SyncText Collaborative Editor
Design Document

CRDT-based Lock-Free Collaborative Editor

November 11, 2025

Contents

1 System Architecture **2**

1.1 High-Level Design Overview 2

1.2 Major Components and Interaction 2

1.3 Key Data Structures 2

2 Implementation Details **3**

2.1 Part 1: User Creation & Local Editing with Automatic Change Detection 3

2.2 Part 2: Broadcasting Local Updates via Message Passing 3

2.3 Part 3: Listening, Merging, and Synchronization using CRDT 4

3 Design Decisions **4**

4 Challenges and Solutions **4**

4.1 Major Challenges 4

4.2 Solutions Implemented 4

5 Conclusion **5**

1 System Architecture

1.1 High-Level Design Overview

SyncText is a multi-user collaborative text editor that ensures eventual consistency without locks using CRDT principles. Each user maintains a local copy of the document. Changes are detected automatically and broadcast to other users via shared memory message queues. Conflicts are resolved using a Last-Writer-Wins (LWW) strategy.

1.2 Major Components and Interaction

- **User Process:** Each user runs an instance of the editor with a unique user ID.
- **Shared Memory Registry:** Tracks all active users and their FIFO/message queue names.
- **Per-User Message Queue:** Each user has a lock-free circular buffer in shared memory for incoming updates.
- **Local Document:** User-specific file (`<user_id>_doc.txt`) that is monitored for changes.
- **Listener Thread:** Continuously receives incoming updates from the user's message queue.
- **Watcher/Main Thread:** Monitors local document changes, accumulates local updates, and triggers CRDT merge.

1.3 Key Data Structures

- `Registry` – stores user slots:

```
    struct Slot {
        char user[USER_LEN];
        char fifo[FIFO_LEN];
        uint64_t ts;
        uint32_t used;
        uint32_t pad;
    };
    struct Registry {
        uint32_t magic;
        uint32_t max_users;
        uint32_t version;
        uint32_t reserved;
        Slot slots[MAX_USERS];
    };

```

- `GlobalMQ` – shared message queues:

```
    struct ShmMsgQueue {
        size_t head;
        size_t tail;
        char msgs[SHM_MQ_CAP][SHM_MQ_MSG_SIZE];
    };
    struct GlobalMQ {
        ShmMsgQueue q[MAX_USERS];
    };

```

- `Update` – encapsulates a document change:

```
struct Update {
    string op; // insert/delete/replace
    int line_no;
    int col_start, col_end;
    string old_content;
    string new_content;
    string timestamp;
    string user;
};

```

2 Implementation Details

2.1 Part 1: User Creation & Local Editing with Automatic Change Detection

- **Program Execution:** `./editor <user_id>`.
- **User Registration:** Registers user in shared memory registry using atomic operations.
- **Local Document:** Each user maintains `<user_id>_doc.txt`, initialized with the same content.
- **File Monitoring:**
  - Tracks `last_mtime` of the file using `stat()`.
  - Periodically polls (every 2 seconds) to detect modifications.
  - Reads updated content and compares line by line with previous content.
- **Update Object Creation:** For each line change, create `Update` objects capturing:
  - Operation type (insert/delete/replace)
  - Line and column numbers
  - Old/new content
  - Timestamp
  - User ID
- **Terminal Display:** Clears terminal and displays updated document and active users.

2.2 Part 2: Broadcasting Local Updates via Message Passing

- **Message Queues:** Each user has a circular buffer in shared memory (`GlobalMQ`) as a message queue.
- **Multi-threading:**
  - Main thread monitors local file and accumulates updates.
  - Listener thread continuously reads the user's queue for incoming updates.
- **Broadcasting:** After accumulating N=5 operations:
  - Serialize each `Update` object into JSON-like string.
  - Send the updates to all other users' message queues.
- **Receiving Updates:** Listener thread pushes incoming updates into a local inbound buffer for merging.

2.3 Part 3: Listening, Merging, and Synchronization using CRDT

- **Conflict Detection:** Two updates conflict if they affect the same line and overlapping columns.
- **CRDT LWW Merge Algorithm:**
  - Collect local and received updates.
  - Resolve conflicts using timestamps (LWW). Tie-breaker: smaller user_id wins.
  - Apply winning updates and non-conflicting updates to local document.
- **File Update and Display:** Merged content is written to the local document file and displayed in the terminal.
- **Periodic Synchronization:** Merge is triggered after every N=5 operations or when new remote updates arrive.

3 Design Decisions

- **Lock-Free Operation:** Used atomic operations for registry slots and message queues. No mutexes.
- **Batching Updates:** Reduces IPC overhead and allows efficient merges.
- **CRDT Strategy:** LWW provides deterministic conflict resolution.
- **File-Based Editing:** Allows users to use standard editors (vim, nano, etc.) without modifying code.
- **Separate Queues:** Each user has their own queue, preventing contention and simplifying merging.

4 Challenges and Solutions

4.1 Major Challenges

- Ensuring updates are not lost under concurrent writes to message queues.
- Conflict detection for overlapping edits.
- Real-time display updates without flickering.
- Maintaining lock-free correctness across multiple threads and processes.

4.2 Solutions Implemented

- Used atomic head/tail pointers in circular buffers.
- Implemented CRDT merge to handle overlapping changes deterministically.
- Cleared terminal and redrew document for smooth display updates.
- Extensively logged state and verified atomic operations correctness during debugging.

5 Conclusion

SyncText demonstrates a fully lock-free, multi-user collaborative editor using CRDT principles. The combination of atomic shared memory operations, LWW-based conflict resolution, and real-time file monitoring ensures correctness, concurrency, and eventual consistency across all users.