BeaverDocs: A Peer-to-Peer Collaborative Editing Solution

Arlene Siswanto
siswanto@mit.edu

Fiona Zhang
fionaz@mit.edu

Grace Yin
graceyin@mit.edu

Justin Restivo
jrestivo@mit.edu

Abstract

Collaborating editing platforms have been growing increasingly popular. However a vast majority of them have been centralized. Sometimes, the nature of the work may require the use of a decentralized system. However this comes with a few challenges. The problem of developing a fast, scalable collaborative text editor with eventual consistency guarantees has been somewhat solved through the existence of a central server to merge all edits. Without this central connection, however, this problem becomes much harder. BeaverDocs is a peer-to-peer collaborative web editor that aims to minimize third party interaction while maintaining real-time content consistency among several users. This application implements a data structure optimized for quick inserts and deletions, a peer-to-peer architecture that fosters direct communication between peers, and a web interface that serves as the connection point for users.

1. Introduction

The increasing popularity of real-time collaboration has moved the process of word editing to the internet. Through services such as Google Docs and Overleaf, different teams can access and modify documents concurrently and on-demand. However, one pressing concern about existing editors is their centralization. Although much of the present-day web architecture relies on central servers and content, concerns about the privacy of users, security of confidential information, and proper use of stored data call for a restructuring of how information is handled online. Depending on the nature of a document, a decentralized web editing architecture may be preferable to a centralized one.

A system that minimizes the use of a centralized server by initiating direct communication between peers would effectively mitigate concerns about privacy, security, and ownership of data. Some other peer-to-peer systems, such as Atom's Teletype, Vim's Tandem, Emacs' RGA, and Conclave have also worked toward tackling these issues. However, after reading through relevant research, our goal was to implement our own peer-to-peer system that would take in the best components of each design along with several key modifications.

1.1. Goals

There were several features we wanted to optimize for in our implementation of BeaverDocs:

1. A data structure that can handle, with low overhead, upstream and downstream insertions and deletions as well as guarantee causal consistency. As elaborated in section 2.1, such a data structure complements peer-to-peer communication.

2. Smart, automatic peer recognition. For a given shared document, we want each peer to be aware of all other directly connected peers and for the document to reach eventual consistency, such that each connected peer will eventually have a convergent view of the document. BeaverDocs accomplishes this by introducing a lightweight central server that makes the initial connection between two peers, then allowing text editor changes to be broadcasted directly between peers without a central server.

3. Low overhead for updates. Since the use case of BeaverDocs involves constant insertions and deletions and necessary real-time updates, it is vital that each character change does not take substantial time or computational resources.

4. Heuristics for the peer selection process. Currently, our selection process randomly selects a peer to connect to. This can be further optimized by taking into consideration the number of connections one node is explicitly responsible for and the longest distance a message would need to travel.

2. System Design

2.1. Newscasting

BeaverDocs uses the Newscast algorithm, a gossip-based algorithm for membership management. Each peer keeps track of a *view*, a map of peerIDs it has directly seen or indirectly heard about to the timestamps at which they were last seen or heard from. Periodically, each peer

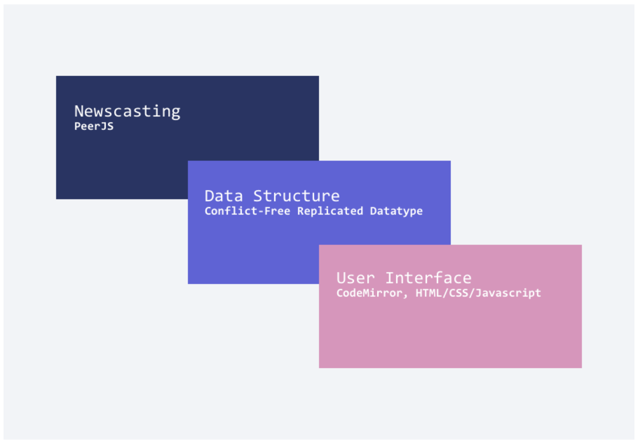

Newscasting
PeerJS
Data Structure
Conflict-Free Replicated Datatype
User Interface
CodeMirror, HTML/CSS/Javascript

Figure 1. Overview of the system

will pick one of its adjacent peers and send its view to them (we call this a `NewscastReq`). The receiver of a `NewscastReq` responds immediately with its current view in a `NewscastResp`, then takes the view it just received and merges it with its current view, keeping only the `x` most recent entries. The receiver of a `NewscastResp` just merges the received view with its current view, also keeping only the `x` most recent entries. [7]

Important to note that a peer's view may contain peers that are not connected to it. The purpose of Newscast is to inform peers of the presence of other peers outside its immediate neighbors, which will prove useful when trying to reconnect a disconnected node to the network.

2.2. Data Structure

BeaverDocs implements a data structure called a ”conflict-free replicated data type”, or CRDT, which allows for quick insertion, quick deletion, and causal consistency. We first introduce why such a data structure is necessary, review the definitions of causal consistency and partial orders, then finally dive into the details and implementation for CRDTs.

2.2.1 The Problem Statement

The problem is as follows. Given a document shared by multiple peers, how can we ensure that each client replica of this document shows the same text eventually and reflects the order of insertions and deletions? A document refers to the body of text contained in the text editor.

2.2.2 Definitions

Consistency models are often used in distributed systems when multiple parties are reading or writing the same resource. Such models serve as a contract between a client and a system, such that if the client behaves within expectations, the system guarantees that the outcome of the action is also expected.

Algorithm 1 Newscast

```
peers := [peerID, ... ]
view := {
    peerID : timestamp,
    ...
}

procedure HEARTBEAT()
    peer := pickRandomPeer()
    viewToSend := view + {me, current_timestamp}
    sendNewscastReq(peer, viewToSend)
end procedure

procedure ONNEWSCASTREQ(PEER, PEERVIEW)
    viewToSend := view + {me, current_timestamp}
    sendNewscastResp(peer, viewToSend)
    mergeWithMyView(peerView)
end procedure

procedure ONNEWSCASTRESP(PEER, PEERVIEW)
    mergeWithMyView(peerView)
end procedure
```

We begin by understanding how consistency applies in the context of a decentralized peer-to-peer web editor.

The intuition behind ”causal consistency” is, as the name suggests, that we do not mind stale data, just as long as we can continue to edit. We do, however, care about the order of concurrent edits. After all, we want the order to be the same across all replicas. The intuition here is that we can **only** accept updates from replicas that are as up-to-date as we are.

To understand this more formally, we review VectorClocks. A VectorClocks is a data structure local to each replica that keeps an internal clock and, via this, can detect concurrent operations. Keeping track of such metadata is necessary to ensure that we are not applying changes to our replica out of order, which would result in an inconsistent state.

We assume that each instance of a shared web editor has a unique site ID. Then, VectorClocks can be thought of as a mapping from site ID to operation number. Each time we perform some action, either an insertion or deletion, we increment the operation number. As seen in figure 2, site 0 is initialized to be all 0's, but then after operation O_1 is integrated into site 0, its vector clock is updated to be $[1, 0, 0]$. Operations are said to be ”casually related”, or concurrent, if all operations are executed in an order that reflects their

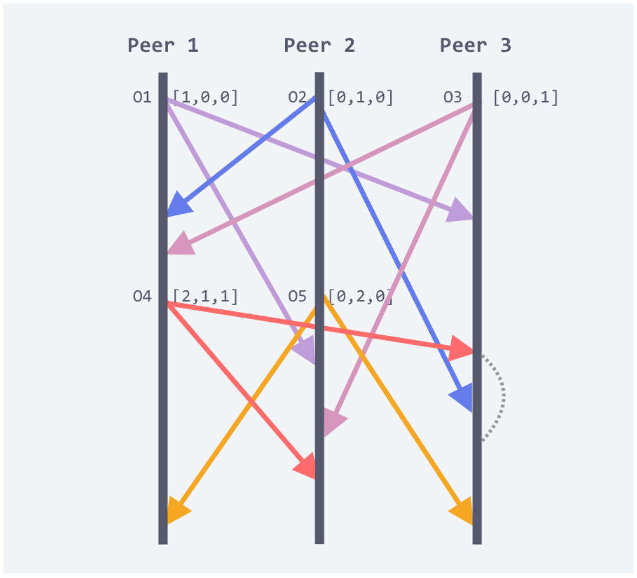


Figure 2. A timespace diagram in which three sites participate. A vector on the right of each operation is its vector clock.

causality. More specifically, we can define partial orderings between concurrent operations, and we would like these operations to be ordered.

Define a "happened-before" relation as Lamport [5] does in his paper:

> The relation $\rightarrow$ on the set of events in a system is the smallest relation satisfying the one two conditions, given two events a and b, $a \rightarrow b$ iff:
>
> - a and b are edits in the same site ID, and a comes before b
> - a is a local operation at site j, and b is the same operation, except at site i.
> - a and b are edits in the same site ID, and a comes before b
> - a is a local operation at site j, and b is the same operation, except at site i.
>
> It is worth noting that happens before is transitive; that is, $a \rightarrow b$ and $b \rightarrow c$ implies $a \rightarrow c$.
>
> Secondly, concurrence between two events is defined as $a \not\rightarrow b$ and $b \not\rightarrow a$.
>
> If we have two vector clocks: v_j, v_i generated at replicas j and i, respectively s.t. $j \neq i$, we define an operation associated with v_j to be causally ready to be applied at replica i (for a set of N known replicas) if:
>
> $$v_j[j] = v_i[j] + 1$$
>
> and
>
> $$\forall k \in \mathcal{Z} \cap [0, N-1] \setminus \{j\}, v_j[k] \leq v_i[k]$$

Intuitively this should make sense: all causally ready execution means is that we're preserving the order of happens-before operations. The happens-before ordering is key to basically making sure that we don't end up with conflicts between replicas.

Now, we choose to perform concurrent operations in an arbitrary order, and claim this allows for eventual consistency, or the idea that eventually all replicas will look the same. However, in this implementation we actually go for causal consistency, and as a result, define a ordering between concurrent operations. The remainder of the theory behind why this works is proved in the RGA paper [6]. Since the emphasis of the RGA paper is based around implementation, we are more concerned with the decentralized aspect, and black box the idea that accepting causally ready operations will provide us with causal consistency.

The RGA paper [6] defines a data structure that is based on Vector Clocks and provides enough information for us to determine orderings in our data structure in a convenient way. The data structure is called an S4Vector. As it is described in that paper, we simply blackbox.

However, we were concerned with performance at the time. As a result, we implemented an augmented RGA split tree data structure [2]. This allows for blocking of operations (inserts/deletes of multiple characters, and based on [8]), and avoids the problem of buildup of tombstone nodes that we would have to traverse when perform operations via a binary search tree [2]. Instead of having a complexity for each operation of θ(number of alive or dead nodes) because of traversing the linked list, we would be looking at a complexity per operation of $\theta \log_2$(number of alive nodes) as we would be traversing a balanced binary search tree.

2.2.3 CRDT

Now, let's discuss the conflict-free replicated data type, the CRDT, itself. At a high level, this data structure is represented through a linked list or a tree. Each node is either (1) a tombstone which has been killed or (2) alive and containing a string of content. Appending the contents of each node in the linked list should represent the entire content of a client's text editor. For faster access time, the data structure is also represented as a binary search tree to allow for faster access time. An in-order traversal of the tree would also reveal the content of the client's string.

Each node in the data structure has a unique key associated with it. This key is used to keep track of the operation that created it and the ordering of that operation in the tree. It also keeps track of other metadata used to determine where future insertions should take place.

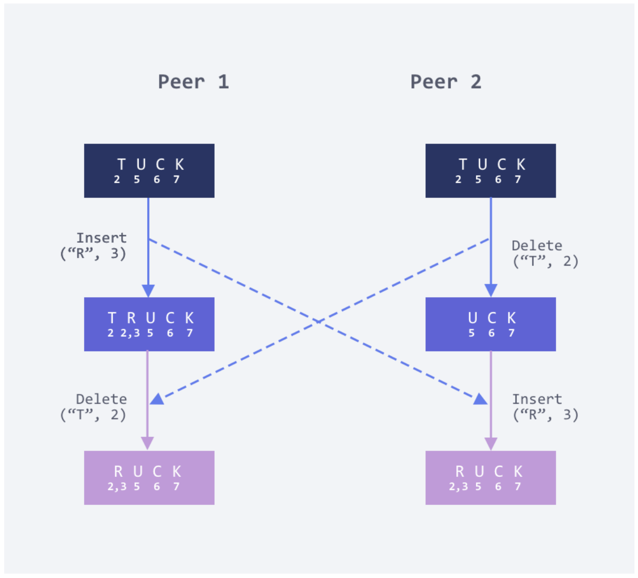


Figure 3. Example of operations using CRDT.

The CRDT API allows for four fundamental operations:

- localInsert(op)
- localDelete(op)
- remoteInsert(op)
- remoteDelete(op)

The first two operations return a sequence of remote operations to be applied to both the local and remote replicas. The latter two integrate remote operations into the local crdt. Before integrating remote operations, we check that the vectorclock associated with operation is causally ready. When an operation O issued at site $j(i \neq j)$ arrives with its vector clock $\vec{v}_0$, O is causally ready if $\vec{v}_O[j] = \vec{v}_i[j] + 1$ and $\vec{v}_O[k]\vec{v}_i[k]$ for $0 \leq k \leq N1$ and $k \neq j$.

The details of the correctness of the CRDT is beyond the scope of this paper, and further information can be found in the resource section. Essentially, our implementation was based off of the pseudocode in original RGA, RGASplitList, and RGASplitTree papers. We found a single, somewhat buggy Java library that contained an implementation that we based our implementation off of [4] and showed correctness through our Jest testing framework (and through inspection of the decentralized application's performance).

2.2.4 Consistency

In order to maintain consistency in the data structure, the operations are integrated as follows:

Algorithm 2 Consistency Operation

```
1: procedure MAIN LOOP()
2:     $\forall k : \vec{v}_i[k] = 0$
3:     Initialize $Q$
4:     Initialize $CRDT$
5:     while not aborted do
6:         if O is a local operation but not read then
7:             $\vec{v}_i[i] = \vec{v}_i[i] + 1$
8:         end if
9:         if (an operation O arrives with $\vec{v}_O$ from site $j$) then
10:            enqueue set $(O, \vec{v}_O)$ into $Q$
11:            while (there is a causally ready set in $Q$) do
12:                $(O, \vec{v}_O)$ = dequeue the set from $Q$
13:                $\forall \vec{v}_i[k]i = [k] = max(\vec{v}_i[k], \vec{v}_O[k])$
14:                Integrate O into CRDT
15:            end while
16:        end if
17:    end while
18: end procedure
```

3. Implementation

3.1. Front End

The client-facing component of our collaborative editor was implemented using Javascript, HTML, and CSS. We used Node.js as our server environment and unit tested with Jest. In order to get this to work, we ended up needing to run server-side javascript client-side. We also used Browserify to compile libraries we wrote into one massive minified javascript file.

Our HTML/CSS code is tightly integrated with Javascript code. Upon page load, the web application creates an instance of CodeMirror, a text editor that serves as the main point of contact for reading and writing to the document. This editor contains listeners that keep track of changes and key presses. It also initializes a client object, which contains metadata about the client and its peers as well as methods that can be called on the client. These methods perform actions using both the CRDT and PeerJS, broadcasting and receiving changes using PeerJS and making the appropriate modifications to the CRDT objects.

3.2. CRDT

We found a buggy implementation of the CRDT described in paper [2] in Java. We then ported this to javascript, fixed a few bugs, and then tied it into the frontend. We had to augment the data structure to include cursor information as well. Essentially each cursor consists of a pointer to the node that the cursor is currently a part of and an offset. When we perform an insertion or deletion, we simply move the cursor to the nearest node (arbitrarily giv-

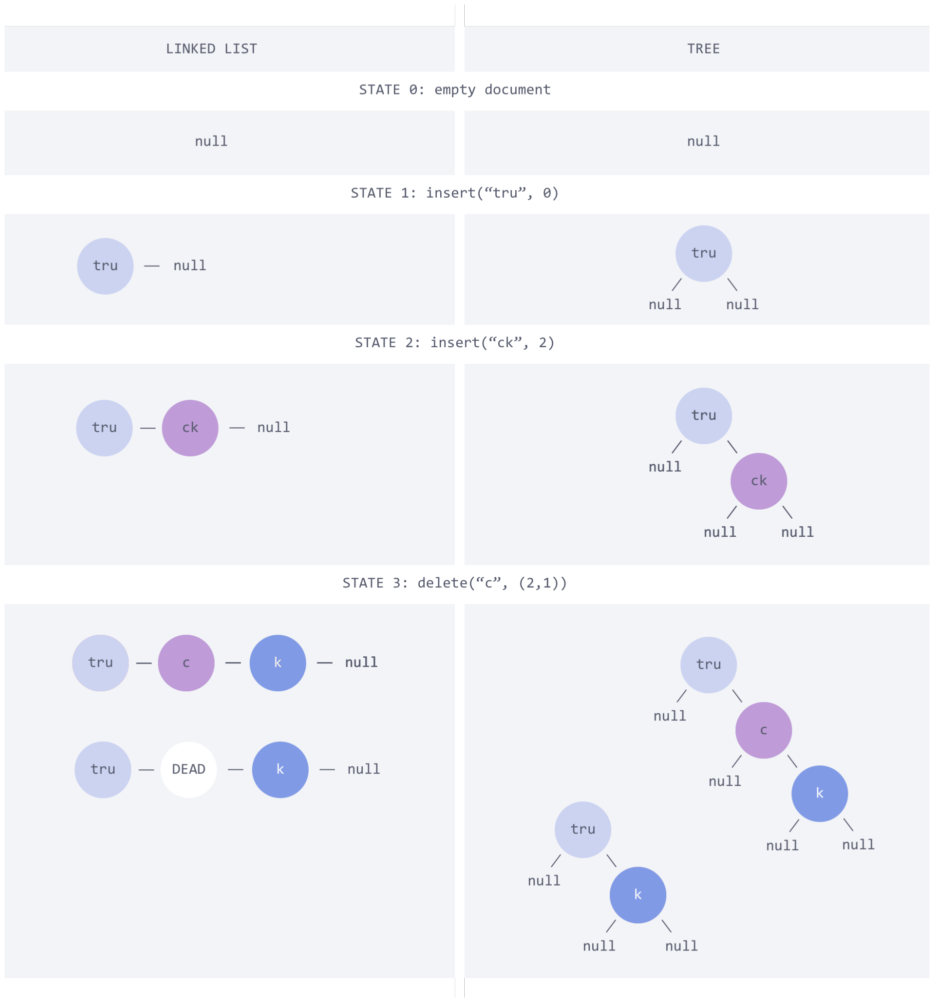


Figure 4. Linked list and tree representation of CRDT

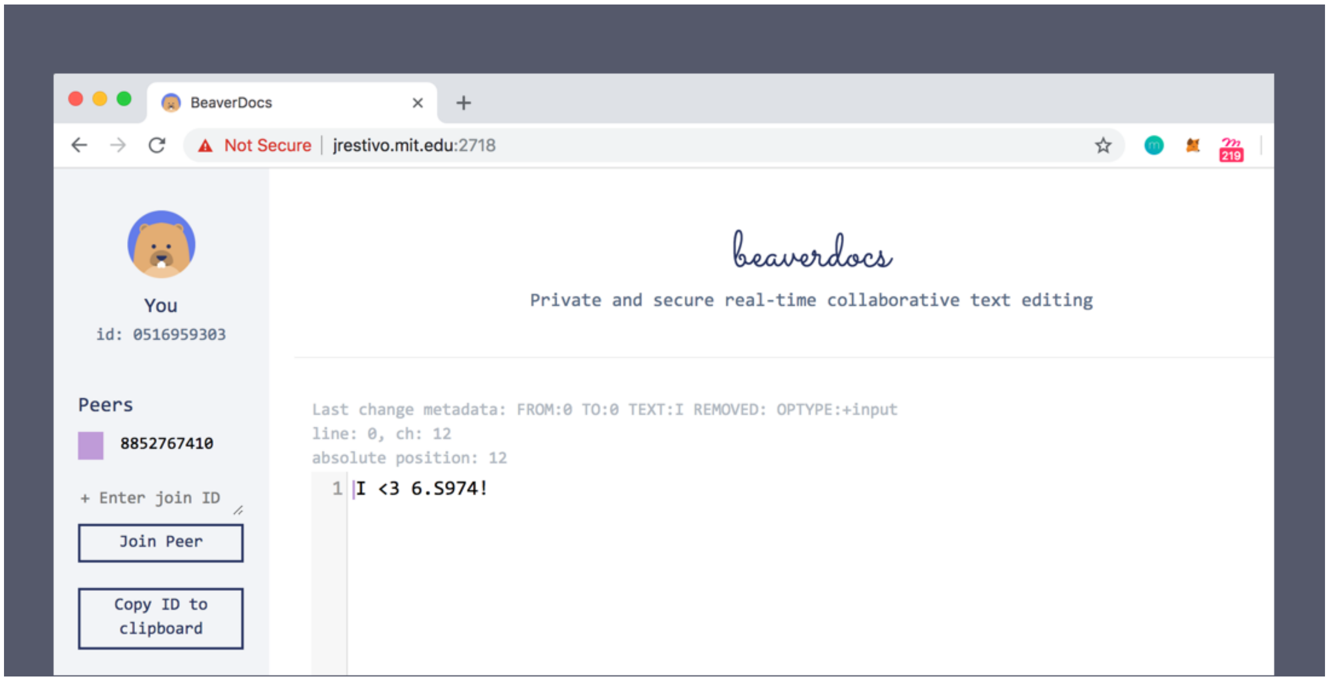
ing left nodes precedence over right). This reasoning works for the local client's cursor. We didn't implement this for other clients.

Here is a basic overview of the code included in *libraries/*:

- **OpTypes**: structures to hold metadata for insertion and deletion ops. Two different structures–one for local and one for remote operations.
- **cursor**: the aforementioned cursor data structure, consisting of a node and a offset into the node
- **vectorclock**
  - vector clock: an implementation of your standard vector clock
  - s3vector: an identifier for nodes in each replica derived from vector clocks
- **rstsplittree**
  - RSTWrapper: executes the API described above, and holds onto metadata
  - RSTReplica: An object holding the instance of the linked list and tree.
  - RSTNode: An actual node in the linked list/BST
- **trees**: BST with key as a RSTNode (not currently balanced)

3.3. PeerJS

PeerJS is a javascript library for peer to peer communication. It provides a clean API coordinates a peer to peer connection through the webRTC protocol/an intermediate ICE or STUN server depending if you're behind an asymmetric NAT. In order to initially establish a connection, you unfortunately need a central server to connect the two nodes. This server will keep a list of current unique replica ids (one for each client). When a client wants to connect to another



```
Last change metadata: FROM:0 TO:0 TEXT:I REMOVED: OPTYPE:+input
line: 0, ch: 12
absolute position: 12
I <3 6.S974!
```

Figure 5. The user interface for beaverDocs

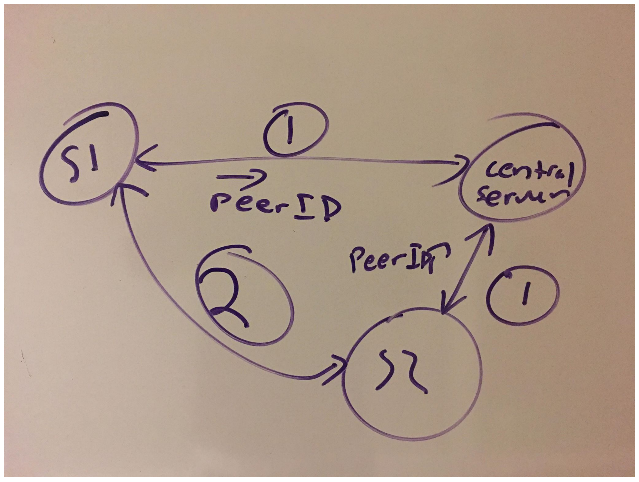


Figure 6. How new clients initialize connections for PeerJS

set of clients, the server will broker the initial connection, then disconnect.

3.4. NewsCast

We integrated the Newscast algorithm into BeaverDocs' communication scheme. Upon receiving any message, each peer will update the timestamp associated with the sender peerID in their view. In this way, peers can keep track of how alive their neighboring peers are.

The core of our Newscast implementation is a heartbeat task, which is run every 2 seconds. This task picks a directly connected peer at random, updates their view with their own peerID and current timestamp, and sends them their view in a `NewscastReq` message. The receiver then sends back their view in a `NewscastResp` message. At this point, both the sender and receiver have copies of each other's views, and they individually merge their views, keeping only the x most recently seen peerIDs (x is currently set to 5).

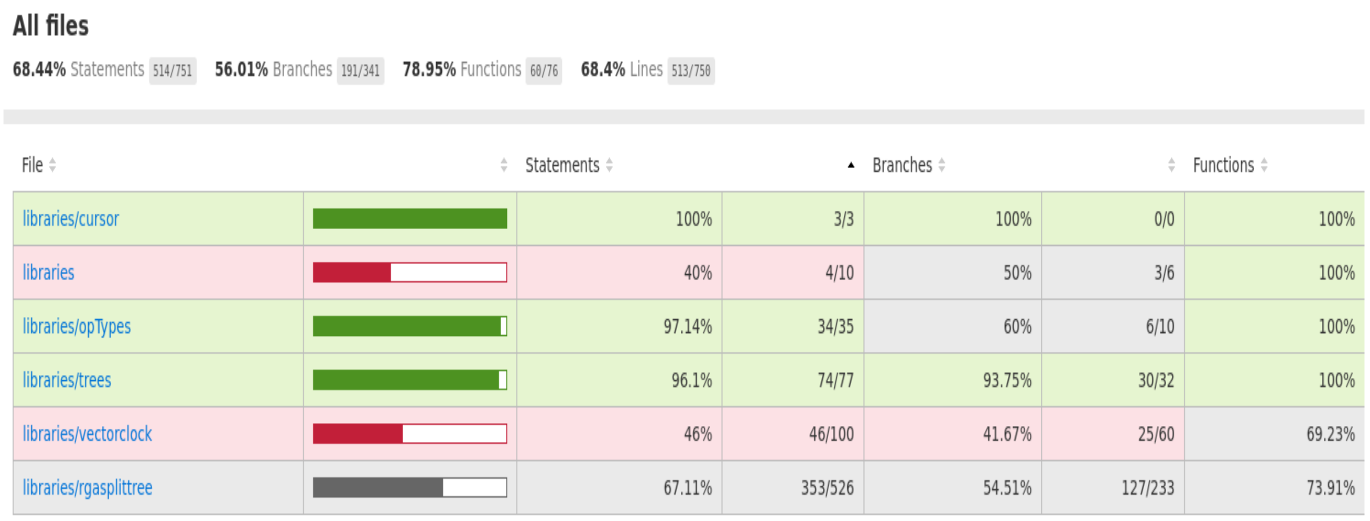
3.4.1 Reconnecting + Partition avoiding

When a peer notices that it is no longer directly connected to any other peer, it spawns a new task that iterates through the peerIDs from the most recent view in order, and tries to connect with them. In particular, we consider the sorted list of peerIDs a ring, and try to connect with peers around the ring starting from the location of our peerID. This is to avoid the scenario where any live, unconnected peers "pair up," forming many small partitions.

Additionally, a peer does not stop searching through its last view until it has started a connection. This is also to discourage small partitions from forming. When starting a new reconnect attempt, we kill any outstanding reconnect attempts.

Similar to heartbeats, this task is run at a set interval until a connection is made with another peer. We set it so one connection attempt is made every 1.5 seconds.

(We decided on an interval of 1.5 seconds to give the peers enough time to forge a connection. Otherwise, we get errors about too many connection attempts. This interval may need to be adjusted to deal with any higher-latency


All files

68.44% Statements 514/751 56.01% Branches 191/341 78.95% Functions 60/76 68.4% Lines 513/750

File		Statements		Branches		Functions
libraries/cursor		100%	3/3	100%	0/0	100%
libraries		40%	4/10	50%	3/6	100%
libraries/opTypes		97.14%	34/35	60%	6/10	100%
libraries/trees		96.1%	74/77	93.75%	30/32	100%
libraries/vectorclock		46%	46/100	41.67%	25/60	69.23%
libraries/rgasplittree		67.11%	353/526	54.51%	127/233	73.91%

Figure 7. Overall Test Coverage for libraries/*

scenarios, for example, if someone on campus wanted to forge a connection with someone in Australia.)

3.4.2 Propagating CRDT Ops

Though Newscast can be used to propagate CRDT ops, We decided against using Newscast this purpose, because we wanted to ensure that operations propagated as quickly as possible and in as predictable a manner possible. The downsides of using Newscast heartbeats for propagating this information is 1) we send Newscast heartbeats infrequetly, and 2) Newscast picks a peer at random to communicate with at each time interval, so it may take a while for a particular neighboring peer to hear about recent operations.

Instead, each peer simply keeps track of all CRDT Ops it's seen, and if they receive a new CRDT Op, they apply it locally and forward it to all their neighboring peers. This message flooding mechanism means that operations reach peers in the network as fast as possible, providing users with a better experience.

3.5. Implementation Difficulties

We ran into implementation difficulties in getting cursors to work, and overall just interfacing CodeMirror's API with the CRDT. The CRDT api allows for local/remote insertions and deletions, and a `toString()` method that allows for us to view the document. However, this means that every time we want to update the document, we overwrite the entire document with the new `toString()`.

This fundamentally is a flaw in the rgasplittree API. It would have been necessary to augment the API to print the discovered local position of the insert or delete. However, we realized that we can edit in real-time with this hack, and the latency is not noticeable, so we did not prioritize fixing this.

Beyond this, we wanted several other features. We wanted the cursors to be tracked (to allow for concurrent typing, as opposed to concurrent insertions. We achieved this by (as mentioned above), augmenting the data structure. But even this is fairly inefficient. We modified the data structure on insertions and deletions to change the cursor position. This is easy enough and efficient. However, when the client clicks on a different portion of the text of the document, we run into issues. We have to find the local position, then traverse the linked list until we find the correct position. Then we update the cursor node and offset. Morally speaking, we should be, instead, traversing the binary search tree. However, this remained performant enough for our purposes.

Similarly, when we press the right or left arrow, we would like the cursor to move right or left. The way we implemented this was also lazy (aimed at getting a working demo), and involves traversing the linked list multiple times. We should add in a backward pointer to the linked list, or use the binary search tree to increase efficiency. These operations are actually incredibly slow.

We would also like to be able to see other users cursors as we type. We currently broadcast the absolute position upon a movement. This is a hack, and completely inaccurate (it just provides a baseline for the general vicinity of where the person is typing. Ideally, we would like to do one of two things here. Either, we would broadcast cursor updates every time we perform an insertion or deletion. Or, we would add cursor movement as an operation that increments the VectorClock (probably the better/more accurate alternative).

Getting cursors to another absolute position in the codemirror frontend was difficult. The codemirror API doesn't make it particularly easy to detect when the user clicks on a different portion of the screen. We implemented a solution that tracks the occurrences of four listeners, and change a global variable based on their sequence of execution.

3.6. Testing

We used the Jest framework to test our Javascript code. There are tests written for each data structure, making for a total of 33 unit tests. Additionally, Jest generates a nice website view of testing code coverage, which is great for our data structures. See coverage/lcov-report/index.html.

For testing and performance of the the frontend, we connected a bunch of peers all over the world, and looked for latency, which there none. TOOD actually do this ...

4. Results

The resulting system is a fun and easy to use peer to peer editing systems. The latency is minimal as tests have been run with no latency run two peers from in Europe. Additionally, the editor has been proved to be consistent as it works between a large number of users a time.

5. Code

Our web application can currently be found on jrestivo.mit.edu:2718. Our repo is hosted on GitHub at: https://github.com/DieracDelta/974fp and additionally on a local self-hosted instance of gitea at: https://gitea.diracdelta.me/jrestivo/974fp

6. Work Split

- Arlene Siswanto: Web application development, HTML/CSS/Javascript and design, CodeMirror integration, peer object methods, message data, cursor movements with multiple peers

- Fiona Zhang: Peer to peer, consistency in CRDT, detection of cursor changes, VectorClock concurrency, cursor crdt updates upon left-right arrow key-presses and click

- Grace Yin: newscast implementation, reconnecting/ partition-healing mechanism, CRDT operation propagation, cursor cleanup

- Justin Restivo: CRDT (RGASplitTree), VectorClock implementation, Cursor CRDT augmentation, Cursor CRDT to local sync, Peer-to-Peer communication. cursor CRDT updates upon left-right arrow key-presses and click

7. Future Work

We would like to implement the following features in the future but didn't get around to them:

- **Undo**. We think this is somewhat feasible to implement. Upon an insertion, we create a pointer to the node with the length of the string of interest. We track this node when we perform more insertions and deletions. If it gets split into multiple nodes, then remove the operation as it is no longer possible to perform. To undo, if the node is untouched we simply kill it. Similarly for a deletion, we simply track the node, and mark it as alive (and update offsets).

- **Storage** We would like client to be able to load and store the file from and to local storage (or some decentralized cloud storage solution).

- **Compiler** We'd like clients to be able to run the code that they write locally. We could do this by include a client-side code compiler.

References

[1] N. Alsulami and A. Cherif. Collaborative editing over opportunistic networks: State of the art and challenges. *International Journal of Advanced Computer Science and Applications*, 8(11), 2017.

[2] L. Briot, P. Urso, and M. Shapiro. High Responsiveness for Group Editing CRDTs. In *ACM International Conference on Supporting Group Work*, Sanibel Island, FL, United States, Nov. 2016.

[3] ConnorsFan. How can i act on cursor activity originating from mouse clicks and not keyboard actions? `https://stackoverflow.com/questions/40282995/how-can-i-act-on-cursor-activity-originating-from-mou` `fbclid=IwAR0b1CRJ978te-BlK_7NPKgHZvtcBjIttxUjwEnfntAWhu0wUqPFNP0RUn8`, 2016.

[4] U. de Lorraine / LORIA / Inria / SCORE Team. replication-benchmarker. `https://github.com/score-team/replication-benchmarker/`, 2012.

[5] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, July 1978.

[6] H.-G. Roh, M. Jeon, J. Kim, and J. Lee. Replicated abstract data types: Building blocks for collaborative applications. *J. Parallel Distrib. Comput.*, 71(3):354–368, 2011.

[7] M. van Steen. Large-scale newscast computing on the internet. 11 2002.

[8] W. Yu. A string-wise crdt for group editing. In *GROUP*, 2012.