ANALYSIS OF RESEARCH PAPER

Research papers used:-

1)https://hal.archives-ouvertes.fr/hal-00921633/document - **LSEQ**

2) https://hal.inria.fr/inria-00432368/document - **LOGOOT**

Logoot and LSEQ adopt CRDT(Conflict Free Replica Data Types) approach to solve the problem of Peer to Peer collaborative code editors.

CRDT is a data structure for achieving consistent data between replicas of data stored on different peers(clients) without any kind of coordination (e.g. transformation) between the replicas and where it is always mathematically possible to resolve inconsistencies that might come up like commutativity and idempotency of data.

In CRDT modifications produced locally are re-executed on remote replicas.

Logoot Model uses an identifier which is unique for every character present on the editor and composed of (pos, hs) where “hs” is the logical clock/counter of the client and “pos” is a list of position identifiers;

Pos : p1 p2 p3 p4 …. pn which designate a path on a tree,

Where p1 = (position, site) , position is an integer which is chosen from 0 to a fixed base number like 32 or 64. Site is the unique site id of the client which generated the operation.

The Code Editor is basically a 2D array of characters and so the CRDT stores the document as a 2D array of unique identifiers for every character.

Insertion and deletions can be done in Logarithmic time using binary search.

Collaborative Code editor can be built using a central relaying server. But that has many disadvantages like :-

a) **Single Point of failure -** If the server goes down , users can not make any more connections with other users.
b) **High Costs to scale -** a single central server performing all requests of connections and message transmission is not a sustainable model.
c) **High Latency between users located in the same geographic location -** Even if users might be sitting in the same room, they would still require to connect to the central server which might be located in another continent.

These problems were highlighted and aimed to solve in the Logoot paper.

Logoot describes a Peer to Peer based architecture where every user is connected to an upper bound of users, although the initial connection requires the use of a central server.

LSEQ improves on the LOGOOT approach by using an adaptive allocation strategy for sequence CRDT resulting in a sub-linear upper bound in its spatial complexity whatever is the editing behaviour.

The Line positions on every level of the tree are doubled , so if the base level has 64 positions then the next level will have 128, and then next level will have 256 and so on.

Also the allocation strategy of choosing a position randomly in LOGOOT is modified by LSEQ by randomly choosing a position within boundary plus or boundary minus of left or right position. So if a position has to be chosen between p and q, and a fixed boundary , of 10 for example, has been defined, then if we randomly choose the left position then the possibilities would be p+1, p+2, till minimum of (q,p+boundary) and same for randomly choosing right position.

Improvements we have been able to perform :-

a) **Automatic Network Balancing** : We use an upper bound of Logarithmic complexity on the number of peers a peer or client can connect to.
b) **Peer Discovery** - We maintain a list of peers or clients active on the network, so if a peer loses connection it can join the code editor again with the help of a peer list maintained on the network.
c) Logarithmic Insert and Delete using a 2D document model - Since Code Editor is a 2D array of lines and characters, we also made the CRDT structure store a 2D document model of position identifiers for the characters.
d) **Simple Code Highlighting** - Keyword highlighting of reserved words of programming languages just for better visual experience.

ALGORITHM / DESCRIPTION OF THE WORK

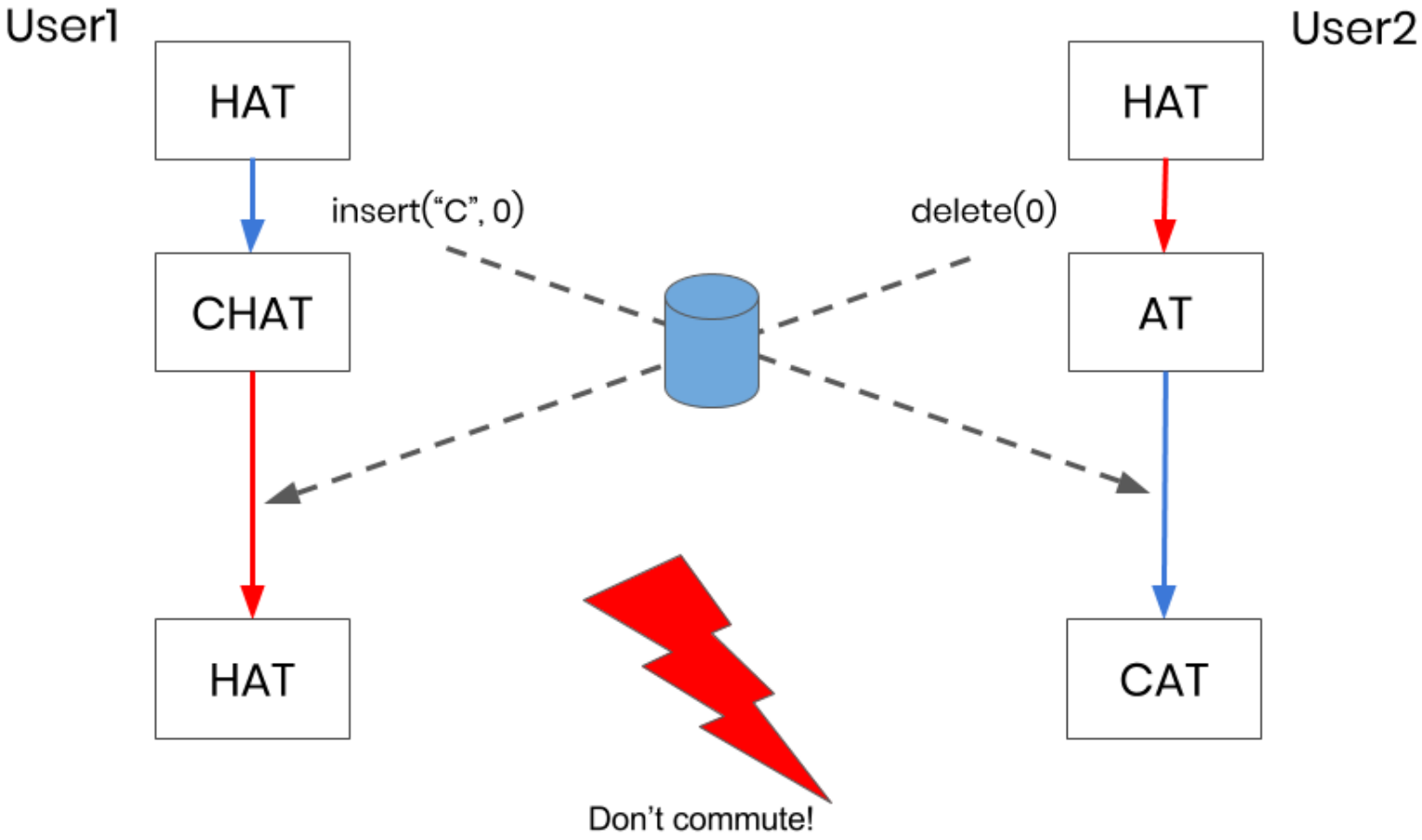
1. **Live Code Editor -**

   A Real Time Collaborative code editor built with CRDTs(conflict free replica data type) and WebRTC which is a peer to peer based framework.

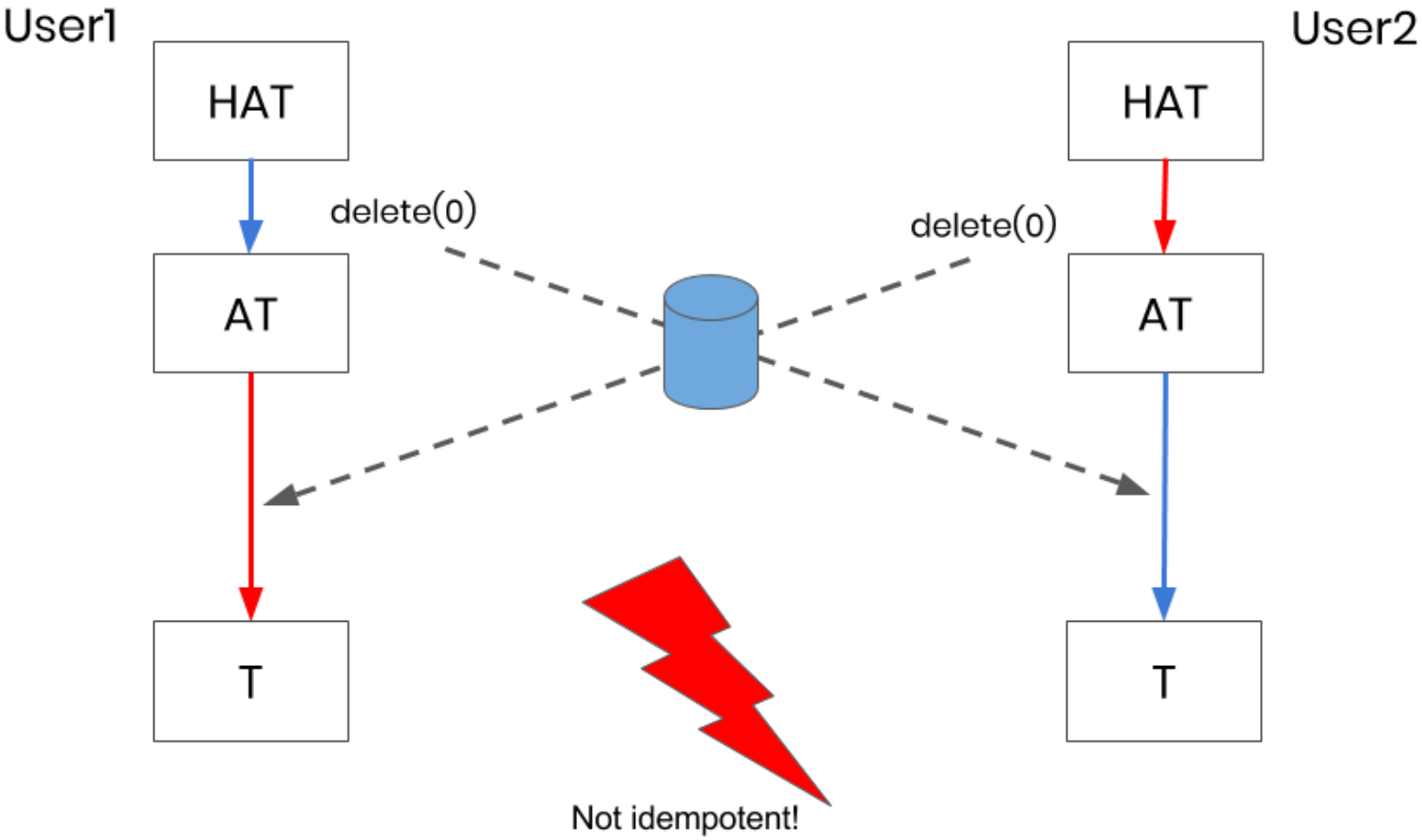
   In a non-real time environment, users perform insert, delete and edit operations on a file either on **Lock** based approach or **Merge** based approach.

   In **Lock** based approach , only one user can perform operations.

   Whereas in **Merge** based approach, users have local copy of same files and they perform operations locally and then upload over a network or software which tries to look for conflicts and either merges those conflicts automatically or informs the user to take action.



Simultaneous insertion and deletion produce different results



Duplicate deletion operations are not idempotent

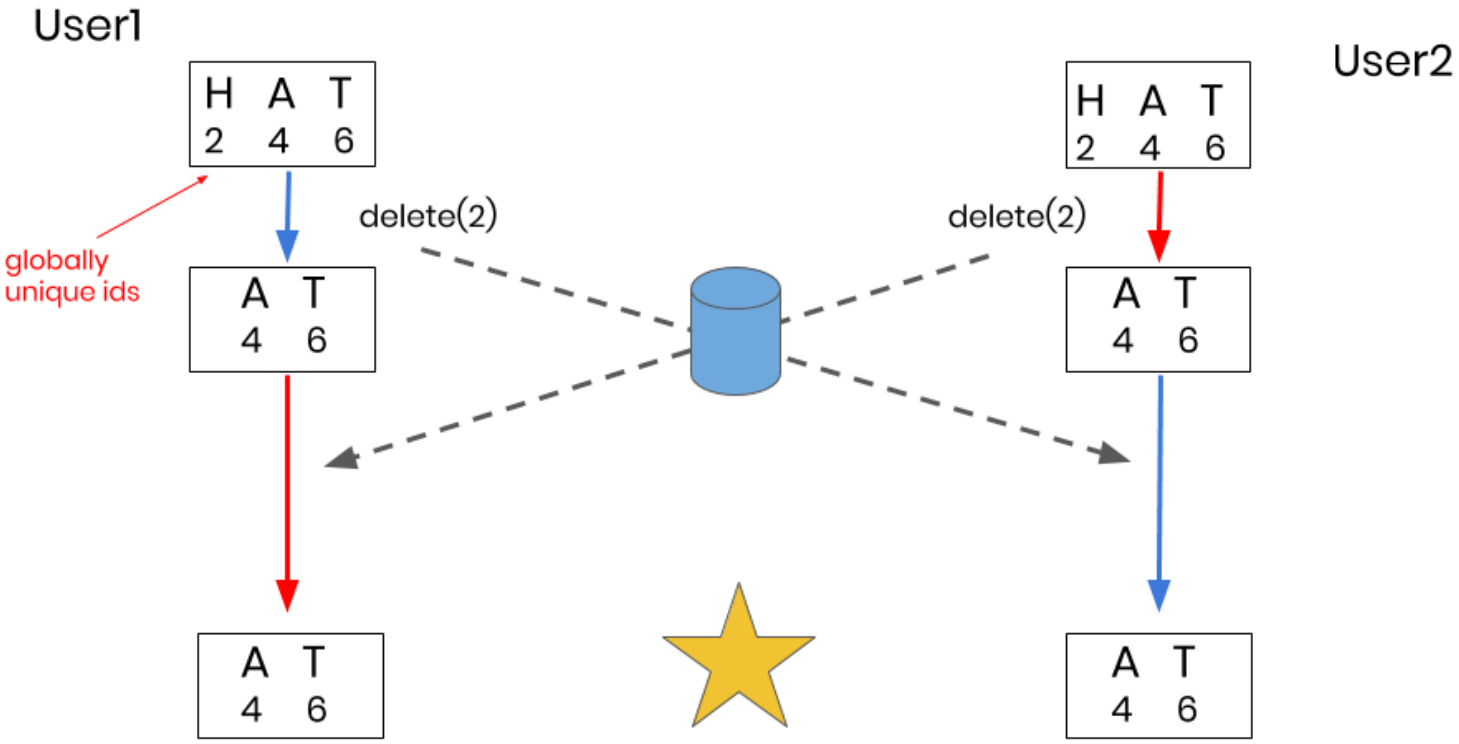
For collaborative code editors either of the approaches is not sustainable.So research was done and CRDT was born.

A CRDT is primarily a strategy for achieving consistent data between replicas of data without any kind of coordination (e.g. transformation) between the replicas. Since a text document requires the characters to be in a specific order, the type of CRDT that we used is usually categorized as a **Sequence CRDT**.

To use CRDTs specifically for a collaborative text editor, there are a couple critical requirements like :-

Globally Unique Characters

The 1st requirement is that each character object must be globally unique. This is achieved by assigning **Site ID** and **Site Counter** properties whenever a new character is inserted. Since the **Site Counter** at each site increments whenever inserting or deleting a character, we ensure the global uniqueness of all characters.

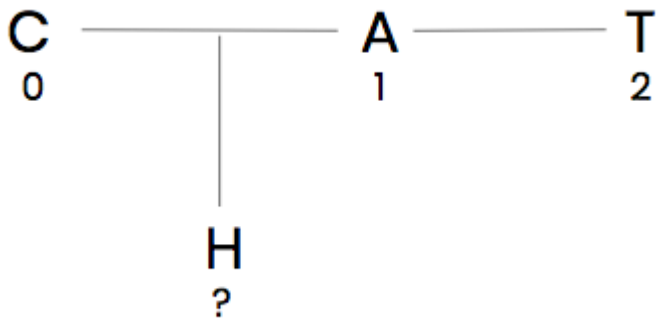


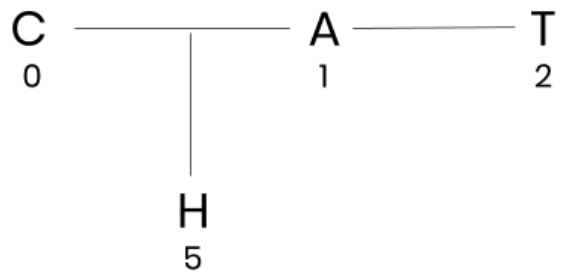
Duplicate deletion operations are idempotent with a CRDT

Globally Ordered Characters

The 2nd requirement for a collaborative text editor CRDT has to do with the positioning of characters. Since we're building a text editor, preserving the order of characters within a text document is required. But for a collaborative text editor where each user has their own copy of the document, we must go a step further. We need all the characters to be globally ordered and consistent. That means that when a user inserts a character, it will be placed in the same position on every user's copy of the shared document.The key is that by using fractional indices to insert characters, we never have to shift the positions of surrounding characters.

We can avoid this problem and ensure commutativity by using fractional indices as opposed to numerical indices. In the example below, when a user insert an "H" at position 1, they specifically intend to insert an "H" in between "C" and "A".





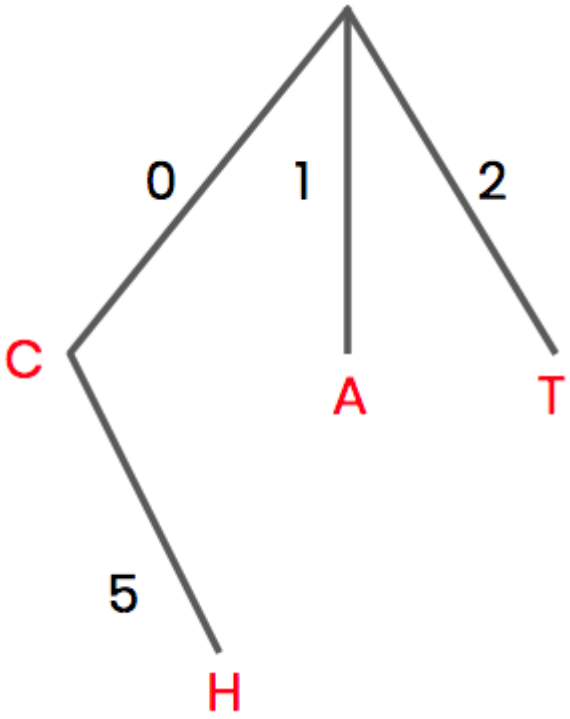
C	A	T
[0]	[1]	[2]

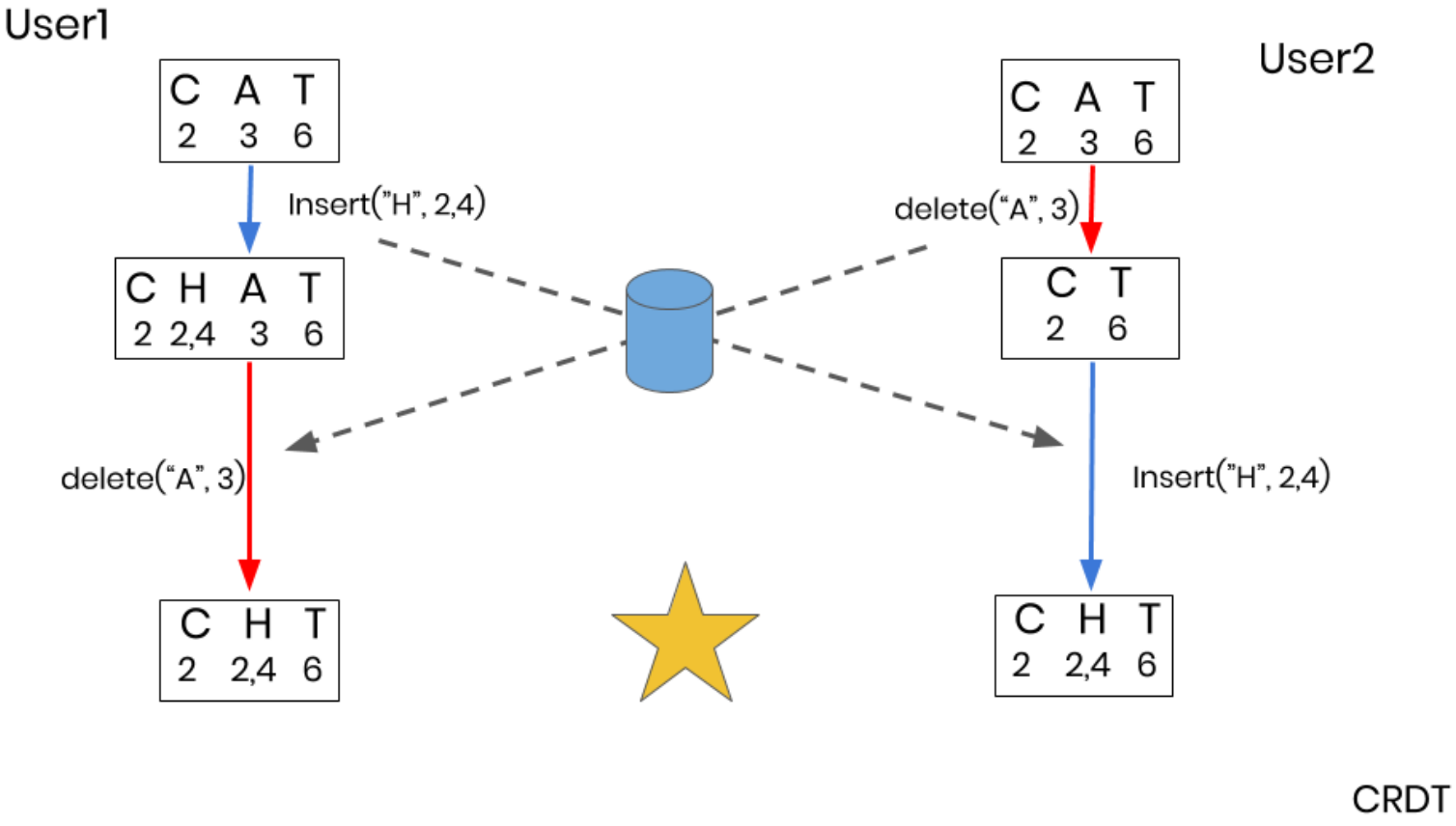
C	H	A	T
[0]	**[0, 5]**	[1]	[2]

Inserting a character at a position in between two existing characters.

Another way to imagine fractional indices is as a tree. As characters are inserted into the document, they can be inserted in between two existing position identifiers at one level of the tree. However, if there is no space between two existing character positions, as demonstrated below, we proceed to the next level of the tree and pick an available position value from there.

There are several academic papers dedicated to how best to “pick an available position”. We implemented an “adaptive allocation strategy for sequence CRDT” called **LSEQ.**





Insert and Delete operations commute using a CRDT

Coding CRDT required :-

a) **Web Text Editor -** We have used open-source **CodeMirror** text editor for it's ease of use and robust API.

b) **CRDT Structure :-**

```
class CRDT {
        constructor(id) {
          this.siteId = id;
          this.struct = [];
        }
 }
```

CRDT must handle 4 basic operations:

- **Local Insert: User inserts character into their local text editor and sends the operation to all other users.**
- **Local Delete: User deletes character from their local editor and sends the operation to all other users.**
- **Remote Insert: User receives a insert operation from another user and inserts it to their local editor.**
- **Remote Delete: User receives a delete operation from another user and deletes it from their local editor.**

c) **Peer to Peer Based framework -** We have used WebRTC. WebRTC is a protocol that was designed for real-time communication over peer-to-peer connections. It's primarily intended to support plugin-free audio or video calling but its simplicity makes it perfect for us even though we're really just sending text messages. While WebRTC enables our users to talk directly to one another, a small server is required to initiate those peer-to-peer connections in a process called "signaling". For signalling we are creating a Node.js server at the backend.

Time Complexity - We can achieve O(Log(N)) insert and delete operation complexity.

Space Complexity - A sub-linear complexity for maintaining the position identifiers for every character.