DESIGNDOC

SyncText: A CRDT-Based Collaborative Text Editor

Course: CS69201 - Computing Lab
Semester: Autumn 2025
Student: Narendra Kumar(25CS60R18)
Institute: IIT Kharagpur

1 System Architecture

1.1 High-Level Design Overview

The **SyncText editor** is a distributed, lock-free collaborative text editing system that allows multiple users to edit the same document simultaneously. Each user process maintains a local replica of the file and synchronizes with others through shared memory and message queues, ensuring real-time, conflict-free updates.

When a user executes:

```
./edtr <user_id>
```

the program registers itself, creates its own message queue, monitors file changes, and exchanges update operations (`UpOp`) with peers. Synchronization occurs every 5 operations globally, guaranteeing eventual consistency.

1.2 Major Components and Interactions

Shared Memory (ShReg): Maintains active user registry and a global operation counter. **Message Queues (POSIX MQ):** Enable lock-free, asynchronous exchange of update operations. **File Monitor Thread:** Detects local changes and creates update objects. **Listener Thread:** Receives and buffers operations from remote users. **Batch Thread:** Merges and synchronizes documents after every 5 operations. **Leader Process:** User with smallest UID writes the central master file.

1.3 Key Data Structures

- **UpOp:** Represents a single change (insert, delete, replace) with timestamps and user ID.
- **UserReg:** Holds each user's UID, queue name, and active status.
- **ShReg:** Shared registry of all users and global operation counter.
- **Vectors:** `locOps` for local edits and `recOps` for received updates.

2 Implementation Details

2.1 Change Detection

Implemented via periodic file monitoring using `stat()`. Every 2 seconds, the file's last modification time is checked. If modified:

1. File is re-read.
2. Old and new versions are compared line-by-line.
3. A diff operation creates a set of `UpOp` objects with timestamps.

Detected changes are stored locally and contribute to the global operation count.

2.2 Message Queues and Shared Memory

Each user creates a POSIX message queue named `/queue_<user_id>`. Message queues are used for broadcasting serialized `UpOp` objects to all other users. Shared memory (`/synctext_registry`) contains the user registry and the atomic `globalOpCount`. No locks are used—synchronization relies purely on atomic operations.

2.3 CRDT Merge Algorithm

Conflict resolution is based on the **Last-Writer-Wins (LWW)** rule:

1. Collect all local and received operations.
2. Sort by timestamp (earliest first).
3. Detect overlapping edits (same line, overlapping columns).
4. The operation with the latest timestamp prevails.
5. Non-conflicting edits are all applied.

This ensures commutativity, associativity, and idempotency—the foundation of CRDT behavior.

2.4 Thread Architecture

- **Main Thread:** Initializes environment and spawns worker threads.
- **fmon():** Monitors local file changes.
- **listen():** Continuously receives operations from peers.
- **refusers():** Keeps user registry fresh by probing queues.
- **batch():** Every 5 operations (global), merges and synchronizes all files.

Threads communicate through shared atomic variables and vectors—never through locks.

3 Design Decisions

3.1 Lock-Free Operation

All inter-process coordination is lock-free:

- Atomic counters manage concurrent updates.
- Message queues provide non-blocking communication.
- CRDT semantics replace traditional locking mechanisms.

3.2 Timestamps and Ordering

Millisecond-precision timestamps determine which change dominates. In case of equal timestamps, lexicographic order of `user_id` acts as a tie-breaker.

3.3 Leader Election

The lexicographically smallest user ID becomes the leader and writes the canonical `centeralFile.txt`. This prevents duplicate master writes.

3.4 Trade-offs

- **Polling:** Portable but introduces a small delay (vs. inotify).
- **Atomic-Only Sync:** Simplifies concurrency, requires cautious design.
- **Operation Threshold:** Reduces message overhead but delays sync slightly.

4 Challenges and Solutions

- **Global Synchronization:** Different users detecting edits asynchronously caused inconsistent merges. $\Rightarrow$ Introduced a global atomic counter; merges occur only after 5 total operations.
- **Conflict Resolution:** Overlapping edits created diverging states. $\Rightarrow$ Used LWW timestamps and UID priority for deterministic resolution.
- **Duplicate Updates:** Repeated broadcasts re-applied the same edit. $\Rightarrow$ Added a `seenOps` set to ignore previously processed operations.
- **File Race Conditions:** File writes triggered false change detection. $\Rightarrow$ Added atomic flag `supMon` to temporarily suspend monitoring.
- **Debugging Multi-Process Behavior:** Hard to trace concurrency visually. $\Rightarrow$ Added time-stamped console logs and tested with multiple terminals.

5 Conclusion

The final implementation achieves a lock-free, CRDT-based collaborative editing system supporting up to five concurrent users. It ensures eventual consistency through timestamp-based merging and atomic synchronization, accurately reflecting the behavior of modern editors like Google Docs. All users' local copies converge to the same state without using any locks or central coordinator.