vicrds: A Collaborative Text Editor

Kyle Trinh
kytrinh@ucsd.edu

Patryk Tomalak
ptomalak@ucsd.edu

Scott Semtner
ssemtner@ucsd.edu

Abstract

We introduce *vicrds*, a web-based, peer-to-peer, collaborative text editor built on Conflict-free Replicated Data Types (CRDTs) and gRPC. To guarantee eventual consistency without a central server, our system relies on a custom Replicated-Growth Array (RGA) CRDT. We engineered a fault-tolerant backend capable of dynamic scaling by integrating a gossip-based SWIM protocol for failure detection, a seamless state transfer mechanism for node synchronization, and distributed tombstone compaction for efficient garbage collection. Together, these features enable a highly available and partition-tolerant editing experience.

1. Introduction

Collaborative text editing in decentralized environments poses significant challenges in ensuring data consistency amid concurrent modifications [1], network partitions, and dynamic group membership [2]. Traditional client-server models avoid many of these issues by relying on a central authority to sequence operations [3].

However, in a peer-to-peer (P2P) setting, achieving consensus on the order of text insertions and deletions requires a fundamentally different approach [4]. To address these challenges, we introduce *vicrds*, a highly available, decentralized collaborative text editor. At the core of *vicrds* is a Replicated-Growth Array (RGA) [5]; it is a Sequence CRDT that tracks character-level operations. By leveraging Lamport clocks [6] and out-of-order message buffering, our RGA implementation deterministically resolves write conflicts and guarantees that all clients eventually converge to a unified state in some asymptotic notion of time.

Beyond the underlying data structure, a robust P2P editor requires scalable distributed mechanisms to handle the network topology. Our primary mechanisms in *vicrds* include:

- **Decentralized Architecture:** A gRPC-based networking layer that allows nodes to broadcast updates directly without a central server.
- **Dynamic Group Membership:** A gossip-based SWIM protocol [2] implementation for scalable and efficient peer failure detection.
- **State Synchronization:** A state transfer mechanism that allows newly joining nodes to dynamically discover peers, request the current document snapshot, and catch up.
- **Distributed Garbage Collection:** A tombstone compaction process designed to safely remove deleted characters across the network, improving memory efficiency and join latency.

2. Background & Related Work

2.1. Operational Transformation vs. CRDTs

Historically, real-time collaborative editors have relied on Operational Transformation (OT) [7] to maintain consistency across multiple clients. In an OT system, when concurrent edits occur, the system transforms the operations so that they achieve the same result regardless of the order they are applied. However, OT fundamentally relies on a central server to dictate the total ordering of operations. In a decentralized, peer-to-peer (P2P) network where no single authority exists and network partitions are common, achieving a strict global order using OT becomes computationally expensive and highly error-prone [1].

To overcome the limitations of OT in distributed environments, Conflict-free Replicated Data Types (CRDTs) were introduced [8]. CRDTs guarantee strong eventual consistency without requiring a central coordinator. This makes them well-suited for P2P applications where clients may be temporarily disconnected or experience high latency.

2.2. Real-World Collaborative Editors using CRDTs

There has been a shift towards CRDTs for collaborative applications. Notable open-source implementations and repositories include:

- **Automerge**: A library built on the Conflict-Free Replicated JSON Datatype [9]. It allows complex, nested JSON documents to be edited concurrently and merged automatically across peers.
- **Yjs**: A highly optimized, production-ready sequence CRDT implementation primarily used in web ecosystems to power collaborative text editors, canvas drawing apps, and 3D modeling tools [10].
- **Zed**: A high-performance collaborative code editor written in Rust. Zed embeds a custom CRDT implementation into its core architecture to seamlessly merge keystrokes from multiple developers in real-time without locking the UI [4].

2.3. The Replicated-Growth Array (RGA)

While there are many Sequence CRDT algorithms designed for text editing (such as Treedoc and Logoot), *vicrds* specifically builds upon the Replicated-Growth Array (RGA) [5]. The RGA was designed to support optimistic insertion and deletion while ensuring operation commutativity and precedence transitivity. We selected the RGA for its simple approach to anchoring new characters to the IDs of preceding characters.

2.4. Decentralized Group Membership

Traditional collaborative web applications manage active users via a centralized database or an authoritative server. In a pure P2P architecture, nodes must dynamically discover each other and detect unexpected failures without a central registry. To achieve this, *vicrds* utilizes the Scalable Weakly-consistent Infection-style Process Group Membership Protocol (SWIM) [2]. SWIM is a gossip-based protocol that provides robust, highly scalable peer discovery and failure detection, forming the underlying network topology that our CRDT operates over.

3. System Design & Requirements

We set out to build a peer-to-peer collaborative text editor where several users can edit a shared document at once, with no central server and no authoritative copy. Every participant holds a full replica, applies edits locally, and exchanges them directly with the others.

The decentralized setting imposes several requirements:

1. No replica sequences operations or owns the document, and any node may fail or leave without halting the rest.
2. Replicas that have applied the same edits on the same state show the same document, and concurrent edits at the same position resolve to one order everywhere. In particular, replica state must converge.
3. A user can read and edit at any time, even while disconnected; an edit never waits on another node, keeping the system available.
4. If two sides of a network split, clients can continue editing and reconcile writes once it heals.
5. A node joins by contacting one existing participant and may leave or fail at any time; failures are detected and there is no fixed member list.
6. Deletions leave tombstones, which should be eventually reclaimed so storage tracks the visible document.

During a network partition, because of the CAP Theorem, a distributed system must choose between consistency or availability. In *vicrds*, we decided to prioritize availability; *vicrds* never blocks a write, never coordinates with other replicas, and does not halt operations during a network partition. The tradeoff is that *vicrds* allows the partitions to diverge. Consistency is eventual, in the manner of Bayou (weakly connected replicated storage) [11]. This suits interactive editing, where blocking a keystroke would provide a substantially worse experience.

No part of the design for tombstone compaction (garbage collection) relies on an agreement or commit round; every replica acts on its own state and on evidence that travels with ordinary edits. One unreachable node never blocks the edits of others, though in our current implementation it can stall garbage collection from taking place. We assume a crash failure model and do not defend against Byzantine failures. Access control is also intentionally out of scope.

4. The *vicrds* Architecture

A *vicrds* participant is a single process that holds one replica of the shared document and communicates with all other participants as a peer. Each participant runs three cooperating components: a Replicated-Growth Array (RGA) that stores the document and resolves concurrent edits (Section 4.2), a SWIM membership layer that discovers peers and detects failures (Section 4.5.1), and a gRPC networking layer that carries operations and state between peers (Section 4.6). A web interface (Section 4.7) sits on top of the replica.

An edit follows the same path at every participant. When the user types or deletes a character, the local replica applies the change immediately and produces an operation describing it. That operation is broadcast to every connected peer, which applies it to its own replica. A node that joins, or that has fallen behind, copies a peer's state directly through a state transfer (Section 4.5.2). Membership changes and tombstone compaction use the same operation streams as regular edits.

4.1. Conflict-free Replicated Data Types

CRDTs [8] are data structures replicated across a set of distributed processes that satisfy the following properties:

1. Updates to any replica of the CRDT must not require synchronization. In particular, there does not exist a single node or process whose purpose it is to order the operations.
2. CRDTs are conflict free. When two replicas of the same CRDT diverge, merging their changes must not produce conflicts. The replicas must reconcile each other automatically, without manual intervention.
3. CRDTs must provably converge to a correct common state.

Properties 2 and 3 are important; if we have a correct implementation of a CRDT, then we have shown eventual consistency.

In *vicrds*, we leverage State-based Convergent Replicated Data Types (CvRDTs). When processes communicate with one another, they transfer the internal data of the CRDT to each other to update their state. We can mathematically model a CvRDT as a monotonic join semilattice: a partially ordered set equipped with a least upper bound (LUB) operation for any pair of elements.

In this model, the state of the CRDT monotonically increases with each local update according to the partial order. When a node receives a state from a remote peer, it merges the remote state with its local state by computing their least upper bound. Because the least upper bound operation is commutative, associative, and idempotent, the order in which updates are received and merged does not matter.

These properties are crucial because they guarantee strong eventual consistency. As long as all updates are eventually delivered to all replicas, every replica will converge to the

exact same state, ensuring safety and liveness despite any number of network partitions or node failures without requiring consensus [12] or synchronization. These qualities are ideal for P2P text editors like *vicrds* because they enable highly scalable and performant decentralized applications.

4.2. The Replicated-Growth Array (RGA)

The Replicated-Growth Array (RGA) is a State-based, Convergent, Sequence CRDT. Our implementation of an RGA (called `Rga`) represents the entire collaborative document as an ordered list. It also maintains historical metadata of when each insertion and deletion was made in `Id` (we will elaborate on this in the next section).

It exhibits join semilattice properties, meaning that the state of the RGA strictly grows (nodes are only ever added or marked as tombstones) and merging two RGAs is simply the union of their nodes. Because all operations are additive or monotonic, the RGA satisfies the commutativity, associativity, and idempotency required of a CvRDT.

In our implementation, `Rga` contains a vector of `Nodes`, which stores the document's data. The nodes are sorted by their `Id`.

4.2.1. `Id`

At the core of our `Rga` implementation is `Id`, which uniquely identifies every operation and element in our sequence. While the overall state of the RGA is a partially ordered semilattice, the `Ids` themselves must form a strict total order to ensure all replicas sequence text identically.

```
#[derive(Eq, PartialEq, Hash)]
pub struct Id {
  pub logical_clock: u64,
  pub client_id: ClientId,
}
impl PartialOrd for Id {
  fn partial_cmp(&self, other: &Self)
    -> Option<Ordering> {
    Some(self.cmp(other))
  }
}
impl Ord for Id {
  fn cmp(&self, other: &Self) -> Ordering {
    self.logical_clock
    .cmp(&other.logical_clock)
    .then(self.client_id.cmp(&other.client_id))
  }
}
```

The Rust traits `Eq` and `PartialEq` guarantee strict equality checks, which are essential for performing $O(1)$ node lookups in our internal hash map and safely ignoring duplicate operations to maintain idempotency. Conversely, the `Ord` and `PartialOrd` traits establish the total ordering mechanism. When two `Ids` are compared, we first compare their Lamport clocks (`logical_clock`). If the clocks are identical (two clients typed a character concurrently at the exact same position), we break the tie using the unique `client_id`. This total ordering guarantees that every replica will resolve concurrent insertion conflicts deterministically, preventing divergence.

4.2.2. `Node`

Each character (or element) in the RGA is encapsulated in a `Node` struct. These nodes form the linked list that represents the sequence of the document.

```
#[derive(Debug, Clone, PartialEq, Eq)]
pub struct Node<T> {
  pub id: Id,
  pub value: T,
  pub left_id: Id,
  pub is_deleted: bool,
}
```

We deliberately implemented `Node` with generic typing (`T`) instead of strictly tying it to text characters. This makes the CRDT flexible; if we wanted to build a collaborative integer array, or even a list of complex nested structs, the underlying CRDT logic and data structures remain unchanged.

The purpose of each field is as follows:

- `id`: The unique identifier for this specific node.
- `value`: The actual data payload of generic type `T` (e.g., a `char` for our text editor).
- `left_id`: A reference to the `Id` of the node immediately preceding this one at the exact moment it was inserted. This acts as the "pointer" in our logical linked list, allowing us to anchor the element correctly in the sequence regardless of when it arrives at other replicas.
- `is_deleted`: A boolean flag that acts as a tombstone.

4.2.3. `Rga` Implementation

The `Rga` struct orchestrates the actual CRDT logic, tying together the elements we discussed above:

```
pub struct Rga<T> {
  nodes: Vec<Node<T>>,
  pub map: HashMap<Id, usize>,
  logical_clock: u64,
  id: ClientId,
  insert_buffer: Vec<(Node<T>, Instant)>,
  delete_buffer: Vec<(Id, Instant)>,
}
```

The `nodes` vector is the authoritative, ordered sequence of elements. Finding a specific node's insertion point by scanning the vector linearly would be inefficient. To solve this, we maintain a HashMap, which provides $O(1)$ lookups from any node's `Id` to its exact index in the `nodes` vector. The `logical_clock` and `id` fields maintain the local state needed to generate new `Ids` when the user types.

4.2.4. The Root Node

Upon initialization, the RGA creates a "root node" with an ID of (0, 0). This sentinel node is prepended to the start of the `nodes` vector. It serves as the "beginning of history" for the system. The very first character a user types must have a valid `left_id` to anchor itself to, and the root node fulfills this role. Furthermore, we intentionally made the root node unaddressable by users to ensure we do not accidentally delete the head of the system, preventing catastrophic failure.

4.3. Out-of-Order Message Buffering

Operations are not self-contained; they refer to other operations. An insert is anchored to the node to its left, and a delete is associated with its target. Our system does not make any assumptions about the ordering of arriving messages, so the operation an arriving message depends on may not have been applied yet.

Instead of dropping these operations, the replica holds them in either the `insert_buffer` or the `delete_buffer`. When a new node is successfully inserted, both of the buffers are drained, retrying every operation whose dependency is now satisfied. Draining repeats until no further operation can be applied.

This buffering allows the protocol to assume nothing about ordering. Waiting to execute an operation until its dependency is present produces the same result as if they had arrived in order, because the merge algorithm places a node deterministically from the set of nodes present, not from the order they were received [8]. The reference node eventually arrives, either as an ordinary broadcast or as part of a state transfer. If the operation stays buffered beyond a short timeout, the replica treats its dependency as lost and pulls a state transfer from a peer.

4.4. RGA Merge Algorithm

In the simple case, when the user types a character, `insert_local` increments the `logical_clock`, generates a new `Id`, creates a `Node`, and immediately splices it into the sequence.

However, handling remote insertions via `insert_node` requires carefully resolving conflicts. If two peers concurrently insert different characters after the same `left_id`, the algorithm must ensure every replica orders them identically. It does this by:

1. Using the `map` to locate the index of the `left_id`.
2. Iterating rightward through the sequence from that point.
3. If it encounters a concurrent sibling (another node sharing the same `left_id`), it orders them descending by their `Id`. If the new node has a higher `Id`, it stops; otherwise, it keeps traversing rightward.
4. If it encounters a descendant of a concurrent sibling (a node anchored to something we already skipped over), it skips over it. This ensures we do not accidentally split apart a contiguous word typed by another user.
5. Once the correct index is found, the node is spliced into the `nodes` vector and the `map` indices are updated.

We abstract these incoming operations into `Rga` via `Operations`. `Operations` will come from the network or the client itself.

4.5. Distributed Systems Mechanisms

4.5.1. Dynamic Group Membership (SWIM)

In a decentralized P2P network, nodes must dynamically discover each other and accurately detect failures without a central registry. To accomplish this, *vicrds* implements a custom variation of the Scalable Weakly-consistent Infection-style Process Group Membership Protocol (SWIM) [2]. A new node joins by contacting a single known active peer. From there, the new node's presence "infects" the network as peers gossip its existence to others.

The `MemberState` enum tracks the health of every node from the local replica's perspective:

```
pub enum MemberState {
    Alive,
    Suspect,
    Dead,
}
```

Each peer in the cluster is tracked using a `MemberInfo` struct, which records its current `state`, its incarnation number (a monotonically increasing counter used to refute false failure detections), and the timestamp of its `last_status_change`.

Nodes communicate these states over our gRPC infrastructure using a `SwimUpdate` struct:

```
pub struct SwimUpdate {
    pub address: String,
    pub state: MemberState,
    pub incarnation: u64,
}
```

Every node runs a local `Swim` state machine. When nodes communicate over gRPC, they "piggyback" an array of `SwimUpdates` onto the message, containing a snapshot of their known membership table.

If an incoming gossip update contains an incarnation number higher than what is currently known, the local membership table is updated. If the incarnation numbers match, ties are broken by state severity: `Dead` overrides `Suspect`, which overrides `Alive`.

Crucially, if a node receives a gossip update stating that it is `Suspect` or `Dead` (a false positive due to network latency), the node will immediately refute the claim. It does this by incrementing its own `local_incarnation` number and broadcasting an `Alive` state. This guarantees that living nodes can defend themselves against false failure detections.

Our SWIM implementation detects failures using a combination of active probing and timeouts. If a node fails to respond to a direct gRPC `Ping`, it is immediately transitioned to a `Suspect` state.

Once a node is marked as `Suspect`, a background task periodically ticks the suspects. If the suspect node does not refute the suspicion within a configured timeout (and no other peer gossips a higher-incarnation `Alive` message for it), the node is formally transitioned to `Dead`. All connections to the dead node are dropped, and its `Dead` state is gossiped to the rest of the network so that other peers can prune it from their routing tables.

4.5.2. State Transfer for Joining Nodes

The list of active members in the network is treated as shared data and synchronized across all nodes. When a new replica joins, an admission record is generated that logs the member that sponsored it, and the position the admission occupies in the sponsor's operation stream (sequences are numbered; see Section 4.5.3). The first replica creates the document by admitting itself, and every later member is admitted by an existing one. A replica that restarts will join as a new member rather than resuming its old identity. Membership records travel both as ordinary operations and inside state snapshots, and are idempotent when applied.

A new replica asks any member for state. The responding member treats a request from an unknown identifier as an application to join, writes the admission into its own operation stream and then, under the same lock, builds the snapshot it returns. The joining replica applies the membership records, then merges the nodes, and only after it's seen its own admission can it start authoring operations. Replicas that have left or are in the process of joining may not serve snapshots since their state may be incomplete. Several members may admit the same newcomer concurrently and the duplicate records are harmless.

A member that wants to leave must write a retirement operation that names the length of the member's operation stream, so a replica that holds the entire stream of a departed member can be sure that it has received all messages that member will ever send. The leaving member waits until the vector of every other member covers its final operation, then exits. If there is a timeout when waiting, the leaver exits anyway, which causes the system to fall into the failure case of the next section, where compaction cannot proceed. Future, more complex designs could provide alternatives to this scenario.

4.5.3. Distributed Tombstone Compaction

We are not able to simply remove deleted characters from the RGA. An insert is positioned by the ID of the character to its left, so an insert that is concurrent to a delete could still reference that newly deleted node. If some but not all replicas have removed that node, the insert can be applied at some replicas and never at the others, causing the documents to diverge. Removing a tombstone is only safe once no operation referencing it can still arrive, which is the idea of causal stability from existing CRDT literature [13]. Our implementation is described in this section.

Each replica numbers its own operations with sequence numbers $(1, 2, 3, \ldots(\text{seq}))$. Membership records, inserts, and deletes all take a slot in the sequence. Deletes can be identified as (`deleter`, `seq`), which are recorded in a `deleted_by` set within the target node. We cannot reuse the existing Lamport clock for this because those clocks jump while merging, so a gap between values is not necessarily a missing operation. With the contiguous numbering, it is possible to make and confirm claims like "this replica holds operations 1 through s from node c."

Every replica r tracks a *received vector*, which for every other replica c is the longest gap-free prefix of c's operations that r has received. Once per second, every member broadcasts this received vector, with an entry for its own last assigned sequence number, to every other replica. These vectors are merged by max at the entry level, so duplicated, reordered, and lost vectors do not cause issues. They are sent over the same peer-to-peer channels as normal operations.

We consider a deleted node able to be removed by a replica when the following three conditions are met:

1. Every member's vector covers the node's insert and every entry in its `deleted_by` set. After that, no member can create a new reference to it, since anchors are only ever chosen from visible characters.
2. The replica holds everything each member had created as of that member's latest vector. An insert referencing the deleted node must have been created before its author applied the delete, meaning before the author sent the vector that acknowledges the delete, so it is counted by the author's entry for itself, and this condition guarantees it has arrived.
3. Nothing anchors at the node (it is a leaf).

The first two conditions are read from the received vectors and the third is embedded into how the compact function is implemented. It walks the array right to left removing deleted, fully acknowledged leaves. The leaf test is one comparison against the next node, and removing a leaf exposes its parent within the same sweep of the array. There is no coordination step; each replica compacts whenever its own conditions are met. Since the removed nodes were invisible anyway, replicas that compact at different times display the same text. One limitation of our approach is the third condition, which means a deleted node with a live descendant is kept because it is the position reference for that descendant.

The first two conditions are based on the member set of Section 4.5.2, minus any departed members whose entire operation stream is already held. Joins are safe because of the order that a sponsor writes. Consider a scenario where a sponsor serves a snapshot which has some character visible while a delete of that character is in flight; the joiner might anchor new text to it. That means no replica can have removed the character yet because the sponsor itself has not acknowledged the delete. When the sponsor's vector does end up covering the delete, that vector's value for the sponsor's own operation count includes the admission, which was written before the snapshot. The second condition then forces any replica that acts on the sponsor's acknowledgement to hold the admission first. Once it does, the first condition waits for the new member's acknowledgements as well. Put together, this means a removal can never outrun a join that might still reference the removed node.

State transfer sends all nodes, and a merged node counts as received. This raises a hazard: a snapshot from a peer that has not yet compacted still contains tombstones the receiver already removed. When merging a snapshot node, we therefore check whether its sequence number falls within the prefix we already hold from its creator. If it does but the node is not in our array, we removed it earlier and skip recreating it. The same transfer also repairs losses, since a delete lives in its target's `deleted_by` set until removal is safe everywhere, so a replica that missed the original delete still picks it up from any peer's snapshot. This sync is run on reconnect and periodically against a random peer, in the style of Bayou's anti-entropy [11].

If a member stops responding, its vector stops advancing and compaction stalls for anything it has not yet acknowledged. Editing is not affected, because placement never reads this state, so clients can keep interacting with the document. Tombstones are held until the member returns and syncs or leaves gracefully. Future work would also add dead peer detection and expulsion, removing them from the members list. For this implementation, we choose stalling instead of guessing with a timeout, as that could cause failure where a peer reconnecting could reintroduce data whose tombstones the other replicas have already purged.

4.6. Networking Layer

Replicas communicate over gRPC (HTTP/2). All replicas act as both a client and server of the same service with four RPCs: `ApplyOperation` to send edits, `GetState` to transfer state, `Ping`, and `PingReq` for SWIM. `ApplyOperation` carries operations in the form of operation variants: insert, delete, received vector, admission, and retirement. Each replica sends messages directly to every peer it is connected to. The set of live replica addresses is supplied by SWIM, and a background task opens missing connections and drops connections to addresses that SWIM says are dead. The network topology therefore converges to a mesh of the live replicas.

The protocol has no assumptions about message ordering. An insert or delete that arrives before something it depends on waits in a buffer, a vector that arrives before the operations it counts only delays compaction, and a stale vector changes nothing because vectors are merged by max. In practice, each peer gets a send queue drained by one task, which keeps a replica's messages in order, but it is not necessarily needed for safety. When a send fails, the peer and everything queued for it are dropped, and later repaired by state transfer.

`GetState` returns the node array and membership records. The receiver applies records first, then merges the nodes. A sync is triggered when a connection is opened, when a buffered operation has been stuck for three seconds, and when a peer's vector claims operations we lack for multiple consecutive checks. It is also triggered unconditionally every 30 seconds against one random peer, as an upper bound on how long the other triggers can miss anything.

4.7. User Interface

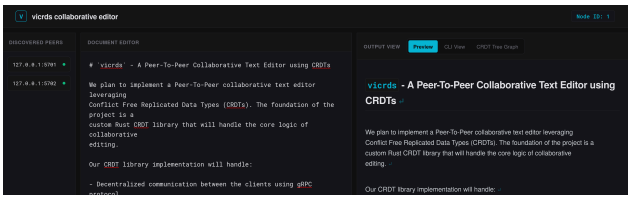

Figure 1: Web demo using 3-node cluster

We also implemented a functional frontend using Axum that communicates with the backend RGA and node connection state of the system via the WebSocket protocol. Additionally, it implements a tree representation of the internal RGA structure for convenient visual debugging of the content state at a given node, as well as internal representation of the RGA consistent with a CLI-based inspection.

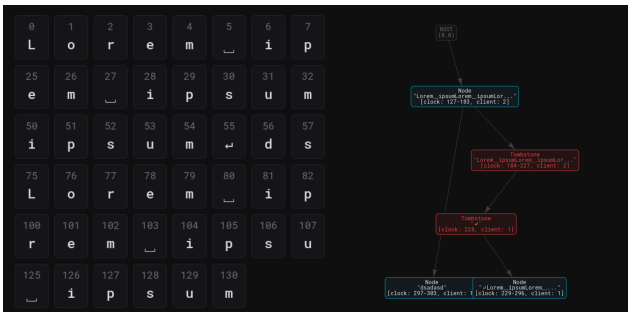

Figure 2: Left: CLI representation, Right: CRDT Graph

In our proof of concept browser editor, we were successfully able to implement the following features: markdown preview, browser-native copy-paste, undo/redo (ctrl+z, ctrl+y), display of connected and alive nodes. Handling native browser batch operations allows for efficient updates across the nodes, processing them as a single delta, regardless of the operation size. In our testing we can maintain a consistent document state across multiple nodes, both within the same system (localhost) and across LAN / WiFi with very responsive update speed. Moreover, thanks to implementing garbage collection as tombstone compaction, we are able to maintain robust performance as the document edits grow (Section 4.5.3).

5. Evaluation

We defined the core experiments in a contained, highly automated test suite. We use easily human-readable YAML files to define the test objectives and execution steps, and allow for either local harness (no network partitions) or containerized execution. Using containers allows us full control over the network conditions. We can emulate partitions, latency, packet loss, jitter, node failure, and then test for convergence at any point of the test. We inspired the scope of our suite by Jepsen-style [14] testing methodology to make sure that we exhaust all reasonable system failures.

5.1. Performance Tests

We started with a local benchmark harness to understand how the system behaves and scales under normal operation. The end result of this work is contained in the `local_benchmark.yaml` scenario, which covers 3 different aspects of the system.

First, system scaling, which runs a workload of writer operations while growing the cluster from 3 to 100 replicas. Despite growing the cluster significantly, our results indicate that the convergence time stays low, and grows at a slower rate than the increase in the data exchange. Each additional node means more peers to broadcast to, so the gossip overhead compounds with cluster size at a steady rate, however the results indicate that the communication latency is amortized by an efficient implementation.

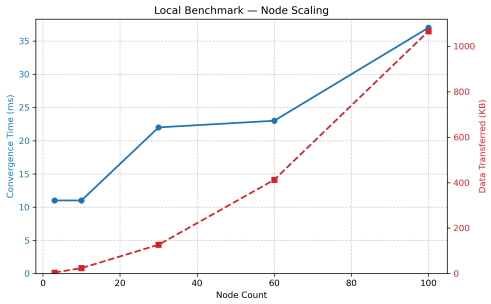

Figure 3: Convergence time and data transferred vs. cluster size, local loopback

Moreover, we tested that as the number of writers increased from 1 to 32, the aggregate throughput scales linearly alongside it. Each writer pushes its own stream of operations, and the merge path handles the concurrency without much contention at these sizes.

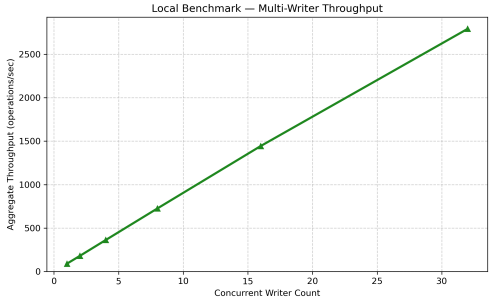

Figure 4: Aggregate throughput vs. concurrent writers, 3 nodes

And finally, we also measured what happens when a new node joins and pulls the full document from a peer. The corresponding figure indicates that only when dealing with a document in the neighborhood of a million characters is the standard sync latency impacted. Otherwise it keeps steady at approx. 300ms for the vast majority of scenarios the system would be subjected to in practice. And even as the document grows, the graph suggests the latency increase is slower than that of the document size.

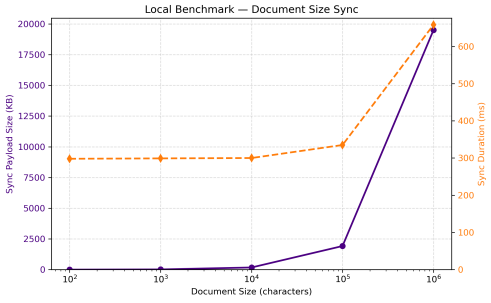

Figure 5: Sync payload size and duration vs. document size

5.2. Containerized Tests

Next, we introduced network conditions using Docker containers with traffic shaping, testing for one variable at a time while keeping the others constant.

In the first scenario we change the one-way latency from 0ms to 300ms on a 5-node cluster with no packet loss. Convergence time grows roughly linearly with the increased latency. Every operation waits on round-trips for acknowledgement-bearing messages, so as the round-trip stretches, each operation takes proportionally longer. Data transferred also creeps up, likely because of the increased node gossip and more time to accumulate retransmission. It's noteworthy to note the steep jump from 0ms to 50ms, which indicates just how much a good network latency can preserve a performance of a distributed system.

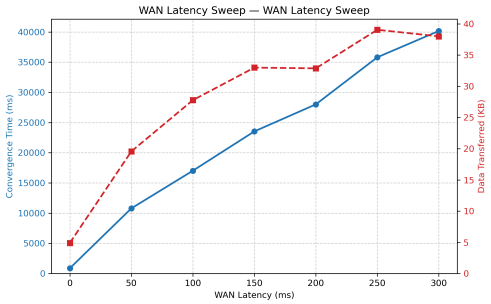

Figure 6: Convergence time and data transferred vs. one-way latency, 5 nodes

The next scenario shows how packet loss impacts the convergence time. Here we keep latency at fixed 80ms, and gradually increase the ratio of lost packets. Surprisingly, it turns out that for all but the more extreme packet loss scenarios, minor packet disruption is not heavily impacting the data transfer in our tests. Only at around 20% we begin to see a noticeable difference due to retransmits, but the system still converges. Compared to latency, loss has a milder effect, probably because gRPC retries and the periodic anti-entropy sync recover most dropped messages before they impact the system.

Figure 7: Convergence time and data transferred vs. packet loss, 5 nodes

And finally, the wan scaling scenario tries to recreate realistic WAN conditions (80ms latency, 1% packet loss, 15ms jitter) with a cluster growing from 3 to 20 nodes. here, the convergence time grows at a much slower rate than the overhead of transferred data. The takeaway here is that the system can handle realistic network scenarios with a predictable, nearly linear, increase in latency and data exchange overhead.

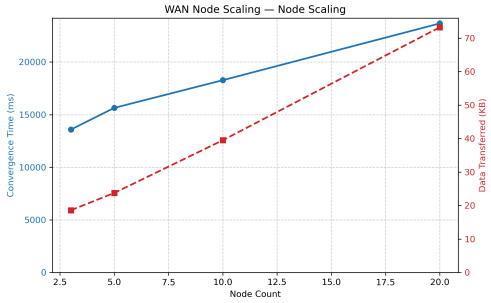

Figure 8: Convergence time and data transferred vs. cluster size under WAN (80ms latency, 1% loss)

5.3. Correctness Tests

Throughput numbers only tell us if the system can scale, however we still miss information about whether it behaves correctly. The Rust-native tests can only ensure the behavior in an isolated hermetic scenario. To test real-world scenarios we needed to expand our harness by a method to robustly verify the mid-run as well as end-result state of each of the nodes, making sure the internal CRDT state is consistent and their properties are maintained, even after subjecting them to various compromising scenarios.

In our first test, `partition_heal.yaml` we prepare a scenario that splits a 6-node cluster twice under moderate network conditions (50ms latency, 0.5% loss). The first split is symmetric: three nodes on each side, both groups writing independently. When the partition heals, the two halves merge their divergent states and all six replicas converge. The second split is deliberately asymmetric: a single node isolated against a five-node majority. Here we wanted to see whether the minority can catch up without data loss after a more severe skew. In both cases the system converges cleanly, and post-heal writes on both sides confirm the cluster remains usable over the test runtime.

Next, in the `fuzz.yaml` scenario we use a deterministic seed to generate a series of 200 random insert and delete operations. The load is split across 5 nodes with 50ms latency and 2% packet loss. After the fuzzing window, all five replicas pass a full document consistency check. The seed makes the test both thorough and debuggable, yet in our tests each time the fuzzing did not produce any failure condition and the documents converged cleanly.

The heaviest scenario is `comprehensive_stress.yaml`. Ten nodes run 25 fault cycles under light baseline latency (30ms). Each cycle interleaves four text insertions and two deletions with randomized faults: partitions at 12% probability, node kills at 6%, and latency spikes between 50ms to 300ms at 15%. These values were tuned so that the cluster is stressed but not completely unusable. In the end the system survives a few dozens fault events. We then heal all impairments, and the cluster fully reconverges before passing both consistency and post-heal liveness checks.

And finally, we use the `swim_convergence.yaml` scenario to target any issues with our SWIM protocol implementation. An 8-node cluster runs with accelerated gossip timings (100ms interval, 3-second suspect timeout) under 40ms latency. We kill two nodes mid-test, the survivors detect the failure and keep accepting writes. When the killed nodes restart, they rejoin the cluster, pull state from their peers, and the group reconverges. A second kill and restart cycle on a different node confirms the recovery process is repeatable. Just like before, all consistency checkpoints pass, meaning the document never diverged despite two rounds of node churn.

5.4. Compaction

We ran a mixed churn benchmark to see how tombstone compaction affects storage and read performance over time. A 3-node cluster does 20 seconds of random inserts and deletes, then wipes the document. We compare two runs of the same workload: one with compaction disabled and one with it enabled, sampling the RGA state every 500ms.

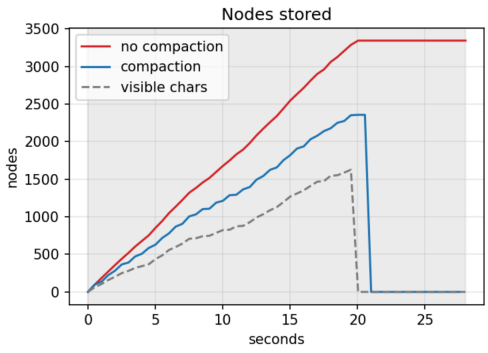

Figure 9: Nodes stored over time, with and without compaction

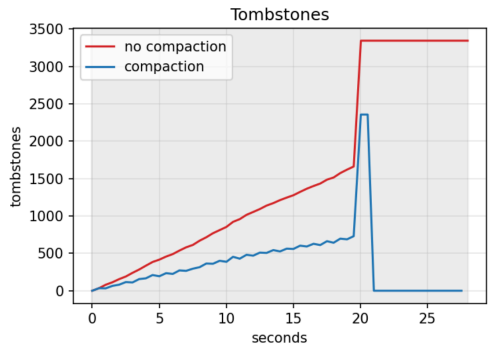

Figure 10: Tombstone count over time, with and without compaction

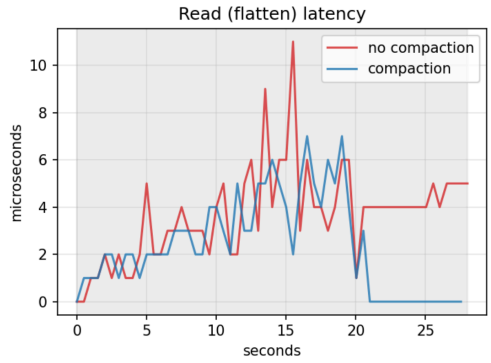

Figure 11: Flatten latency during mixed churn, with and without compaction

During the churn window the two modes closely follow each other: node count and tombstone count grow at the similar linear rate. After the delete wave the lines reveal the main difference. Without compaction, the array freezes at over 3,000 nodes, all tombstones. With compaction, the array drains to zero within a second. A late-joining node confirms the practical difference: syncing state takes over 100ms against an uncompacted cluster and pulls thousands of dead nodes, but finishes in single-digit milliseconds when the other replicas have already reclaimed everything.

6. Conclusion

Over the course of our project we successfully built *vicrds*, a collaborative text editor that works peer-to-peer, with no central server, and still can survive the kinds of failures that real networks throw at distributed systems. However, after implementing and testing the full stack, it did come with some caveats.

The RGA CRDT at the center of the system meets it basic function with full correctness. Concurrent edits at the same position resolve deterministically everywhere, and our final evaluation never produced a document divergence. The SWIM layer detects failures quickly, so that writes continue through node joins and leaves. State transfer lets a new node join by contacting a single peer and catch up without unnecessary overhead. The compaction mechanism reclaims tombstones independently on each replica, reducing the stored node count to zero within a second of a full-document delete, and cuts join time by an order of magnitude.

The evaluation gave us a clear picture of where the system is strong and where it is not. Convergence time degrades predictably under latency and packet loss, and per-message delay dominates over cluster size as the primary cost, at least in a reasonable use case setting. The correctness scenarios: symmetric and asymmetric partitions, seeded fuzzing, sustained chaos injection, all passed without a single divergence. That is probably the biggest achievement of our work.

7. References

[1] M. Ahmed-Nacer, C.-L. Ignat, G. Oster, H.-G. Roh, and P. Urso, "Evaluating CRDTs for Real-time Document Editing," in *11th ACM Symposium on Document Engineering*, ACM, Ed., Mountain View, California, United States, Sept. 2011, pp. 103–112. doi: 10.1145/2034691.2034717.

[2] A. Das, I. Gupta, and A. Motivala, "SWIM: scalable weakly-consistent infection-style process group membership protocol," in *Proceedings International Conference on Dependable Systems and Networks*, 2002, pp. 303–312. doi: 10.1109/DSN.2002.1028914.

[3] B. Liskov, S. Ghemawat, R. Gruber, P. Johnson, L. Shrira, and M. Williams, "Replication in the harp file system," *SIGOPS Oper. Syst. Rev.*, vol. 25, no. 5, pp. 226–238, Sept. 1991, doi: 10.1145/121133.121169.

[4] N. Sobo, "How CRDTs make multiplayer text editing part of Zed's DNA." 2022.

[5] H.-G. Roh, M. Jeon, J.-S. Kim, and J. Lee, "Replicated abstract data types: Building blocks for collaborative applications," *Journal of Parallel and Distributed Computing*, vol. 71, no. 3, pp. 354–368, 2011, doi: https://doi.org/10.1016/j.jpdc.2010.12.006.

[6] L. Lamport, "Time, clocks, and the ordering of events in a distributed system," *Communications of the ACM*, vol. 21, no. 7, pp. 558–565, July 1978, doi: 10.1145/359545.359563.

[7] C. A. Ellis and S. J. Gibbs, "Concurrency control in groupware systems," in *Proceedings of the 1989 ACM SIGMOD international conference on Management of data - SIGMOD '89*, Portland, Oregon, United States: ACM Press, 1989, pp. 399–407. doi: 10.1145/67544.66963.

[8] M. Shapiro, N. Preguiça, C. Baquero, and M. Zawirski, "Conflict-Free Replicated Data Types," *Stabilization, Safety, and Security of Distributed Systems*, vol. 6976. Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 386–400, 2011. doi: 10.1007/978-3-642-24550-3_29.

[9] M. Kleppmann and A. R. Beresford, "A Conflict-Free Replicated JSON Datatype," *IEEE Transactions on Parallel and Distributed Systems*, vol. 28, no. 10, pp. 2733–2746, Oct. 2017, doi: 10.1109/TPDS.2017.2697382.

[10] K. Jahns, “Yjs: Shared data types for building collaborative software.” [Online]. Available: https://github.com/yjs/yjs

[11] D. B. Terry, M. M. Theimer, K. Petersen, A. J. Demers, M. J. Spreitzer, and C. H. Hauser, “Managing update conflicts in Bayou, a weakly connected replicated storage system,” *ACM SIGOPS Operating Systems Review*, vol. 29, no. 5, pp. 172–182, Dec. 1995, doi: 10.1145/224057.224070.

[12] L. Lamport, “Paxos Made Simple,” *ACM SIGACT News (Distributed Computing Column) 32, 4 (Whole Number 121, December 2001)*, pp. 51–58, Dec. 2001, [Online]. Available: https://www.microsoft.com/en-us/research/publication/paxos-made-simple/

[13] C. Baquero, P. S. Almeida, and A. Shoker, “Making Operation-Based CRDTs Operation-Based,” in *Distributed Applications and Interoperable Systems*, K. Magoutis and P. Pietzuch, Eds., Berlin, Heidelberg: Springer, 2014, pp. 126–140. doi: 10.1007/978-3-662-43352-2_11.

[14] K. Kingsbury, “Jepsen: A framework for distributed systems verification, with fault injection.” [Online]. Available: https://github.com/jepsen-io/jepsen