DS Project: Collaborative text editor

Hitesh Goel (2020115003)
Shreyansh Aggarwal
Anirudh Kaushik (2020111015)

21 April 2024

Introduction

This project presents the design and implementation of a collaborative text editor utilizing a Sequence-based Conflict-free Replicated Data Type (CRDT) as its underlying data structure. The system is developed using Flask and Javascript for the web development part, with a custom CRDT sequence implementation. The CRDT sequence is designed to handle concurrent edits from multiple users, ensuring eventual consistency across distributed replicas.

The key contribution of this work lies in the development of a robust CRDT sequence, facilitating real-time collaborative editing while preserving document integrity. The sequence CRDT offers efficient methods for adding and removing elements, as well as merging concurrent updates to maintain a consistent state across distributed replicas. Additionally, the project explores the implementation of a two-phase CRDT, although this feature is not utilized in the demonstration.

Through a demonstration, we showcase the effectiveness of our CRDT implementation in enabling seamless collaboration among users. The system allows multiple users to concurrently edit a shared document, with changes propagated and synchronized across all replicas in a consistent manner.

Literature review

Collaborative text editing systems have been the subject of extensive research and development efforts over the past few decades, driven by the need to enable seamless collaboration among users in distributed environments.

Operational Transform

Until the mid-2000s, Operational Transform was the predominant approach used in collaborative text editing systems. OT algorithms were developed to synchronize changes made by multiple users to a shared document in a distributed environment. Systems such as Google Docs, Etherpad, and Microsoft Office Online relied on OT to handle concurrent edits and resolve conflicts.

Conflict-free Replicated Data Type

In recent years, CRDTs have emerged as an alternative approach to handling concurrency control in collaborative editing systems. Unlike OT, which requires complex conflict resolution mechanisms, CRDTs offer a simpler and more efficient way to achieve eventual consistency across distributed replicas.

OT algorithms require complex conflict resolution mechanisms and are susceptible to scalability and performance issues, especially in large-scale distributed systems. In contrast, CRDTs offer a simpler and more efficient approach to handling concurrent edits, with strong theoretical guarantees of eventual consistency. CRDTs eliminate the need for explicit conflict resolution by ensuring that updates **commute and are idempotent** with each other, leading to a more robust and scalable solution for collaborative text editing.

Methodology

Description of the overall architecture

- Frontend: The frontend of the system is developed using Flask, a lightweight web framework for Python. Flask serves as the interface through which users interact with the collaborative text editor, providing features such as document editing, user authentication, and real-time synchronization.

- Backend: The backend consists of server-side logic responsible for handling client requests, managing user sessions, and maintaining the state of the shared document. The backend utilizes a custom Conflict-free Replicated Data Type (CRDT) sequence implementation to manage concurrent edits and ensure eventual consistency across distributed replicas.

CRDT

The implemented CRDT sequence is designed to represent an ordered list of elements, allowing for efficient handling of concurrent edits in a distributed environment. The Sequence CRDT employs three main operations: **add, remove, and merge.** The add operation appends a new element to the sequence, while the remove operation marks an element for deletion. The merge operation combines the state of two sequences to reconcile concurrent updates.

A second CRDT was also implemented, known as the "two-phase set CRDT." This was done out of interest in the subject and for learning purposes. However, this particular architecture has not been utilised for the demonstration.

Performance

We will present a live demo during the in-person evaluations. Thus, we leave out tests which show two people updating the document, conflict-resolution etc for the live demo. We perform 3 different kinds of tests for latency and performance:

- Average insert and delete latency (performed 3 times)
- Consecutive insertion latency (100)
- Consecutive deletion latency (100)

0.1 Test No. 1

Insert Request	**Latency (ms)**
1	14.2
2	15.4
3	16.4
Average	**15.33**

Table 1: Latency for insert requests

Remove Request	**Latency (ms)**
1	10.1
2	14.5
3	17.6
Average	**14.067**

Table 2: Latency for remove requests

0.2 Test No. 2

Insert Request	**Latency (ms)**
1	13.89
2	15.39
3	16.2
Average	**15.16**

Table 3: Latency for insert requests

Remove Request	**Latency (ms)**
1	13.1
2	21.6
3	15.6
Average	**20.1**

Table 4: Latency for remove requests

0.3 Test No. 3

Insert Request	**Latency (ms)**
1	18.1
2	21.4
3	23.2
Average	**20.9**

Table 5: Latency for insert requests

Remove Request	**Latency (ms)**
1	12.6
2	17.4
3	22.5
Average	**17.5**

Table 6: Latency for remove requests

0.4 Consecutive insertion test

Request	**Latency (ms)**
1	16
10	268.8
25	280.2
50	314.1
75	448.9
100	651.5

Table 7: Latency for consecutive insert requests

0.5 Consecutive Deletion Test

Request	Latency (ms)
1	207
10	207.8
25	341.4
50	609.9
75	841.8
100	1049.9

Table 8: Latency for consecutive delete requests

Conclusion

- We are able to achieve real-time editing performance with average latency of 15ms for insertion and 14ms for deletion.
- On average deletion was faster than insertion for general use case.
- We noticed that the latency increases with increasing number insertions and deletions due to our CRDT being a linked list type structure with an insert complexity of $O(n)$.
- The latency increases considerable with increasing insertion and deletion. The reason for the latency of consecutive deletions being higher than insertions is because we perform the deletion test immediately after consecutive insertion test.
- The deleted elements are tomb-stoned instead of being removed due to which array size increases with insertions and does not reduce with deletions, thus we start with an array of size 100.
- This is evident from the fact that the latency of the first deletion in the consecutive deletion test is 200ms.
- To address such issues, a data structure with a better average time complexity can be used as a CRDT, such as a Grow-only Tree which is based on a B-Tree without deletions. This will allow for a uniform latency across different edits and will be independent of the total number of edits performed.