SyncText-CRDT — Design Document

Shreyan Naskar
Roll: 25CS60R41
M.Tech. Computer Science and Engineering.

Nov 10, 2025

Overview

SyncText-CRDT is a multi-process, lock-free collaborative text editing system. Each user edits a local text file. The system automatically detects edits, broadcasts them to other users using POSIX message queues, and merges updates using a Last-Writer-Wins (LWW) CRDT conflict-resolution strategy. A shared memory registry is used for peer discovery, and the terminal UI displays active users, file state, and change notifications. The system design is structured according to the assignment's three-part rubric.

1 Part 1 — User Creation & Local Editing with Automatic Change Detection

1.1 Program Execution

The system is launched as:

```
./editor <user_id>
```

The main function in `control.cpp` validates the user ID and stores it globally in `gUID`. Signal handlers are registered so that interrupts trigger `cleanExit`.

1.2 User Registration & Discovery

Shared memory is allocated and mapped by `openReg`. The registry is stored in `gReg` and represents the global user list. A new user claims a slot using `regUser`, storing:

- `uid`
- message queue name
- active flag

Up to five concurrent users are supported. Active users are discovered by scanning the registry, particularly at broadcast time and when updating the display.

1.3 Local Document Lifecycle

Each user maintains a local text document named:

```
<user_id>_doc.txt
```

`verifyLocalDoc` ensures the file exists and contains the shared initial text. Two internal states are maintained:

- `doc_lines`: authoritative merged state
- `observed_lines`: last version seen on disk

Initial display is handled by `dispDocUpdatesSimp`.

1.4 Editing Workflow

Users edit the file in any editor. The system periodically checks file metadata (`stat`) to detect saves. When a change is detected, `diffLinesMakeUpdates` compares the new file state with `observed_lines`, generating one or more `Update` objects containing operation type, line number, affected columns, removed text, inserted text, timestamp, and originating user.

Detected updates are appended to:

- `local_unmerged` for later merging
- `outgoing_bufferfer` for future broadcasting

1.5 Terminal Display

`dispDocUpdatesSimp` re-renders the document, showing:

- updated lines
- active user list
- recent edit indicators based on `gPrevEdits`
- notifications stored in `g_recent_notifications`

1.6 Update Object Creation

The `Update` structure stores: operation type, line number, column bounds, previous text, new text, timestamp, and user ID. `updateClassification` labels the type for display, and `dispBoundesup` formats text for clarity.

1.7 Acceptance Criteria (Part 1)

- User starts program via command line with unique user_id.
- User joins and discovers peers via shared memory registry.
- Local edits detected automatically without user interaction.
- Update objects correctly encode changed regions.
- Terminal display updates live and highlights edited lines.

2 Part 2 — Broadcasting Local Updates via Message Passing

2.1 Message Queue Setup

Each user creates its own POSIX message queue with name:

`/mq_<user_id>`

This name is stored into the registry. `listenerThreadFunc` continuously reads from the queue and stores incoming updates into `gRingRecv`, a lock-free ring buffer.

2.2 Broadcasting Pipeline

Local update objects accumulate in `outgoing_bufferfer`. When the number of pending updates reaches:

$$BROADCAST\_BATCH\_SIZE = 5$$

each update is serialized and sent to all other active users using `sendRetriesUpdatesToQ`.

2.3 Concurrency Model

- Main thread detects local edits, buffers updates, and triggers merges.
- Listener thread receives updates from message queue and buffers them.
- `gRingRecv` implements single-producer/single-consumer lock-free communication.
- Registry access is read-mostly and uses atomic flags.

2.4 Dynamic Users

Broadcast logic enumerates registry entries at send time. Newly joined users immediately begin receiving future updates.

2.5 Acceptance Criteria (Part 2)

- Each user has a dedicated message queue.
- System broadcasts updates after every five edits.
- Listener thread collects updates continuously.
- No locks are required in update transmission.

3 Part 3 — Listening, Merging, and Synchronization using CRDT

3.1 Continuous Listening

Incoming messages are pushed to `gRingRecv`. The main loop extracts updates from the ring and places them into `recv_unmerged`.

3.2 Merge Trigger

When:

$$|\texttt{local\_unmerged}| + |\texttt{recv\_unmerged}| \geq BROADCAST\_BATCH\_SIZE$$

the system merges all pending updates.

3.3 Conflict Detection and Resolution

Conflicts are detected using `collisionUpdates`, which checks:

- Same line number
- Overlapping or matching column regions

`updatesAonB` applies Last-Writer-Wins based on timestamp, with user ID lexicographic comparison as a deterministic tiebreaker. `crdtMerge` removes losing updates and outputs the winning set.

3.4 Applying Merged Updates

`applyLineUpdates` updates `doc_lines`. The file is written using `writeLinesFile`, then `observed_lines` is aligned so local writes do not trigger new change detection. Notifications are displayed through `dispDocUpdatesSimp`.

3.5 Convergence Guarantee

Because all users evaluate the same merge rules on the same operations, all replicas converge to identical final documents.

3.6 Acceptance Criteria (Part 3)

- Listener thread buffers remote updates correctly.
- CRDT merge resolves conflicts deterministically.
- Local document converges to global state.
- Terminal UI reflects merged data.
- No mutexes or locks required.

4 Runtime Sequence

1. User launches program and joins registry.
2. Local document is initialized and displayed.
3. Listener thread begins receiving updates.
4. User edits document; updates detected and buffered.
5. After every 5 edits, updates are broadcast.
6. Received updates accumulate into merge buffer.
7. Merge is performed; final state applied and displayed.
8. On exit, queue and registry entries are cleaned up.

5 Data & Interfaces

Update objects store mutation information. **ShmRegistry** stores user session and queue data. Filesystem, POSIX MQ, POSIX SHM, and signals form the external interfaces.

6 Error Handling & Resilience

Queue creation failures, message oversize conditions, and shared memory exhaustion are handled gracefully. The system avoids echo loops by syncing observed state after merges.

7 Performance & Scalability

Polling intervals control responsiveness. Batching reduces communication overhead. Lock-free ring buffers avoid blocking contention.

8 Testing Strategy

Testing covers editing, broadcasting, concurrency, conflict resolution, and final convergence across multiple terminals.

9 Limitations & Future Enhancements

Clock skew may affect LWW ordering. Multi-line structural edits are treated line-by-line. The system assumes single-machine deployment and does not handle UTF-8 multi-byte alignment fully.

10 Traceability Matrix

Requirement	Implementation Reference
User identification	main, gUID, signalHandler
Shared registry management	openReg, regUser, deregSlot, ShmRegistry
Local file monitoring	stat polling, diffLinesMakeUpdates
Message passing	createSelfQ, sendRetriesUpdatesToQ, listenerThreadFunc
CRDT resolution	collisionUpdates, updatesAonB, crdtMerge, applyLineUpdates

Conclusion

The system successfully implements collaborative editing using lock-free coordination, message-based synchronization, and deterministic CRDT merging to ensure eventual consistency across all user replicas.