```
/// A join-semilattice: a state-based CRDT.
/// `join` must be commutative, associative, and idempotent.
pub trait Lattice: Clone + PartialEq + Debug {
    fn join(&mut self, other: &Self); // self = self  $\sqcup$  other

    fn joined(mut self, other: &Self) -> Self where Self: Sized {
        self.join(other);
        self
    }

    /// The lattice order: self  $\leq$  other iff self  $\sqcup$  other == other.
    fn leq(&self, other: &Self) -> bool { &self.clone().joined(other) == other }
}

impl<K: Ord + Clone + Debug, V: Lattice> Lattice for Map<K, V> {
    fn join(&mut self, other: &Self) { /* pointwise: your job in Part 1 */ }
}
```

- The laws live in a **doc comment**: the type system can't check them. Property tests will
- **One generic impl** covers every map of lattices (Lecture 06), including maps of maps
- `PartialEq` is in the supertraits so that `a  $\sqcup$  a == a` can be tested

Canonical Representations, and No `HashMap`

- Property tests compare states with `==`, so two states that mean the same thing must be `==`
 - Version vector: `{A: 0}` and `{}` mean the same thing, so store no zero entries
 - OR-Set: keep each tag **either** live **or** tombstoned, never both
 - Otherwise a correct join can "fail" a law, and a wrong one can pass
- **Determinism rule:** CRDT state uses `BTreeMap` / `BTreeSet`, not `HashMap` / `HashSet`
 - `HashMap`'s iteration order depends on a random per-process seed (`RandomState`), so anything that iterates (serializing, picking "the first", tie-breaking) can differ between runs and between replicas
 - Thursday: why this matters for deterministic testing

HW4 Part 1 at a Glance (~5 hours)

You implement	Key idea
<code>VersionVector</code> (<code>lattice.rs</code>)	pointwise max; <code>contains(dot)</code> ; no zero entries
<code>GCounter</code> , <code>PNCounter</code> (<code>crdts.rs</code>)	a PN-Counter is two G-Counters
<code>LwwRegister<T></code>	Lamport stamps, ties broken by replica id
<code>OrSet<T></code>	tags are dots; tombstones; add-wins
<code>Map<K, V: Lattice></code>	pointwise join; no deletion
<code>KvStore</code> 's <code>delta_for</code>	a write becomes a one-entry map
properties (<code>properties.rs</code>)	the lattice laws, plus semantic properties

- **Provided:** the traits (`Lattice` , `Counter` , `SetOps` , `DeltaSource`), `Dot` , and proptest **generators** that build random states through each type's public API, so your internal representation is up to you

Property Testing Idempotence

A **property** is a claim about *every* input, here $a \sqcup a = a$. A **property test** checks it on many **random** inputs. The starter code provides this:

```
/// A proptest runner: 256 random cases, generated from a fixed seed
fn runner() -> TestRunner {
    let config = Config { cases: 256, failure_persistence: None, ..Config::default() };
    TestRunner::new_with_rng(config, TestRng::deterministic_rng(RngAlgorithm::ChaCha))
}

/// a  $\sqcup$  a = a, for ANY lattice type L
pub fn join_is_idempotent<L: Lattice + Arbitrary>() -> Result<(), String> {
    runner()
        .run(&any::<L>(), |a| {
            prop_assert_eq!(a.clone().joined(&a), a); // for each random `a: L` ...
            Ok(()) // ... check the law
        })
        .map_err(|e| e.to_string()) // failure: a message with a minimal input
}
```

- `runner()` makes a proptest **test runner** that tries **256** inputs, all from a **fixed seed**: every run tests the same inputs, so a failure reproduces. (`failure_persistence: None` : don't save failures to a file)
- `.run(generator, check) : any::<L>()` generates random `L` s using the provided `Arbitrary` impls; the closure checks one; `prop_assert_eq!` fails it. On a failure, proptest **shrinks** the input to a minimal one
- **Generic over `L`**, so the whole test is `join_is_idempotent::<GCounter>().unwrap()`. You write the other laws the same way

Watching It Catch a Bug

A `SumCounter` whose join does `+=` instead of `max`:

```
join_is_idempotent::<GCounter>(): ok (256 cases)
join_is_commutative::<SumCounter>(): ok (256 cases)

join_is_idempotent::<SumCounter>(): FAILED
Test failed: assertion failed: `(left == right)`
  left: `SumCounter({0: 2})`,
  right: `SumCounter({0: 1})` at law-check\src\main.rs:84.
minimal failing input: SumCounter(
  {
    0: 1,
  },
)
```

- Commutativity **passes**: `+` is commutative. Only idempotence catches it
- proptest **shrinks** the failure to a minimal counterexample: one increment by replica 0 (more on Thursday)

[Live demo] in 10-crdts/demos/: `cargo run -p law-check`

Your Properties Are Graded by Mutation

- The autograder runs **your** properties against plausible **buggy** CRDTs, and your properties must reject them. For example:
 - a G-Counter join that sums (we just saw this one)
 - an LWW register that keeps whichever value **arrived** last
 - a set where remove beats a concurrent add
 - a PN-Counter storing one net count per replica, merged with `max`
- The last two are **perfectly good lattices**. They pass every law, and they're still the wrong data type
- So Part 1 also asks for **semantic** properties:
 - a PN-Counter's value = increments – decrements, under any merge order
 - an OR-Set's concurrent add and remove leave the element present

Limitations of CRDTs

- Some operations need everyone to agree **before** acting:
 - "set **k** to **v** only if it's still **u**" (compare-and-swap)
 - "no two users get the same username"
 - "sell the last ticket only if it's still available"
- Each replica decides **alone**. Alice and Bob both see one ticket left, and both buy it: both writes succeed locally, and no merge can undo either without discarding a write someone was told succeeded
- Real systems **combine** them: CRDTs for the data people edit, consensus for the few operations that need a single order

HW4 Part 4 asks you to name one such operation for your store, and explain why.

Summary

- Replicas that accept writes independently **will** disagree; consensus prevents that at the cost of latency and availability
- **Strong eventual consistency**: same updates $\Rightarrow$ same state, with no coordination
- The merge must be a **join**: commutative, associative, and idempotent. `max`, `u`, and pointwise combinations are; `+` is not
- Counters (G, PN), registers (LWW with **Lamport clocks**), sets (**OR-Set**, add-wins), and maps of lattices
- **Dots** name events; **version vectors** summarize what a replica has seen
- A lattice that obeys laws can still have the wrong operations, so test **semantics** as well as laws

Next Time, and HW4

- HW4 comes out tonight, due Oct 20. Part 1 is today's material, so you can start right away
- Thursday:
 - Text is harder: is a **sequence** a lattice? (RGA: the CRDT behind collaborative editing)
 - Anti-entropy: exchanging **version vectors** to find what a peer is missing, and a replication log that delivers ops in causal order
 - Testing convergence: proptest in depth, and **deterministic simulation** of a whole network

Demo code: [26fa/lectures/10-crdts/demos/](#) (see its README)

Bonus Slides

Keeping every concurrent write, cheaper OR-Sets, and deleting map keys

After-class material: not covered in lecture

The Multi-Value Register (MV-Register)

- LWW discards one of two concurrent writes. The MV-Register **keeps both**, and lets the application (or the user) resolve them
- Each value is tagged with the **version vector** of the write that produced it
 - A write at replica **r** supersedes every value it has seen, and gets a version vector that dominates theirs
 - The join keeps every value that no other value's version vector dominates
- After concurrent writes **"red"** and **"blue"**, reading returns **{"red", "blue"}**, until someone writes a value that has seen both
- This is Amazon Dynamo's shopping cart and Riak's "siblings". The cost is that every reader must handle a *set* of values

Cheaper OR-Sets: Dropping the Tombstones

- HW4's OR-Set keeps a tombstone for every removed tag **forever**, so the state grows with the history, not with the set
- The optimized OR-Set (the "add-wins observed-remove set without tombstones") replaces the tombstones with a **version vector** of every tag the replica has ever seen
 - A tag that the other side has seen (its version vector contains the dot) but no longer has live must have been removed
 - A tag the other side has never seen is a concurrent add, so keep it
- Same add-wins semantics, and the state is proportional to the live elements plus one entry per replica
- This works only if each replica's tags are seen in order, the contiguous-prefix property that Thursday's replication log also relies on

Why HW4's **Map** Can't Delete Keys

- Deleting a key has the same problem as removing a set element: joining with an older state that still has the key brings it back
- An **OR-Map** tracks the keys with an OR-Set: a delete removes the *observed* tags of the key, and a concurrent update to the key's value re-adds it (update-wins)
- The subtle part is the **value**: after a delete and a concurrent update, which parts of the value survive? For a counter, "the increments the deleter hadn't seen" is a reasonable answer, but it requires the value to track causality too
- Production CRDT libraries (Automerge, Yjs, Riak's data types) each pick a policy for this; Riak's maps, for example, share one causal context between the map and its values