CRDT-Based Memory for Distributed Agents with Provable Convergence and GDPR-Compliant Deletion

Ryan Hunter
Helaix Applied Research Institute
research@helaix.com

November 6, 2025

Abstract

Multi-agent systems require shared memory that tolerates partitions and concurrent updates without coordination while meeting regulatory obligations such as GDPR data subject rights. We present a CRDT-based memory for production AI systems that achieves strong eventual consistency with merge-safe semantics and supports GDPR-compliant deletion via tombstones. Building on LWW-element-set and delta-state techniques, we formalize operations, give convergence proofs (associative/commutative/idempotent joins), and integrate causal metadata (vector clocks) for happens-before reasoning. We show how tombstones preserve merge laws while implementing right-to-erasure at the query/export interfaces, and how compaction strategies bound storage growth without violating correctness. The design is realized as a typed-effects capability and composes with a tamper-evident audit ledger, enabling deterministic replay and automated compliance evidence. We outline evaluation of convergence, overhead, and GC trade-offs, demonstrating that CRDT memory provides reliable, compliant state for distributed agents.

1 Introduction

Agents collaborate across services and geographies, learning facts about customers, tasks, and environments. Updates are concurrent; networks partition; replicas drift. Traditional approaches either coordinate (hurting availability/latency) or risk conflicts and data loss (last-write-wins). Meanwhile, GDPR requires right-to-access and right-to-erasure with timely, provable behavior. We need memory that is convergent without coordination and compliant by design.

1.1 Contributions

1. A memory design based on LWW-element-set augmented with explicit tombstones and causal metadata.
2. Formalization of add/remove/query/merge with proofs of associativity, commutativity, and idempotence (join-semilattice).
3. GDPR-compliant deletion via tombstones providing legal effect at the query/export interfaces and auditability via a ledger.
4. Delta-state dissemination and compaction to bound storage growth without violating correctness.
5. Typed-effects integration and production-oriented ergonomics.

1.1.1 Assumptions & threat model

Replicas may partition and later heal; messages may be delayed, duplicated, or reordered. Clocks may skew; timestamps are logical/monotonic where available. The adversary can replay stale updates and attempt "resurrection" by injecting old adds; it cannot break cryptographic hashes if used in audit/export. Our objective is strong eventual consistency with *erasure non-resurrection* while supporting bounded compaction.

1.1.2 Design goals

1. **Strong eventual consistency** under asynchronous networks and crash-stop failures.
2. **Interface-level erasure**: deletion is enforced at query/export boundaries without violating merge laws.
3. **Bounded storage growth**: compaction with explicit safety barriers.
4. **Auditability**: evidence of adds/removes and causal order for compliance export.

2 Background and Related Work

CRDTs converge by making merge a join (associative, commutative, idempotent); state-based CvRDTs and op-based CmRDTs are well-studied [@shapiro-crdts]. Delta-state CRDTs disseminate joinable deltas instead of full state [@almeida-delta-crdt]. JSON/document CRDTs (Automerge, Yjs) show practicality for complex datatypes [@automerge; @yjs]. Logical time and vector clocks capture happens-before [@lamport-time; @fidge-vector; @mattern-vector]. GDPR Articles 15/17 motivate deletion and export semantics [@gdpr].

3 Model

We model facts as key-addressed entries; memory is a pair of maps:

```
type Key = string
type Fact = unknown
type LWWElementSet = {
  adds: Map<Key, { value: Fact; timestamp: number }>
  removes: Map<Key, number> // tombstones
}
```

Operations.

- `add(fact, ts)`: `adds[key] = { value: fact, timestamp: ts }`
- `remove(fact, ts)`: `removes[key] = ts` (tombstone)
- `query(pattern)`: return values where `addTs > (removeTs || −∞)` and `matches(value, pattern)`

- `merge(other)`: pointwise `max` by timestamp for both maps (LWW for adds; max for removes)

Design intent. Adds and removes commute and are idempotent under `max`. Queries enforce deletion at the interface while preserving merge laws.

Formalization (set-theoretic). Let K be the key domain, $A : K \rightharpoonup (V \times T)$ the partial map of adds, and $R : K \rightharpoonup T$ the partial map of remove timestamps. The partial order on states is pointwise:

$$(A, R) \preceq (A', R') \iff \forall k.\ \big(A(k)\!\downarrow \Rightarrow A(k).t \le A'(k).t\big)\ \wedge\ \big(R(k)\!\downarrow \Rightarrow R(k) \le R'(k)\big).$$

The join $(A, R) \sqcup (A', R')$ is pointwise max by timestamp; $(\mathcal{S}, \sqcup)$ is a join-semilattice with least element $\bot = (\varnothing, \varnothing)$. Visibility is $vis(k) \iff A(k)\!\downarrow\ \wedge\ \big(R(k)\!\uparrow\ \vee\ A(k).t > R(k)\big)$.

Timestamp discipline. Timestamps may come from physical clocks, logical counters, or vector clocks. When physical skew is non-negligible, prefer monotonic logical time for intra-replica order, optionally pairing with coarse physical time for GC horizons.

Variations. Where duplicate adds must be tracked distinctly (e.g., multivalued registers), substitute OR-Set semantics: tag adds with unique identifiers; removals record *observed* identifiers; merge by union, query by set difference. LWW-Element-Set remains a fast default when per-key value replacement is intended.

4 Convergence Proof (Sketch)

Let the state space be `S = MapK(AddEntry) 𝔼 MapK(RemTs)` with merge $\sqcup$ defined pointwise by `max` (per key). Since `max` is associative, commutative, and idempotent, $(S, \sqcup)$ is a join-semilattice. Under eventual delivery (e.g., anti-entropy/gossip), replicas converge to the same least upper bound; thus strong eventual consistency holds.

5 GDPR-Compliant Deletion

Tombstones. A `remove(fact, ts)` creates a durable tombstone. Queries and exports omit any `add` dominated by `remove`:

```
visible(fact) <=> addTs > removeTs
```

This preserves CRDT laws and makes deletion observable as absence in query/export results, while keeping auditability.

Lemma (Erasure non-resurrection). Let $a(k, t_a)$ be the latest observed add for key k and $r(k, t_r)$ a tombstone with $t_r \ge t_a$ at some replica. After any sequence of merges (with pointwise max on adds and removes), k is invisible to queries/export. *Proof sketch.* Merges compute max for k in both maps; since $t_r \ge t_a$, the visibility predicate $t_a > t_r$ remains false in any upper bound.

Compaction. Retain tombstones for a horizon sufficient to dominate any in-flight add with lower timestamp, then safely compact. With causal metadata (vector clocks or epochs), no late "older" add can still arrive before compaction.

Export semantics. *Right to access (GDPR Art. 15).* Export all visible facts with metadata $(k, A(k).t)$; include ledger pointers to add/remove events for audit.
Right to erasure (GDPR Art. 17). Record $r(k, t)$; honor erasure at query/export while retaining minimal metadata (key digest, timestamps, controller id) necessary to (i) prove erasure occurred and (ii) preserve chain integrity. Redaction markers are carried in exports.

6 Causality and Vector Clocks

Track a vector clock alongside state to reason about happens-before:

```
type VectorClock = Map<string, number>
function happensBefore(a: VectorClock, b: VectorClock): boolean {
  for (const [r,c] of a) if (c > (b.get(r) ?? 0)) return false
  return true
}
```

If one clock dominates the other, it supersedes; if concurrent, join via CRDT merge and join clocks component-wise. Causality enables safe GC and auditable ordering.

Theorem (Causal-compaction safety). Suppose each replica maintains a vector clock and compacts a tombstone (k, t) only after observing a barrier B such that for all replicas r, $B[r] \geq t[r]$. Then no add dominated by t can later arrive and become visible after compaction. *Proof sketch.* A join of any state with clock $\preceq B$ cannot introduce an add with clock component greater than B in every dimension; by construction, such an add would have been in flight before the barrier and therefore is accounted for.

7 Implementation (Typed Effects)

Expose memory as a capability in an Effect-style runtime:

```
interface MemoryTag {
  add(f: Fact): Effect<never, never, void>
  remove(f: Fact): Effect<never, never, void>   // tombstone
  query(p: Pattern): Effect<never, never, ReadonlyArray<Fact>>
  merge(delta: LWWElementSet): Effect<never, never, void>
  compact(ttlMs: number): Effect<never, never, number>
}
```

Operational hooks.

- Emit audit spans (subject id, controller, purpose) before mutation; bind vector-clock and tombstone metadata into the ledger for replay.
- Store deltas for dissemination; periodically checkpoint to a full snapshot.

Anti-entropy & deltas. Peers gossip delta-intervals $(\Delta A, \Delta R)$ tagged with their current clocks. Receivers join locally and advance their "ack" clock. To bound replay, retain recent delta intervals for a window W; beyond W, peers must request a snapshot. This yields $O(|\Delta|)$ bandwidth per sync in the common case.

Storage layout. Maintain (i) an *active index* of visible keys, (ii) a *tombstone index* (k, t), and (iii) a *barrier table* tracking last-known replica clocks. Compaction scans tombstones whose clocks are dominated by all barrier rows; upon compaction, remove k from both indexes while preserving a minimal proof record (key digest + compaction epoch) in the audit ledger.

8 Performance and Engineering

Complexities. Query: `O(n)` over active facts; Merge: `O(m)` over incoming delta; Memory: without GC, `O(total_ops)`; with GC, `O(active + recent_deletes)`.

Delta-state replication. Disseminate compact deltas periodically; peers join them locally.

Clock skew. Prefer logical/monotonic clocks for adds/removes when available.

Cost model. Let n_v be the number of visible keys, n_t tombstones, and $\rho = n_t/(n_v + n_t)$ the tombstone ratio. Query cost is $O(n_v)$ with a small constant if the active index is maintained; merges are $O(|\Delta|)$; compaction cost per run is $O(n_t)$ but amortized to $O(\rho)$ per operation over a window. The delta gossip bandwidth is $O(|\Delta|)$ with snapshots bounded by $O(n_v + n_t)$.

Clock skew sensitivity. When using physical LWW timestamps, skew ϵ can cause temporary mis-visibility. Prefer logical clocks for ordering and physical time *only* to set loose GC horizons $> \epsilon_{max}$.

9 Evaluation Plan

Metrics.

- Convergence: fraction converged vs time/fanout under churn.
- Overhead: CPU/memory vs tombstone ratio; query latency P50/P99 with tombstone checks.
- GC efficacy: reclaimed bytes per run; impact on query latency and merge time.
- Erasure SLA: time to propagate tombstones to all replicas; export correctness.

Workloads. Concurrent add/remove storms; long partitions with late merges; mixed CRDTs in multi-agent scenarios.

Adversarial workloads. Inject add/remove races with long partitions and late-arriving "old adds"; measure false-visibility rate (should be zero) under the visibility predicate and after compaction barriers; report compaction-induced query latency changes.

Methodology. We evaluate on clusters of 3, 5, and 9 replicas with churn (join/leave) and partitions lasting up to 15× the gossip interval. Metrics include: time-to-convergence (fraction converged vs time), query P50/P99 under varying ρ, delta vs snapshot bandwidth, and erasure SLA (time from $r(k, t)$ to global invisibility).

Baselines. Compare to (i) naive LWW without tombstones (unsafe for erasure), (ii) OR-Set without compaction (unbounded storage), and (iii) centralized coordinator (linearizable, but unfit under partitions). Our design should match naive LWW on throughput while maintaining erasure correctness and bounded growth.

10 Case Studies

1. Customer preferences. Concurrent updates converge; erasure generates tombstones invisible to queries/exports while replicas reconcile.
2. Distributed research memory. Offline edits merge without conflict; deletions remain effective.
3. Contact graph. Tombstones prevent "resurrection" after late merges; compaction bounded by causal thresholds.

11 Discussion and Limitations

Tombstone accumulation requires GC; LWW is a good default but some domains need OR-Set/registers/counters; consider encrypted payloads or keyed hashing. Cross-org federation implies federated audit.

Security & privacy. If values carry personal data, encrypt-at-rest and export via redaction/tombstones; optionally hash keys to mitigate membership inference on sparse domains; bind subject identifiers and purposes into ledger spans for compliance audit.

Limitations. LWW semantics collapse concurrent writes to a single winner per key; use OR-Set or MV-registers where per-insert identity matters. Vector clocks grow with replica count; for large federations, consider dotted version vectors or epoch-based barriers. Compaction depends on observing barriers from all replicas; laggards can delay GC.

12 Related Work

Shapiro et al. on CRDT theory; Almeida et al. on delta-state CRDTs; Automerge/Yjs; Lamport/Fidge/Mattern on causality; GDPR Articles 15/17.

13 Conclusion

CRDT-based memory with tombstones provides merge-safe, convergent state and GDPR-compliant deletion for distributed agents. With causal metadata, delta-state replication, and typed-effects integration, it delivers a practical foundation for production AI systems that need both availability under partition and compliance under audit.

Appendix A Operations (reference)

```
add(fact, ts):
  key = hash(fact)
  adds[key] = { value: fact, timestamp: ts }

remove(fact, ts):
  key = hash(fact)
  removes[key] = ts   // tombstone

query(pattern):
  results = []
  for (key, {value, addTs}) in adds:
    delTs = removes.get(key)
    if (delTs == null || addTs > delTs) && matches(value, pattern):
      results.push(value)
  return results

merge(other):
  // adds: LWW by timestamp
  for (key, {value, ts}) in other.adds:
    if !adds[key] || ts > adds[key].timestamp:
      adds[key] = { value, timestamp: ts }
  // removes: max by timestamp
  for (key, ts) in other.removes:
    if !removes[key] || ts > removes[key]:
      removes[key] = ts
```

Appendix B Causality-Aware Compaction

- Maintain `clock` with state; propagate `(state, clock)` in deltas.
- Compact tombstone `(k, t)` only after observing a barrier where all replica clocks dominate `t` (epochs/snapshots certify no earlier adds can still arrive).

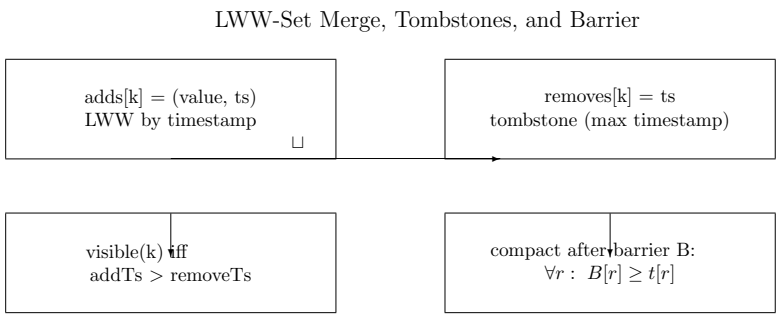

Figure 1: LWW-set with explicit tombstones: pointwise max merge on adds/removes, visibility predicate (addTs > removeTs), and causal barrier B for safe compaction.