A Major Project Report

on

IMPLEMENTATION OF CRDT IN REAL-TIME COLLABORATIVE APPLICATOINS

Submitted to the

JAWAHARLAL NEHRU TECHNOLOGICAL UNIVERSITY HYDERABAD

In partial fulfillment of the requirement for the award of the degree of

BACHELOR OF TECHNOLOGY

IN

COMPUTER SCIENCE AND ENGINEERING

BY

TARAKANT SETH	18WJ1A05Y3
VARUN PAPISHETTY	18WJ1A05Z5

Under the Esteemed Guidance Of

Mr. V. DEVASEKHAR,

Head Of The Department, CSE.

GURU NANAK INSTITUTIONS TECHNICAL CAMPUS (AUTONOMOUS)

School of Engineering and Technology

Ibrahimpatnam, R.R District- 501506

2021-2022

GNI
www.gniindia.org

GURU NANAK INSTITUTIONS TECHNICAL CAMPUS

Approved by AICTE - New Delhi | Affiliated to JNTU - Hyderabad | Accredited by National Assessment and Accreditation Council | Accredited by National Board of Accreditation

AUTONOMOUS
under Section 2 (f) & 12 (b) of University Grants Commission Act

Department of Computer Science and Engineering

CERTIFICATE

This is to certify that the Major project report entitled "**IMPLEMENTATION OF CRDT IN REAL-TIME COLLABORATIVE APPLICATOINS**" by **TARAKANT SETH (18WJ1A05Y3), VARUN PAPISHETTY (18WJ1A05Z5)** submitted in partial fulfillment of the requirements for the degree of **Bachelor of Technology** in **Computer Science and Engineering** of the **Jawaharlal Nehru Technological University Hyderabad** during the academic year 2021-2022, is a bonafide record of work carried out under our guidance and supervision.

INTERNAL GUIDE	**CO-ORDINATOR**	**HOD CSE**
Mr. V. DEVASEKHAR	**Ms. V. SWATHI**	**Mr. V. DEVASEKHAR**

EXTERNAL EXAMINER

ACKNOWLEDGEMENT

We wish to communicate our true gratitude to **Dr. Rishi Sayal**, **Associate Director**, GNITC for giving us the helpful condition to bringing through our scholastic calendars and undertaking effortlessly.

We would like to say our sincere thanks to **Mr. V. Devasekhar, Associate Professor & HOD of CSE**, GNITC for providing seamless support and right suggestions for the development of the project.

We particularly thank our project coordinator **Ms. V. Swathi, Associate Professor, Department of CSE**, Project Coordinator for offering consistent help and right recommendations are given in the improvement of the project.

We have been blessed to have a wonderful internal guide **Mr. V. Devasekhar, Associate Professor and HOD of CSE**, GNITC for managing us to investigate the implication of our work and we offer our true thanks towards him for driving us through the consummation of undertaking.

TARAKANT SETH	**18WJ1A05Y3**
VARUN PAPISHETTY	**18WJ1A05Z5**

IMPLEMENTATION OF CRDT IN REAL-TIME COLLABORATIVE APPLICATOINS

ABSTRACT

Real-time collaborative applications allow users to edit a shared document concurrently and see each other's changes in real-time. Most collaborative applications such as apache wave, etherpad, google docs, uses OT (operational transformation) based algorithms that treat a document as a single ordered list with no support for nested tree structures, which degrades rapidly with an increase in simultaneous operations. These systems rely on a single server, with no peer-to-peer collaboration, which hinders the user's privacy. CRDT (conflict-free replicated data type) is a family of data structures that support concurrent transformation and ensure convergence of concurrent updates and works by attaching additional metadata to the data structure, making modification operations commutative by construction. In this project, we use CRDTs for implementing a general-purpose data store that supports real-time collaborative editing of semi-structured data. The intent of the data store is that it drastically streamlines the development of collaborative and state-synchronizing applications for mobile devices with poor network connectivity, in peer-to-peer networks, and in applications systems with end-to-end encryption.

Contents

ABSTRACT I
Contents II
List of Figures IV
List of Symbols V

1. Introduction 1
    1.1 Definition 1
    1.2 Objective 2
    1.3 Existing System 2
        1.3.1 Existing system advantages 2
        1.3.2 Existing system disadvantages 2
    1.4 Proposed System 3
        1.4.1 Proposed system advantages 3
        1.4.2 Proposed system disadvantages 3

2. Literature Survey 4
    2.1 Sliding Window CRDT Sketches 4
    2.2 Database development supporting offline update using CRDT 4
    2.3 An efficient collaborative editing algorithm for string-based operations 5
    2.4 Merkle Search Trees: Efficient State-Based CRDTs in Open Networks 6
    2.5 A Control Loop-based Algorithm for Operational Transformation 6
    2.6 A Conflict-Free Replicated JSON Datatype 7
    2.7 Concurrency control and awareness support for multi-synchronous collaborative editing 8
    2.8 Delta-State-Based Synchronization of CRDTs in Opportunistic Networks 8
    2.9 Efficient Synchronization of State-Based CRDTs 9
    2.10 Edge Applications: Just Right Consistency 9
    2.11 Conflict-Free Partially Replicated Data Types 10

3. Project Description 11
    3.1 Text Editor 11
    3.2 Real-time collaborative text editor 12
        3.2.1 Operational transformation 15
        3.2.2 Conflict-free replicated data types 16
    3.3 CRDT 17
        3.3.1 State-based CRDTs 17
        3.3.2 Operation-based CRDTs 17
        3.3.3 Globally Unique Characters 18
        3.3.4 Globally Ordered Characters 19

4. Software Design 23
    4.1 Architecture 23
        4.1.1 Peer-to-Peer Architecture 23
        4.1.2 System Architecture 25
        4.1.3 Version Vector 26
        4.1.4 Final System Architecture 28

4.2 UML Diagram 29
4.2.1 Use Case Diagram 30
4.2.2 Activity Diagram 31
4.2.3 Sequence diagram 32
4.2.4 Class Diagram 33
4.2.5 Object Diagram 34

5. Software and Hardware Requirements 35
5.1 NodeJS 35
5.2 ExpressJS 36
5.3 jQuery 37
5.4 PUG 38
5.5 PeerJS 39
5.6 RxJS 40
5.7 Acorn 41
5.8 UUID 42
5.9 SimpleMDE - Markdown Editor 43

6. Code Templates 45
6.1 CRDT Structure 45
6.2 Local Insert/Delete 45
6.3 Remote Insert/Delete 47
6.4 Remote Cursors 48
6.5 Download 55
6.6 Upload 55
6.7 Automatic Network Balancing 56
6.8 What is the maximum number of connections? 56
6.9 Version Vector 57

7. Testing 59

8. Output Screens 60

9. Conclusion 66

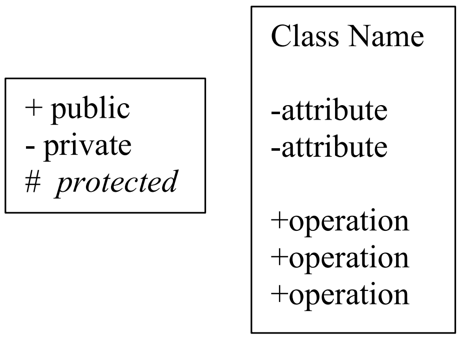
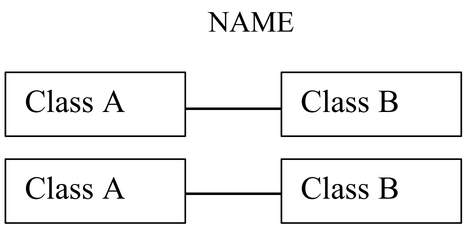
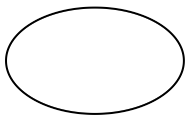
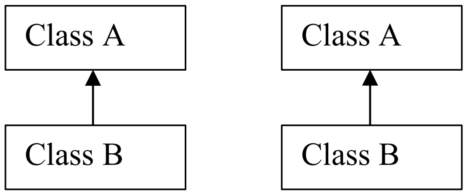
10. Further Enhancements 67
10.1 Better Connection Distribution 67
10.2 Large Insertions/Deletions 67
10.3 Comments 67
10.4 Suggests edits 67
10.5 Better functionality and UI 67

REFERENCES 68

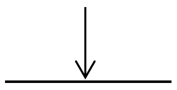
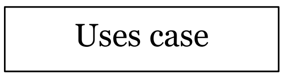
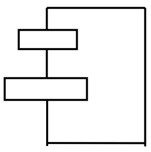
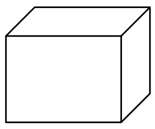
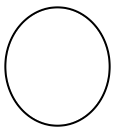
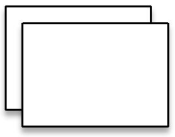
List of Figures

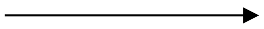
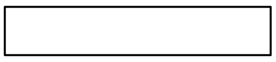
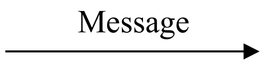
Fig.1. The character positions may shift as surrounding characters are inserted or deleted. 11
Fig.2. Two users connected through a central relay server. 12
Fig.3. Simultaneous insertion and deletion produce different results. 13
Fig.4. Duplicate deletion operations are not idempotent. 14
Fig.5. Insertion and deletion operations commute with OT. 16
Fig.6. Duplicate deletion operations are idempotent with a CRDT. 19
Fig.7. Inserting a character in between two existing characters. 20
Fig.8. Inserting a character at a position in between two existing characters. 20
Fig. 9. Relative positions are like a tree 21
Fig.10. Insert and Delete operations commute using a CRDT. 22
Fig.11. peer to peer network. 24
Fig.12. System architecture. 25
Fig.13 Version Vector insertion and deletion. 26
Fig.14. Version Vector insertion and deletion operations process 27
Fig.15. Final System Architecture 28
Fig.16. Use Case Diagram 30
Fig.17. Activity Diagram 31
Fig.18 Sequence Diagram 32
Fig.19 Class Diagram 33
Fig.20 Object Diagram 34

List of Symbols

S.NO	NOTATION NAME	NOTATION	DESCRIPTION
1.	Class	 + public - private # *protected* Class Name -attribute -attribute +operation +operation +operation	Represents a collection of similar entities grouped together.
2.	Association		Associations represent static relationships between classes. Roles represents the way the two classes see each other.
3.	Actor		It aggregates several classes into a single classes.
4.	Aggregation		Interaction between the system and external environment

5.	Relation (uses)	uses	Used for additional process communication.
6.	Relation (extends)	extends →	Extends relationship is used when one use case is similar to another use case but does a bit more.
7.	Communication	———	Communication between various use cases.
8.	State	$\boxed{\text{State}}$	State of the processes.
9.	Initial State	◯	Initial state of the object
10.	Final state	⊙	Final state of the object
11.	Control flow	→	Represents various control flow between the states.

12	Decision box		Represents decision making process from a constraint
13	Use Case		Interact ion between the system and external environment.
14.	Component		Represents physical modules which are a collection of components.
15.	Node		Represents physical modules which are a collection of components.
16.	Data Process/State		A circle in DFD represents a state or process which has been triggered due to some event or action.
17.	External entity		Represents external entities such as keyboard,sensors,etc.

18.	Transition		Represents communication that occurs between processes.
19.	Object Lifeline		Represents the vertical dimensions that the object communications.
20.	Message		Represents the message exchanged.

Chapter 1

Introduction

Collaboration through software is a core part of many peoples lives. We communicate by email and chat systems and work together with other people on documents, spreadsheets and presentations. The most advanced types of such collaborative software are those that support real-time collaborative editing, i.e. that multiple users can edit the same document simultaneously and see each other's changes in real-time.

A popular example of a real-time collaborative product is Google Docs. It achieves eventual consistency by using a technique called Operational transformation (OT), which was introduced by Ellis and Gibbs in 1989. However, more recent advancements in distributed systems that promise strong eventual consistency are conflict-free replicated data types (CRDTs), which was formulated by Shapiroet al. in 2011. CRDTs is a collective term for data structures that can be replicated across nodes and guarantees that any nodes that have received the same updates will end up in the same state.

1.1 Definition

With the emergence of modern cloud computing and mobile applications, we typically need to support offline editing, i.e. the ability to continue using the application without having a network connection. These edits are later synchronized to the cloud and the other devices when the network comes back.

What these features have in common is that the application state needs to be replicated across multiple devices. Each replicated state can be updated locally and later synced to other devices. This replication requires a fundamental shift in the mindset of the application from only being a view of a server side state to being a node in a distributed system where each node has a separate state. This shift comes with a series of new challenges such as how to achieve consistency between nodes and how to handle conflicts that can arise when different nodes edit a shared state in parallel.

1.2 Objective

The objective of this project is to build a real-time collaborative editing application that uses conflict-free replicated data types (CRTDs). The theoretical part of the project is to identify the requirements of such an application, identify challenges, explore different solutions and proposed implementation strategies.

1.3 Existing System

In a real-time collaboration system, all the application state needs to imitate several devices, which may modify the state locally. Due to the conventional approach towards concurrency management and serialization, an application becomes unstable.

1.3.1 Existing system advantages

- Transformation functions are simpler and less error-prone
- Widely used by companies

1.3.2 Existing system disadvantages

- Treats a document as single ordered list
- No support for nested tree structure
- Lack of peer-to-peer collaboration without server
- Algorithms are mostly proprietary
- Privacy issues

1.4 Proposed System

The proposed system allows messages to be delivered through an ad-hoc network as well as a secure messaging protocol with end-to-end encryption.

1.4.1 Proposed system advantages

- Replication and conflict resolution
- Uses JSON format
- Uses decentralized network
- Better privacy

1.4.2 Proposed system disadvantages

- Transformation functions aren't simpler.
- Active area of research and technology isn't ready for our needs yet.

Chapter 2

Literature Survey

A literature survey or a literature review in a project report is that a section which shows the various analyses and research made in the field of your interest and the results already published, taking into account the various parameters of the project and the extent of the project.

2.1 Sliding Window CRDT Sketches[1]

Author : D. Adas and R. Friedman

Year : 2021

Description:

Sketches maintain compact approximate statistics about streams of data, thereby enabling quickly answering queries regarding the data stream without having to reprocess it. Often, recent data is considered more important than older one, which is captured by the sliding window model. In distributed settings, where parts of the stream are seen by different, potentially geographically distributed components of the system, it makes sense to collect global statistics about the stream, but in a decentralized manner. Further, in order to ensure availability, scalability, and good performance, it is appealing to treat sketches as a CRDT data-type. In this work we introduce the notion of sliding window CRDT sketches. We then present the CRDT All Timestamps (aka CRDT-AT) and CRDT Last Timestamp (aka CRDT-LT) algorithms for implementing such sketches and analyze them. We also study the performance of CRDT-AT and CRDT-LT using real workloads, to establish their viability.

2.2 Database development supporting offline update using CRDT[2]

Author : E. Chandra and A. I. Kistijantoro

Year : 2017

Description:

Database is nowadays a crucial data storage for application needs. Currently, there are several existing popular databases such as relational database like SQL and non-relational database (i.e. NoSQL). However, both types have their own drawbacks which could not be utilized in both online and offline modes concurrently. Therefore, the authors develop a database that supports both modes, especially availability in offline mode without sacrificing consistency. The database is integrated with the CRDT (conflict-free replicated data types) algorithm to handle and solve conflicts in the synchronization process. Some tools and technologies are used in the database development including Apache Thrift for middleware, LevelDB for server databases, and IndexedDB for temporary client databases. The purpose of this development is to build a CRDT database with a usable API for developer users. Finally, it is tested with scenarios to satisfy essential operations: add, remove, and update.

2.3 An efficient collaborative editing algorithm for string-based operations[3]

Author : X. Lv, F. He, W. Cai and Y. Cheng

Year : 2016

Description:

Recently, Commutative Replicated Data Type(CRDT) algorithms have been proposed and proved by many references to outperform traditional algorithms in real-time collaborative editing. Replicated Growable Array(RGA) has the best average performance among CRDT algorithms. However, RGA only supports character-based primitive operations. This paper proposes an efficient collaborative editing approach supporting string-based operations(RGA Supporting String). Firstly, RGASS is presented under string-wise architecture to preserve operation intentions of collaborative users. Secondly, the time complexity of RGASS has been analyzed in theory to be lower than that of RGA and the state of the art OT algorithm(ABTSO). Thirdly, the experiment evaluations show that the computative performance of RGASS is better than that of RGA and ABTSO. Therefore, RGASS is more adaptable to large-scale collaborative editing with higher performance than a representative class of CRDT and OT algorithms in publications.

2.4 Merkle Search Trees: Efficient State-Based CRDTs in Open Networks[4]

Author : A. Auvolat and F. Taïani

Year : 2019

Description:

Most recent CRDT techniques rely on a causal broadcast primitive to provide guarantees on the delivery of operation deltas. Such a primitive is unfortunately hard to implement efficiently in large open networks, whose membership is often difficult to track. As an alternative, we argue in this paper that pure state-based CRDTs can be efficiently implemented by encoding states as specialized Merkle trees, and that this approach is well suited to open networks where many nodes may join and leave. At the core of our contribution lies a new kind of Merkle tree, called Merkle Search Tree (MST), that implements a balanced search tree while maintaining key ordering. This latter property makes it particularly efficient in the case of updates on sets of sequential keys, a common occurrence in many applications. We use this new data structure to implement a distributed event store, and show its efficiency in very large systems with low rates of updates. In particular, we show that in some scenarios our approach is able to achieve both a 66% reduction of bandwidth cost over a vector-clock approach, as well as a 34% improvement in consistency level. We finally suggest other uses of our construction for distributed databases in open networks.

2.5 A Control Loop-based Algorithm for Operational Transformation[5]

Author : C. Gadea, B. Ionescu and D. Ionescu

Year : 2020

Description:

Operational Transformation (OT) has emerged as a viable theoretical principle for the implementation of real-time collaboration applications. In such systems, the collaboration consists of operations generated by members of a group who are performing concurrent actions on the same document or content. This powerful multi-user co-editing has been researched ever since the seminal works of the late 1980s. As the web evolved into a dominant platform for content consumption and creation, classes of algorithms like OT and Conflict-free Replicated Data Types (CRDT) have enabled flexible content synchronization for applications such as online word processors. Despite their long history in academia, OT and CRDT continue to have unsolved issues due to the centralized approach required for scalable and reliable web-based document editing. This paper proposes a Control Loop-based OT approach based on a serverless architecture and on Finite State Automata (FSA). A control loop principle is used to design a series of algorithms for distributed conflict resolution. The proposed architecture consists of a series of blocks which internally contain a number of multi-level Finite State Machines. The architecture of the new serverless approach for OT is introduced and the basic FSAs that model the co-editing processes are described. Cases encountered in the dynamics of the co-editing processes were modeled to prove that the essential OT properties of causality preservation, convergence, and intention preservation are all satisfied. Simulation results are given at the end of the paper.

2.6 A Conflict-Free Replicated JSON Datatype[6]

Author : M. Kleppmann and A. R. Beresford

Year : 2017

Description:

Many applications model their data in a general-purpose storage format such as JSON. This data structure is modified by the application as a result of user input. Such modifications are well understood if performed sequentially on a single copy of the data, but if the data is replicated and modified concurrently on multiple devices, it is unclear what the semantics should be. In this paper we present an algorithm and formal semantics for a JSON data structure that automatically resolves concurrent modifications such that no updates are lost, and such that all replicas converge towards the same state (a conflict-free replicated data type or CRDT). It supports arbitrarily nested list and map types, which can be modified by insertion, deletion and assignment. The algorithm performs all merging client-side and does not depend on ordering guarantees from the network, making it suitable for deployment on mobile devices with poor network connectivity, in peer-to-peer networks, and in messaging systems with end-to-end encryption.

2.7 Concurrency control and awareness support for multi-synchronous collaborative editing[7]

Author : M. Ahmed-Nacer, P. Urso, V. Balegas and N. Preguiça

Year : 2013

Description:

Collaborative editing tools have become increasingly popular in the last decade, with some systems being used by massive numbers of users. While traditionally collaborative editing systems would either target synchronous or asynchronous collaboration settings, some recent systems support both types of collaboration, even supporting disconnected work. The proposed concurrency control algorithm, based on conflict-free data types, builds on the ideas previously developed for synchronous collaboration, extending them to support asynchronous collaboration. The evaluation of our algorithm shows that comparing our solution with traditional solutions in collaborative editing, the conflict resolution strategy proposed in this paper leads to results closer to the ones expected by users.

2.8 Delta-State-Based Synchronization of CRDTs in Opportunistic Networks[8]

Author : F. Guidec, Y. Mahéo and C. Noûs

Year : 2021

Description:

Conflict-Free Replicated Data Types (CRDTs) are distributed data types that support optimistic replication: replicas can be updated locally, and updates propagate asynchronously among replicas, so consistency is eventually obtained. This ability to tolerate asynchronous communication makes them ideal candidates to serve as software building blocks in opportunistic networks (OppNets), that is, mobile networks in which the dissemination of information can only depend on unpredicted transient radio contacts between pairs of nodes. In this paper we investigate the problem of implementing CRDTs in an Opp-Net, and we propose a delta-state-based algorithm to solve this problem. Experimental results confirm that this algorithm ensures the synchronization of CRDT replicas in an OppNet, and that it outperforms a pure state-based synchronization algorithm when dealing with container CRDTs.

2.9 Efficient Synchronization of State-Based CRDTs[9]

Author : V. Enes, P. S. Almeida, C. Baquero and J. Leitão

Year : 2019

Description:

To ensure high availability in large scale distributed systems, Conflict-free Replicated Data Types (CRDTs) relax consistency by allowing immediate query and update operations at the local replica, with no need for remote synchronization. State-based CRDTs synchronize replicas by periodically sending their full state to other replicas, which can become extremely costly as the CRDT state grows. Delta-based CRDTs address this problem by producing small incremental states (deltas) to be used in synchronization instead of the full state. However, current synchronization algorithms for delta-based CRDTs induce redundant wasteful delta propagation, performing worse than expected, and surprisingly, no better than state-based. In this paper we: 1) identify two sources of inefficiency in current synchronization algorithms for delta-based CRDTs; 2) bring the concept of join decomposition to state-based CRDTs; 3) exploit join decompositions to obtain optimal deltas and 4) improve the efficiency of synchronization algorithms; and finally, 5) experimentally evaluate the improved algorithms.

2.10 Edge Applications: Just Right Consistency[10]

Author : A. Ahuja, G. Gupta and S. Sidhanta

Year : 2019

Description:

CRDTs are distributed data types that make eventual consistency of a distributed object possible and non ad-hoc. Geo-distributed systems are spread across multiple data centers at different geographic locations to ensure availability and performance despite network partitions. These systems must accept updates at any replica and propagate these updates asynchronously to every other replica. Conflict-Free Replicated Data Types (CRDTs) ensures eventual consistency in the replicas despite asynchronous delivery of updates. Extending this idea to fog computing servers where connection reliability is low, eventual consistency amongst the servers is required. We configure Kubernetes, an open-source container orchestration system used for automating deployment, scaling, and management of containerized applications, and use it for cluster deployment of CRDT based low resource intensive AntidoteDB can be used for deployment on fog servers to ensure eventual consistency amongst these servers. We have developed an automated benchmarking tool for benchmarking edge computing applications.

2.11 Conflict-Free Partially Replicated Data Types[11]

Author : I. Briquemont, M. Bravo, Z. Li and P. Van Roy

Year : 2015

Description:

Designers of large user-oriented distributed applications, such as social networks and mobile applications, have adopted measures to improve the responsiveness of their applications. Latency is a major concern as people are very sensitive to it. Geo-replication is a commonly used mechanism to bring the data closer to clients. Nevertheless, reaching the closest data center can still be considerably slow. Thus, in order to further reduce the access latency, mobile and web applications may be forced to replicate data at the client-side. Unfortunately, fully replicating large data structures may still be a waste of resources, especially for thin-clients. We propose a replication mechanism built upon conflict-free replicated data types (CRDT) to seamlessly replicate parts of large data structures. The mechanism is transparent to developers and gives improvements without increasing application complexity. We define partial replication and give an approach to keep the strong eventual consistency properties of CRDTs with partial replicas. We integrate our mechanism into Swift Cloud, a transaction system that brings Geo-replication to clients. We evaluate the solution with a content-sharing application. Our results show improvements in bandwidth, memory, and latency over both classical Geo-replication and the existing Swift Cloud solution.

Chapter 3

Project Description

3.1 Text Editor

A text editor is a space where you can **insert** or **delete** text characters and then save the resulting text to a file. Each character has a value and a numerical index that determines its position in the document. For example, with the text “HAT”, the first character has a value “H” and a position of 0, “A” has position 1, and “T” has position of 2.

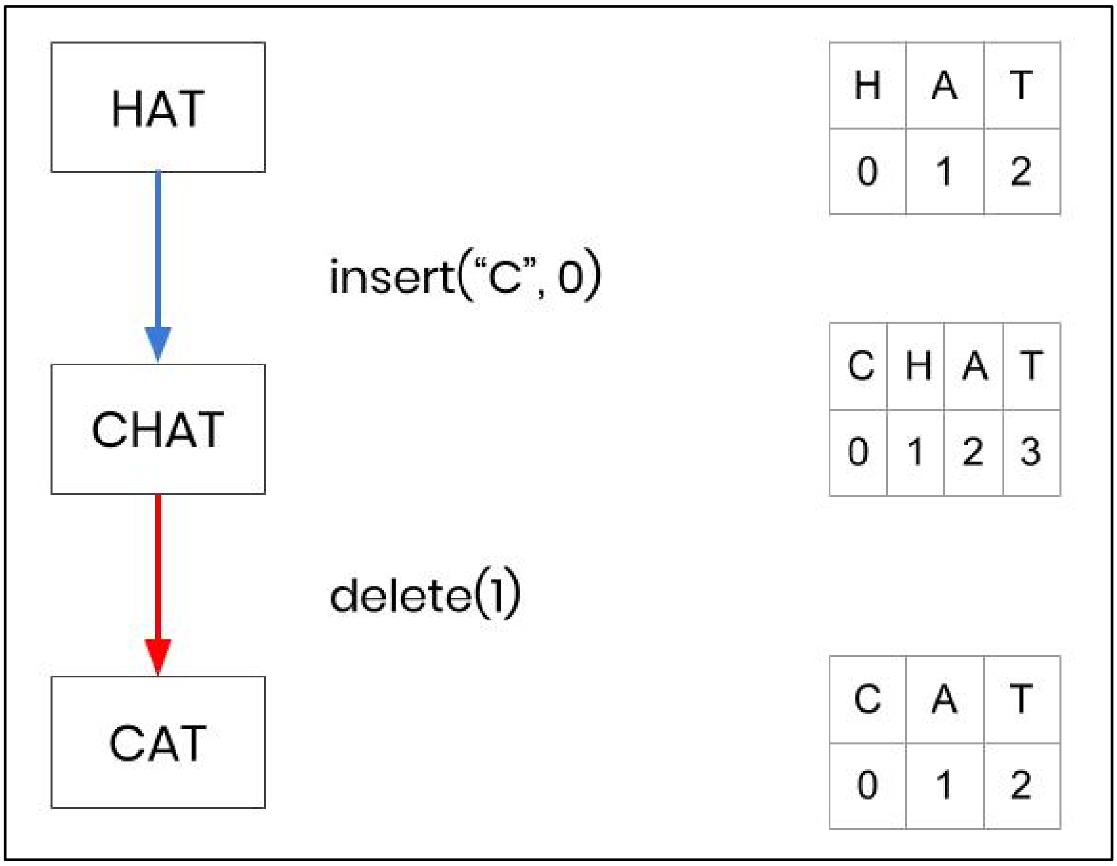


Fig.1. The character positions may shift as surrounding characters are inserted or deleted.

A character can be inserted or deleted from the text simply by referencing a positional index. To insert a “C” at the beginning of the text, the operation is insert("C", 0). This insertion results in the remaining positions being shifted (or incremented) by 1. Now to delete the “H”, the operation is delete(1).

3.2 Real-time collaborative text editor

Collaborative editing is an advanced feature of many document edit-ing software. In such distributed systems, each node (for example a client application) has a replicated instance of the object being edited,called a replica. In order for the editing to be done in real-time, providing a responsive and interactive experience for the user, the local replica is first updated and the change is later sent to the other replicas. This approach is called optimistic replication and comes with a series of challenges which are described in the following sections. In order to build a real-time collaborative data store, these challenges need to be addressed. A general purpose data store should support a variety of data types, such as Map, List, Boolean, Number and String.

Next, we need a way for users to inform other users of edits they made to the document. We'll introduce a Central Relay Server to facilitate this communication.

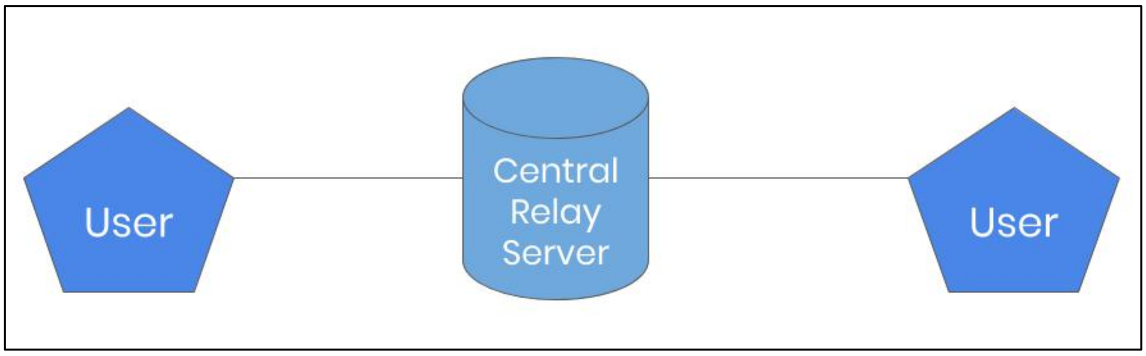


Fig.2. Two users connected through a central relay server.

Now inserting a "C" and deleting the "H" as we did with our simple text editor. This time, however, we'll have one user perform the insertion and the other user perform the deletion. After performing their respective operations, each user will send a message to the other user informing them about the operation.

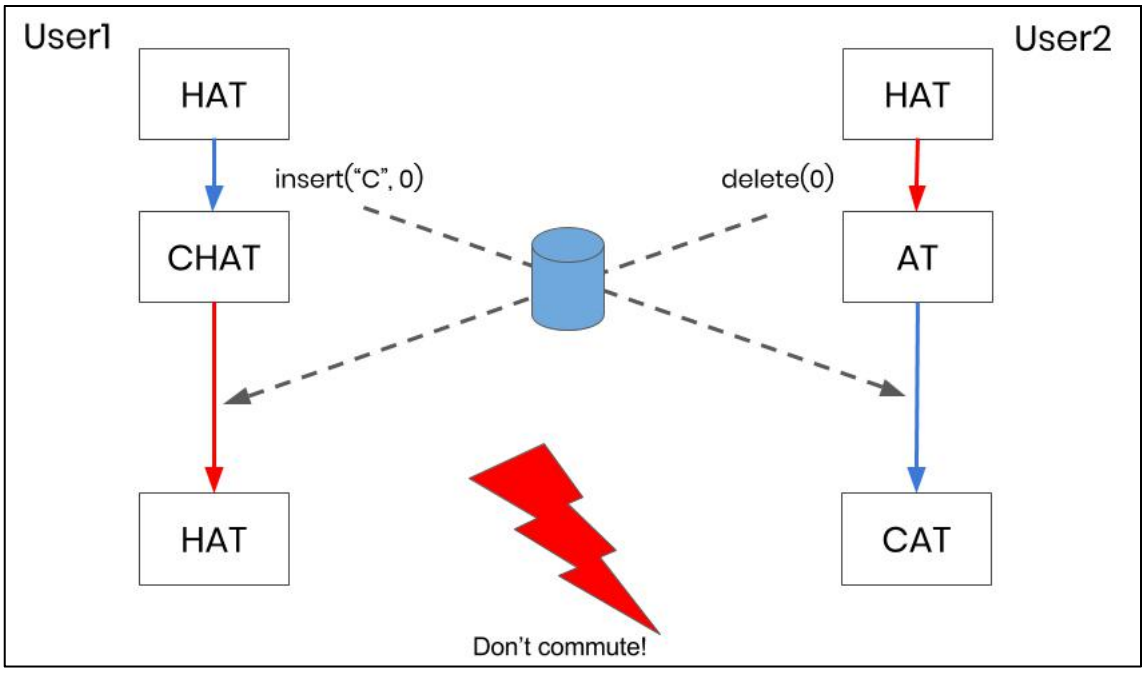


Fig.3. Simultaneous insertion and deletion produce different results.

Now, One user has a “HAT” and the other user has a “CAT”. In this particular example, their documents did not converge to the same state. This example demonstrates one of the primary challenges with building a collaborative text editor — getting all users to converge to the same document state.

The reason that the users’ document didn’t converge is because the insert and delete operations were applied in different orders. In mathematical terms, the operations did not commute. For non-mathematicians like ourselves, let’s review what commutativity means. Commutativity occurs when operations applied in different orders produce the same result. For example, addition is commutative because A + B = B + A. Subtraction, however, is not commutative because A - B != B - A.

Let’s try another example where the users simultaneously decide they want to delete the “H” from “HAT” to get “AT”.

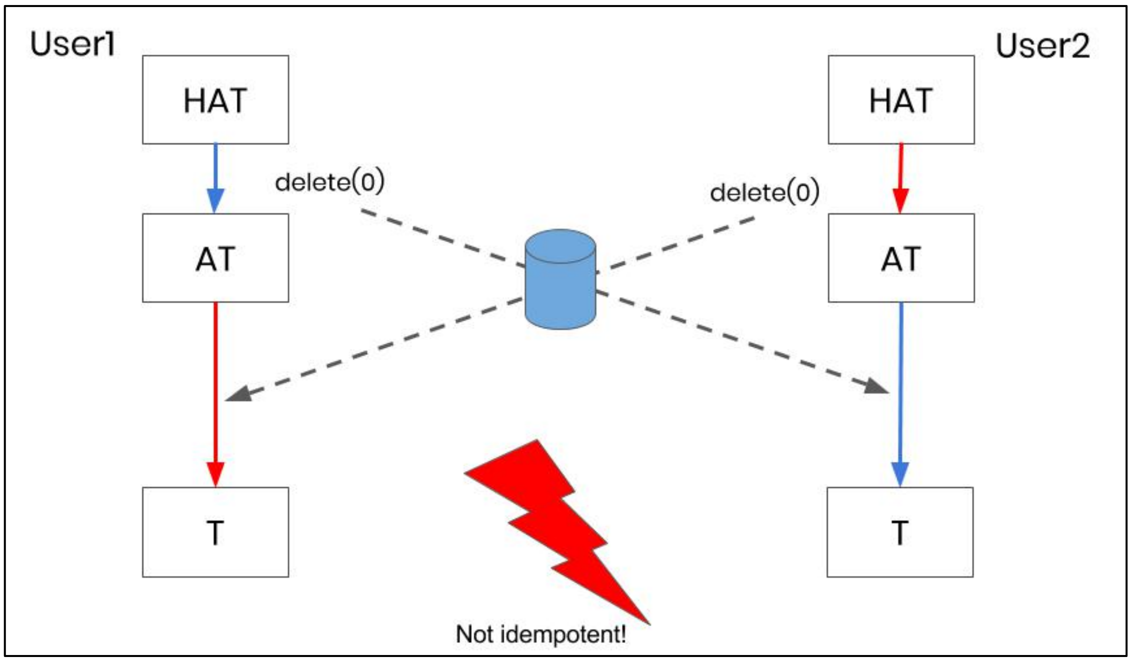


Fig.4. Duplicate deletion operations are not idempotent.

Though it converged, we have another problem! Both users ended up with an “T” instead of a “AT”, which neither of them wanted. In mathematical terms, the delete operations are not idempotent. What the heck does that mean? Idempotent occurs when repeated operations produce the same result. For example, multiplying by 1 is an idempotent operation. No matter how many times you multiply a number by 1, the result is the same.

With our text editor use case, if both users try to insert the same letter in the same position at the same time, it’s acceptable for the result to be two duplicate characters because it’s easy to find and delete the unnecessary character. When it comes to deletes, however, the character is simply gone without a record of it ever existing; a user would have to remember the value and position of the character in order to insert it back in the document.

Through these examples, we’ve seen that a collaborative text editor is different from a simple text editor. Further, we’ve concluded that when users make concurrent edits to a shared document, the insert and delete operations must commute and the delete operations must be idempotent.

- **Commutativity**: Concurrent insert and delete operations converge to the same result regardless of the order in which they are applied.
- **Idempotent**: Repeated delete operations produce the same result.

3.2.1 Operational transformation

Operational transformation (OT) is one of the first collaborative editing approaches and was described by Ellis and Gibbs in 1989[14]. It works by a central server which keeps track of the state that each client is in. When a client updates the state, the node optimistically executes the operation locally. The operation is then sent to the server which transforms the operation for each other client so that all clients end up in the same state after executing the operation. The transformed operations are sent to the clients which execute them in their local state.Conflicting updates are resolved using tie breaking rules.A number of OT algorithms for collaborative text editing have been developed, for example GOTO[17]. One of the most well known applications of OT is Google Docs. All OT algorithms require a central server that transforms the operations for each client are also complex to implement and use ad-hoc rules to resolve conflicts. A peer-to-peer solution for collaborative text editing called WOOT (Without Operational Transform)was presented in 2005[15]. The solution was simpler than previous OT algorithms and is formally proven to be correct. WOOT created a new direction for collaborative editing techniques which were later labeled as conflict-free replicated data types.

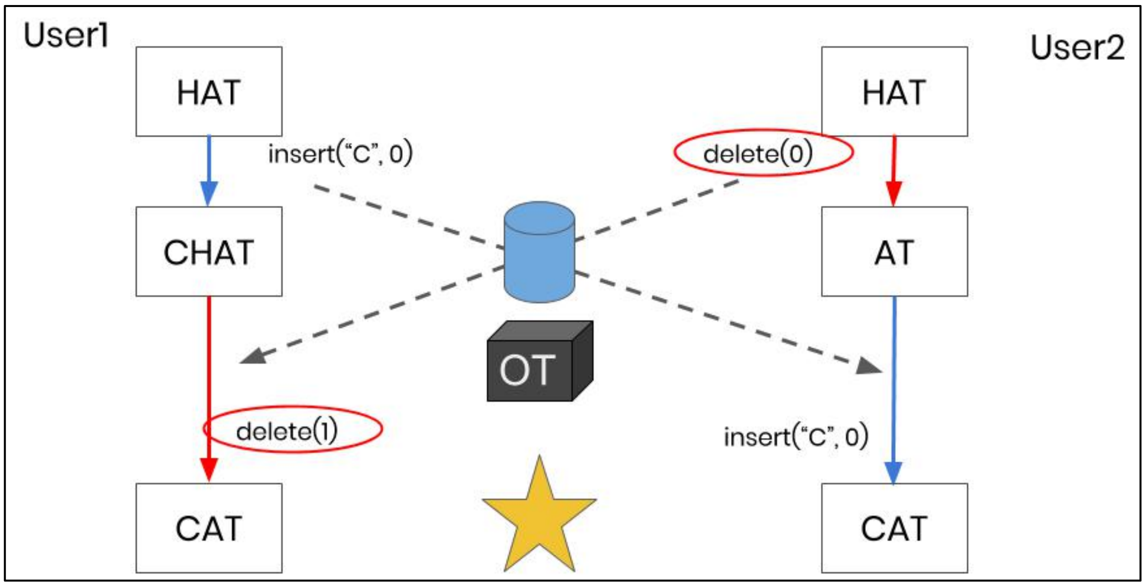


Fig.5. Insertion and deletion operations commute with OT.

We can imagine that when a user tries to delete a character that's already been deleted, OT can easily recognize this and skip the operation. OT solves our challenge by providing a strategy for achieving the commutativity and idempotency we need.

OT was the first popular way to allow for collaborative editing. The first collaborative editors, Google Wave, Etherpad, and Firepad, all used OT. Unfortunately, the verdict from those products is that it's extremely tough to actually implement.

3.2.2 Conflict-free replicated data types

Conflict-free replicated data types (CRDTs) were formally introduced in 2011[20]. CRDTs are replicated objects that provide SEC. Assuming Eventual delivery, CRDTs guarantee that they converge to the samestate. Unlike OT algorithms, CRDTs do not require a central server that transforms the operations and can thus be used for peer-to-peer replication of data. CRDTs need to fulfill a set of mathematical prop-erties and can therefore be proven to be correct. CRDTs have existed long before the concept was formalized and use cases have been found in distributed file systems and databases, among others. CRDTs can be categorized into two general types, state-based andoperation-based. The main difference is the way that the data type replicates its state. The state-based version applies operations locally and then sends the new state to the other nodes for replication. The Operation-based version instead sends the operations themselves for replication.

3.3 CRDT

3.3.1 State-based CRDTs

State-based CRDTs, also known as convergent replicated data types,send the full state to replicate data[19][20]. On receiving a remotestate, the node merges the remote with the local state. In order for the replicas to converge to the same state, the merge function must fulfill the following properties:

- Commutative: f (a, b) = f (b, a)
- Associative: f (a, f (b, c)) = f (f (a, b), c)
- Idempotent: f (f (a)) = f (a)

where f is the merge function a and b are two states. The key characteristic of state-based CRDTs is that the number of transmissions and delivery order does not matter, the data will still be SEC. However, one issue with state-based CRDTs is that they can be inefficient if the data transmitted is big. There are solutions to this such as delta state replication[18], but they require complex protocols to resolve what needs to be transmitted in order to converge.

3.3.2 Operation-based CRDTs

Operation-based CRDTs, also known as commutative replicated data types, replicate by sending individual operations over the network[19][20]. An operation-based CRDT has to fulfill the following properties:

- Operations have to be delivered exactly once in causal order to all replicas.
- Causally independent operations have to be commutative when applied to the local state.

The main benefit of operation-based CRDTs is the small amount of data is sent over the network compared to state-based. However, the implementation of the delivery mechanism is a lot harder. The efficient use of the network makes operation-based CRDTs preferable for real-time collaborative editing since we want to send small incremental updates frequently in order to achieve a real-time feeling of the application.

CRDTs take a different approach. Researchers realized that there’s no reason to treat the characters as just having a value and absolute position; they set out to change the underlying data structure of the text editor. Instead, properties are added to each character object that enables commutativity and idempotent. Using a more complex data structure allows for a much simpler algorithm than OT.

Note: There are many different types of CRDTs with different requirements for different use cases. In this context, 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.

3.3.3 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.

With globally unique characters, when a user sends a message to another user to delete a character, it can indicate precisely which character to delete. Let’s see how this changes our earlier example.

Note: For the purpose of simplifying the example, the globally unique ids are just integers. In reality, they’ll be objects with Site ID and Site Counter properties.

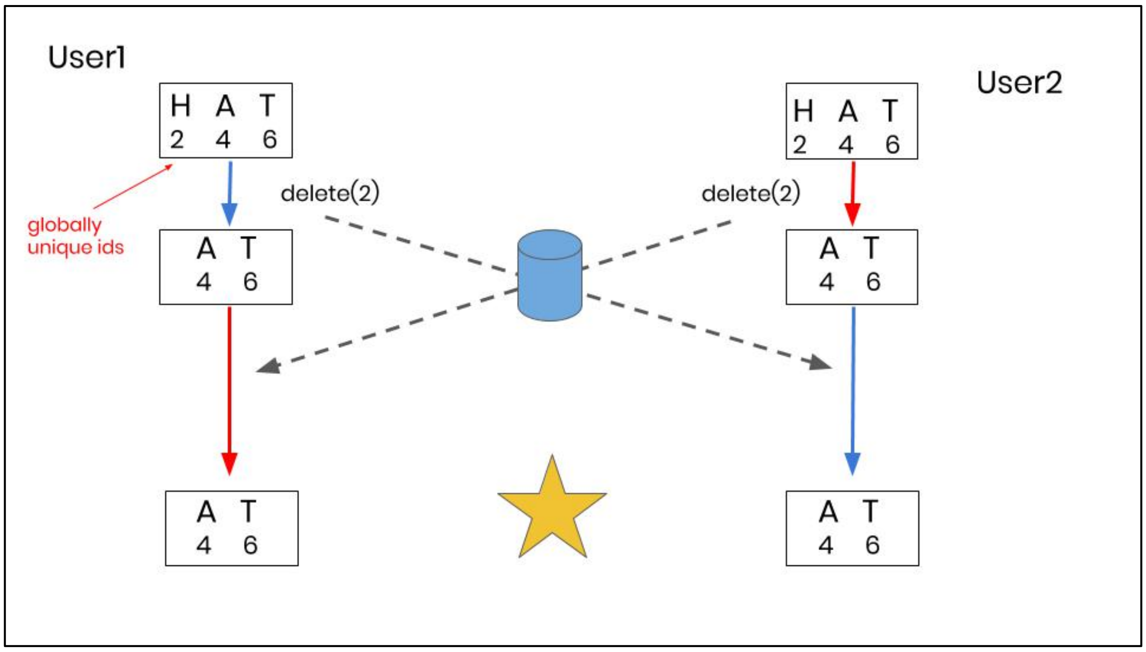


Fig.6. Duplicate deletion operations are idempotent with a CRDT.

With our CRDT, when a user receives a delete operation from another user, it looks for a globally unique character to delete. And if it has already deleted that character, then there's nothing more it can delete. By ensuring globally unique character objects, we've achieved idempotent of delete operations.

3.3.4 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.

In our initial text editor example, we noted that when inserting or deleting a character, the positions of surrounding characters would sometimes need to be shifted accordingly. And since characters can be shifted by a user without the other users' knowledge, we can end up in a situation where insert and delete operations do not commute.

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

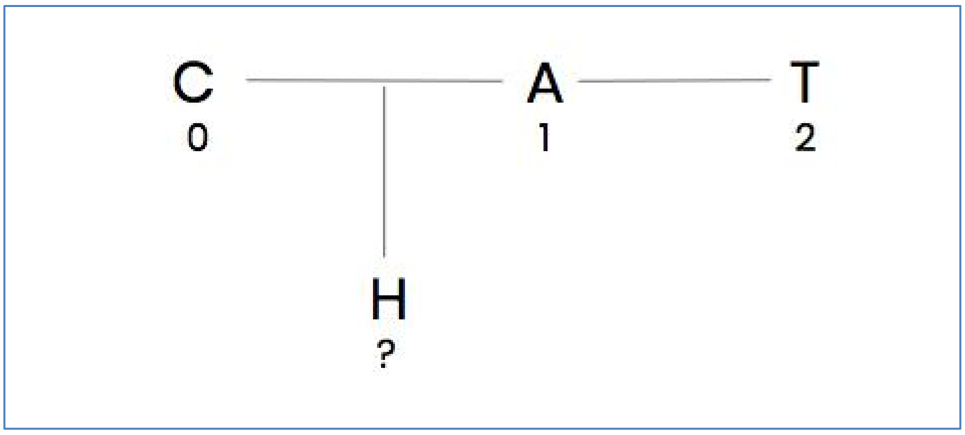


Fig.7. Inserting a character in between two existing characters.

Using fractional indices, instead of inserting “H” at position 1, we insert “H” at a position between 0 and 1 (e.g. 0.5). We represent fractional indices in code as a list of integers (or position identifiers). For example, O.5 is represented as [0, 5].

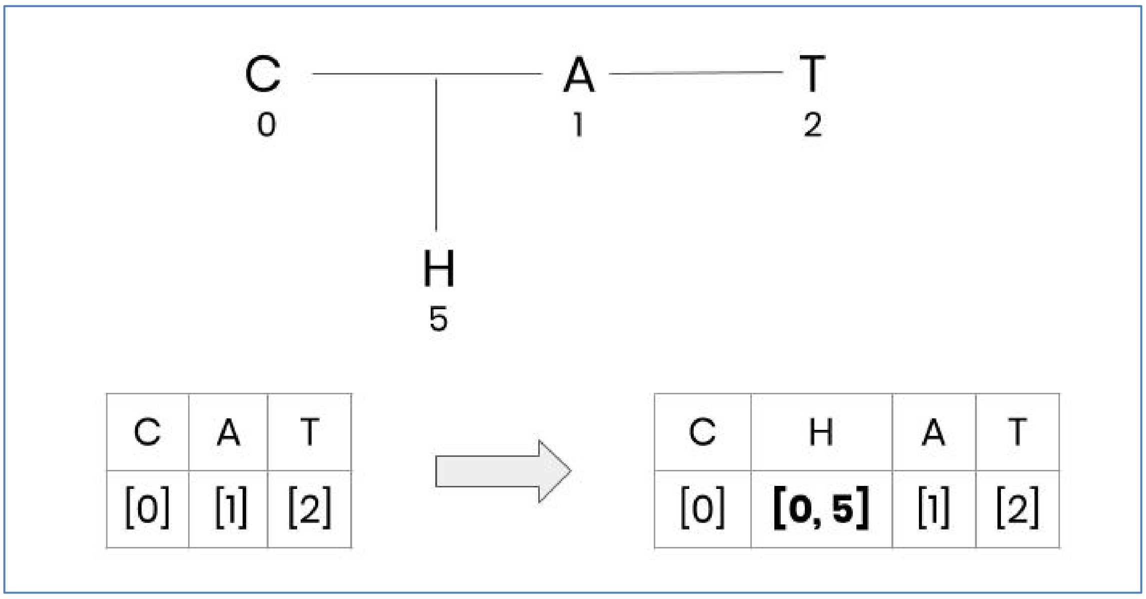


Fig.8. Inserting a character at a position in between two existing characters.

The key is that by using fractional indices to insert characters, we never have to shift the positions of surrounding 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.

Note: 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.

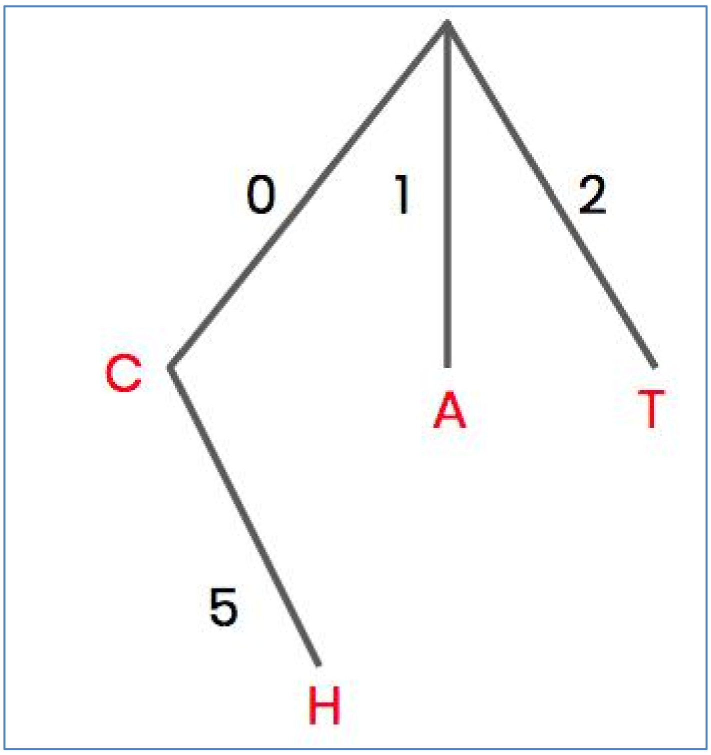


Fig. 9. Relative positions are like a tree

With globally ordered characters using position identifiers, let's revisit our example of simultaneous insert and delete operations.

Note: For the purpose of simplifying the example, we've omitted the Site ID and Site Counter properties. The globally unique ids here are just the fractional positions. In reality, they'll be more complex objects.

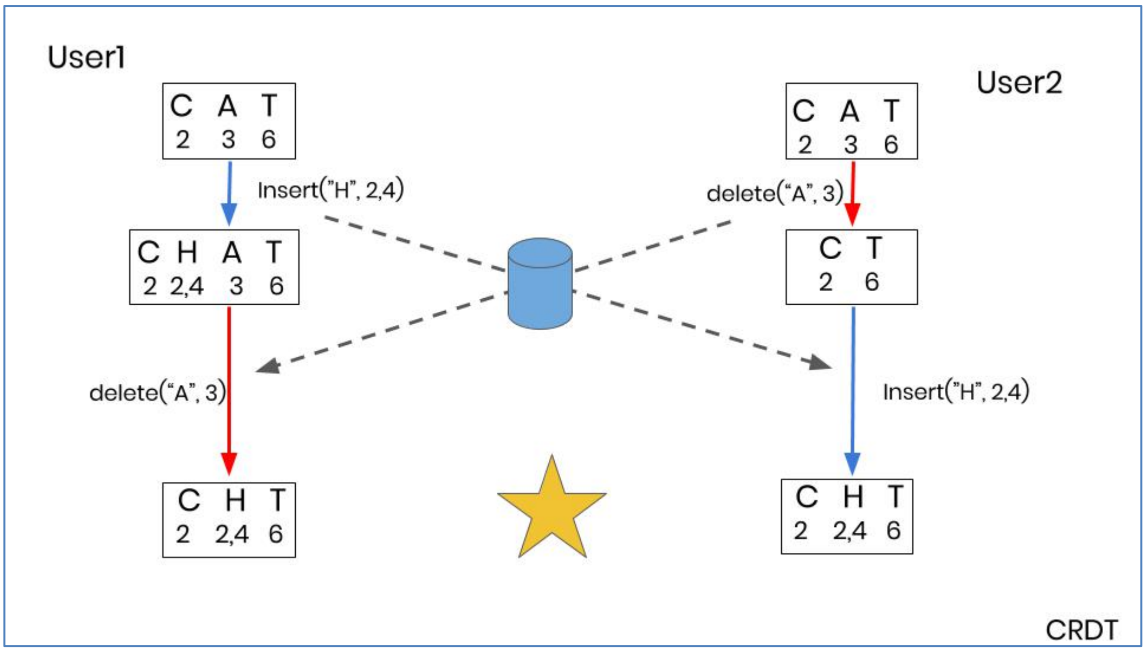


Fig.10. Insert and Delete operations commute using a CRDT.

With our CRDT, we now insert globally unique characters with fractionally indexed positions. The result is that deleting a particular character has absolutely no effect on the insertion of a new character. In other words, our insertion and deletion operations now commute!

If you're paying close attention, you might ask what happens if two users insert the same character in the same position at the same time. To ensure that these characters are still globally unique, we attach the Site ID to the end of the character's position identifier, and that Site ID is used as a tiebreaker in the event that multiple users inserted the same character in the same position.

Chapter 4

Software Design

The current system architecture relies on the client-server model of communication. It supports multiple users editing a shared document, and between all of our users lies a central server that acts as a relay by forwarding operations to every user in the network of that shared document.

4.1 Architecture

We started with this model because it allowed us to first focus on resolving editing conflicts among users. Now that we have achieved that, could we change our application architecture to a better model? Before we get into what we changed, let's talk about the limitations of our current central server architecture.

The first limitation is that we currently have an unnecessarily high latency between users. All operations are currently routed through the server, so even if users are sitting right next to each other, they still must communicate with each other through the server.

4.1.1 Peer-to-Peer Architecture

We can remove these limitations by switching to a peer-to-peer architecture where users send operations directly to each other. In a peer-to-peer system, rather than having one server and many clients, each user (or peer) can act as both a client and a server. This means that instead of relying on a server to relay operations, we can have our users perform that work for free (at least in terms of money). In other words, our users will be responsible for relaying operations to other users they're connected to.

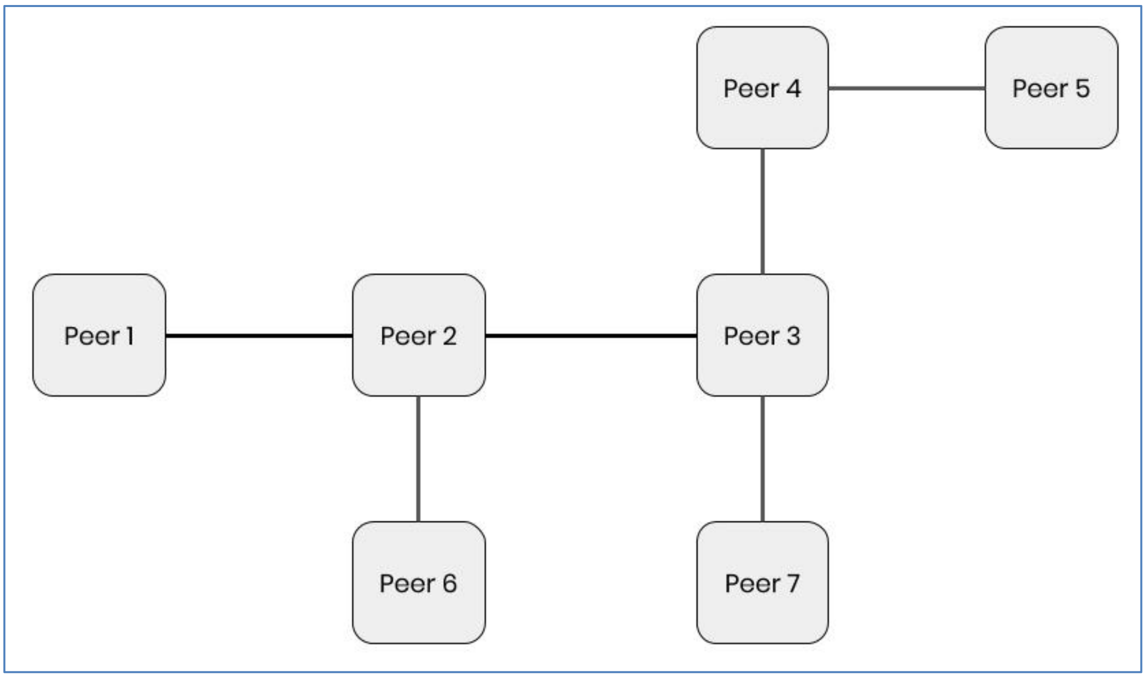


Fig.11. peer to peer network.

To allow nodes to send and receive messages, we used a technology called 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".

It's important to mention that while WebRTC relies on this signaling server, no document content will ever travel through it. It's simply used to initiate the connection. Once a connection is established, the server is actually no longer necessary.

When a user first opens Conclave, the application establishes a WebSocket connection with the server. Using that connection, the app "registers" with the signaling server, essentially letting it know where it's located. The server responds by assigning a random, unique Peer ID to the user. The application then uses the assigned Peer ID to create a Sharing Link to display to each user.

4.1.2 System Architecture

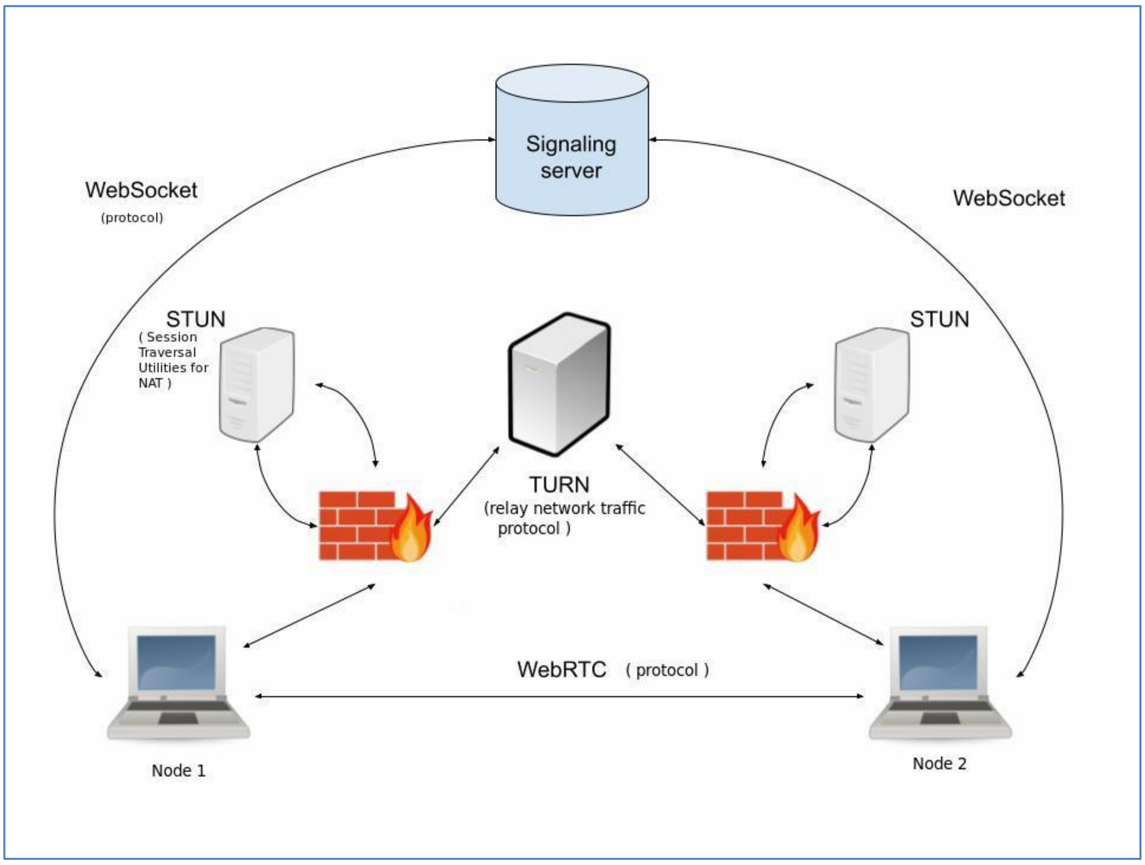

Signaling server
WebSocket (protocol)
WebSocket
STUN (Session Traversal Utilities for NAT)
STUN
TURN (relay network traffic protocol)
WebRTC (protocol)
Node 1
Node 2

Fig.12. System architecture.

The link is unique to each user and is essentially a pointer to a particular user. A user can share their link with anyone, and upon clicking the link, the collaborator will automatically be connected to the user and able to collaborate on the shared document. To implement signaling and WebRTC messaging, we used a library called PeerJS which took care of a lot of this stuff behind the scenes for us. For example, when a user clicks another user's sharing link, it's essentially asking the signaling server to broker a connection between them. The server responds by providing the user with the other user's IP address, allowing the user to send messages to the other user.

Since most internet users use wireless routers, the public IP address is found using a STUN server. The connecting user then uses the IP address to establish a WebRTC connection, and once established, content can be sent directly between users. In the case that a connection with the STUN server cannot be made and the WebRTC connection fails, a TURN server is used as a backup to send operations between users.

4.1.3 Version Vector

WebRTC uses the UDP transport protocol. UDP is a lightweight message protocol that allows it to send messages quickly and without waiting for a response from the other user.

One drawback to UDP is that it does not guarantee in-order packet delivery. That means that our messages may be received in a different order than they were sent. This presents a potential issue. What if a user receives a message to delete a particular character before it's actually inserted?

Let's say we have 3 peers collaborating on a document. Two of the peers are next to each other while the third is far away. Peer1 types an "A" and sends the operation out to both peers. Since Peer2 is nearby, it quickly receives the operation but decides it doesn't like it and promptly deletes it.

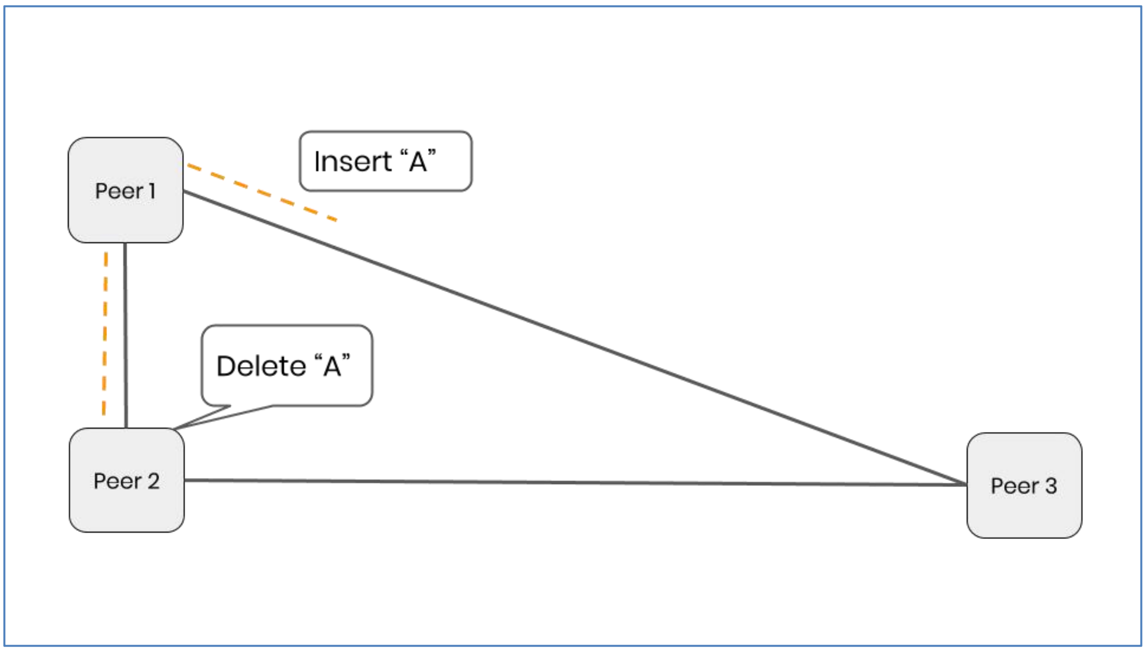


Fig.13 Version Vector insertion and deletion.

What happens if the delete operation arrives at Peer3 before the insert operation? We wouldn't want to apply the delete first because there'd be nothing to delete and the operation would be lost. Later, when the insert is applied, Peer3's document would look different from the others. We need to find a way to wait to apply the delete operation only after we've applied the insert.

To solve the out-of-order messages problem, we built what's called a Version Vector. It sounds fancy but it's simply a strategy that tracks which operations we've received from each user.

Whenever an operation is sent out, in addition to the character object and whether it's an insertion or deletion, we also include the character's Site ID and Site Counter value. The SiteID indicates who originally sent the operation, and the Counter indicates which operation number it is from that particular user.

When a peer receives a delete operation, it's immediately placed in a Deletion Buffer. If it were an insert, we could just apply it immediately. But with deletes, we have to make sure the character has been inserted first.

After every operation (insert or delete), the deletion buffer is "processed" to check if the characters have been inserted yet. In this example, the character has a SiteID of 1 and Counter of 24.

To perform this check, Peer3 consults its version vector. Since Peer3 has only seen 23 operations from Peer1, the delete operation will remain in the buffer.

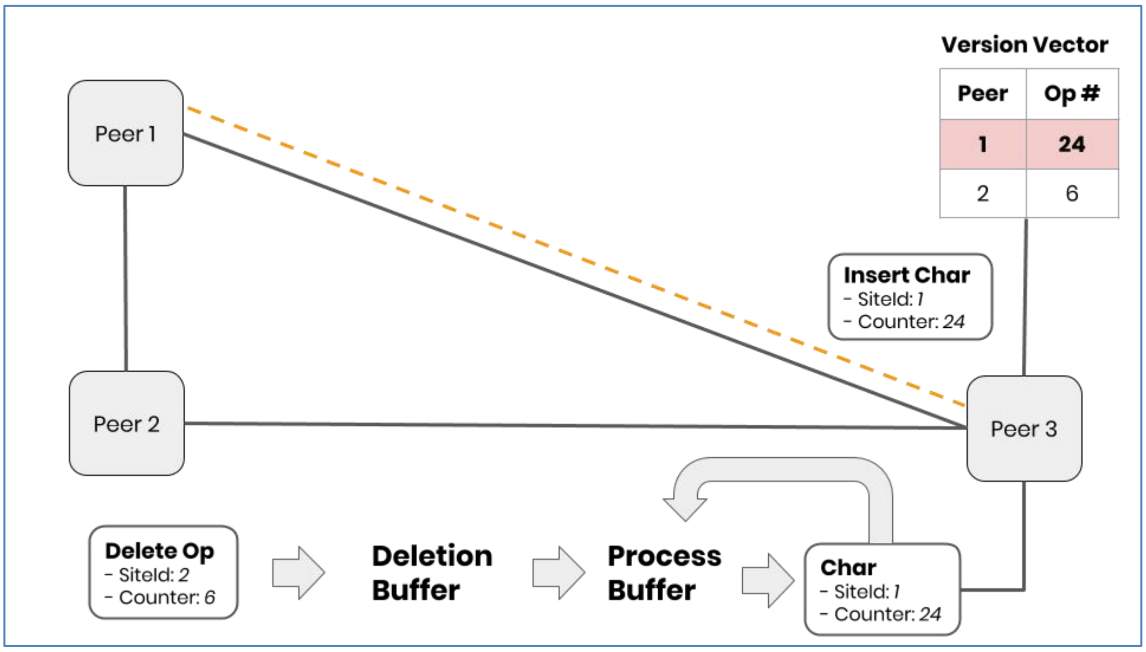


Fig.14. Version Vector insertion and deletion operations process

4.1.4 Final System Architecture

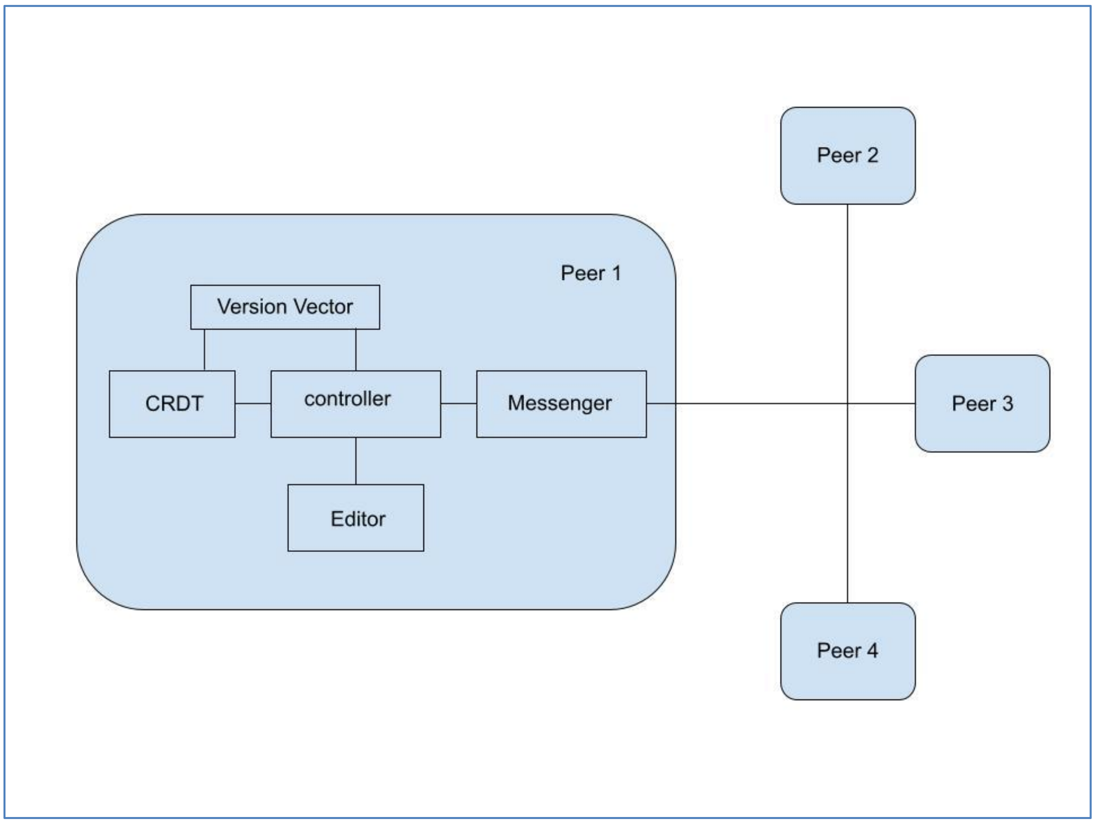


Fig.15. Final System Architecture

At this point, we've described the major components of our system architecture. Within every instance of our application, a custom-built CRDT works together with a Version Vector to make sure our document replicas all converge. The Messenger is responsible for sending and receiving WebRTC messages. And of course, the Editor allows a user to interact with their local copy of the shared document.

4.2 UML Diagram

Modelling is a process through which the requirements are translated into representation of the software with the help of UML. [Unified Modelling Language]. Design is the means of accurately transferring the customer requirements into the finished product. UML is a standard language for specifying, visualizing, constructing, and documenting the artifacts of software systems.

UML was created by the Object Management Group (OMG) and UML 1.0 specification draft was proposed to the OMG in January 1997.
OMG is continuously making efforts to create a truly industry standard.

- UML stands for Unified Modelling Language.
- UML is different from the other common programming languages such as C++, Java, COBOL, etc.
- UML is a pictorial language used to make software blueprints.
- UML can be described as a general-purpose visual modelling language to visualize, specify, construct, and document software system.
- Although UML is generally used to model software systems, it is not limited within this boundary. It is also used to model non-software systems as well. For example, the process flow in a manufacturing unit, etc.

UML is not a programming language but tools can be used to generate code in various languages using UML diagrams. UML has a direct relation with object oriented analysis and design. After some standardization, UML has become an OMG standard.

4.2.1 Use Case Diagram

Use case diagram is a behavioral UML diagram type and frequently used to analyse the type of the system. They enable you to visualize the different type of the roles that interact with the system. Use case diagrams are used to gather the usage requirements of the system. Depending on your requirements you can use the data in different ways.

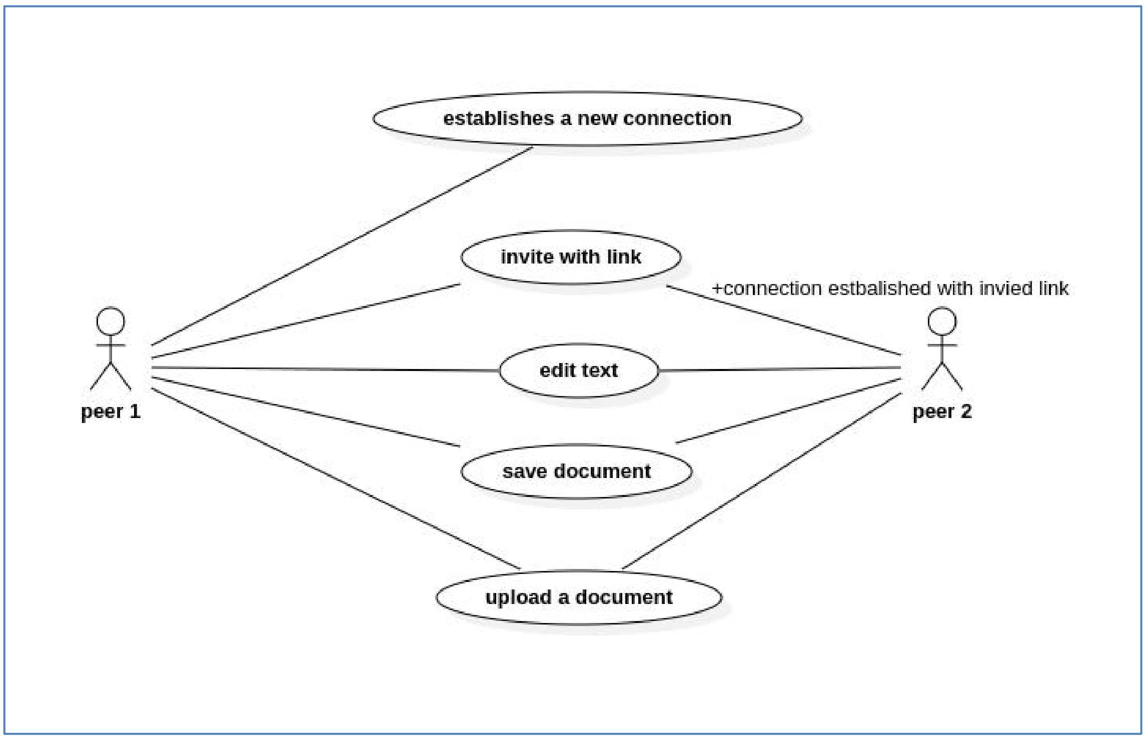

establishes a new connection
invite with link
+connection estbalished with invied link
edit text
peer 1
peer 2
save document
upload a document

Fig.16. Use Case Diagram

Purpose of Use Case Diagrams

The purpose of use case diagram is to capture the dynamic aspect of a system. However, this definition is too generic to describe the purpose, as other four diagrams (activity, sequence,collaboration, and State chart) also have the same purpose. We will look into some specific purpose, which will distinguish it from other four diagrams.

In brief, the purposes of use case diagrams can be said to be as follows −

- Used to gather the requirements of a system.
- Used to get an outside view of a system.
- Identify the external and internal factors influencing the system.
- Show the interaction among the requirements are actors.

4.2.2 Activity Diagram

When it comes to a project, the entire project is divided into many independent tasks, the sequences or the order of the tasks is quite important. If the sequence is wrong, the end result of the project might not be what the management has expected. Some tasks in the project can safely be projected to other tasks. In a project activity diagram, the sequence of the tasks is simply illustrated.

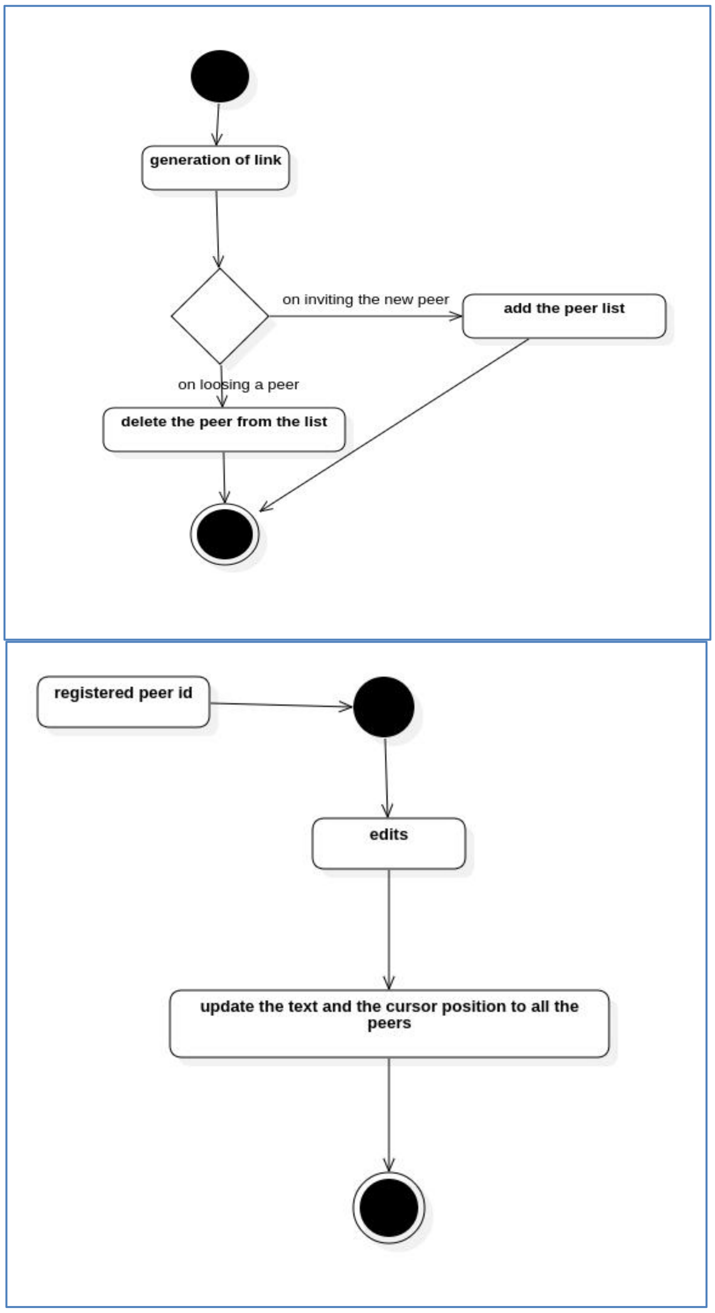


Fig.17. Activity Diagram

4.2.3 Sequence diagram

A sequence diagram is and interaction diagram that shows how objects operate with one another and in what order. It is a construct of a message sequence chart. A sequence diagram shows object interactions arranged in time sequence. It depicts the objects and classes involved in the scenarios and the sequence of the message exchanged between the objects needed to carry out the functionality of the scenario. Sequence diagrams are typically associated with the use case realizations in the logical view of the system under development. Sequence diagrams are sometimes called event scenarios.

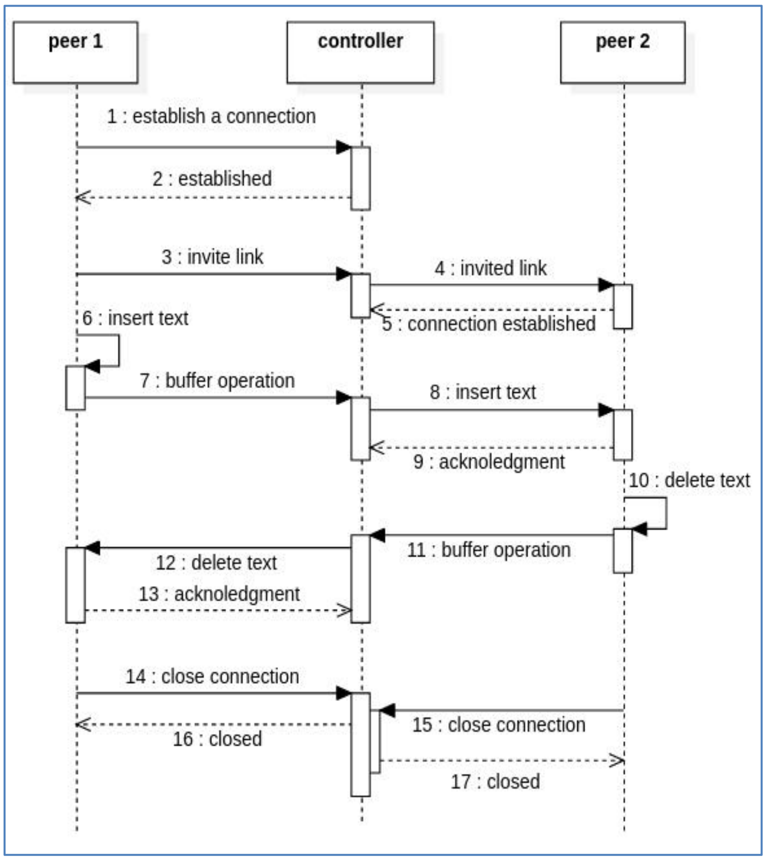


Fig.18 Sequence Diagram

From the term Interaction, it is clear that the diagram is used to describe some type of interactions among the different elements in the model. This interaction is a part of dynamic behaviour of the system.

This interactive behaviour is represented in UML by two diagrams known as Sequence diagram and Collaboration diagram. The basic purpose of both the diagrams are similar.

4.2.4 Class Diagram

In software engineering, a class diagram in the Unified Modeling Language (UML) is a type of static structure diagram that describes the structure of a system by showing the system's classes, their attributes, operations (or methods), and the relationships among objects

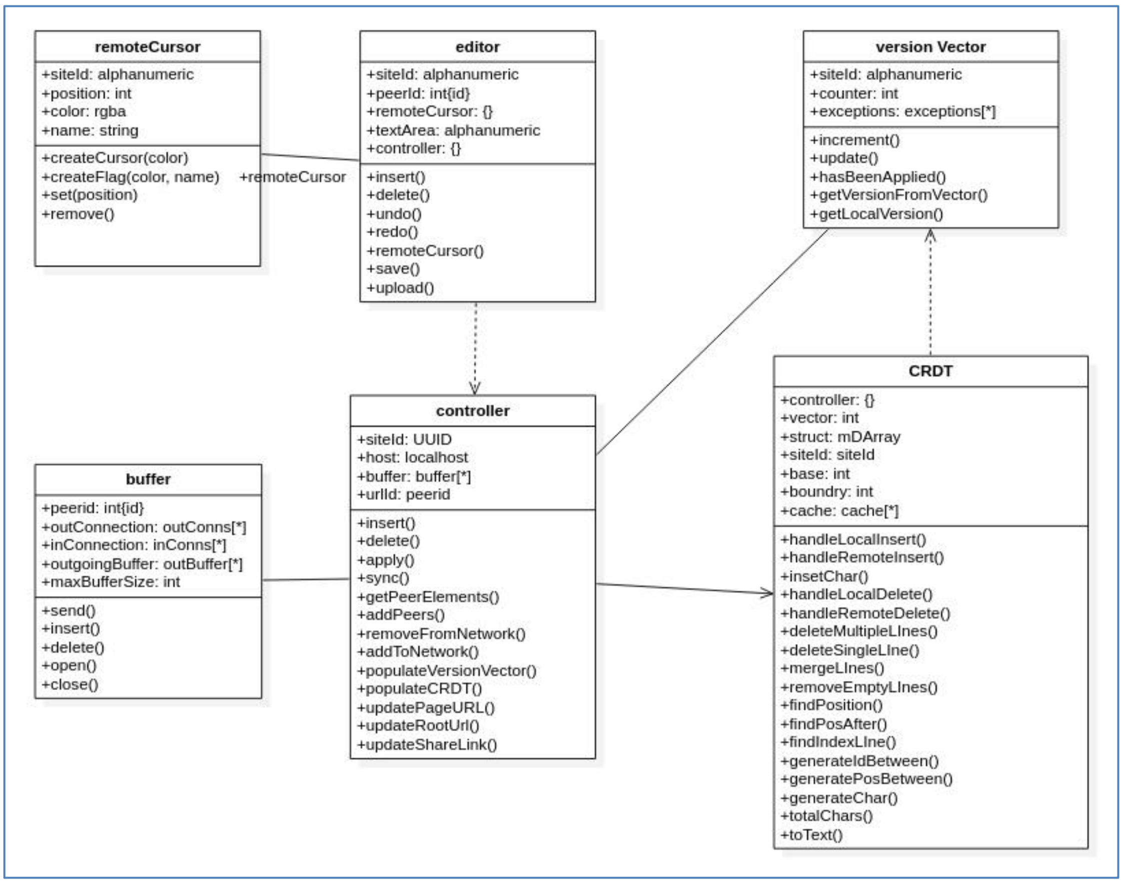


Fig.19 Class Diagram

Purpose of Class Diagrams

- Shows static structure of classifiers in a system
- Diagram provides a basic notation for other structure diagrams prescribed by UML
- Helpful for developers and other team members too
- Business Analysts can use class diagrams to model systems from a business perspective.

4.2.5 Object Diagram

An **object diagram** in the Unified Modeling Language (UML), is a diagram that shows a complete or partial view of the structure of a modeled system at a specific time.

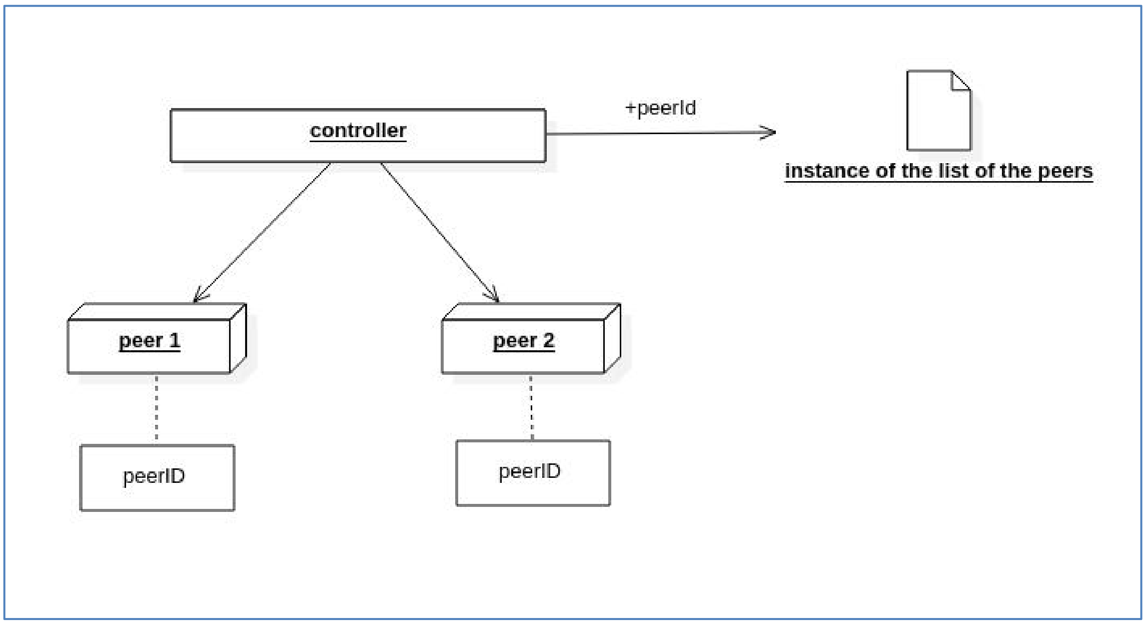


Fig.20 Object Diagram

Purpose of Object Diagrams

The purpose of a diagram should be understood clearly to implement it practically. The purposes of object diagrams are similar to class diagrams.
The difference is that a class diagram represents an abstract model consisting of classes and their relationships. However, an object diagram represents an instance at a particular moment, which is concrete in nature.

It means the object diagram is closer to the actual system behavior. The purpose is to capture the static view of a system at a particular moment.

The purpose of the object diagram can be summarized as −

- Forward and reverse engineering.
- Object relationships of a system
- Static view of an interaction.
- Understand object behaviour and their relationship from practical perspective

Chapter 5

Software and Hardware Requirements

5.1 NodeJS

Node.js is an open-source, cross-platform, back-end JavaScript runtime environment that runs on the V8 engine and executes JavaScript code outside a web browser. Node.js lets developers use JavaScript to write command line tools and for server-side scripting—running scripts server-side to produce dynamic web page content before the page is sent to the user's web browser. Consequently, Node.js represents a "JavaScript everywhere" paradigm, unifying web-application development around a single programming language, rather than different languages for server-side and client-side scripts.

As an asynchronous event-driven JavaScript runtime, Node.js is designed to build scalable network applications.

```
const http = require('http');
const hostname = '127.0.0.1';

const port = 3000;
const server = http.createServer((req, res) => {
  res.statusCode = 200;
  res.setHeader('Content-Type', 'text/plain');
  res.end('Hello World');
});

server.listen(port, hostname, () => {
  console.log(`Server running at http://${hostname}:${port}/`);

});
```

Node.js are free from worries of deadlocking the process, since there are no locks. Almost no function in Node.js directly performs I/O, so the process never blocks except when the I/O is performed using synchronous methods of Node.js standard library. Because nothing blocks, scalable systems are very reasonable to develop in Node.js.

5.2 ExpressJS

Express.js, or simply Express, is a back end web application framework for Node.js, released as free and open-source software under the MIT License. It is designed for building web applications and APIs. It has been called the de facto standard server framework for Node.js.

Example:

```
const express = require('express')
const app = express()
const port = 3000
app.get('/', (req, res) => {
  res.send('Hello World!')
})
app.listen(port, () => {
  console.log(`Example app listening on port ${port}`)
})
```

This app starts a server and listens on port 3000 for connections. The app responds with “Hello World!” for requests to the root URL (/) or route. For every other path, it will respond with a 404 Not Found.

Running Locally

In the myapp directory, create a file named app.js and copy in the code from the example above.

Run the app with the following command:

```
$ node app.js
```

Then, load http://localhost:3000/ in a browser to see the output.

5.3 jQuery

jQuery is a lightweight, "write less, do more"JavaScript library. The purpose of jQuery is to make it much easier to use JavaScript on your website. jQuery takes a lot of common tasks that require many lines of JavaScript code to accomplish, and wraps them into methods that you can call with a single line of code. jQuery also simplifies a lot of the complicated things from JavaScript, like AJAX calls and DOM manipulation.

The jQuery library contains the following features:

- HTML/DOM manipulation
- CSS manipulation
- HTML event methods
- Effects and animations
- AJAX
- Utilities

There are two versions of jQuery available for downloading:

- Production version - this is for your live website because it has been minified and compressed
- Development version - this is for testing and development (uncompressed and readable code)

Both versions can be downloaded from jQuery.com.

The jQuery library is a single JavaScript file, and you reference it with the HTML <script> tag (notice that the <script> tag should be inside the <head> section):

```
<head>
<script src="jquery-3.6.0.min.js"></script>
</head>
```

5.4 PUG

Pug is a high-performance template engine heavily influenced by Haml and implemented with JavaScript for Node.js and browsers.The general rendering process of Pug is simple. pug.compile() will compile the Pug source code into a JavaScript function that takes a data object (called “locals”) as an argument. Call that resultant function with your data, and voilà!, it will return a string of HTML rendered with your data.

The compiled function can be reused, and called with different sets of data.

```
//- template.pug
p #{name}'s Pug source code!



const pug = require('pug');

// Compile the source code
const compiledFunction = pug.compileFile('template.pug');

// Render a set of data
console.log(compiledFunction({
  name: 'Timothy'
}));
// "<p>Timothy's Pug source code!</p>"

// Render another set of data
console.log(compiledFunction({
  name: 'Forbes'
}));
// "<p>Forbes's Pug source code!</p>"
```

Pug also provides the pug.render() family of functions that combine compiling and rendering into one step. However, the template function will be re-compiled every time render is called, which might impact performance. Alternatively, you can use the cache option with render, which will automatically store the compiled function into an internal cache.

5.5 PeerJS

PeerJS wraps the browser's WebRTC implementation to provide a complete, configurable, and easy-to-use peer-to-peer connection API. Equipped with nothing but an ID, a peer can create a P2P data or media stream connection to a remote peer.

Setup

Include the library

```
<script src="https://unpkg.com/peerjs@1.4.5/dist/peerjs.min.js">
</script>
```

Create a peer

```
var peer = new Peer();
```

Data connections

Connect

```
var conn = peer.connect('another-peers-id');
//will be launch when you successfully connect to PeerServer
conn.on('open', function(){
  // here you have conn.id
  conn.send('hi!');
});
```

Receive

```
peer.on('connection', function(conn) {
  conn.on('data', function(data){
    // Will print 'hi!'
    console.log(data);
  });
});
```

PeerServer

To broker connections, PeerJS connects to a PeerServer. Note that no peer-to-peer data goes through the server; The server acts only as a connection broker.

5.6 RxJS

RxJS is a library for composing asynchronous and event-based programs by using observable sequences. It provides one core type, the Observable, satellite types (Observer, Schedulers, Subjects) and operators inspired by Array methods (map, filter, reduce, every, etc) to allow handling asynchronous events as collections.

ReactiveX combines the Observer pattern with the Iterator pattern and functional programming with collections to fill the need for an ideal way of managing sequences of events.

The essential concepts in RxJS which solve async event management are:

- Observable: represents the idea of an invokable collection of future values or events.
- Observer: is a collection of callbacks that knows how to listen to values delivered by the Observable.
- Subscription: represents the execution of an Observable, is primarily useful for canceling the execution.
- Operators: are pure functions that enable a functional programming style of dealing with collections with operations like map, filter, concat, reduce, etc.
- Subject: is equivalent to an EventEmitter, and the only way of multicasting a value or event to multiple Observers.
- Schedulers: are centralized dispatchers to control concurrency, allowing us to coordinate when computation happens on.

Example

Normally you register event listeners.

```
document.addEventListener('click', () => console.log('Clicked!'));
```

Using RxJS you create an observable instead.

```
import { fromEvent } from 'rxjs';
fromEvent(document, 'click').subscribe(() => console.log('Clicked!'));
```

5.7 Acorn

Acorn is designed to support plugins which can, within reasonable bounds, redefine the way the parser works. Plugins can add new token types and new tokenizer contexts (if necessary), and extend methods in the parser object. This is not a clean, elegant API—using it requires an understanding of Acorn's internals, and plugins are likely to break whenever those internals are significantly changed. But still, it is possible, in this way, to create parsers for JavaScript dialects without forking all of Acorn. And in principle it is even possible to combine such plugins, so that if you have, for example, a plugin for parsing types and a plugin for parsing JSX-style XML literals, you could load them both and parse code with both JSX tags and types.

A plugin is a function from a parser class to an extended parser class. Plugins can be used by simply applying them to the Parser class (or a version of that already extended by another plugin). But because that gets a little awkward, syntactically, when you are using multiple plugins, the static method Parser.extend can be called with any number of plugin values as arguments to create a Parser class extended by all those plugins. You'll usually want to create such an extended class only once, and then repeatedly call parse on it, to avoid needlessly confusing the JavaScript engine's optimizer.

```
import { fromEvent } from 'rxjs';
fromEvent(document, 'click').subscribe(() => console.log('Clicked!'));

const {Parser} = require("acorn")
const MyParser = Parser.extend(
  require("acorn-jsx")(),
  require("acorn-bigint")
)
console.log(MyParser.parse("// Some bigint + JSX code"))
```

Plugins override methods in their new parser class to implement additional functionality. It is recommended for a plugin package to export its plugin function as its default value or, if it takes configuration parameters, to export a constructor function that creates the plugin function.

5.8 UUID

To create a globally unique id, we used a library that generates UUIDs. To hold our character objects, we decided on a linear array to make things as simple as possible to start.

To create a random UUID

1. Install

```
npm install uuid
```

2. Create a UUID (ES6 module syntax)

```
import { v4 as uuidv4 } from 'uuid';
uuidv4(); // ⇨ '9b1deb4d-3b7d-4bad-9bdd-2b0d7b3dcb6d'
```

... or using CommonJS syntax:

```
const { v4: uuidv4 } = require('uuid');
uuidv4(); // ⇨ '1b9d6bcd-bbfd-4b2d-9b5d-ab8dfbbd4bed'
```

Example:

```
import { stringify as uuidStringify } from 'uuid';
const uuidBytes = [ 0x6e, 0xc0, 0xbd, 0x7f, 0x11, 0xc0, 0x43, 0xda, 0x97, 0x5e,
0x2a, 0x8a, 0xd9, 0xeb, 0xae, 0x0b,
];
uuidStringify(uuidBytes);
// ⇨'6ec0bd7f-11c0-43da-975e-2a8ad9ebae0b'
```

Example using options:

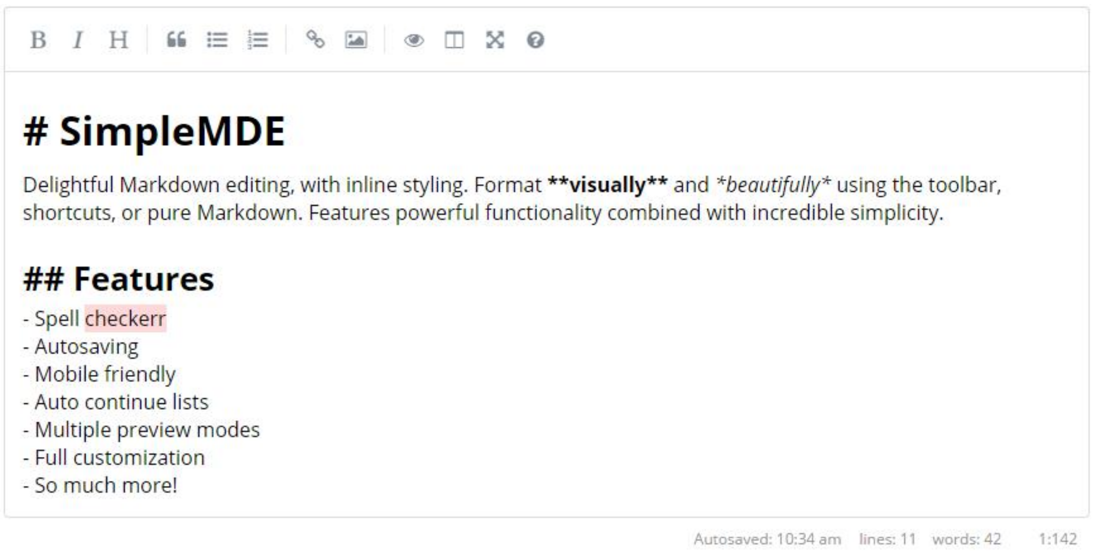
```
import { v1 as uuidv1 } from 'uuid';

const v1options = {
  node: [0x01, 0x23, 0x45, 0x67, 0x89, 0xab],
  clockseq: 0x1234,
  msecs: new Date('2011-11-01').getTime(),
  nsecs: 5678,
};
uuidv1(v1options); // ⇨ '710b962e-041c-11e1-9234-0123456789ab'
```

5.9 SimpleMDE - Markdown Editor

A drop-in JavaScript textarea replacement for writing beautiful and understandable Markdown. The WYSIWYG-esque editor allows users who may be less experienced with Markdown to use familiar toolbar buttons and shortcuts. In addition, the syntax is rendered while editing to clearly show the expected result. Headings are larger, emphasized words are italicized, links are underlined, etc. SimpleMDE is one of the first editors to feature both built-in autosaving and spell checking.

WYSIWYG editors that produce HTML are often complex and buggy. Markdown solves this problem in many ways, plus Markdown can be rendered natively on more platforms than HTML. However, Markdown is not a syntax that an average user will be familiar with, nor is it visually clear while editing. In other words, for an unfamiliar user, the syntax they write will make little sense until they click the preview button. SimpleMDE has been designed to bridge this gap for non-technical users who are less familiar with or just learning Markdown syntax.



```
# SimpleMDE
Delightful Markdown editing, with inline styling. Format **visually** and *beautifully* using the toolbar,
shortcuts, or pure Markdown. Features powerful functionality combined with incredible simplicity.

## Features
- Spell checkerr
- Autosaving
- Mobile friendly
- Auto continue lists
- Multiple preview modes
- Full customization
- So much more!
```

Fig. Simple MDE text editor

Install

Via npm.

```
npm install simplemde --save
```

Via CDN.

```
<link rel="stylesheet"
href="https://cdn.jsdelivr.net/simplemde/latest/simplemde.min.css">
<script src="https://cdn.jsdelivr.net/simplemde/latest/simplemde.min.js"></script>
```

Quick start

After installing, load SimpleMDE on the first textarea on a page

```
<script>
var simplemde = new SimpleMDE();
</script>
```

Using a specific textarea

Pure JavaScript method

```
<script>
var simplemde = new SimpleMDE({ element:
document.getElementById("MyID") });
</script>
```

jQuery method

```
<script>
var simplemde = new SimpleMDE({ element: $("#MyID")[0] });
</script>
```

Get/set the content

```
simplemde.value();
simplemde.value("This text will appear in the editor");
```

Chapter 6

Code Templates

6.1 CRDT Structure

For our CRDT data structure, we simply need a globally unique Site ID and a structure to hold our characters objects. To create a globally unique id, we used a library that generates UUIDs. To hold our character objects, we decided on a linear array to make things as simple as possible to start.

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

6.2 Local Insert/Delete

When inserting a character locally, the only information needed is the character value and the editor index at which it is inserted. A new character object will then be created using that information and spliced into the CRDT array. Finally, the new character is returned so it can be sent to the other users.

```
class CRDT {
   localInsert(value, index) {
     const char = this.generateChar(value, index);
     this.struct.splice(index, 0, char);
     return char;
   }
 }
```

The bulk of the generateChar logic is determining the globally unique fractional index position of the new character. Since each new character's position is relative to its adjacent characters, the positions of these adjacent characters are used to generate the position of the new character.

```
generateChar(val, index) {
   const posBefore = (this.struct[index - 1] && this.struct[index - 1].position) || [];
   const posAfter = (this.struct[index] && this.struct[index].position) || [];
   const newPos = this.generatePosBetween(posBefore, posAfter);
 }
```

We took advantage of that structure to create a recursive algorithm that traverses down that tree and dynamically generates a position.

```
generatePosBetween(pos1, pos2, newPos=[]) {
   let id1 = pos1[0];
   let id2 = pos2[0];

   if (id2.digit - id1.digit > 1) {
     let newDigit = this.generateIdBetween(id1.digit, id2.digit);
     newPos.push(new Identifier(newDigit, this.siteId));
     return newPos;

   } else if (id2.digit - id1.digit === 1) {

     newPos.push(id1);
     return this.generatePosBetween(pos1.slice(1), pos2, newPos);

   }
 }
```

Deleting a character from the CRDT is much simpler. All that is needed is the index of the character. That index is used to splice out the character object from our linear array and return it.

```
localDelete(idx) {
   return this.struct.splice(idx, 1)[0];}
```

6.3 Remote Insert/Delete

Remote operations are where each character object's relative position comes in handy. When a user receives an operation from another user, it's up to their CRDT to figure out where to insert it.

To make this as efficient as possible, we implemented a binary search algorithm. When applying a remote delete, the binary search uses the character's relative position to find it's index in the array. In the case of remote inserts, the binary search is used to find where it should be inserted in the array.

The return values are passed along to our Editor where the operations are applied in our user's local CodeMirror text editor.

```
remoteInsert(char) {
   const index = this.findInsertIndex(char);
   this.struct.splice(index, 0, char);

   return { char: char.value, index: index };
 }

remoteDelete(char) {
   const index = this.findIndexByPosition(char);
   this.struct.splice(index, 1);

   return index;
 }
```

6.4 Remote Cursors

Having several people edit a document at the same time can be a chaotic experience. It becomes even more hectic when you don’t know who else is typing and where.

That is the situation we ran into. Without a way to identify another person on the page, users would end up writing over each other and turning the real-time collaboration experience into a headache. We solved this problem with remote cursors.

```
addRemoteCursor() {
   // ...

   const color = generateItemFromHash(this.siteId, CSS_COLORS);
   const name = generateItemFromHash(this.siteId, ANIMALS);

   // ...
 }

 function generateItemFromHash(siteId, collection) {
   const hashIdx = hashAlgo(siteId, collection);

   return collection[hashIdx];
 }

 function hashAlgo(input, collection) {
   const filteredNum = input.toLowerCase().replace(/[a-z\-]/g, '');
   return Math.floor(filteredNum * 13) % collection.length;
 }
```

Source code for CRDT [crdt.js]:

```
import Identifier from './identifier';
import Char from './char';

class CRDT {
  constructor(controller, base=32, boundary=10, strategy='random') {
    this.controller = controller;
    this.vector = controller.vector;
    this.struct = [[ ]];
    this.siteId = controller.siteId;
    this.base = base;
    this.boundary = boundary;
    this.strategy = strategy;
    this.strategyCache = [ ];
  }

  handleLocalInsert(value, pos) {
    this.vector.increment();
    const char = this.generateChar(value, pos);
    this.insertChar(char, pos);
    this.controller.broadcastInsertion(char);
  }

  handleRemoteInsert(char) {
    const pos = this.findInsertPosition(char);
    this.insertChar(char, pos);
    this.controller.insertIntoEditor(char.value, pos, char.siteId);
  }

  insertChar(char, pos) {
    if (pos.line === this.struct.length) {
      this.struct.push([]);
    }

    // if inserting a newline, split line into two lines
    if (char.value === "\n") {
      const lineAfter = this.struct[pos.line].splice(pos.ch);

      if (lineAfter.length === 0) {
        this.struct[pos.line].splice(pos.ch, 0, char);
      } else {
        const lineBefore = this.struct[pos.line].concat(char);
        this.struct.splice(pos.line, 1, lineBefore, lineAfter);
      }
    } else {
      this.struct[pos.line].splice(pos.ch, 0, char);
    }
  }

  handleLocalDelete(startPos, endPos) {
    let chars;
    let newlineRemoved = false;

    // for multi-line deletes
    if (startPos.line !== endPos.line) {
      // delete chars on first line from startPos.ch to end of line
      newlineRemoved = true;
      chars = this.deleteMultipleLines(startPos, endPos)

      // single-line deletes
    } else {
      chars = this.deleteSingleLine(startPos, endPos)

      if (chars.find(char => char.value === '\n')) newlineRemoved = true;
    }

    this.broadcast(chars);
    this.removeEmptyLines();

    if (newlineRemoved && this.struct[startPos.line + 1]) {
      this.mergeLines(startPos.line);
    }
  }

  broadcast(chars) {
    chars.forEach(char => {
      this.vector.increment();
      this.controller.broadcastDeletion(char, this.vector.getLocalVersion());
    });
  }
```

```
deleteMultipleLines(startPos, endPos) {
  let chars = this.struct[startPos.line].splice(startPos.ch);
  let line;

  for (line = startPos.line + 1; line < endPos.line; line++) {
    chars = chars.concat(this.struct[line].splice(0));
  }

  // todo for loop inside crdt
  if (this.struct[endPos.line]) {
    chars = chars.concat(this.struct[endPos.line].splice(0, endPos.ch));
  }

  return chars;
}

deleteSingleLine(startPos, endPos) {
  let charNum = endPos.ch - startPos.ch;
  let chars = this.struct[startPos.line].splice(startPos.ch, charNum);

  return chars;
}

// when deleting newline, concat line with next line
mergeLines(line) {
  const mergedLine = this.struct[line].concat(this.struct[line + 1]);
  this.struct.splice(line, 2, mergedLine);
}

removeEmptyLines() {
  for (let line = 0; line < this.struct.length; line++) {
    if (this.struct[line].length === 0) {
      this.struct.splice(line, 1);
      line--;
    }
  }

  if (this.struct.length === 0) {
    this.struct.push([]);
  }
}
```

```
  handleRemoteDelete(char, siteId) {
    const pos = this.findPosition(char);

    if (!pos) return;
    this.struct[pos.line].splice(pos.ch, 1);

    if (char.value === "\n" && this.struct[pos.line + 1]) {
      this.mergeLines(pos.line);
    }

    this.removeEmptyLines();
    this.controller.deleteFromEditor(char.value, pos, siteId);
  }
isEmpty() { return this.struct.length === 1 && this.struct[0].length === 0;
  }

  findPosition(char) {
    let minLine = 0;
    let totalLines = this.struct.length;
    let maxLine = totalLines - 1;
    let lastLine = this.struct[maxLine];
    let currentLine, midLine, charIdx, minCurrentLine, lastChar,
          maxCurrentLine, minLastChar, maxLastChar;

    // check if struct is empty or char is less than first char
    if (this.isEmpty() || char.compareTo(this.struct[0][0]) < 0) {
      return false;
    }

    lastChar = lastLine[lastLine.length - 1];

    // char is greater than all existing chars (insert at end)
    if (char.compareTo(lastChar) > 0) {
      return false;
    }

    // binary search
    while (minLine + 1 < maxLine) {
      midLine = Math.floor(minLine + (maxLine - minLine) / 2);
      currentLine = this.struct[midLine];
    lastChar = currentLine[currentLine.length - 1];

    if (char.compareTo(lastChar) === 0) {
      return {line: midLine, ch: currentLine.length - 1}
    } else if (char.compareTo(lastChar) < 0) {
      maxLine = midLine;
    } else {
      minLine = midLine;
    }
  }

  // Check between min and max line.
  minCurrentLine = this.struct[minLine];
  minLastChar = minCurrentLine[minCurrentLine.length - 1];
  maxCurrentLine = this.struct[maxLine];
  maxLastChar = maxCurrentLine[maxCurrentLine.length - 1];

  if (char.compareTo(minLastChar) <= 0) {
    charIdx = this.findIndexInLine(char, minCurrentLine);
    return { line: minLine, ch: charIdx };
  } else {
    charIdx = this.findIndexInLine(char, maxCurrentLine);
    return { line: maxLine, ch: charIdx };
  }
}

findIndexInLine(char, line) {
  let left = 0;
  let right = line.length - 1;
  let mid, compareNum;

  if (line.length === 0 || char.compareTo(line[left]) < 0) {
    return left;
  } else if (char.compareTo(line[right]) > 0) {
    return this.struct.length;
  }

  while (left + 1 < right) {
    mid = Math.floor(left + (right - left) / 2);
    compareNum = char.compareTo(line[mid]);
 if (compareNum === 0) {
      return mid;
    } else if (compareNum > 0) {
      left = mid;
    } else {
      right = mid;
    }
  }

  if (char.compareTo(line[left]) === 0) {
    return left;
  } else if (char.compareTo(line[right]) === 0) {
    return right;
  } else {
    return false;
  }
}

// could be refactored to look prettier
findInsertPosition(char) {
  let minLine = 0;
  let totalLines = this.struct.length;
  let maxLine = totalLines - 1;
  let lastLine = this.struct[maxLine];
  let currentLine, midLine, charIdx, minCurrentLine, lastChar,
      maxCurrentLine, minLastChar, maxLastChar;

// check if struct is empty or char is less than first char
  if (this.isEmpty() || char.compareTo(this.struct[0][0]) <= 0) {
    return { line: 0, ch: 0 }
  }

  lastChar = lastLine[lastLine.length - 1];

 // char is greater than all existing chars (insert at end)
  if (char.compareTo(lastChar) > 0) {
    return this.findEndPosition(lastChar, lastLine, totalLines);
  }
```

6.5 Download

Since we eliminated the central relay server, there is no server to store a user's documents. We needed to provide a way for users to save what they were working on. Adding a download button was simple. Any user can click the Download button to save the current document text to their computer.

```
downloadButton.onclick = () => {
   const text = editor.value();
   const blob = new Blob([text], { type:"text/plain" });
   const link = document.createElement("a");
   link.style.display = "none";
   link.download = "Conclave-"+Date.now();
   link.href = window.URL.createObjectURL(blob);
   link.onclick = e => document.body.removeChild(e.target);

   document.body.appendChild(link);
   link.click();
 }
```

6.6 Upload

In a similar vein, what if someone wants to continue editing a previously downloaded file or some existing document? To support this, we added the ability to upload a file.

We used JavaScript's built-in FileReader to read the contents of the file selected for upload. The text is inserted into the user's CRDT and then their editor text is replaced entirely by the data in the CRDT. Updating the editor is fast but insertion into the data structure happens one character at a time, so large documents can take a while. This is an area for future improvement.

```
fileSelect.onchange = () => {
   const file = document.querySelector("#file").files[0];
   const fileReader = new FileReader();
   fileReader.onload = (e) => {
     const fileText = e.target.result;
     localInsert(fileText, { line: 0, ch: 0 });
     replaceText(crdt.toText());
   }
   fileReader.readAsText(file, "UTF-8"); }
```

6.7 Automatic Network Balancing

In Peer-to-Peer Architecture, we explained how users immediately connect to a user upon clicking that user's sharing link. The problem is that it's likely that the first user of a document is going to share their link with several users resulting in that single point-of-failure. Our p2p architecture ends up looking pretty much like the client-server architecture we wanted to move away from.

Our solution was to add an evaluation step before connecting a new user to the network. When a new user attempts to join the document (or network), they send a connection request first.

```
evaluateRequest(peerId, siteId) {
   if (this.hasReachedMax()) {
     this.forwardConnRequest(peerId, siteId);
   } else {
     this.acceptConnRequest(peerId, siteId);
   }
 }
```

6.8 What is the maximum number of connections?

To answer that question, we had to calculate the average number of connections we wanted for every user. Since the network can grow and change, a fixed number (O(1)) would not be reliable. At the same time, having each user connected to every person in the network (O(N)) could cause bandwidth issues. We settled on a logarithmic scale (O(log(N))).

However, through testing and trial-and-error, we discovered that this solution causes unusual bugs when the network is less than 5 users. Therefore, we use 5 as our baseline and when the network grows beyond the point, it transitions to logarithmic growth.

```
hasReachedMax() {
   const partOfNetwork = Math.ceil(Math.log(this.network.length));
   const tooManyConns = this.connections.length > Math.max(partOfNetwork, 5);
   return tooManyConns; }
```

6.9 Version Vector

WebRTC uses the UDP transport protocol. UDP is a lightweight message protocol that allows it to send messages quickly and without waiting for a response from the other user.

Version vector code of the project [versionVector.js]

```
import SortedArray from './sortedArray';
import Version from './version';

// vector/list of versions of sites in the distributed system
// keeps track of the latest operation received from each site (i.e. version)
// prevents duplicate operations from being applied to our CRDT
class VersionVector {
  // initialize empty vector to be sorted by siteId
  // initialize Version/Clock for local site and insert into SortedArray vector object
  constructor(siteId) {
    // this.versions = new SortedArray(this.siteIdComparator);
    this.versions = []
    this.localVersion = new Version(siteId);
    this.versions.push(this.localVersion);
  }

  increment() {
    this.localVersion.counter++;
  }

  // updates vector with new version received from another site
// if vector doesn't contain version, it's created and added to vector
// create exceptions if need be.
  update(incomingVersion) {
    const existingVersion = this.versions.find(version => incomingVersion.siteId ===
version.siteId);
    if (!existingVersion) {
      const newVersion = new Version(incomingVersion.siteId);
      newVersion.update(incomingVersion);
      this.versions.push(newVersion);
    } else {
      existingVersion.update(incomingVersion);
    }
  }
  // check if incoming remote operation has already been applied to our crdt

  hasBeenApplied(incomingVersion) {
    const localIncomingVersion = this.getVersionFromVector(incomingVersion);
    const isIncomingInVersionVector = !!localIncomingVersion;

if (!isIncomingInVersionVector) return false;
const isIncomingLower = incomingVersion.counter <=
localIncomingVersion.counter;
const isInExceptions =
localIncomingVersion.exceptions.includes(incomingVersion.counter);
    return isIncomingLower && !isInExceptions;
  }

  getVersionFromVector(incomingVersion) {
    return this.versions.find(version => version.siteId === incomingVersion.siteId);
  }

  getLocalVersion() {
    return {
      siteId: this.localVersion.siteId,
      counter: this.localVersion.counter
    };
  }
}

export default VersionVector;
```

Chapter 7

Testing

Developing peer-to-peer (P2P) systems is hard because they must be deployed on a high number of nodes, which can be autonomous, refusing to answer to some requests or even unexpectedly leaving the system. Such volatility of nodes is a common behavior in P2P systems and can be interpreted as fault during tests.

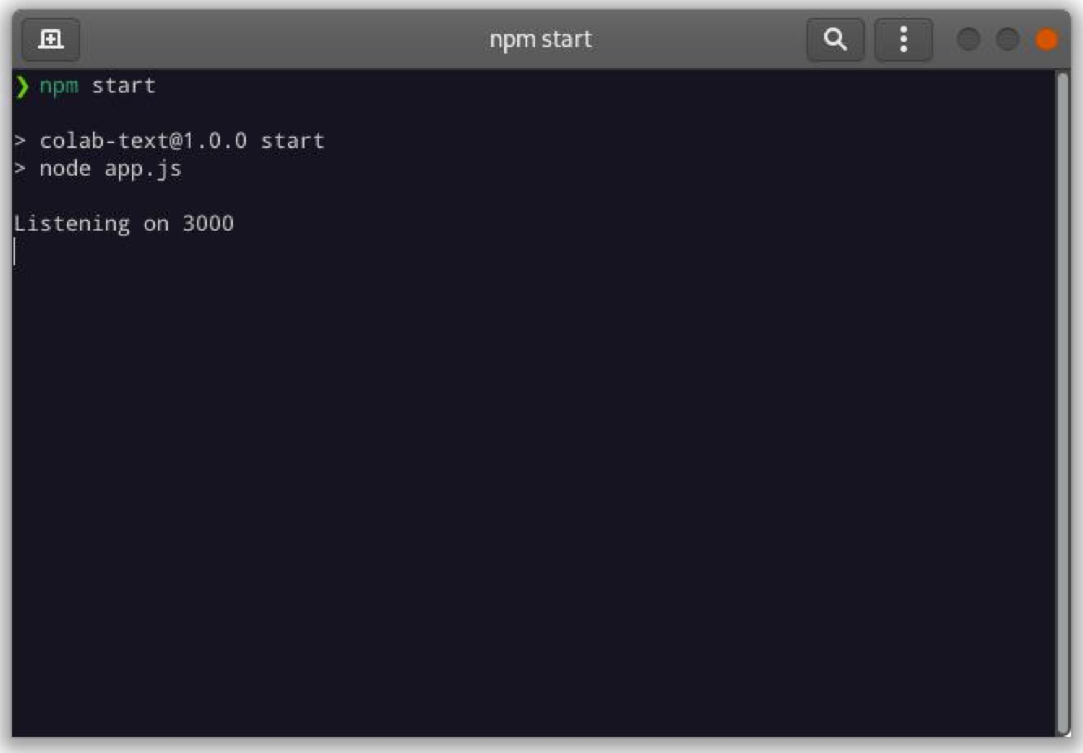
Testing the p2p applications is difficult. The majority of our bugs were found through manual testing, which is inefficient and risky but most of the bugs we found are very core to the CRDT itself which is a research in progress.

Since this application is confined within the local area network these do depend upon the hardware's used in the network. If the network has higher bandwidth capability this would result in better performance and efficiency. With the network hardware in mind the users hardware that is deployed on too has the advantages as it depends upon the browser's engine which is very memory and processor hungry.

In order to simulate real world latency scenarios, we would need to buy servers in data centers across the country and world. It's feasible but we don't currently have the resources to achieve such a feat.

Chapter 8

Output Screens



```
❯ npm start

> colab-text@1.0.0 start
> node app.js

Listening on 3000
```

Fig.21. screenshot of local nodejs

- We have used nodejs as our server.
- local server running at port 3000 as shown in the screenshot above.
- The application can be assesed with the localhost://3000 in the browser

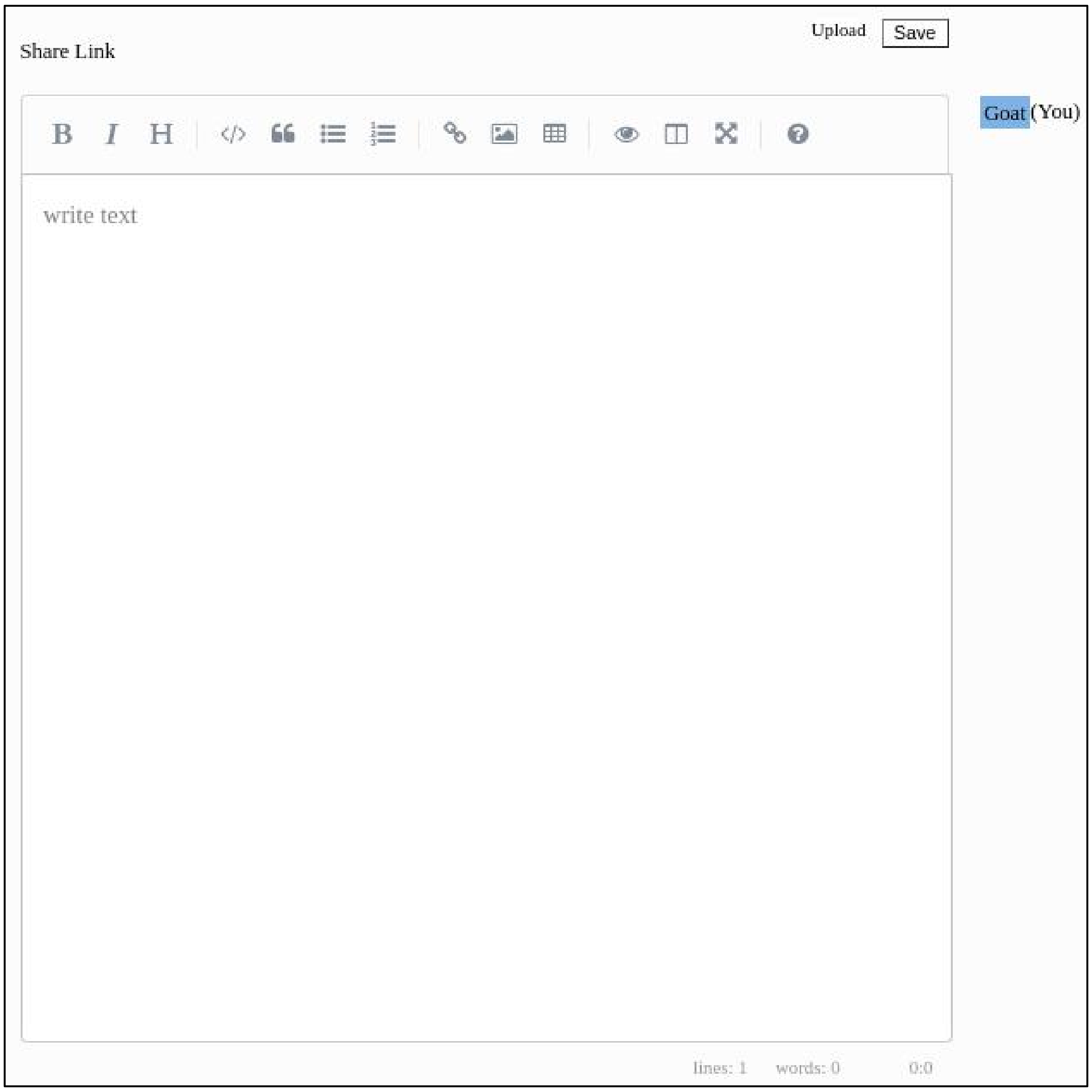
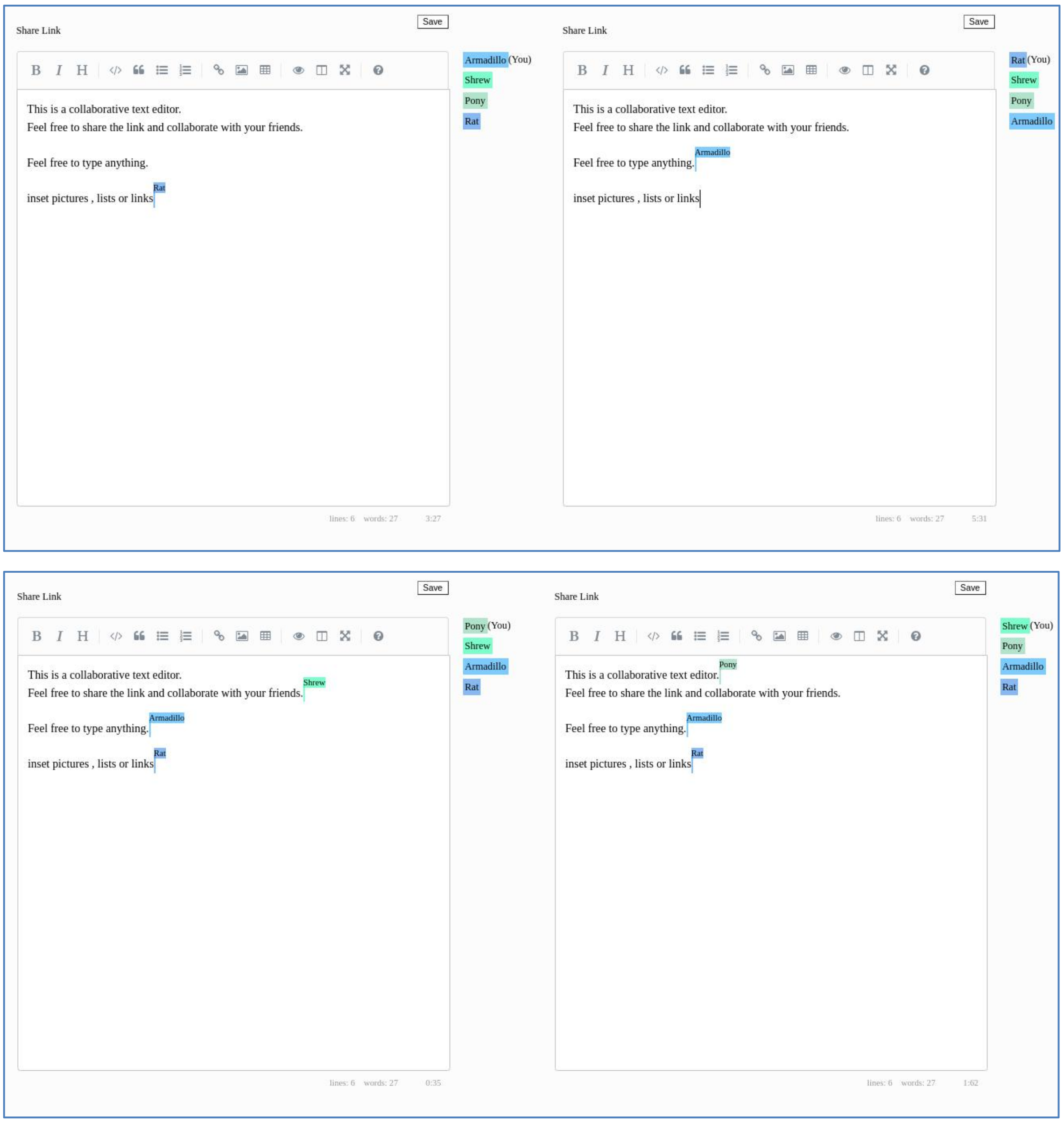


Fig.22. Single collaborator

- Share link is used for inviting others for collaborations.
- Save to save the document.
- Upload to re-edit the saved documents.
- Random name of the editor in the color.



This is a collaborative text editor.
Feel free to share the link and collaborate with your friends.

Feel free to type anything.

inset pictures , lists or links

lines: 6 words: 27 3:27

This is a collaborative text editor.
Feel free to share the link and collaborate with your friends.

Feel free to type anything.

inset pictures , lists or links

lines: 6 words: 27 5:31

This is a collaborative text editor.
Feel free to share the link and collaborate with your friends.

Feel free to type anything.

inset pictures , lists or links

lines: 6 words: 27 0:35

This is a collaborative text editor.
Feel free to share the link and collaborate with your friends.

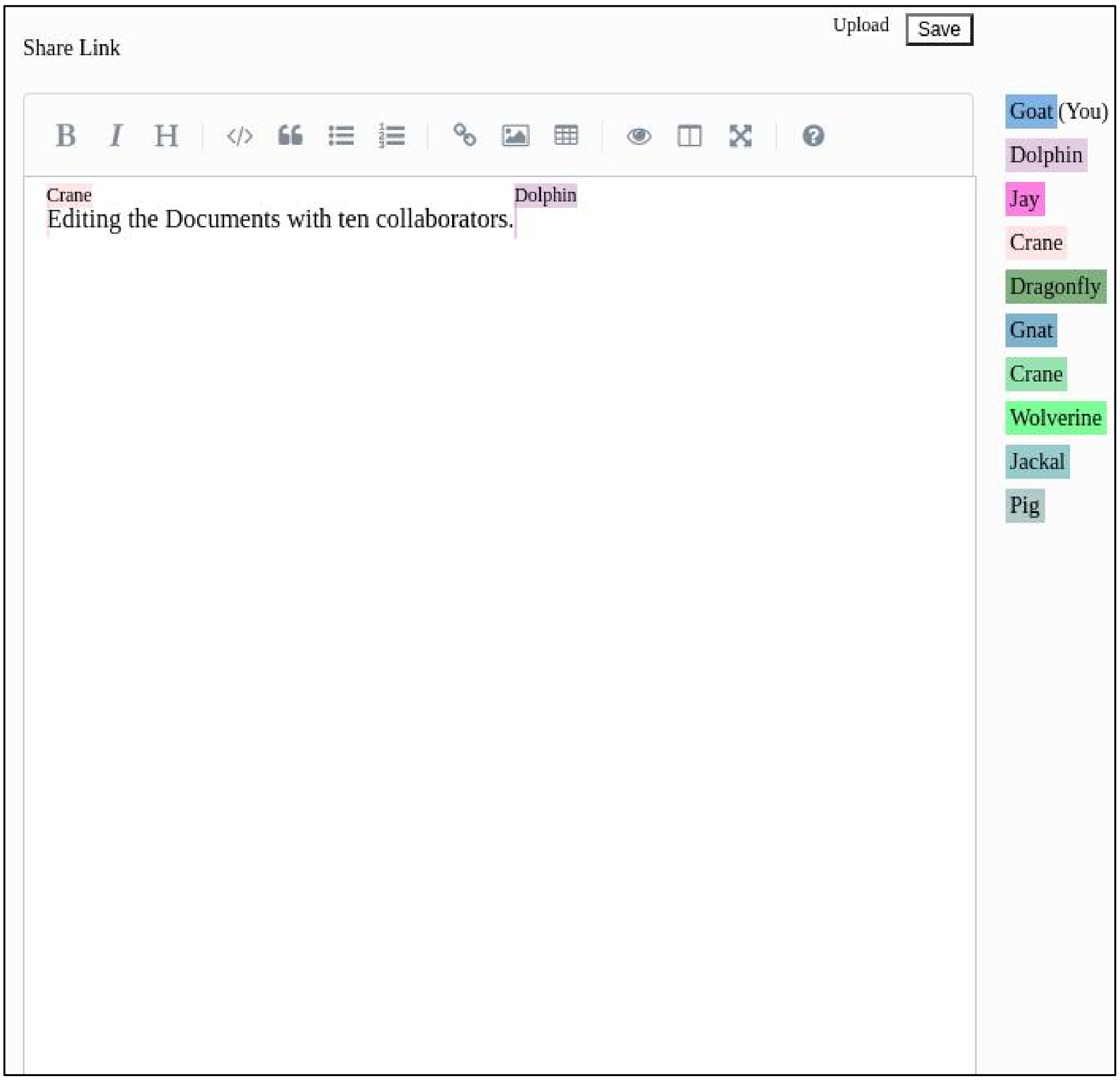
Feel free to type anything.

inset pictures , lists or links

lines: 6 words: 27 1:62

Fig.23. Multiple Collaborator

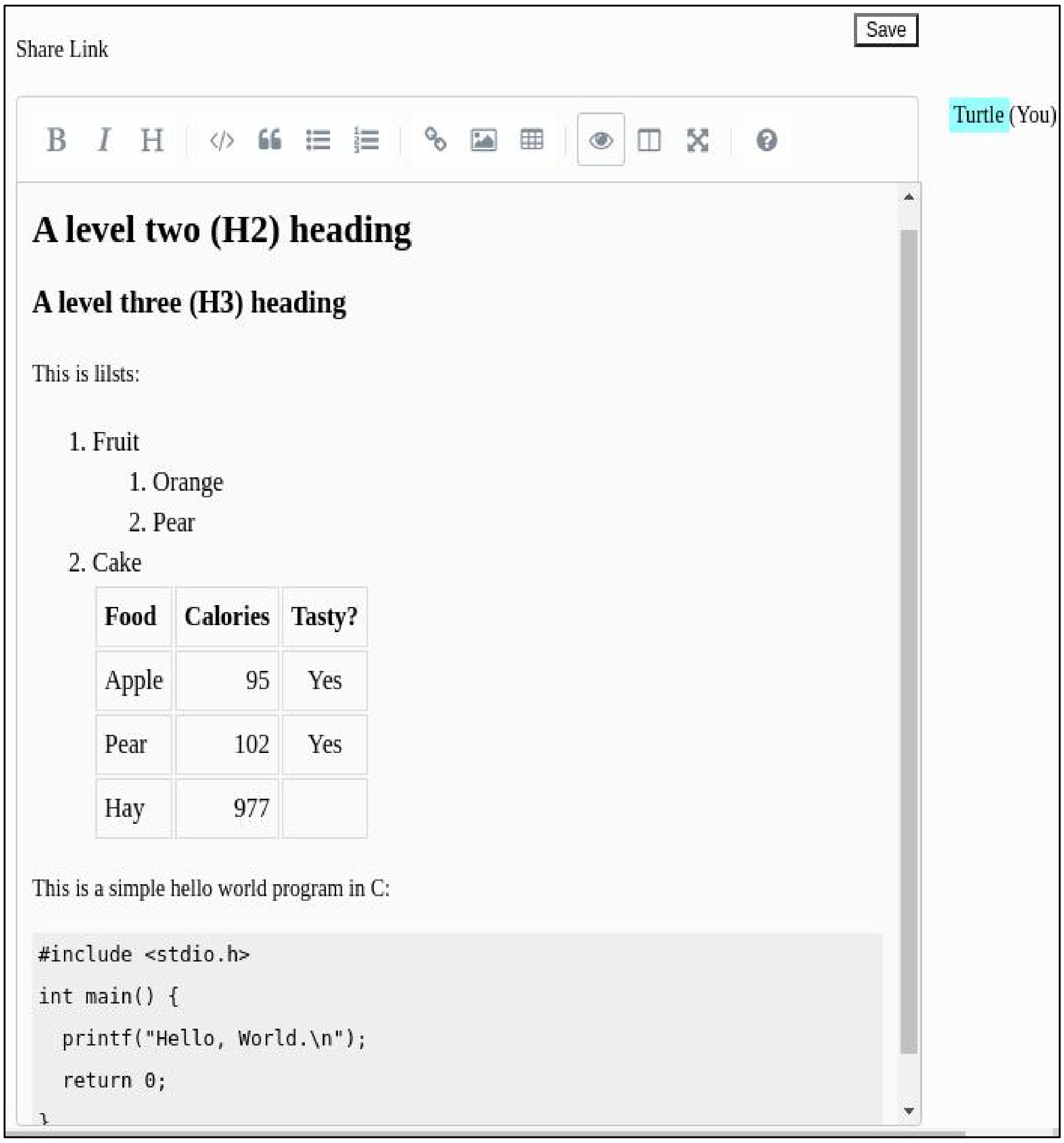
- Here multiple collaborators are collaborating with each other.
- All the collaborators names are unique and random with unique colors.
- The flag shows the collaborator editing the documents.
- The collaboration is in the multiple devices.



Editing the Documents with ten collaborators.

Fig.24. Editing with ten collaborators.

Note: During higher no of collaborators collaboration do take a bit time to replicate throughout the documents of every collaborators due to the hardware constraints.



A level two (H2) heading

A level three (H3) heading

This is lilsts:

1. Fruit
    1. Orange
    2. Pear
2. Cake

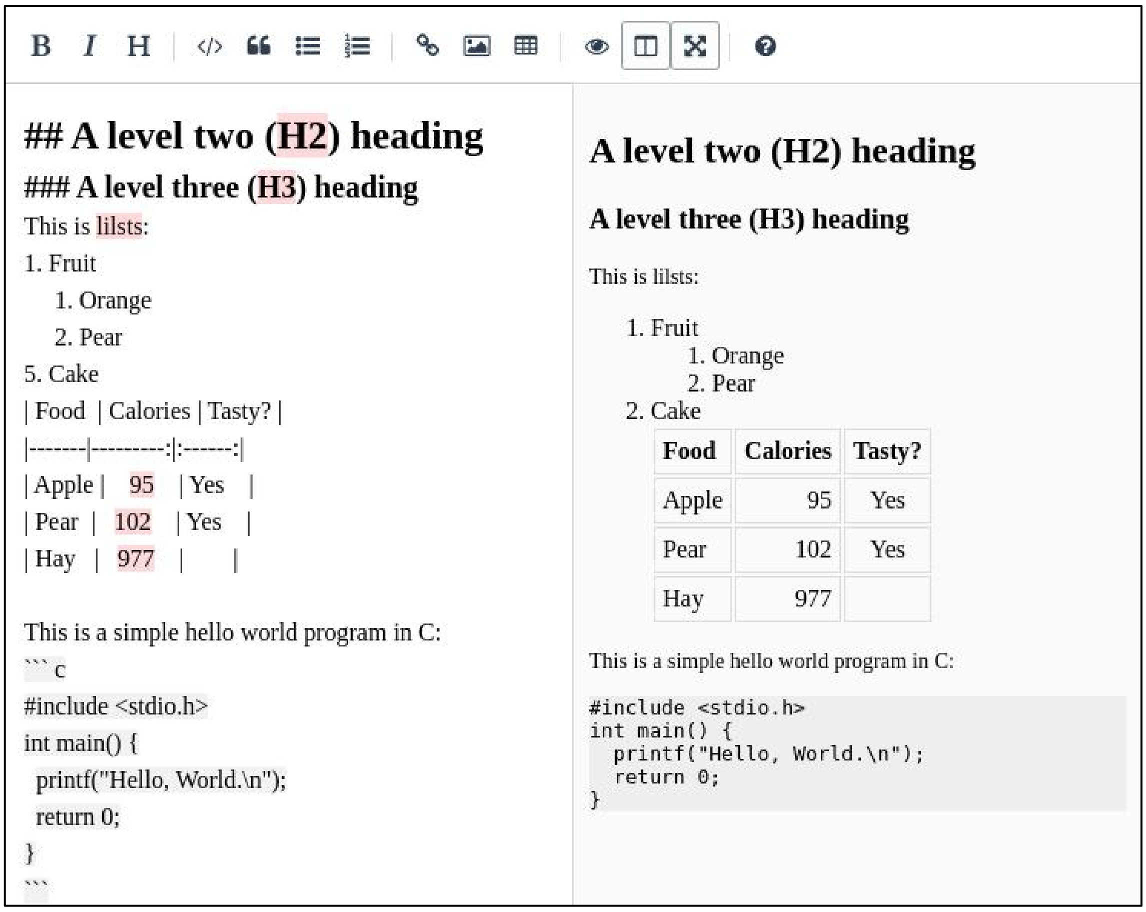
Food	Calories	Tasty?
Apple	95	Yes
Pear	102	Yes
Hay	977	

This is a simple hello world program in C:

```
#include <stdio.h>
int main() {
  printf("Hello, World.\n");
  return 0;
}
```

Fig.25. This is a simple example of few editing features.

All the elements like tables, lists, bullets, code snippets, different font typos can be added with the functionalities of the editor from the toolbar just like any text editors.



~~~
## A level two (H2) heading
### A level three (H3) heading
This is lilsts:
1. Fruit
    1. Orange
    2. Pear
5. Cake
| Food  | Calories | Tasty? |
|-------|---------:|:------:|
| Apple |    95    | Yes    |
| Pear  |   102    | Yes    |
| Hay   |   977    |        |

This is a simple hello world program in C:
```c
#include <stdio.h>
int main() {
  printf("Hello, World.\n");
  return 0;
}
```
~~~

A level two (H2) heading

A level three (H3) heading

This is lilsts:

1. Fruit
    1. Orange
    2. Pear
2. Cake

Food	Calories	Tasty?
Apple	95	Yes
Pear	102	Yes
Hay	977	

This is a simple hello world program in C:

```
#include <stdio.h>
int main() {
  printf("Hello, World.\n");
  return 0;
}
```

Fig.26. Markdown editor with split

- The documents can be splitted into two for easy preview of markdown elements.
- This functionality can be used using the split window in the toolbar.
- Functionality of fullscreen mode can also be used with full-screen tool

Chapter 9

Conclusion

The objective of this project was to implement CRDT in the real world applications for a general purpose collaborative text editor.

The development of this project taught us few libraries and packages that we have almost never heard of like PUG for efficient layout editing, ExpressJS for the better backend support with NodeJS, jQuery for the better javascript features.

Even though the project is not as elegant as most commercialized editors, adding new features and the functionalities will be carried on continuously.

Chapter 10

Further Enhancements

10.1 Better Connection Distribution

The first thing to further improve would be the connection distribution for users in the network. A possibility we are entertaining is to have new users connect to more than one user initially. This will prevent collaborators from becoming stranded if one of their connections drops and they are forced to find a new user to connect to.

10.2 Large Insertions/Deletions

Currently, our CRDT can only insert and delete one character at a time. Being able to add or remove chunks of text at once will drastically improve overhead and improve the efficiency of large cuts and pastes (as well as uploads).

10.3 Comments

This feature would allow the collaborators to comment on the documents lines or text. This would really improve the quality of collaborations.

10.4 Suggests edits

Suggest edits are the recommendations that are given by the users to other users for enhancing their part of the documents. Generally very helpful while large no of collaborations in bigger documents.

10.5 Better functionality and UI

Adding more functionality to the editor for an efficient editing experience for the users will be the major factor for the newer version of this project.
The UI of this version of the collab-editor is not so perfect. Few enhancements with the UI would definitely enrich the editor.

REFERENCES

1. D. Adas and R. Friedman, "Sliding Window CRDT Sketches," 2021 40th International Symposium on Reliable Distributed Systems (SRDS), 2021, pp. 288-298, doi: 10.1109/SRDS53918.2021.00036.

2. E. Chandra and A. I. Kistijantoro, "Database development supporting offline update using CRDT: (Conflict-free replicated data types)," 2017 International Conference on Advanced Informatics, Concepts, Theory, and Applications (ICAICTA), 2017, pp. 1-6, doi: 10.1109/ICAICTA.2017.8090961.

3. X. Lv, F. He, W. Cai and Y. Cheng, "An efficient collaborative editing algorithm supporting string-based operations," 2016 IEEE 20th International Conference on Computer Supported Cooperative Work in Design (CSCWD), 2016, pp. 45-50, doi: 10.1109/CSCWD.2016.7565961.

4. A. Auvolat and F. Taïani, "Merkle Search Trees: Efficient State-Based CRDTs in Open Networks," 2019 38th Symposium on Reliable Distributed Systems (SRDS), 2019, pp. 221-22109, doi: 10.1109/SRDS47363.2019.00032.

5. C. Gadea, B. Ionescu and D. Ionescu, "A Control Loop-based Algorithm for Operational Transformation," 2020 IEEE 14th International Symposium on Applied Computational Intelligence and Informatics (SACI), 2020, pp. 000247-000254, doi: 10.1109/SACI49304.2020.9118822.

6. M. Kleppmann and A. R. Beresford, "A Conflict-Free Replicated JSON Datatype," in IEEE Transactions on Parallel and Distributed Systems, vol. 28, no. 10, pp. 2733-2746, 1 Oct. 2017, doi: 10.1109/TPDS.2017.2697382.

7. M. Ahmed-Nacer, P. Urso, V. Balegas and N. Preguiça, "Concurrency control and awareness support for multi-synchronous collaborative editing," 9th IEEE International Conference on Collaborative Computing: Networking, Applications and Worksharing, 2013, pp. 148-157, doi: 10.4108/icst.collaboratecom.2013.254113.

8. F. Guidec, Y. Mahéo and C. Noûs, "Delta-State-Based Synchronization of CRDTs in Opportunistic Networks," 2021 IEEE 46th Conference on Local Computer Networks (LCN), 2021, pp. 335-338, doi: 10.1109/LCN52139.2021.9524978.

9. V. Enes, P. S. Almeida, C. Baquero and J. Leitão, "Efficient Synchronization of State-Based CRDTs," 2019 IEEE 35th International Conference on Data Engineering (ICDE), 2019, pp. 148-159, doi: 10.1109/ICDE.2019.00022.

10. A. Ahuja, G. Gupta and S. Sidhanta, "Edge Applications: Just Right Consistency," 2019 38th Symposium on Reliable Distributed Systems (SRDS), 2019, pp. 351-3512, doi: 10.1109/SRDS47363.2019.00047.

11. I. Briquemont, M. Bravo, Z. Li and P. Van Roy, "Conflict-Free Partially Replicated Data Types," 2015 IEEE 7th International Conference on Cloud Computing Technology and Science (CloudCom), 2015, pp. 282-289, doi: 10.1109/CloudCom.2015.81.

12. M.Kleppmann and A.R.Beresford,"A Conflict-Free Replicated JSON Datatype,"in IEEE Transactions on Parallel and Distributed Systems,vol.28,no. 10,pp.2733-2746, 1 Oct.2017,doi:10.1109/TPDS.2017.2697382.

13. F.Jacob,C.Beer,N.Henze and H.Hartenstein,"Analysis of the Matrix Event Graph Replicated Data Type,"in IEEE Access,vol.9,pp.28317-28333,2021, doi:10.1109/ACCESS.2021.3058576.

14. Clarence A Ellis and Simon J Gibbs.“Concurrency control in groupware systems”.In:Acm Sigmod Record.Vol.18.2.ACM.1989.

15. Gérald Oster et al.“Real time group editors without operation transformation”. PhD thesis.INRIA,2005.

16. Martin Kleppmann,Victor B.F.Gomes,Dominic P.Mulligan,and Alastair.Beresford. "Interleaving anomalies in collaborative text editors" March 2019.

17. Chengzheng Sun and Clarence Ellis.“Operational transforma-tion in real-time group editors:issues,algorithms,and achieve-ments”.In:Proceedings of the 1998 ACM conference on Computersupported cooperative work.ACM.1998.

18. Paulo Sérgio Almeida, Ali Shoker, and Carlos Baquero. “Efficient state-base crdts by delta-mutation”. In: International Conference on Networked Systems. Springer. 2015.

19. Marc Shapiro et al. “A comprehensive study of convergent and commutative replicated data types”. PhD thesis. Inria–Center Paris-Rocquencourt; INRIA, 2011.

20. Marc Shapiro et al. “Conflict-free replicated data types”. In: Symposium on Self-Stabilizing Systems. Springer. 2011.