

17-350 / 17-650

Lecture 11: Collaborative Text, Anti-Entropy, and Testing Convergence

Jonathan Aldrich

Oct 1, 2026

Lecture Plan

Today	HW4
Finishing Lecture 10: deltas, canonical forms, property tests	Part 1
Collaborative text: the RGA sequence CRDT, and making it fast	Part 2
Anti-entropy: a causal log, synced by version vectors	Part 3a
Testing convergence: proptest, then deterministic simulation	Parts 1–3

HW4 is due Oct 20. After today, you've seen everything it needs.

Deltas: State-Based Merging, Op-Sized Messages

	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	exactly-once, causal delivery	nothing for convergence
G-Counter <code>{A: 3, B: 7}</code> , A increments	<code>{A: 4, B: 7}</code>	"A: +1"	<code>{A: 4}</code>

- A **delta-mutator** `ms` returns just the part that changed, so that `m(X) = X  $\sqcup$  ms(X)`. A delta joined twice, or late, changes nothing
- HW4 Part 1: the key-value store's delta for a write is a **one-entry map**, holding that key's register re-stamped newer than now

```
fn delta_for(&self, edit: KvEdit, replica: ReplicaId) -> Self // DeltaSource, for KvStore
```

HW4 uses all three: Part 1's lattices, the key-value store's deltas, and RGA's ops (today). Almeida, Shoker, and Baquero, 2018.

Canonical Representations, and No HashMap

- 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: a tag is **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`, never `HashMap` / `HashSet`
 - `HashMap`'s iteration order comes from a random per-process seed (`RandomState`), so anything that iterates can differ between runs
 - Later today: why that matters for testing

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. From HW4's `src/properties.rs`:

```
/// A proptest runner: 256 random cases, generated from a fixed seed
pub 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_idempotent::<SumCounter>(): FAILED
Test failed: assertion failed: `(left == right)`
  left: `SumCounter({0: 2})`,
 right: `SumCounter({0: 1})` at law-check\src\main.rs:86.
minimal failing input: SumCounter(
  {
    0: 1,
  },
)
```

- `+` is commutative and associative, so `SumCounter` passes those laws (you'll write them). Only idempotence catches it
- proptest **shrinks** the failure to a minimal counterexample: one increment by replica 0

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

Your Properties Are Graded by Mutation Testing

- 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 it)
 - an LWW register that keeps whichever value **arrived** last
 - a PN-Counter storing one net count per replica, merged with `max`
 - a set where remove beats a concurrent add
- It also runs them on **correct** implementations, built differently from yours: a property that rejects correct code catches nothing
- The last two bugs are **lawful lattices** with the wrong meaning, so Part 1 also asks for **semantic** properties: a PN-Counter's value is increments – decrements; a concurrent add survives a remove; an observed remove sticks
- Laws alone can't even tell `max` from `min`: a version vector joined with **min** passes all four laws (it's the dual lattice)

Homework 2 Awards

- Fastest Threaded Interpreter: Yang Hu
 - About 3x faster than the sample solution!
 - Avoids indirection and bounds checking on the instruction pointer
 - Small, well-isolated use of unsafe
 - Macros make the code more ergonomic
- Fastest JIT: Tie between Alan Joseph and Viren Dodia
 - About 2x faster than other handins!

Text Is a Sequence, and Positions Shift

Alice and Bob share "cat". Alice inserts h at position 1; Bob deletes position 2. Each sends the other "what I did, by index":

	Alice	Bob
local edit	insert h at 1: "chat"	delete at 2: "ca"
apply the other's op	delete at 2: "cht"	insert h at 1: "cha"

- They diverge, and Alice's result is wrong: once anything before a position changes, **its index no longer names the same character**
- Operational Transformation (Google Docs) rewrites indices to compensate, with a central server ordering the edits
- The CRDT approach: give every character a **permanent identity**, and refer to characters, never to positions

RGA: Every Character Gets an Identity

RGA (Replicated Growable Array; Roh et al., 2011):

```
pub struct Id { pub lamport: u64, pub replica: ReplicaId } // compared counter first

pub enum RgaOp {
    Insert { id: Id, after: Option<Id>, ch: char }, // None = at the start
    Delete { target: Id },
}
```

- A new character's id is a **Lamport stamp**: one more than the largest counter this replica has seen, paired with its replica id. Unique, and every character is newer than everything its creator had seen
- An insert names the character it goes **after**, by id
- A delete **marks** its target as deleted: a **tombstone**. It stays in the sequence, because a concurrent insert may name it as its **after**
- Ops must arrive in causal order: an insert can't be placed before the character it follows exists (more on this soon)

Concurrent Inserts at the Same Spot

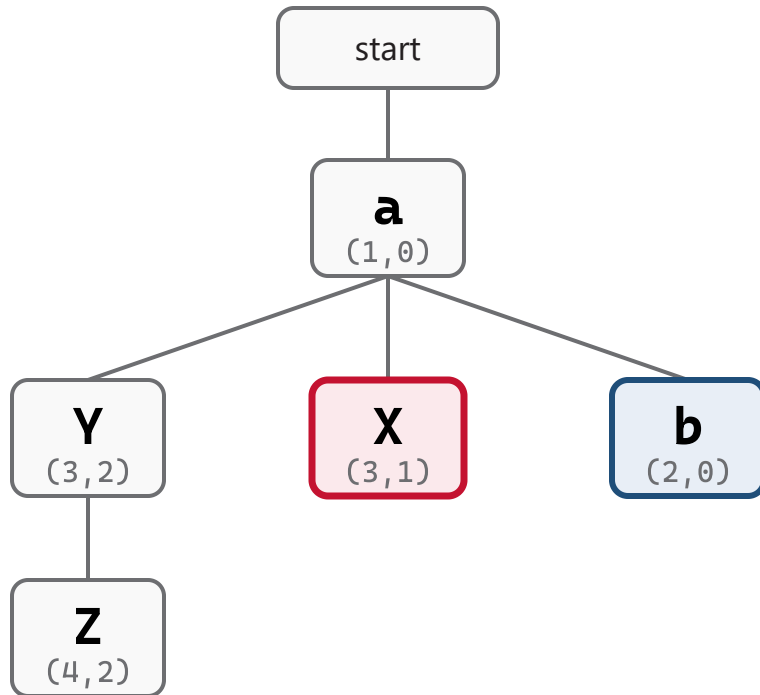
Replica 0 typed "ab" : a = (1,0) , b = (2,0) . Then, concurrently:

Replica 1	Replica 2
inserts x after a : id (3,1)	inserts y after a : id (3,2)

- x , y , and b all follow a . Which order?
- **Rule:** characters inserted after the same element are ordered by **descending id**. (3,2) > (3,1) > (2,0) , so every replica shows "aYXb"
- Arbitrary, but the **same everywhere**, like LWW's tie-break
- A later id sorts first, so text you type right after a character appears right after it, even if someone else typed there earlier

RGA as a Tree

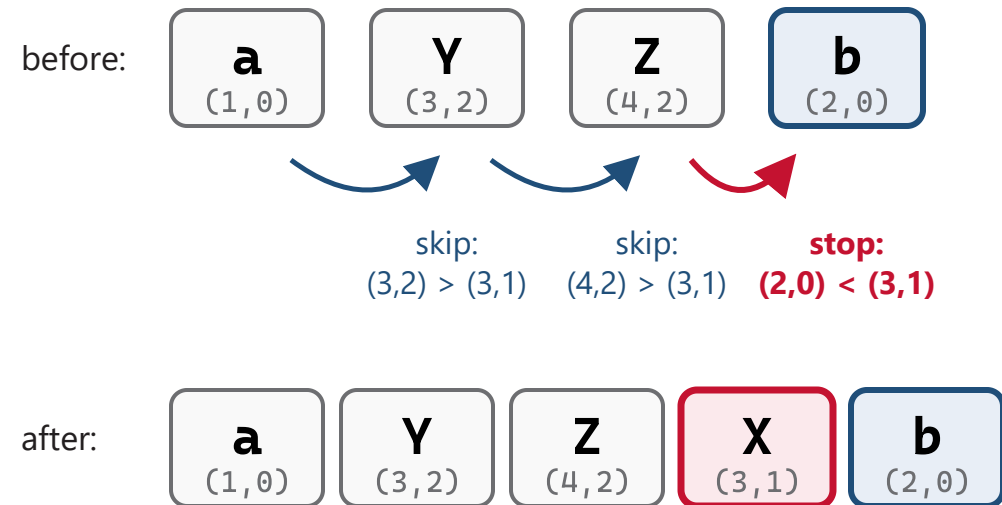
Each character hangs off the one it follows



Siblings are ordered by **descending** id, left to right.

The text is a depth-first walk: **a Y Z X b**

Integrating X (3,1) after a, at replica 2



Z is Y's child, so it was created after its creator saw Y:
its Lamport counter is larger. A subtree's ids all exceed its root's,
so "skip while id > X's id" skips larger siblings **with their subtrees**,
and stops exactly where the tree says X goes.

The Skip Rule, and Why It Works

To apply `Insert { id, after, ch }` to the flat sequence:

```
i = position just after `after` (or 0 if `after` is None)
while i < len and elems[i].id > id: i += 1 // skip what sorts first
insert (id, ch) at i
```

- The elements right after `after` are its larger-id children and **their subtrees**; we must skip all of them
- Every descendant has a **larger id than its ancestor**: its creator saw the ancestor first, and Lamport stamps exceed everything already seen
- So the scan skips larger siblings with their whole subtrees, and stops at the first smaller id: a smaller sibling, or the end of `after`'s subtree
- One comparison per element, no tree needed: the tree is implicit in the order

HW4 Part 2: `prepare` and `apply`

```
pub enum RgaEdit { Insert { pos: usize, ch: char }, Delete { pos: usize } } // positions

impl OpCrdt for Rga {
    fn prepare(&mut self, edit: RgaEdit, replica: ReplicaId) -> Option<RgaOp>; // positions → ids
    fn apply(&mut self, op: &RgaOp); // local or remote
    fn snapshot(&self) -> String;
}
```

- **Local path:** a user edits at a *visible* position; `prepare` turns it into an id-based op (without applying it), and the replica then applies it
- **Remote path:** a peer's op arrives; `apply` alone. Same code for both
- Tests: unit tests, a proptest **convergence** property, and replaying a real editing trace, which must reproduce the final document **exactly**

Replaying a Real Editing Trace - Performance Challenges

Martin Kleppmann and Alastair Beresford recorded every keystroke of writing a LaTeX paper: **182,315 inserts** and **77,463 deletes**. HW4 replays it through `prepare / apply` (local) and through `apply` alone (remote).

Implementation (release mode)	Local replay
linear scans over a <code>Vec</code>	~15 s
a <code>Vec</code> , plus a cursor remembering the last edit	~6.6 s
our sample solution	~ 0.07 s (remote: ~0.04 s)

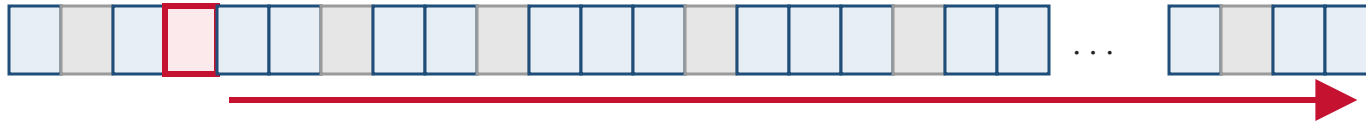
Budget: 1 s for each path. Get a simple version correct first, then make it fast

- Real editing is **local**: most keystrokes are next to the previous one. A cursor exploits that for finding positions.
 - **Discussion: Why isn't it enough? How can this be optimized?**

The automerge-paper trace, CC BY 4.0, via github.com/josephg/editing-traces. Timings on the instructor's machine.

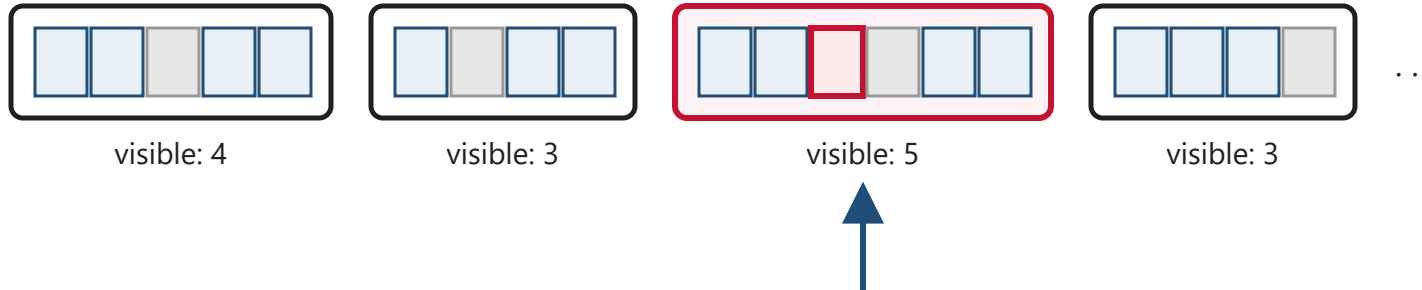
Why a Cursor Isn't Enough: Chunks

One flat Vec: an insert shifts everything after it



every later element moves, tombstones included (the document reaches ~180,000 elements), on each of 182,315 inserts. A cursor makes *finding* the spot cheap, but not the move.

A chunked list: small Vecs, each with a count of its visible characters



cursor: chunk 2, with 7 visible characters before it

An insert moves at most one chunk's elements.

Skip a whole chunk in $O(1)$ using its visible count.

A full chunk (say 512) splits in two.

- The remote path finds characters **by id**: search outward from the cursor (edits are local), or keep an index from ids to chunks

RGA Needs Exactly-Once, Causal Delivery

- Deltas tolerate duplicates and reordering. A lost delta must still be sent again, but it can arrive whenever
- RGA's ops don't:

If an op is...	RGA...
delivered twice	inserts the character twice
delivered before an op it depends on	can't place it: its <code>after</code> doesn't exist yet
never delivered	never converges

- "Depends on": everything its creator had applied when it made the op, such as the character it follows
- So HW4 puts every op in a **replication log** that guarantees exactly-once, causally ordered delivery, for any `opCrdt`: RGA's ops and the key-value deltas alike (Part 3a)

The Causal Log

- Each op is tagged with a **dot** `(origin, seq)` : the origin replica numbers its own ops 1, 2, 3, ...
- `Replica<C>` keeps the CRDT, the **log** of every op it has applied, in the order applied, and a **version vector**

Invariant: for every origin `r`, the log holds exactly the ops `(r, 1) ..= (r, version[r])`, each once, in an order consistent with causality

- Contiguous prefixes are what let one version vector summarize **exactly** which ops a replica has: `{A: 3, B: 1}` means A's first three and B's first
- `local(op)` takes the next dot for this replica, applies the op, and appends it: it follows everything already in the log

Anti-Entropy: Pull What You're Missing

On every gossip tick, a replica picks a peer and **pulls**:

```
B → A: SyncRequest { from: B, version: {A: 1, B: 2} }  
A → B: SyncResponse { from: A, ops: A.missing_for({A: 1, B: 2}) }
```

- `missing_for(theirs)` : every op in my log whose dot `theirs` doesn't contain, **in my log order**
- Log order is causal, so a relayed op arrives **after** its dependencies:

Step	
A types <code>x</code>	A's log: <code>x</code>
B pulls from A, then types <code>y</code> after <code>x</code>	B's log: <code>x</code> , <code>y</code>
C pulls from B (never talked to A)	gets <code>x</code> , then <code>y</code> : <code>y</code> 's <code>after</code> exists

- The version vector makes the request small, and the response contains only what's missing

receive: Skip Duplicates, Stop at a Gap

For each op in the batch, in order: its dot is (r, s) . Compare s with one number, $\text{version}[r]$: how many of r 's ops this replica has

s is...	e.g. $\text{version}[A] = 2$, op (A, s)	then
$\leq \text{version}[r]$	$(A, 1)$, $(A, 2)$	a duplicate : skip it
$= \text{version}[r] + 1$	$(A, 3)$	r 's next op: apply it, log it, increment $\text{version}[r]$ (now $= s$)
$> \text{version}[r] + 1$	$(A, 5)$	a gap ($(A, 3)$, $(A, 4)$ missing): stop , ignore the rest of the batch

- Why stop rather than skip ahead? A later op in the batch, from **any** origin, might depend on the missing one
- Why this is enough: a batch from **missing_for** is **whole**. Every op's dependencies are already here or earlier in the batch
 - A batch with an op cut out of the middle could hide a missing dependency from **another** origin, and a per-origin check can't see that. It's safe because a batch travels as **one message**: it arrives whole, or not at all

Replica IDs Must Be Unique Forever

- What if node crashes, loses its state, and restarts with its **old** id?
- It numbers its ops from 1 again: its new `(r, 1)` is a **different** op from the old `(r, 1)`
- Peers already have `(r, 1)`, so they skip the new op as a **duplicate**. No error, no crash: the replicas **silently diverge**
- The invariant assumed "a dot names one op, forever"
- HW4's fix: a restarted node takes a **fresh id** (the demo's `--id`)
 - `restarted_node_catches_up` crashes a simulated node, restarts it with a new id, and checks that it catches up and its new edits spread
- Real systems persist their state, or draw ids randomly from a huge space

HW4 Part 3: From a Replica to a Node

- 3a (the log), `src/replica.rs`: `local`, `missing_for`, and `receive`. Tested with direct calls (no network)
- 3b (event loop), `src/node.rs`: `run_node`. It handles four kinds of events:

When...	<code>run_node</code> ...
the user makes an edit (it arrives on the <code>edits</code> channel)	applies it with <code>local_edit</code> , and publishes the new snapshot to the UI
the gossip timer ticks (<code>tokio::time::interval</code>)	pulls from a peer chosen with <code>SeededRng</code> : connect, send <code>SyncRequest</code> with its version vector, <code>receive</code> the ops in the <code>SyncResponse</code> , publish
a peer connects and sends a <code>SyncRequest</code>	replies <code>SyncResponse { ops: missing_for(their version) }</code>
a peer is down, slow, or partitioned	gives up on that round after <code>config.timeout</code> , and never panics

- The network I/O runs in **spawned tasks**, one per pull and one per incoming connection, so a slow peer can't stall local edits. You can use an **actor** (Lecture 09): one loop owns the replica, and the tasks send it messages. Or you can use `Arc<Mutex<_>>`
- 3c (demo): run three nodes on localhost with the provided CLI. Kill one, restart it with a new `--id`, and show that it catches up

"Publish" = `snapshots.send_replace(...)`: tests and the demo watch it. A provided check fails to compile if `run_node`'s future isn't `Send`.

proptest Strategies: Recipes for Random Inputs

A property test needs random inputs of the right **shape**. A **strategy** describes how to generate them. You build one from small pieces, each row here built from the row above:

Strategy	Generates	One value it might generate
<code>0..4u32</code>	a number in the range	<code>2</code>
<code>(0..4u32, 0..6u64)</code>	a tuple: one value from each part	<code>(2, 1)</code>
<code>prop::collection::vec((0..4u32, 0..6u64), 0..6)</code>	a <code>Vec</code> of 0 to 5 such tuples	<code>[(0, 3), (2, 1)]</code>
<code>prop::collection::vec(...)</code> <code>.prop_map(VersionVector::from_iter)</code>	each generated <code>Vec</code> , passed through the function <code>from_iter</code>	<code>{0: 3, 2: 1}</code>
<code>any::<VersionVector>()</code>	the provided <code>Arbitrary</code> impl: exactly the row above	<code>{0: 3, 2: 1}</code>
<code>prop_oneof![3 => edit, 1 => sync]</code>	a value from strategy <code>edit</code> 3/4 of the time, else from <code>sync</code>	an <code>Edit</code> (next slide)

- Run a check on many generated inputs: `runner().run(&strategy, |x| { ... })` (Slide 5; 256 inputs), or in a test, `proptest! { #[test] fn t(x in strategy) { ... } }`
- `src/arbitrary.rs` builds each Part 1 type's `any::<T>()` this way. Your Part 1 properties need tuples, ranges, and `vec`, to generate **histories** of operations

Example: A Random Editing Session

From `tests/rga_tests.rs` : one step of a session among three replicas.

```
enum Step {
    Edit { r: usize, insert: bool, pos_seed: usize, ch: char }, // replica r edits
    Sync { from: usize, to: usize }, // the "network": `to` applies every op `from` has that it lacks
}
fn step() -> impl Strategy<Value = Step> {
    prop_oneof![
        3 => (0..3usize, any::<bool>(), 0..64usize, prop::char::range('a', 'e')) // 4-tuple...
            .prop_map(|(r, insert, pos_seed, ch)| Step::Edit { r, insert, pos_seed, ch }), // ...to an Edit
        1 => (0..3usize, 0..3usize).prop_map(|(from, to)| Step::Sync { from, to }),
    ]
}
// The test's input is a whole session, prop::collection::vec(step(), 0..60), for example:
// [Edit { r: 1, insert: true, pos_seed: 37, ch: 'c' }, Sync { from: 1, to: 0 }, Edit { r: 0, .. }, ...]
```

- About three edits per sync, from the weights `3 =>` and `1 =>`
- `pos_seed` : the session is generated before the test runs, so it can't know how long the document will be. The test will turn the seed into a valid position:
insert at `pos_seed % (len + 1)`
- **Shrinking**: when a session fails, proptest retries **simpler** sessions built from the same pieces (fewer steps, smaller numbers, `'a'` rather than `'e'`, an `Edit` rather than a `Sync`), and keeps any that still fail

The Convergence Property

```
proptest! {  
  #[test]  
  fn replicas_converge(steps in prop::collection::vec(step(), 0..60)) {  
    let mut replicas = vec![TestReplica::default(); 3];  
    for s in steps {  
      match s {  
        Step::Edit { .. } => { /* prepare an insert or delete at replica r; record it */ }  
        Step::Sync { from, to } => sync(&mut replicas, from, to), // from's ops, in order  
      }  
    }  
    for _ in 0..2 { // then a full sync, around the ring 0 → 1 → 2 → 0, twice  
      for (from, to) in [(0, 1), (1, 2), (2, 0)] { sync(&mut replicas, from, to); }  
    }  
    let t0 = replicas[0].rga.text();  
    prop_assert_eq!(&replicas[1].rga.text(), &t0);  
    prop_assert_eq!(&replicas[2].rga.text(), &t0);  
  }  
}
```

- The oracle is **agreement**: we never say what the text should be
- It generalizes HW3's "a mismatch is a data race" to "a divergence is a merge bug"

Demo: Shrinking a Real Bug

The same property, on an RGA that compares ids by **counter only**, forgetting the replica-id tie-break:

```
replicas_converge, Rga (ids compared as (counter, replica)): ok (256 cases)

replicas_converge, Rga (counter only): FAILED
Test failed: replicas diverged: ["ea", "ea", "ae"] at src/main.rs:133.
minimal failing input: [
  Insert { r: 2, pos: 0, ch: 'e' },
  Insert { r: 0, pos: 0, ch: 'a' },
]
```

- From up to 40 random steps down to **two concurrent inserts** at the start of an empty document, with equal counters: the whole bug
- **'e'** and **'a'** stay distinct: two equal characters wouldn't show the divergence, so the shrinker can't merge them

[Live demo] rga-shrink: cargo run (output reformatted onto fewer lines). Its code will be posted after HW4 is due.

Testing the Node: Where Nondeterminism Comes From

Part 3's node is async and networked. Test it over real TCP, and whether replicas converge before your timeout depends on **timing**: a failure you may never see again.

Source	In a real run	Under simulation
time	<code>Instant::now</code> , real sleeps	<code>tokio::time</code> runs on simulated time
scheduling	OS threads, tokio's worker pool	every host on one thread , stepped by the simulator
the network	latency, loss, reordering	simulated links; delays drawn from the seed ; partitions on demand
randomness	<code>thread_rng</code> , <code>RandomState</code>	seed every RNG from the test's seed
<code>HashMap</code> order	random per process	use <code>BTreeMap</code>

Put every source of nondeterminism under **one seed**.

turmoil: A Network Simulator for tokio Programs

- **turmoil** is a testing library from the tokio project. It runs a whole distributed system **inside one test**: several simulated machines ("hosts"), a simulated network between them, and a simulated clock
- The hosts run **ordinary tokio code**. Only the edges are swapped out:
 - **network**: `turmoil::net::TcpStream` replaces `tokio::net::TcpStream`. Bytes travel through the simulator, which delays them, and can partition or crash hosts
 - **time**: `tokio::time::sleep` advances the simulated clock, instantly
- The test is a **client** host that drives the system: it makes edits, cuts links, and checks the outcome
- In HW4, you don't call turmoil yourself. The provided `net.rs` and `tests/sim_tests.rs` do, and your `run_node` is the software on each host

Deterministic Simulation with turmoil

From HW4's `tests/sim_tests.rs`, abbreviated:

```
let mut sim = Builder::new()
    .rng_seed(seed) // latency, ordering, ... all from here
    .min_message_latency(Duration::from_millis(1))
    .max_message_latency(Duration::from_millis(25))
    .build();
for i in 0..4 {
    sim.host(format!("node{i}"), move || async move { // a simulated machine
        let edits = ...; let snapshots = ...; ...
        node::run_node(config(i), Rga::new(), edits, snapshots).await?; // YOUR code
        Ok(())
    });
}
sim.client("test", async move {
    /* random edits at random nodes, with random partitions ... */
    turmoil::partition("node0", "node2");
    tokio::time::sleep(Duration::from_millis(150)).await; // simulated: no real wait
    repair_all(); tokio::time::sleep(SETTLE).await; /* then: all snapshots equal? */
    Ok(())
});
sim.run()?;
```

- Your node doesn't change: inside a simulation, `net.rs` uses turmoil's simulated TCP instead of tokio's

Demo: `sim-toy`

Three nodes gossip their max value (Lecture 09's `gossip-toy`) under turmoil. node2 is partitioned away while node0 updates:

```
[t=0.050s] partition: node2 | node0, node1
[t=0.800s] node0 local update: 99
[t=0.810s] node1 learned 99 (had 30)
[t=2.000s] partition healed
[t=2.127s] node2 learned 99 (had 30)
```

seed 7: final values [99, 99, 99]

205 events, trace fingerprint 56ef638bc26f1401

<- every run with seed 7

- **Fingerprint** (the demo's, not turmoil's): a hash of the log of every send and delivery, with its simulated time. Same fingerprint, same execution. Seed 7 gives the same one every run; seed 8, a different one
- `--unseeded` **plants a bug**: the seed is still 7, but each node picks its gossip peer with `RandomState`, which the seed doesn't control. Two runs give **different** fingerprints, **and both still converge**: a passing test can't tell you your seed stopped working

[Live demo] in 11-crdt-text-testing/demos/: `cargo run -p sim-toy -- --seed 7` (then add `--unseeded`)

HW4's Simulation Tests

Test	Checks
<code>kv_replicas_converge</code> , <code>text_replicas_converge</code>	random edits and partitions on 4 nodes, then heal: all snapshots equal (50 seeds)
<code>simulation_is_deterministic</code>	the same seed twice: identical snapshots at every step, same bytes sent
<code>restarted_node_catches_up</code>	crash a node, restart it with a fresh id: it catches up

```
HW4_SEED=17 cargo test --release --test sim_tests text_replicas_converge # replay one failure
HW4_SEEDS=500 cargo test --release --test sim_tests # try seeds 0..499
```

- A failure prints its seed, and that seed replays **exactly**: add `println!` s and see the same run again
- `simulation_is_deterministic` catches the `sim-toy --unseeded` bug: HW3's "a mismatch is a data race" becomes "a mismatch between identical seeds is nondeterminism you introduced"
- Use `--release` : about 4 s instead of about 35 s

From Property Testing to Simulation

	Property-based testing	Deterministic simulation
Randomizes	the inputs to a function	inputs and schedules, delays, faults, crashes
Tests	pure code (Parts 1–2)	a whole distributed system (Part 3)
On failure	a shrunk counterexample	a seed that replays it exactly

- **In industry:** FoundationDB built its deterministic simulator first, before the database itself, and runs whole clusters in one process under injected faults
 - TigerBeetle's VOPR simulates its cluster the same way
 - Antithesis makes arbitrary software deterministic by running it in a deterministic hypervisor
- In Rust: turmoil (from the tokio project) and madsim

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 key-value store, and explain why.

Summary, and HW4

- Text needs **identities**, not positions. RGA: Lamport ids, insert-after, tombstones, and siblings in descending id order, so one skip rule places every insert
- Fast replay is about **data layout**: chunks bound the memmove, and a cursor exploits locality
- Op-based CRDTs need a **causal log**: dots, contiguous prefixes, a version vector per replica, pull what you're missing, stop at a gap
- **Test convergence** with agreement as the oracle: proptest for pure merge logic, deterministic simulation for the async node
- **Suggested order**: Part 1 → Part 3 with the key-value store → Part 2 → Part 3 with text → Part 4. HW4 is due **Oct 20**

Demo code: [26fa/lectures/11-crdt-text-testing/demos/](https://github.com/26fa/lectures/11-crdt-text-testing/demos/) (see its README); rga-shrink after HW4 is due

Bonus Slides

trait Lattice, interleaving, and CRDTs in production

After-class material: not covered in lecture

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 do
- **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

Interleaving, and Fugue

- RGA keeps a run typed **forward** together: each character follows the previous one, so the run is one chain in the tree
- A run typed **backward** (each character inserted *before* the last, at the same position) makes siblings instead. Alice types `abc` backward while Bob types `xyz` backward, at the same spot:

Alice's ids	Bob's ids	Descending order (Bob's replica id is larger)
<code>c (1,A)</code> , <code>b (2,A)</code> , <code>a (3,A)</code>	<code>z (1,B)</code> , <code>y (2,B)</code> , <code>x (3,B)</code>	<code>x a y b z c</code>

- Both users' text is there, but **interleaved** character by character
- **Fugue** (Weidner and Kleppmann, "The Art of the Fugue", 2023) is a sequence CRDT designed to minimize interleaving: concurrent runs stay together whether they're typed forward or backward

CRDTs in Production

- **Automerge**: a JSON-like document CRDT (maps, lists, text, counters), whose core is written in Rust and compiled to WebAssembly for the browser
- **Yjs**, and its Rust port **yrs**: collaborative editing, used in many editors and note-taking apps
- Neither keeps a flat array of per-character nodes: Yjs merges consecutive characters into one item, and Automerge keeps its operations in a B-tree. The same concern as our chunks
- Yjs syncs by exchanging a **state vector** (a version vector): "here's what I have; send me what I lack", as in HW4. Automerge exchanges hashes of the document's latest changes instead
- The editing trace you replay in HW4 comes from Automerge's performance work