NUpad: Collaborative Text Editor using CRDTs

SUMEET MAHENDRA GAJJAR, Khoury College of Computer Sciences, Northeastern University
MADHUR AKSHAY JAIN, Khoury College of Computer Sciences, Northeastern University

1 INTRODUCTION

Collaborative editing is a relatively old concept. People have been trying to efficiently edit documents in a collaborative way for many years. Collaborative work often requires several people to simultaneously edit the same text document, spreadsheet, presentation, etc. with each person's edit reflected on other collaborator's copies with minimal delay. The fundamental problem in collaborative editing is reaching the same consistent state when multiple users are concurrently editing the document, each of which may modify the state locally.

Collaborative editing has been a significant area of research since the 1990s with the first instance of a real-time collaborative editor being presented in 1968. The research primarily focuses on the issue of consistency in maintaining the documents under the constraints of free and concurrent editing from the users [15]. Consider an example highlighted in Figure 1 which reflects the issue of inconsistency, it has two users, A and B, starting with the same initial state $S = abc$, and simultaneously editing the same document. User A wants to perform the operation "append $ at end of the string S" while User B wants to perform "append @ at the end of the string S". Each operation is applied locally before sending it to each other. A brute-force approach is to apply each operation in the order they arrive, but as we can see from the figure doing so would result in an inconsistent state.

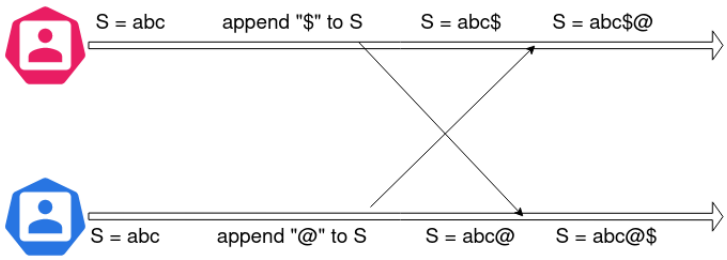

Fig. 1. Concurrent Operations Problem

Research on collaborative real time editors have led to an innovative technique called Operational Transform (OT) [14] that includes a family of algorithms which enable us to resolve conflicts. One such deployed OT collaboration system is Google Docs which uses the Jupiter algorithm [13]. The Jupiter algorithm requires a central server to receive the local updates and broadcast those to the peers. The central server looks at the incoming operations or updates and checks whether any operation has been performed in the past at the given index. If yes, it transforms the operation on a peer basis such that, applying this operation will result in a consistent state across all peers. Hence the name operational transform. From figure 2, we can see that the remote changes applied locally are different from the originated change.

Authors' addresses: Sumeet Mahendra Gajjar, Khoury College of Computer Sciences, Northeastern University; Madhur Akshay Jain, Khoury College of Computer Sciences, Northeastern University.

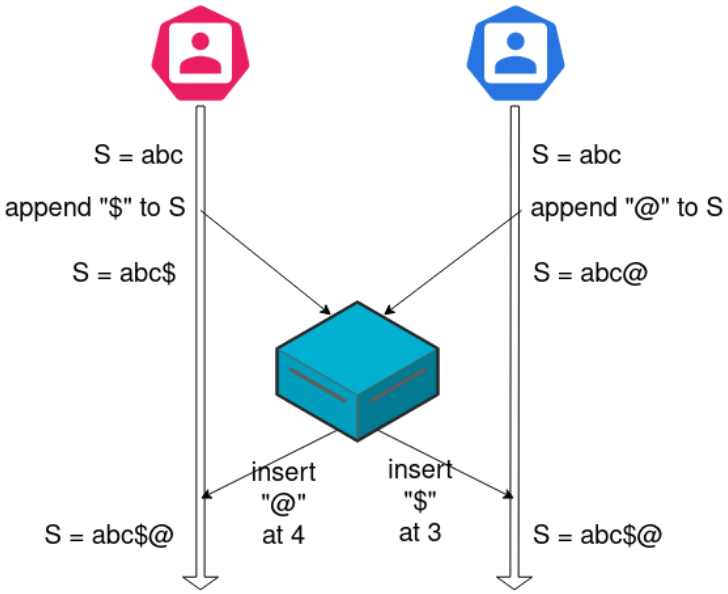

Fig. 2. Google Docs using Operation Transform

2 CONSENSUS TO RESCUE? NOT NECESSARILY!

Collaborative editing seems to be all about arriving at a consistent state, so can we use consensus to arrive at a consistent state here? In the following text, we will argue why consensus is not a good solution to the concurrent editing problem.

In order to achieve consistent state using consensus, we model the document as a list of characters and for each concurrent operation on a particular position in the list of the document, we will run consensus to determine the value. An operation here would consist of a change to a single character in the document. Any local change performed on a single character in the document by, say, Peer A, would be sent across to all the other peers whereby Peer A would be acting as a leader for that position and then waiting for the majority to agree on that particular change. Once a majority is reached, that change would then be applied on all the other peers thereby reaching a consistent state.

Consensus requires a majority of peers agreeing on a single value, but no majority is ever reached in a two-user scenario. Also, as a document is modeled using list of characters and each change is transmitted as a single character operation, so if multiple people are editing the document concurrently, running consensus for every character will be very slow, thereby defeating our purpose of transmitting the change with minimal delay.

Consensus also doesn't work really well in a partitioned network environment. If a peer becomes disconnected from the network, and then performs some local changes to the document, these changes would be overridden when the peer rejoins the cluster again. Consensus is usually performed within a LAN environment, so when we extend the consensus to apply at the WAN level, there will be not be a single global timeout value associated with all the peers thereby increasing the complexity for reaching majority.

Consensus, on a fundamental level, selects only one value out of all the proposed values. Consider figure 3, there are three users A, B, C (top to bottom) each starting with the initial state $S = abc$. A, B, C each wants to perform some operation. A, B, C will each apply the change locally and then transmit this change to all the other peers. On running consensus here, we can see that the @ change will be decided; discarding the $ change. At

this point, User A will be wondering where did its change go? This would cause confusion to the user, thereby creating inconsistent state of the document.

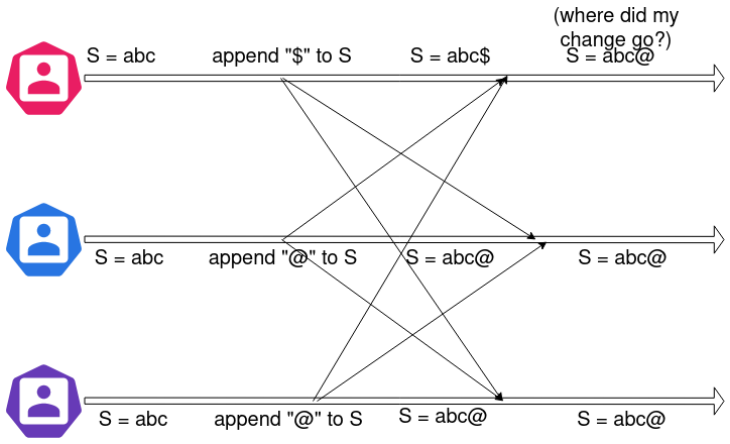

Fig. 3. Consensus causing confusion

3 WHAT WE DID?

Our proposed solution to the concurrent editing problem does not require any form of consensus or agreement on the final consistent state, but relies on the concept of **Conflict-Free Replicated Data Types (CRDT)**. CRDTs are a family of data structures that support concurrent modification and guarantee convergence of concurrent updates. They work by attaching additional metadata to the data structure, making modification operations commutative by construction. There are two types of CRDTs - **Commutative or CmRDTs** in which only *updates* on the local state are sent across to all the other peers and they work by applying those updates onto their local state and **Convergent or CvRDTs** where the *entire state* of a peer is sent and all the other peers merge it with their local state.

CRDTs for registers, counters, maps, ordered lists and sets are already well-known and have been implemented in various systems [11]. We provide one such implementation of a commutative CRDT list for our collaborative text editor. A document in our collaborative text editor is modeled as a linked list of characters. Each character has a corresponding index in the list, and the valid operations on those characters are insertion/deletion. There's an user interface also developed for providing visual feedback for our editor. For simplicity, we only allow single character insertion and deletion operations. If we want to allow multi-insertion and multi-deletion operations, that will require handling concurrency at both the UI and the CRDT list level. To reduce complexity and decouple the core logic from the UI, a decision was made to only allow single-character operations.

4 ARCHITECTURE

4.1 Initial Architecture

Our architecture involves three main components - **the UI, Local App Instance and the gRPC server**. Starting from the top, we have our UI - WebUI and VSCode, each connecting to the Local App Instance via WebSockets. Websockets allow us to open a two-way communication between the Local App instance and the UI to send messages and receive event-driven responses without having to poll for those responses [10]. The Local App instance represents the current state of the application. The UI queries the Local App Instance to always display

the final consistent state of the application. The gRPC server [6] acts as a central server that allows us to maintain a bidirectional communication channel to broadcast and receive messages with the local app instance.

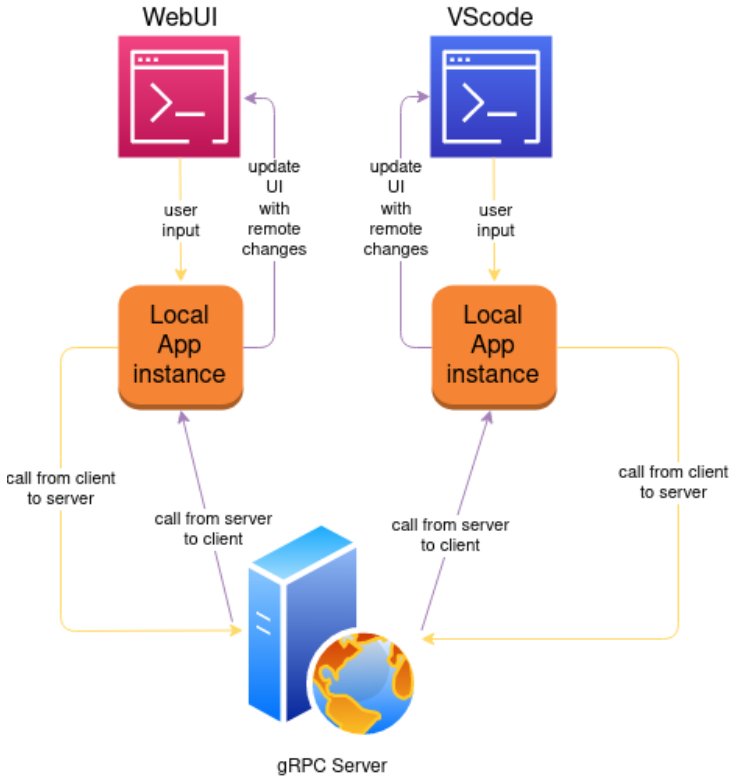

Fig. 4. Initial Architecture Overview

Protocol Buffers [4] is used for serialization/deserialization (serdes) of messages exchanged between the three components. Protocol Buffers is Google's open-source, language and platform neutral mechanism for serializing structured data. Protocol Buffers also quickly integrates with gRPC by providing service definitions for the RPC calls within the proto files itself. This allows us to concentrate on the core functionality of the application and delegate the serdes functionality of the message structure to the Protocol Buffers.

4.2 Final Architecture

The role of the central gRPC server seems very similar to a pub/sub system which can be easily served with a message bus. For the message bus, there were two options - we could go with the uber-popular **Kafka** or with something simple like **NSQ** (New Simple Queue). From experience, Kafka [1] is a resource-hogging service which requires a lot of configuration changes to setup even a simple quickstart application. For example, Kafka has multiple message delivery guarantee of at-least once, at-most once and exact once which requires change in the configurations. For our application, we required a simple lightweight message bus with minimal configuration changes and NSQ is a message bus which exactly fits that purpose. NSQ has a message delivery guarantee of only at-least once, with duplicate messages possible. Figure 5 provides us with the updated architecture.

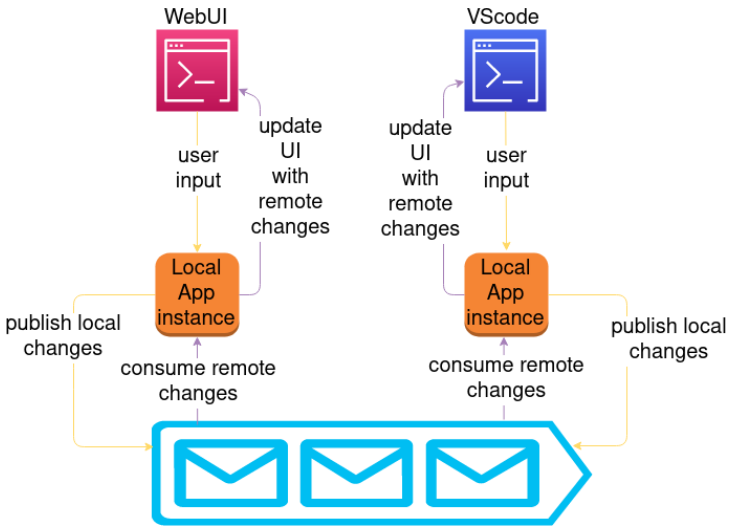

Fig. 5. Final Architecture Overview

5 IMPLEMENTATION DETAILS

We initially referred "A conflict-free replicated JSON datatype" [11] to get a general idea about a JSON CRDT, however, later decided on the fact that a JSON CRDT would be an awfully complex overkill for a collaborative editor. Instead, we decided to model a document as CRDT list of characters which reduced the implementation complexity. The CRDT list is implemented using C++ as a doubly linked-list. This design choice helps in reducing the time complexity of applying the remote updates to the local list to $O(1)$.

5.1 Globally Unique Element ID

Each node in the linked list has a globally unique Element ID associated with it. This element id is constructed as a pair of the peer name p and the Lamport timestamp [12] c of that peer at the instance - $ID(c, p)$. Here a subtle assumption is that each peer will have a unique name. For now the peer name is user specified however, a unique name policy can be easily enforced by using UUID [16] in the name. The element id acts as building block for sending and applying a local insert or delete operation to a remote peer. Two element IDs can be compared as follows:

$$ID_1(c_1, p_1) < ID_2(c_2, p_2) \iff (c_1 < c_2) \vee (c_1 = c_2 \wedge p_1 < p_2)$$

5.2 Element ID to Node Map

Whenever a new node is inserted in the list, it is added to a map with key as its element id and value as the pointer to the node. This helps in optimizing applying remote operations to the local list.

5.3 Insertion

The insert operation requires two values, a value x and a integer index ix representing the position in the list. Internally, we construct a node with a new element id which holds x, find the $(ix - 1)th$ node in the list, and insert the newly created node next to it. This leads to time complexity of $O(n)$ where n is the number of elements in the list.

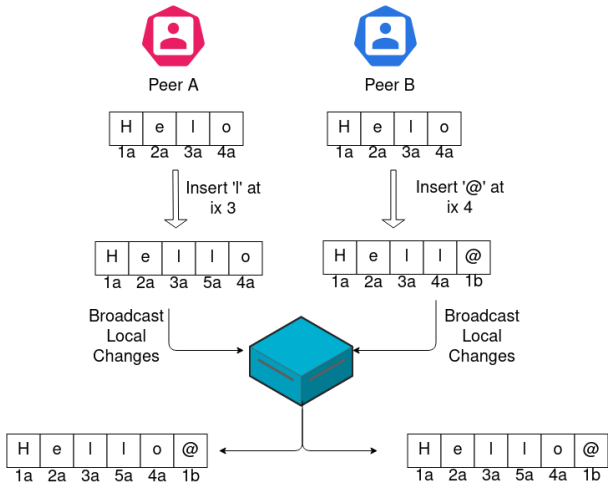

Fig. 6. Inserting in CRDT list

Once the value is inserted, an InsertOperation is generated which will be sent to all the peers. It contains the element id of the newly constructed node, the value to be inserted, and the element id of the previous node i.e. the id of $(ix-1)th$ node (we call this the reference id).

On receiving an InsertOperation, the reference id is used to lookup for the node in the Element ID to Node Map (which is a constant time operation) and the received value x is inserted after it. The nodes that hold x on all the peers will have the same element id i.e. the element id generated at the peer where the insert operation originated.

It is trivial to assume that multiple peers can start a local insert at an index ix concurrently. In order to define the order of inserts deterministically but arbitrary of the value of x, element id of the insertion operation is used to break ties. The value with larger element id will appear first in the list. Since each corresponding node in the list in each peer will have the same element id, the order will be consistent across all peers.

In case of k concurrent inserts, in order to reach the node after the reference node where the condition holds true, we need to traverse the list at most k positions, just the time complexity of applying the remote insert operation is $O(k)$.

Figure 6 shows gives an overview of insert operation.

5.4 Deletion

The delete operation requires the index of the element in the list. Internally, we traverse the linked list till the corresponding index is reached and simply mark the corresponding node as tombstone. This leads to time complexity of $O(n)$ where n is the number of elements in the list. Thus a node is never deleted from the list it is just hidden from the user.

Once the node is marked, an DeleteOperation is generated which will be sent to all the peers. It contains the element id of the node which was marked and the timestamp at which the node was marked.

On receiving an DeleteOperation, the element id is used to lookup for the node in the Element ID to Node Map and the node is marked as tombstone. If the node is already marked tombstone (which indicates that concurrent delete was performed on the same index), then nothing happens. This leads to time complexity of $O(1)$.

Figure 7 shows gives an overview of delete operation.

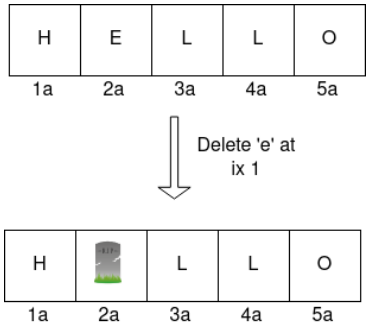

Fig. 7. Deletion in CRDT list

5.5 Casual Dependencies

The operations are broadcasted to remote peer in form of a Change Message which contains the name of the peer broadcasting the change, the change counter, list of operations and causal dependencies of the current change.

Causal dependencies is defined as follows: if operation o_2 was generated by peer p, then a causal dependency of o_2 is any operation o_1 that had already been applied on p at the time when o_2 was generated.

A peer keeps tracks of causal dependencies by using a Vector Clock [12] which stores the number of change messages it has received from other peers. On receiving a change c, a peer p only applies if and only if $p.vector\_clock \geq c.causal\_dependencies$

Causal dependencies enforces a partial ordering on operations ensuring a change is only applied after all changes that happened before it have been applied. This implies a sequence of operations generated at any particular peer will be applied in the same order at every other peer.

5.6 Out Of Order Delivery

Our initial architecture included a Central server to broadcast the changes to peers. The central server would guarantee total ordering of messages originating from a given peer. This is a strong guarantee to have and the cost of implementing this is high. NSQ [7], which we used in our final architecture does not have this guarantee. Furthermore, making this ordering a requirement would prevent us from plugging a P2P network in place of NSQ. In order to solve this, we maintain a receive buffer on a peer by peer basis and on arrival of a new change message, we sort the buffer for the corresponding peer based on the change id.

6 USER INTERFACE

The primary role of the UI is to take input from the user and display the updated content. We decided to abstract the UI component so that it is easier to plug-in different UIs as necessary. By doing so, we were able to achieve collaborative editing on custom made react Web UI [3] and on VS Code using its extension API [9].

6.1 WebUI

The Web interface is developed using ReactJS [8]. This decision was made to avoid writing boilerplate code to wire various HTML components. The WebUI connects to the local App using WebSockets [10]. It prevents multi-character inserts like pasting a word and multi-character deletes like selecting a bunch of characters and deleting them together. This enables us to not have any logic at the WebUI which deals with concurrent inserts and deletes, making WebUI a pure i/o interface.

6.2 VS Code Extension

The VS Code extension was developed using the VS Code extension API. The sole purpose of this extension was to demonstrate how easy it is to plugin a new UI. It not as robust as WebUI against failure scenarios e.g. connection error. And it does not prevent multi-characters edits, it's users responsibility to avoid it.

7 TESTING

Since CRDT list is a client side library completely decoupled from the networking stack, it is fairly easy to test it using Unit tests. We used GTest [5] unit testing framework to write unit tests. The unit tests helped us a lot to catch concurrency bug and trivial issues even before plugging the CRDT list with networking layer. We further integrated these unit tests in our GitHub workflow as a pre-merge step to avoid accidentally merging incorrect changes to the "main" branch.

8 CHALLENGES

8.1 C++ dependency management

We decided to use C++ for its zero overhead principle. However, this decision had a very subtle and an important underlying assumption with it - that it would be easy to build or install the required 3rd party libraries. We both spent literally a couple sleepless nights individually to get the 3rd party library working with our binary build process. This brings us to an important learning, while developing C++ applications, the tool-chain you use plays an crucial role in determining how productively a developer time is spent. We discovered later that using Bazel [2] as our build system, we could have avoided the dependency management issues.

8.2 Working Remote

As of now, the members for this project were situated in different locations in different timezones. Zoom played an important role in the success of this project. No matter how much we planned, sometimes it was not feasible for us to communicate synchronously, so we left comments on each others Pull Requests on GitHub which helped both of us to work asynchronously.

REFERENCES

[1] 2020. Apache Kafka - a distributed streaming platform. https://kafka.apache.org/.
[2] 2020. Bazel - Build and test software of any size, quickly and reliably. https://www.bazel.build/.
[3] 2020. Create React apps with no build configuration. https://github.com/facebook/create-react-app.
[4] 2020. Google Protocol Buffers - Protocol buffers are a language-neutral, platform-neutral extensible mechanism for serializing structured data. https://developers.google.com/protocol-buffers.
[5] 2020. Google Test. https://github.com/google/googletest.
[6] 2020. gRPC - A high-performance, open source universal RPC framework. https://grpc.io/.
[7] 2020. NSQ - A realtime distributed messaging platform. https://nsq.io/overview/design.html.
[8] 2020. ReactJS. https://reactjs.org/.
[9] 2020. VSCode Extension. https://code.visualstudio.com/api.
[10] 2020. WebSockets API. https://developer.mozilla.org/en-US/docs/Web/API/WebSockets_API.
[11] Martin Kleppmann and Alastair R Beresford. 2017. A conflict-free replicated JSON datatype. *IEEE Transactions on Parallel and Distributed Systems* 28, 10 (2017), 2733–2746.
[12] Leslie Lamport. 2019. Time, clocks, and the ordering of events in a distributed system. In *Concurrency: the Works of Leslie Lamport*. 179–196.
[13] David A Nichols, Pavel Curtis, Michael Dixon, and John Lamping. 1995. High-latency, low-bandwidth windowing in the Jupiter collaboration system. In *Proceedings of the 8th annual ACM symposium on User interface and software technology*. 111–120.
[14] Chengzheng Sun and Clarence Ellis. 1998. Operational transformation in real-time group editors: issues, algorithms, and achievements. In *Proceedings of the 1998 ACM conference on Computer supported cooperative work*. 59–68.

[15] Chengzheng Sun and Clarence (Skip) Ellis. 1998. Operational Transformation in Real-Time Group Editors: Issues, Algorithms and Achievements. *ACM Conference on Computer Supported Cooperative Work* (1998), 59–68.

[16] Wikipedia contributors. 2020. Universally unique identifier — Wikipedia, The Free Encyclopedia. https://en.wikipedia.org/w/index.php?title=Universally_unique_identifier&oldid=989664972. [Online; accessed 11-December-2020].