Design Document

SyncText - A CRDT-Based Collaborative Text Editor

Course: Computing Lab (CS69201)
Author: Harsh Jain
Date: November 10, 2025

Contents

1 System Architecture

1.1 High-Level Design Overview

The Collaborative Lock-Free Text Editor is a distributed, multi-process system that allows multiple users to edit a shared document concurrently without using locks or centralized coordination. Each user runs an independent process that communicates with others using **POSIX shared memory** and **named pipes (FIFOs)**.

The system achieves eventual consistency through an algorithm inspired by **CRDT (Conflict-Free Replicated Data Type)** with a *Last Writer Wins* merge strategy.

Each editor instance performs the following roles:

- Monitors its local document for changes.
- Broadcasts detected changes to other users through FIFO pipes.
- Listens for updates from others and merges them into its local file.

All synchronization between users happens using atomic operations and snapshot-based data structures, ensuring that the system remains fully lock-free.

1.2 Major Components and Interaction

The architecture consists of five core modules that interact as shown below:

- **Shared Memory Registry:** Stores all active user IDs in a shared memory region (`/sync_registry`). It allows new users to discover others and enables broadcasting without prior connections.
- **Named Pipes (FIFOs):** Each user creates a named pipe (for example, `/tmp/pipe_userA`) to receive incoming updates. Every broadcasted update is sent through all registered FIFOs except the sender's.
- **Change Detection Engine:** Continuously monitors the user's local file (for example, `userA_doc.txt`) using the file modification timestamp. Whenever a change is detected, it identifies which lines and columns changed and encapsulates the information into an update object.
- **Listener Thread:** Runs concurrently in the background for each user. It listens for incoming updates via the FIFO and appends them to a received updates buffer using copy-on-write semantics.
- **Merge Engine:** Integrates updates from both local and remote buffers. It uses a timestamp-based conflict resolution strategy (Last Writer Wins). Once enough updates accumulate, it merges and rewrites the file.

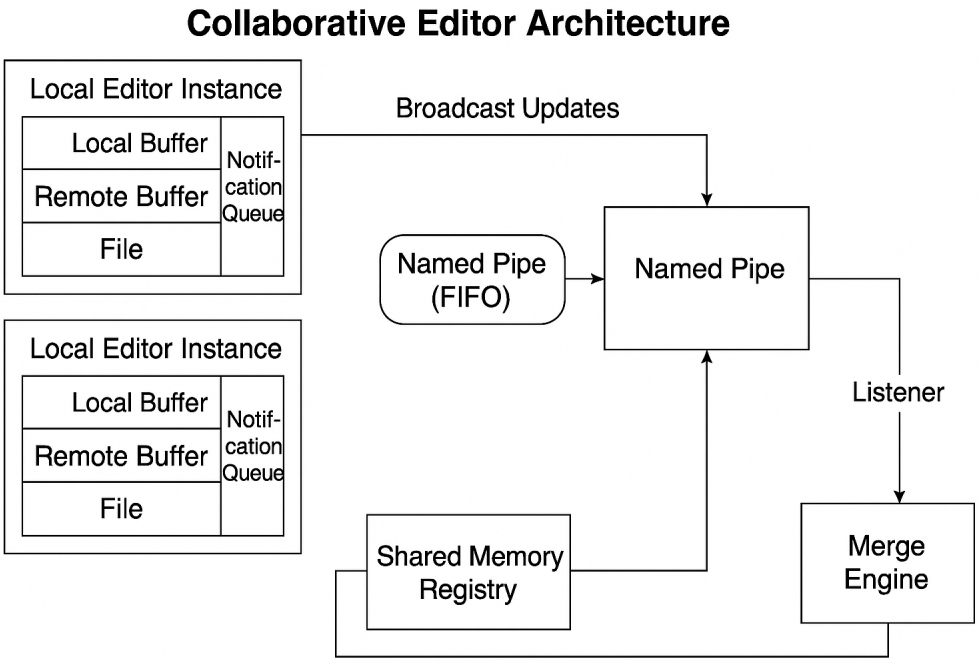

Figure 1: High-level System Architecture of Collaborative Lock-Free Editor

1.3 Key Data Structures

- **Registry:** A shared structure holding all active users:
  - `user_count` – total number of registered users.
  - `users[]` – array of user IDs.
- **UpdateObject:** Represents an atomic change operation:
  - Line number and affected column range.
  - Old and new content.
  - Timestamp (`ts`) and user ID (for conflict resolution).
- **Shared Pointers with Atomic Snapshots:** Two shared vectors – one for local updates (`local_ptr`) and one for received updates (`recv_ptr`) – are implemented using `std::shared_ptr` and `atomic_thread_fence`. These allow concurrent readers and writers without locks.

2 Implementation Details

2.1 Change Detection (File Monitoring Approach)

Each user continuously monitors their own text file for changes using the file's last modification timestamp, retrieved via the `stat()` system call. When the timestamp differs from the last recorded one, the file is re-read, and its content is compared line by line with the previous version.

The change detection process identifies:

- The line number(s) that changed.

- The start and end positions of the modification.
- The replaced and inserted substrings.

A compact change description (the update object) is created and added to the local buffer. This approach avoids polling the entire file continuously and detects modifications efficiently.

2.2 Message Queues and Shared Memory Structure

Instead of POSIX message queues, the system uses **Named Pipes (FIFOs)** as unidirectional message channels for each user. Each editor instance:

- Creates a personal FIFO at `/tmp/pipe_<user_id>` to receive updates.
- Broadcasts updates by writing to the FIFOs of all other registered users.

The list of all active users is stored in the shared memory registry, implemented via:

- `shm_open()` – creates or opens the shared memory region.
- `mmap()` – maps the region into process memory space.

This setup ensures that users can dynamically discover each other without direct connections.

2.3 CRDT Merge Algorithm (Last Writer Wins)

The merge logic is inspired by CRDTs, ensuring that all users converge to the same final document state, regardless of update order.

Each update object includes:

- Epoch timestamp (`ts`).
- User ID (to break ties deterministically).

During merging:

1. All updates (local and remote) are combined.
2. Conflicting edits (same line and overlapping ranges) are resolved by:
    - Keeping the update with the higher timestamp.
    - If timestamps are equal, choosing the one from the lexicographically smaller user ID.
3. Updates are applied to the document in reverse order of column indices to preserve offsets.

This deterministic rule ensures **eventual consistency** – all users will reach the same file content after all updates propagate.

2.4 Thread Architecture and Communication

The program uses two main threads per user:

- **Main Thread:**
  - Monitors file changes using timestamps.
  - Detects differences and generates update objects.
  - Broadcasts updates to other users' FIFOs.
  - Periodically triggers merges when the threshold is reached.
- **Listener Thread:**
  - Opens its own FIFO in read-only mode.
  - Receives updates from other users.
  - Appends received updates to the remote buffer using copy-on-write.
  - Calls the merge engine when new data arrives.

Inter-thread communication is indirect – both threads modify shared buffers (`local_ptr`, `recv_ptr`) atomically, without explicit locks or condition variables.

3 Design Decisions

3.1 Lock-Free Operation and Rationale

Traditional synchronization using mutexes or semaphores was avoided to eliminate the risk of deadlocks and reduce contention. Instead, the system adopts a **lock-free snapshot model** based on the following principles:

- Writers always work on a private copy of data.
- Once updates are ready, a new pointer is atomically published.
- Readers never block because they access consistent snapshots.

This ensures that even under heavy concurrent operations, no thread waits or interferes with others.

3.2 Trade-offs Made

- **Pros:**
  - No locking overhead.
  - High concurrency and responsiveness.
  - Safe for multiple writers and readers.
- **Cons:**
  - Slightly higher memory usage due to snapshot copies.
  - Updates may appear delayed by up to one merge cycle.
  - Not ideal for very large files due to repeated copying.

3.3 Implementation Choices

- Used **POSIX Shared Memory** for cross-process user registry (simpler and faster than sockets).
- Chose **FIFOs** instead of message queues for simplicity and easy cleanup.
- Implemented merging and broadcasting at the **line level** for better performance.
- Used **timestamp-based resolution** for deterministic conflict handling.
- Designed a **threshold-based merge trigger** to batch small updates, reducing overhead.

4 Challenges and Solutions

4.1 1. Detecting Changes Without Polling the Entire File

Challenge: Continuously re-reading the file to detect edits was inefficient.

Solution: Used the `stat()` system call to track only the file's modification timestamp. When it changes, the file is re-read once, and differences are computed line by line.

4.2 2. Preventing Deadlocks and Race Conditions

Challenge: Concurrent access to update buffers could lead to inconsistent states.

Solution: Replaced traditional locks with atomic fences and shared pointer snapshots, allowing multiple writers without blocking and ensuring memory visibility across threads.

4.3 3. Handling FIFO Write Failures

Challenge: If a receiver process exited unexpectedly, `write()` to its FIFO would block or fail.

Solution: Used non-blocking writes (`O_NONBLOCK`) and error-checked them gracefully without crashing the sender.

4.4 4. Debugging Concurrent Behavior

Challenge: Simultaneous updates and merges made debugging order of operations difficult.

Solution: Added color-coded logs (cyan for registration, green for received updates, blue for local changes, magenta for merges) to visually differentiate threads and actions in real time.

4.5 5. Synchronizing Without a Central Server

Challenge: Without a central coordinator, it was difficult to ensure users discovered each other.

Solution: Implemented a shared memory registry that automatically maintains a list of active users. This registry acts as a lightweight directory service for broadcast targets.

4.6 6. Ensuring Eventual Consistency

Challenge: Simultaneous edits could cause diverging document states.

Solution: Implemented a deterministic merge algorithm based on timestamps and user IDs, ensuring that all users always converge to the same state.

5 Conclusion

This project demonstrates how collaborative editing can be achieved using only low-level UNIX system programming techniques. By combining shared memory, named pipes, and atomic operations, the system maintains synchronization between multiple users without any locking mechanism or central server.

It successfully showcases:

- Practical use of shared memory for inter-process coordination.
- FIFO-based asynchronous message passing.
- Lock-free, copy-on-write synchronization.
- Deterministic CRDT merging for eventual consistency.

Outcome: All users can edit concurrently, view real-time updates, and converge on the same document without data loss or deadlocks – purely using OS-level constructs.