

SyncNotes:

A Consistency-Oriented Replicated Store for Collaborative Notes

Marzia De Maina

Alma Mater Studiorum - University of Bologna

Computer Science, Campus of Bologna

Distributed Software Systems, A.Y. 2025/2026

About SyncNotes

What it does

One shared set of plain-text notes: browse, open, edit, create, delete, always the same for the whole team.

Who uses it

A small, non-technical team, each person on their own laptop, connecting and disconnecting independently.

Why it's distributed

Many independent clients read and write the same mutable state at once, and it must survive a machine failing without lost work.

No edit lost

Disjoint changes merge automatically; a directly conflicting line resolves deterministically.

Distributed System Features

Tr.

Transparency: replication and failures masked to clients as long as quorum holds.

Sc.

Scalability: growing to any N replicas is a deployment change, not a code change.

Co.

Consistency: CP in CAP terms; writes linearized through a Raft log, reads served only by the leader.

Op.

Openness: plain HTTP and JSON contracts; another client could target the same API.

FT

Fault tolerance: tolerates one replica down; a content hash lets a replica detect and repair its own corruption.

Se.

Security: no authentication, authorization or encryption; the design assumes a trusted network.

Pa.

Partition tolerance: the minority side loses quorum, excluded from writes; the majority keeps serving.

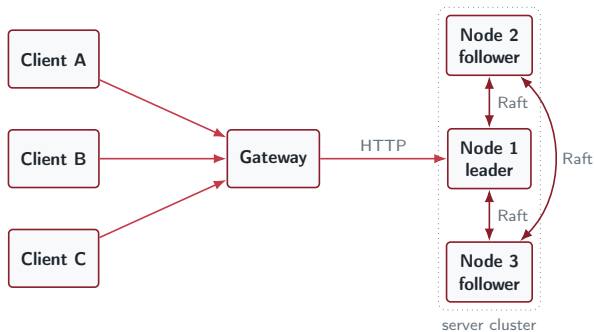
Ec.

Economy: lightweight containers on one host, no external infrastructure; cost scales linearly with N .

Architecture: Three Tiers

One direction of authority

- The **client** interacts through a full-screen TUI, holds provisional copies and tracks a base version; it has no authority over state.
- The **gateway** is a thin, stateless reverse proxy: discovers the current Raft leader and forwards every request there.
- The **cluster** is a replicated state machine: every replica runs the same note-store logic over the same Raft log.



Requirements

Functional

- F1** List available notes
- F2** Download a note
- F3** **Save with conflict-aware merging**
- F4** Create a note
- F5** Delete a note for everyone
- F6** Automatic propagation to open clients
- F7** Usable by a non-technical user

Non-Functional

- N1** Replication/failure transparency
- N2** **Strong consistency of state**
- N3** Total order of writes
- N4** **Tolerates one replica (or leader) failure**
- N5** **No split brain under partition**
- N6** Corruption self-healing
- N7** Automatic rejoin after restart
- N8** Deletion convergence
- N9** Idempotent, retry-safe writes
- N10** Responsive UI, never blocks
- N11** Unavailable, not corrupt, if quorum is lost entirely

Implementation

- I1** Python 3.12 for client, server and gateway
- I2** Persistence in a SQL database (SQLite)

Design

Design Rationale & Data Model

Why this style

- Client/gateway split → N1: clients bind to the gateway, never to a replica.
- Replicated state machine → N2, N3: consensus gives one order, merge runs inside the replicated operation.
- Leadership plus quorum → N4, N5: failure tolerance without split brain.

Note

name, version, content, content_hash, updated_at, deleted; the authoritative aggregate, held identically by every replica.

Sidecar (client)

{name, version} next to a downloaded note: the base version for the next three-way merge.

Node / replica

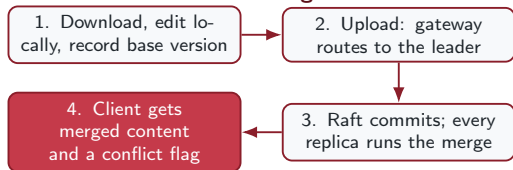
replica_id, consensus state, has_quorum, peers_connected.

Client & Gateway

Client holds provisional Notes and Sidecars; the Gateway holds no domain state at all.

Editing Workflow & Merge Rule

From edit to merged result



The merge rule

- Disjoint lines on both sides: merged, both survive.
- Same line changed on both sides: server wins, client is told.
- Unknown or missing base: server content kept, conservative by design.

Why it is safe: the merge runs inside the Raft commit, so its result depends only on log order, never on which replica evaluated it.

Merge in Action: A Mini Example

Two clients read the same base version of a note and edit concurrently:

```
1:  Agenda:  budget review    2:  Attendees:  Alice, Bob
```

Disjoint lines: automatic merge

Client 1 commits first (line 1):

```
1:  Agenda:  budget review + hiring
```

Client 2 commits next (line 2):

```
2:  Attendees:  Alice, Bob, Carol
```

Client 2 receives:

```
1:  Agenda:  budget review + hiring
```

```
2:  Attendees:  Alice, Bob, Carol
```

Conflicting line: server wins

Client 1 commits first (line 1):

```
1:  Agenda:  hiring plan only
```

Client 2 commits next (line 1):

```
1:  Agenda:  postpone to Friday
```

Client 2 receives:

```
1:  Agenda:  hiring plan only
```

```
had_conflict = true
```

Write Operations and Background Reconciliation

upload

Looks up the base version, computes the three-way merge, bumps the version, archives it, clears any tombstone. An unknown-content upload *creates* the note.

delete

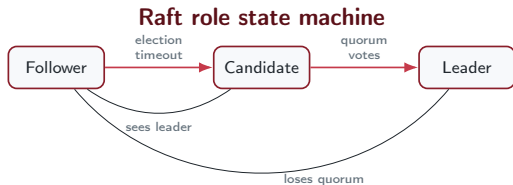
Sets the tombstone and bumps the version; a repeated delete is a no-op. Writes a tombstone even for a note never seen, so a stale peer cannot resurrect it.

reconcile

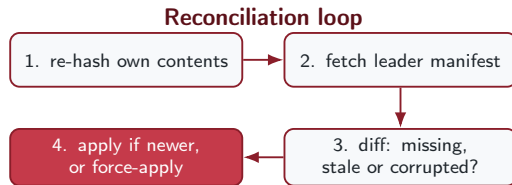
Runs on every non-leader replica on a fixed interval: diffs its manifest against the leader's and pulls what it is missing, stale on, or corrupted on.

upload and delete are committed through the Raft log; reconcile is a pull-based safety net layered on top, not a replacement for consensus.

Consensus & Reconciliation



- Only a leader accepts writes, only while a quorum is reachable.
- Missed heartbeats trigger an election; quorum of votes wins.



- Fixed interval, non-leader only: anti-entropy on top of consensus.
- Catches what Raft cannot: a storage-level overwrite that never went through the log.

Client Concurrency & Consistency Model

Client side

- **Two threads:** a curses main loop (never blocks indefinitely, fixed timeout) and a background watcher (holds the events stream open).
- **One shared, lock-protected state:** note list, status message, which note is open for editing.
- While a note is open, the watcher must not touch its local file or sidecar.

Server side

- Writes are **linearizable** with respect to the Raft commit order.
- Reads go to the leader only: **read your writes** and **monotonic reads**.
- A follower may transiently lag; it is never read directly by a client.

Without the client side lock, a background refresh could rewrite the sidecar's base version mid-edit, corrupting the merge on data that looked perfectly fine. Server side, an upload with an unknown base version against an existing note keeps the server content entirely, conservative by design.

Fault Tolerance and Availability

Failure	Effect
Follower crashes	Quorum still 2 out of 3, service continues; rejoins and reconciles on restart
Leader crashes	A follower is elected, gateway re-routes, service resumes
Gateway crashes	Clients cannot reach the cluster until restart (the one single point of failure)
Client crashes	No cluster impact; provisional edits are simply lost

- **Detection:** Raft detects a dead leader within an election timeout and triggers a new election; the gateway does one re-discovery and one retry before a 503.
- **Under partition:** the majority side keeps serving; the minority reports `has_quorum=false` and accepts nothing, so there is no split brain.

Implementation and Deployment

Server and Gateway Internals

Server cluster

- The note store subclasses the consensus layer's replicated-object base; upload and delete are plain methods wrapped with a `@replicated` decorator.
- An `async` adapter bridges Raft's callback API to FastAPI: the handler awaits an `asyncio` future with a 10 second timeout.
- SQLite uses `check_same_thread=False` since the consensus tick thread and the Uvicorn worker thread both touch the engine; each unit of work opens its own session.

Gateway (stateless reverse proxy)

- Discovers the leader by polling each replica's `/health`, forwards every request there with one retry across re-discovery.
- Never mutates cluster state; holds no durable state of its own.
- A server-sent-events stream pushes the note list on every change; a low-frequency poll catches out-of-band changes such as reconciliation.

Client: Modules and Controls

Modules (tui.py is the actual client)

cli.py

scriptable subcommands, used by chaos scripts and the collaborative-edit demo.

api.py

thin HTTP calls plus the idempotency-key functions.

sync_state.py

read, write, clear the hidden JSON sidecar.

notes.py

fetch, open, save, delete, create workflow and wording.

tui.py

curses full-screen UI, the F7 entry point (syncnotes.py); the other modules are its building blocks, not separate clients.

TUI controls (keyboard commands)

↑/↓ (k/j)

move the selection.

Enter

open the selected note for editing.

n / d

create / delete a note (with confirmation y / n).

r / q

refresh now / quit.

Persistence and the HTTP API

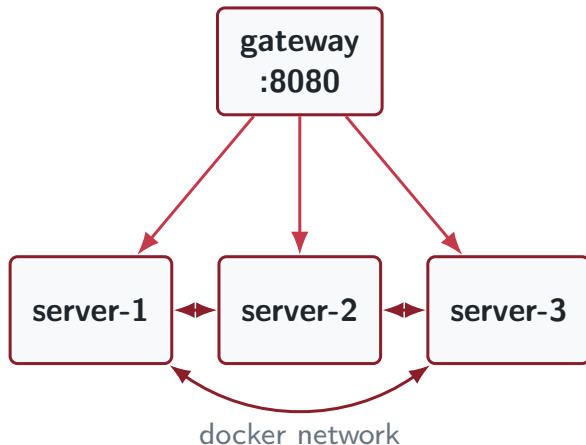
Each replica's SQLite stores

- files: one row per note
- file_versions: one row per (note, version)
- applied_operations: idempotency ledger

7 HTTP ENDPOINTS

GET /health · **GET** /files · **GET** /files/{name}
· **POST** /files/{name} · **DELETE** /files/{name}
· **GET** /internal/manifest · **POST** /internal/reconcile

Deployment Topology



- One compose file, four services, one bridge network.
- Only the gateway's port is published.
- Each replica has its own named volume: database and Raft log survive a restart.
- Growing the cluster is a configuration change: add a service and a peer address, not a code change.

Validation and Results

Validation Strategy & Test Suite

- 167 automated unit and integration tests, all passing.

Module	Type	Covers
test_merge.py	Unit	Every branch of the merge rule
test_db.py	Unit	Version monotonicity, tombstones, corruption detection
test_server.py	Integration	FastAPI surface, round-trips, op-id dedup
test_reconciliation.py	Integration	Pull/repair paths, no resurrection
test_quorum.py	Integration	Three real Raft nodes, in-process
test_gateway.py	Integration	Leader discovery, routing, events stream
test_api.py	Unit	List/download/upload requests to the gateway
test_cli.py	Unit	Download/upload/list commands, base-version handling, merge-outcome messages
test_notes.py	Unit	Fetch/open/save orchestration, error wrapping, outcome wording
test_sync_state.py	Unit	Sidecar file read/write for the local base version
test_tui.py	Unit	Username default, note-name validation, text size and normalization

End-to-End Demo & Chaos Scenarios

- `demo_collaborative_edit.py` drives four real client directories through both merge scenarios; two live TUI sessions show the “your line was dropped” status; `inspect_db.py` confirms multi-client convergence; a cold `docker compose up` elects a leader from scratch.

Script	Injects and asserts
<code>kill_leader</code>	Kills the leader mid-flight, waits for a new one, writes again through the gateway.
<code>network_partition</code>	Disconnects a follower; checks no split brain and reconvergence on heal.
<code>corruption</code>	Overwrites stored content directly (hash left intact); checks reconciliation repairs it.
<code>restart_node</code>	Force-recreates a follower on a new IP; checks it rejoins and catches up.
<code>delete_converges</code>	Deletes a note while a replica is down; checks the tombstone converges everywhere.

DEMO

Wrapping Up

Future Work

Availability & scale

Highly-available gateway (multiple instances or a client-side replica-list fallback); follower reads once freshness can be proven.

Integrity & security

Periodic cross-replica hash comparison with majority vote against a corrupted leader; TLS, client/inter-replica authentication, per-note ACLs.

Housekeeping

Garbage-collect old versions and durably-seen tombstones; a scrolling TUI editor to lift the window-fit restriction.

Conclusion

Consistent by design

A single Raft log orders every write; the merge runs inside the replicated operation, so its outcome depends only on that order.

Fault-tolerant and self-healing

One replica, including the leader, can fail without losing data; reconciliation repairs corruption consensus cannot see.

Usable by anyone

A full-screen UI needs no commands, no note names, no file paths, arrow keys only.

Thank you for
your attention