TimDocs: Decentralized P2P Collaborative Editor

Prabhakar Kafle, Grace Kim, Natasha Maniar

May 12 2023

1 Introduction

With remote collaboration surging due to the post-COVID WFH protocols, it's critical to have reliable and privacy-aware collaboration tools, especially for documents. Our goal was to design a decentralized document editor that didn't rely on a centralized server or required much overhead of data. Decentralization ensures high performance since clients aren't relying on one server to merge edits. Currently, other state-of-the-art real-time collaborative editors include Google Docs or Microsoft Word Online. The two main decisions real-time editors require are automatic conflict resolution and features to ensure reliability. Google Docs uses operational transform which is difficult to implement in a distributed fashion. Another open source editor, Conclave, uses CRDT (Conflict-Free Replicated Data Types) for merging edits which is based on state vs operation level changes [4].

1.1 Goals

1. Decentralized client-server system to reduce reliance on one server and improve privacy such that only clients that are on the same document have access to the data. In addition, prevent attackers from hacking the centralized server.
2. Quick communication between clients for real-time updating: Gossip protocol uses UDP which is faster than a TCP protocol, however, we handle the misordering through a buffer pool.
3. CRDT for automatic conflict resolution to ensure consistency and fast convergence. [2]

2 System Overview

We designed TimDocs with consistency, fault tolerance, and performance in mind. Multiple peers can make concurrent edits to the same document and the system makes sure that every edit's intent is preserved while achieving the same consistent document state for all peers. And the document is preserved as long as at least one peer remains alive.

To combat the issue of availability, all of the peers act as replicas of one another even though they represent different users. If one peer crashes then the document state is not lost since other peers must have the most updated document state.

First, a new *peer* contacts the broker service to get a list of all other *peer*s' IDs. The broker is a *Node.js* server keeping track of the *peer*s. It gives each requesting *peer* a unique ID which they then use to establish a WebRTC connection with other *peer*s.

Each *peer* has 3 components as indicated in Figure 1: Editor, Controller, and Messenger. The Editor is the user-facing interface showing the current state. The user makes edits via the Editor. This edit is then transferred to the Controller which keeps track of the state of the document. This state is maintained in a tree (detail in section 3.1) which is mutated by the edit from the Editor. A CRDT operation is generated as a result of this mutation. The Messenger broadcasts this operation to all the connected *peer*s via Gossip protocol [3] using the WebRTC connection established at the start. In the *peer* receiving the operation, the message flows from Messenger to Controller (mutating the tree with the CRDT operation received) to Editor (reflecting the change in UI).

Since the operations are sent peer-to-peer without going through the central server, the communication is very fast. Combine this with the fact that CRDT ensures no conflict, meaning no operation is aborted- the system achieves *high performance.* This is because we do not need to wait for other peers' responses before committing (displaying in UI) any change.

3 System Design

3.1 CRDT

3.1.1 WOOT Overview

To handle merging multiple users' edits, we used a conflict-free replicated data type (CRDT), specifi-

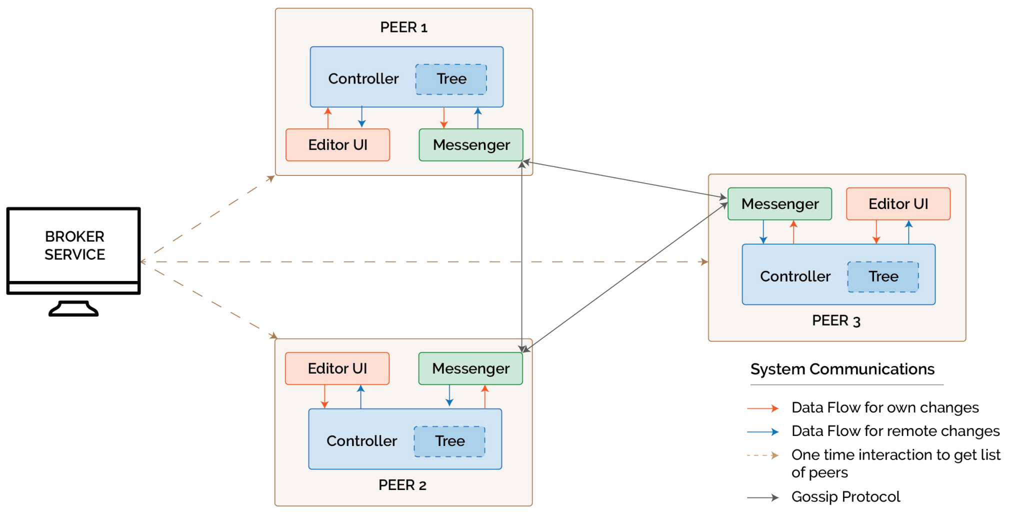


Figure 1: System Overview

cally the WOOT framework [1]. Within this framework, a user's document is represented with a tree structure, such that each node corresponds to a text edit. Each node has a unique ID based on the ID of the client and the order in which it was created. It also contains metadata about where it places relative to other edits in the document.

We implemented a standard tree, meaning that a node can have several children. Our search and traversal operations take O(n) time, but our editing operations take O(1) time. A complete traversal of the tree will return the content of the document. Notably, WOOT keeps deleted characters in the tree but marks them with an "invisible" flag so that when you traverse the tree, it will return only the nodes that are visible.

Finally, we implemented the WOOT operations so that they are idempotent. The order in which the client sees or executes them does not matter, ensuring eventual *convergence* and eventual *consistency* across all clients.

3.1.2 WOOT Generation

The logic behind WOOT can be divided into two categories: generation and reception (as outlined in the WOOT paper) [1]. Generation refers to what happens when a user makes an edit to their local copy of the document.

To support generation, the WOOT framework has four fundamental operations:

- IntegrateInsert
- IntegrateDelete
- GenerateInsert
- GenerateDelete

The two integrate operations take in a CRDT operation and apply it to the client's local copy of the document.

The two generate operations do three crucial things in the following order: (1) populate the CRDT operation with the appropriate metadata, (2) call the respective integrate operation, and (3) broadcast this CRDT operation to its peers.

These four operations ensure that clients execute all local edits to their local documents immediately. Only after executing locally will clients broadcast the new CRDT operations to its peers.

3.1.3 WOOT Reception

The logic in Figure 2 gets invoked when a client receives a CRDT operation from another peer. Each client keeps track of pending operations in a pool of operations, which we implemented with a buffer. We only capture the essence of the logic in Figure 2, but we implemented all the details that are mentioned in the original paper [1].

3.1.4 Our WOOT Implementation

We tried our best to completely implement the algorithms and data structures found in Section 3 of the cited WOOT paper. Our language of choice was Javascript to better integrate with our frontend. Though we tried to search for an existing library that had already implemented this type of framework, we ultimately decided to implement it from scratch because it seemed relatively feasible.

```
Algorithm 2: Reception(op)
  add op to pool
```

```
Algorithm 3: Main()
  while true do
      find op in pool such that isExecutable(op);
      if type(op) = delete then
          IntegrateDelete(op);
      end
      IntegrateInsert(op);
  end
```

Figure 2: Reception Pseudocode

3.2 Gossip

As a decentralized editor, our system is based on peer-to-peer (P2P) communication between the clients. Even though we have a signaling server to keep track of the list of active peers, this server does not store any information about the document state. We implemented the gossip protocol via the PeerJS library to make sure all peers disseminate document state information to all other n-1 peers. In addition, the protocol is used to ensure that all peers are active/ have not lost connection or crashed. Traditional gossip protocols use a ring topology of the peer nodes; however, we decided to use a star (fully connected) topology to reduce the consistency issues among peers.

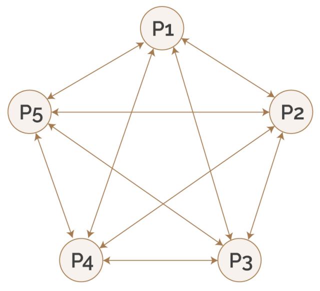
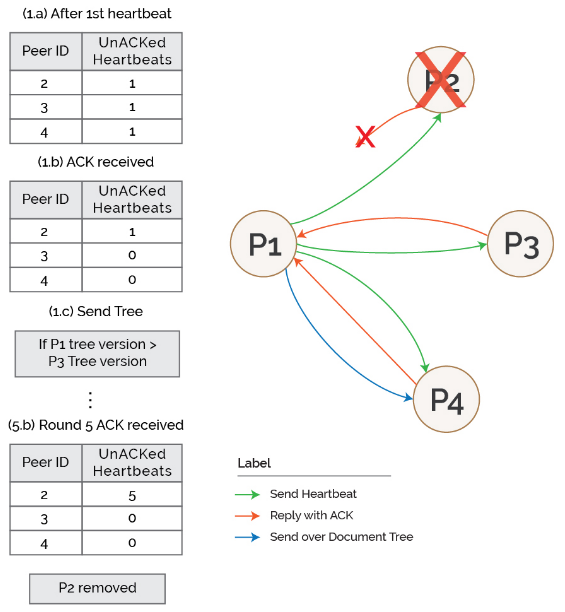


Figure 3: Connection between peers in gossip

Communication is executed in 2 different ways: heartbeats and broadcast changes. Similar to Raft, heartbeats are sent every 5s from every peer to every n-1 other peer (following the scheme below). Once a peer sends out a heartbeat, it waits for an acknowledgment from every other peer. If all peers send an acknowledgment within 5s, the peer updates its mapping from peer id: acknowledgment from (1) to (0) signifying that outgoing heartbeats were received. When the peer sends an acknowledgment, it also sends its tree version number.



(1.a) After 1st heartbeat

Peer ID	UnACKed Heartbeats
2	1
3	1
4	1

(1.b) ACK received

Peer ID	UnACKed Heartbeats
2	1
3	0
4	0

(1.c) Send Tree

If P1 tree version > P3 Tree version

⋮

(5.b) Round 5 ACK received

Peer ID	UnACKed Heartbeats
2	5
3	0
4	0

P2 removed

Figure 4: Heartbeat

Versioning: Each peer has a versionNum associated with their document state (i.e. tree.versionNumber), which is a counter that incrementally updates monotonically whenever it sends changes or receives changes from other peers. If there are no unreliability issues or crashes, the version numbers associated with all peers should be equal. In the case of these issues, a peer may lag behind and have a lower version number, triggering a call to sendDocument from the peer with the most updated version number.

Removing peers: If a peer doesn't send an acknowledgement within the first heartbeat we retry 5 times before removing the peer from active peer list ensuring there are no partitions. If that peer wants to reconnect it will have to join back on the network and can be added to the active list again and be sent the most updated document state.

Buffer pool: Since WebRTC uses UDP instead of TCP, packets may be received out of order, so we created a buffer within each client to keep track of ordered changes. Like the CRDT WOOT paper, we pushed the new operation to the buffer pool and only apply changes to the tree if the operation is executable (Reference to isExecutable definition in CRDT WOOT paper). We only process new operations when the buffer pool is empty.

3.3 Tools Used

We implemented the whole project in JavaScript. The broker service uses Express[1] framework on top of Node.js. For the peer clients, we used Peer.js[2] to implement WebRTC and CodeMirror[3] for the editor UI.

4 Evaluation

4.1 Results

The resulting system is a prototype of a decentralized collaborative editing platform. Several users can access one document at once through the website.

4.2 Testing

We used Jest as our testing platform for backend and end-to-end testing. We have tests for the major data structures that we could test, resulting in 26 unit tests.

We had over 85% test coverage for the files that we did make test cases for, which can be found in the "/TimDocsCoverage.txt" file in our Git repository. You will see that the test coverage for client.js and messenger.js is comparatively much lower. That is because we could not figure out how to integrate PeerJS with Jest. However, we tried our best to test edge cases on our actual computers.

5 Code

You can find our Git repository here: https://github.com/kafleprabhakar/TimDocs.

6 Conclusion

We created a distributed and decentralized remote collaborative editor to ensure privacy, fast convergence, and fault tolerance. By utilizing CRDT for automatic conflict resolution, gossip protocol for peer to peer communication, and a buffer pool to mitigate out of order convergence issues, TimDocs will enable users to collaborate in real time.

6.1 Future Work

- Scalability: Currently our system lacks scalability since we send over the entire document state (in the form of a tree) during heartbeats if the versions don't match. A more scalable way would be to create a merge function that merges the 2 tree versions by only sending the changes that were not applied.
- Persistence: If the signaling server crashes there is no way to retrieve the global list of active peers. To solve this issue, every peer could send every other peer the active peer list upon receiving it from the signaling server. Since we have a fully connected network, this is highly feasible. Alternatively, we can persist the active peer list to stable storage every time a new peer joins.
- History: Since we are tracking versions, we believe retrieving history would be relatively easy to implement on top of our tree structure.
- Performance: Our tree operation is currently essentially a linked list, meaning that our find operations are taking O(n) time. To make these find operations significantly faster, we could implement our tree to be a balanced binary tree instead.

References

[1] G. Oster, P. Urso, P. Molli, and A. Imine, "Data consistency for p2p collaborative editing," in *Proceedings of the 2006 20th anniversary conference on Computer supported cooperative work*, 2006, pp. 259–268.

[2] M. Kleppmann, *Conflict resolution for eventual consistency*, Nov. 2016. [Online]. Available: `https://martin.kleppmann.com/2016/11/03/code-mesh.html`.

[3] M. Chaturvedi, *Multicast and the gossip protocol*, Jan. 2022. [Online]. Available: `https://maneesh-chaturvedi.medium.com/multicast-and-the-gossip-protocol-6bdb78d1053e`.

[4] C. Team, *Conclave case study*. [Online]. Available: `https://conclave-team.github.io/conclave-site/`.

[1]https://expressjs.com/
[2]https://peerjs.com/
[3]https://codemirror.net/