Gollaborate - Final Report

In this report, we introduce Gollaborate, a terminal-based collaborative document editor. We describe its design and implementation. We further demonstrate the correctness of our system and address its limitations. Finally, we certify that all work (both code & report) is original and citations or acknowledgements are provided where appropriate.

Andrew Thappa (c1l0b), Yiyun Xie (c9g0b), Haotian Xiao (v9i0b), Mengyu Han (q8h0b)

Abstract

Gollaborate is a collaborative document editor written in Golang and fully deployed on Azure. Its design is based on an existing solution entitled *Conclave* and the project proposal introduces its specification [1, 2]. The application consists of a server and supports multiple client and application nodes. The application nodes utilize *gocui*, a third-party library, to provide a terminal-based GUI [3]. Gollaborate offers eventual consistency, the notion that operations between users will be consistent at some indeterminate time in the future, via conflict-free replicated data types (CRDT). The system has been tested to support 4 users concurrently inserting and deleting characters on a shared document, but can theoretically support up to 10 concurrent users per document.

1 - System Design & Implementation

1a - Network Architecture

There are 3 nodes in the Gollaborate system - the *server* node, the *client* node, and the *app* node (Figure 1). The server maintains a mapping between document ID to the list of active clients editing the document. The client sends heartbeats to the server to signify that it is alive and actively participating in the network. When a client node wants to join the network, they initiate a connection to the server and supply it with a document ID. If the document ID is not known by the server, then the server will add the client and the document to the map. If the ID is known, then the server will return a list of peers to which the new client should connect to. Thus, collaborators editing the same document form a strongly connected graph (Figure 1). A strongly connected network topology is suitable for a constrained system in which no more than 4 - 5 users concurrently edit a single document.

The client exposes an interface to a text editor application (the 3rd node type in Gollaborate) to perform insertion and deletion operations. Operations performed by one client will be broadcast to all of its peers who are editing the same document. Each peer will then forward the operation to its neighbours based on a time-to-live (TTL) field.

The Gollaborate system enforces a 1-to-1 mapping between text editor to client. Thus, if a user wishes to edit 2 additional documents concurrently, they must open 2 new instances of the text editor app and connect them to 2 new client nodes. The text editor application receives user keyboard input from the command line terminal, displays the characters on the terminal GUI, and invokes RPC calls on the client node to disseminate these operations to the network. The application will also handle RPC calls made by other client nodes in the network and perform the computation to display the inserted character or remove the deleted character.

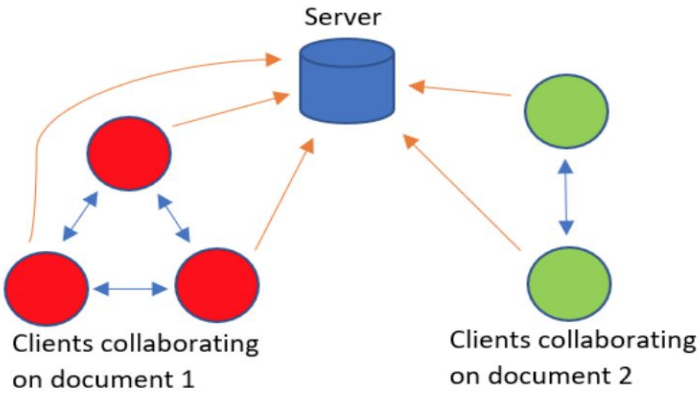


Figure 1 - Network topology in Gollaborate. A central server stores information regarding which client is editing which document. The server coordinates connection of a new node to its peers for a given document. Clients editing the same document form a strongly connected graph and communicate in a peer-2-peer fashion to maintain a consistent state.

1b - Authenticity/Threat Model

Collaboration between peers is built on trust and we assume for this project that if a client has acquired the document ID, it is trusted by its peers. Moreover, there is no incentive to replay operations because operations neither consume nor generate currency/resources or otherwise provide any form of reward. Nevertheless, a trusted client can still be malicious.

The main threat vector we consider is a malicious client assuming the identity of another node and performing edits on its behalf. They could accomplish identity spoofing by substituting the *author* field of the character data structure (*charObj)* prior to sending it to its neighbours. Additionally, they could edit the payload message (e.g. characters to be inserted or deleted) from another client prior to forwarding it on the client's behalf to more nodes in the system.

To prevent such malicious edits, we require that all data structures sent across the network contain the public key of the client who performed the operation and require all fields to be digitally signed with the client's private key. Thus, a CRDT packet is only considered valid if it can be decrypted by the public key of said author. A consequence of this system is that operations cannot be made anonymously since every CRDT is digitally signed.

2 - Model of Consistency

2a - Data Structure & Design

Gollaborate guarantees eventual consistency via conflict-free replicated data types (CRDT). We define eventual consistency to mean that all operations (e.g. insertion and deletion) across all users of a specific document will be harmoniously integrated after some indeterminate time. Thus, two users can temporarily observe inconsistent state in the shared document; however, given time and absence of operations, the two inconsistent states will converge. In this section, we introduce two key data structures used by Gollaborate to guarantee eventually consistency, and work through several scenarios to demonstrate its correctness.

CRDT that is stored locally on client & editor	**CRDT sent via remote procedure call (RPC)**

`type charObj struct {`
`  Author  string`
`  CharValue  rune`
`  EditorPosX int`
`  EditorPosY int`
`  CRDTPos  []int`
`  Opcount  int`
`}`
	`type RPCcharObj struct {`
	`  Author  string`
	`  RPCauthor  string`
	`  CharValue  rune`
	`  CRDTPos  []int`
	`  Opcount  int`
	`  Type  string`
	`  Hash  []byte`
	`  TTL  int`
	`}`

Figure 2. All insertion and deletion operations are captured as a conflict-free replicated data type (CRDT).

When a character is inserted locally, information such as its position in the terminal GUI is encapsulated in a CRDT entitled *charObj* and stored locally. Local deletions are not encapsulated as a *charObj*. However, both local insertions and deletions are encapsulated as a *RPCcharObj,* to facilitate its sending and incorporation into the documents of collaborators.

The first key data structure comes in two flavours (Figure 2). A *charObj* encapsulates all the information necessary to store and display an inserted character. A *RPCcharObj* is created subsequent to an insertion or deletion to capture enough information to enable neighbours to reconstruct the operation consistently. We would like to note 4 potentially non-obvious characteristics of the 2 data structures:

(1) CRDTpos represents the immutable position of a character in the global document distributed across a set of collaborators. This position is translated to and from a mutable position in the terminal GUI to be displayed (EditorPosX & EditorPosY) by the *app* node.
(2) Opcount is the n-th operation by the Author.

(3) Author always indicates the node that inserted the character. RPCauthor indicates the node that inserted the character or deleted the character (which may not be the same as the Author).
(4) Type indicates whether the operation is an insertion or deletion.

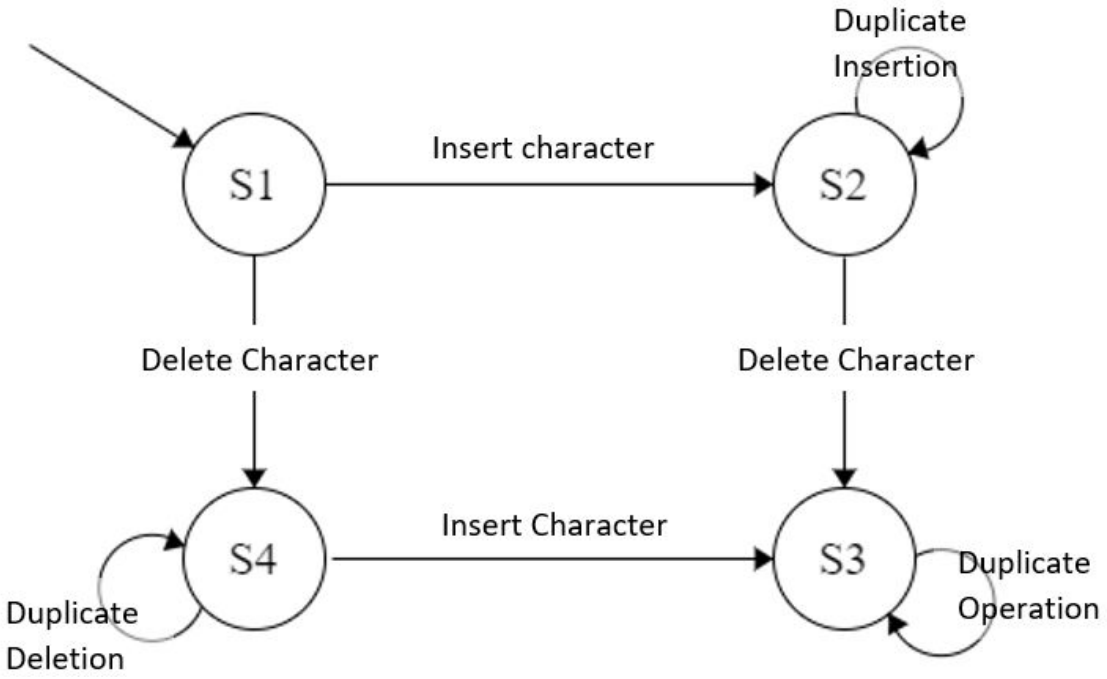


Figure 3. Finite state machine depicting per character state for all sequences of operations. Each character is identified locally by a 2-tuple: (1) the author of the character and (2) the operation count (e.g. it's the 7th character inserted by the author).
S1: By default, the space of all such characters starts at S1 - *the n-th character by author XYZ has not yet been inserted*.
S2: When the n-th character by XYZ is received (via RPC call or inserted locally by XYZ), the character transitions from S1 to S2 - *said character has been observed and inserted*. If a duplicate insertion is received, the FSM remains at S2.
S3: When the character is deleted locally or remotely, the FSM transitions from S2 to S3 - *the character has been observed and subsequently deleted*. The FSM will remain at S3 upon further duplicate requests by the local user or neighbours to insert or delete the character.
S4: If a request to delete the n-th character by XYZ is received before it is observed and inserted, the FSM transitions to S4 - *a request to delete the character that has not yet been received from the network*. Duplicate deletions are dropped at S4 and the *RPCcharObj* is inserted into a deletion buffer. Upon further insertions of any character, the deletion buffer is checked to see if any pending deletion requests can be serviced.

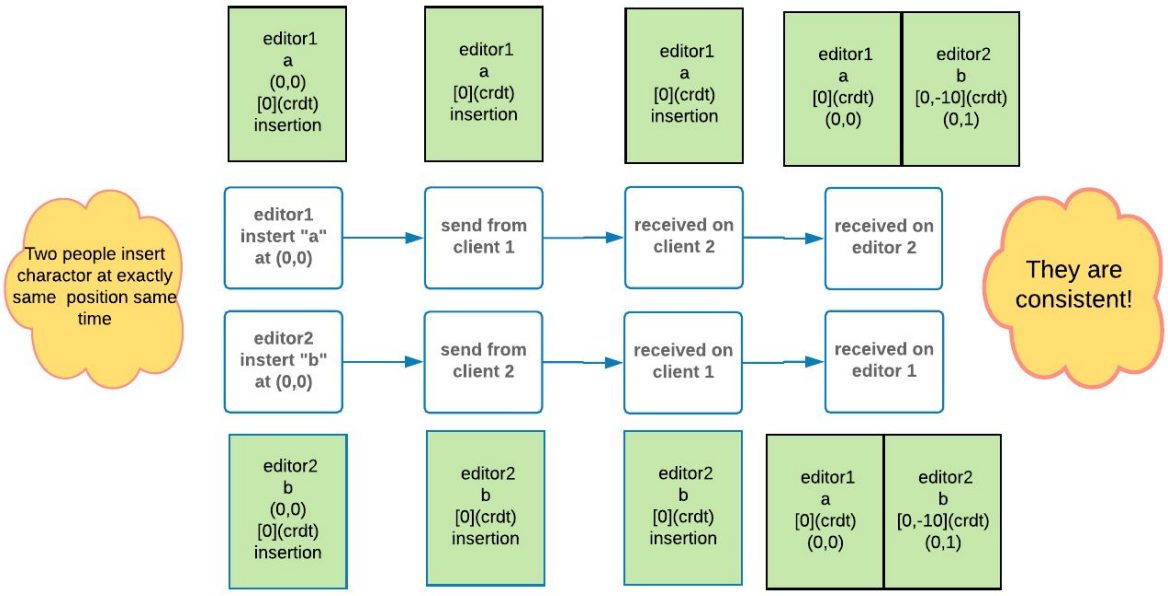
The second data structure is captured by the finite state machine in figure 3. All client nodes are responsible for keeping track of a map between a 2-tuple of author and operation count to a state S1 - S4. Local and remote operations trigger transitions of the n-th character by author XYZ between states.

In conjunction with the first data structure, the map will help ensure all users eventually converge to the same document view.

2b - Operation Scenarios

In this section, we describe 5 scenarios where a set of users undertake a sequence of operations and how our system maintains consistency across the users. One important point to note is that all users maintain a linear slice of *charObj* data structures, and that “enter” and “space” are treated as characters.

Scenario 1: Two distinct users simultaneously insert at the same position in the document

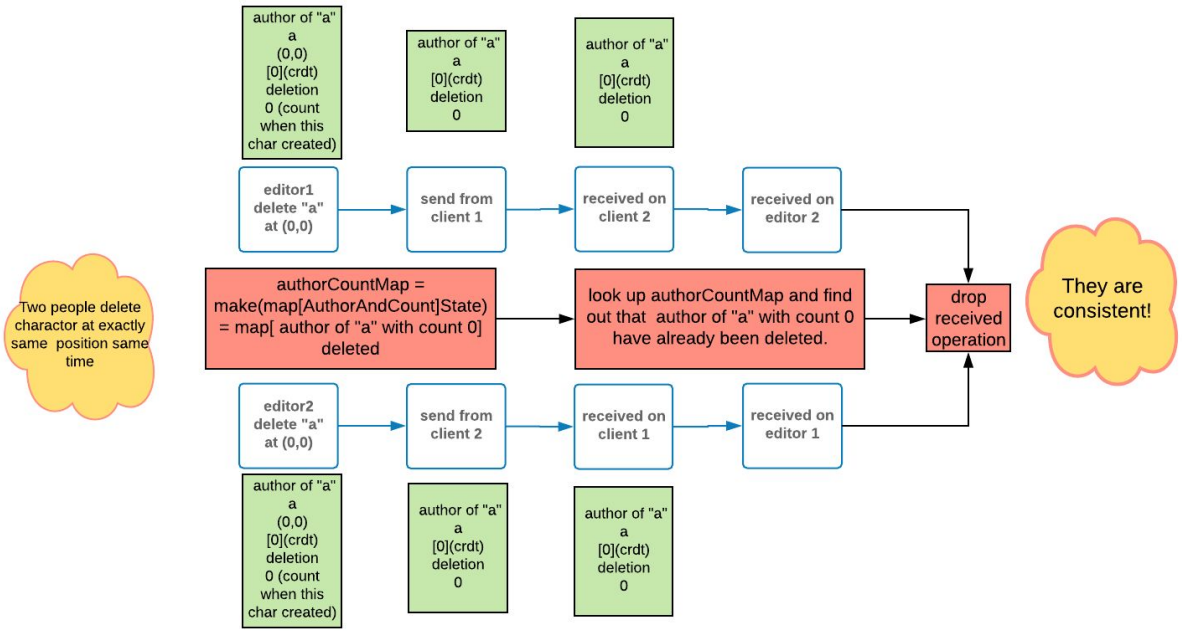


The diagram above depicts the sequence of events that take place, from left to right, when two users insert a character at the same position in the shared document. The blue outlined boxes in the middle depict actions taken by the users: (1) insert a character into editor, (2) send the *RPCcharObj* to neighbours via RPC, (3) and receive *RPCharObj* from neighbours. The green boxes depict the state of the *charObj* data structures as stored on each client. We observe that when both clients insert a character, the CRDTpos value of the generated *charObj* and *RPCcharObj* is the same because the users are not yet cognizant of neighbour actions. Upon receiving an RPC from a neighbour with the same CRDTpos, the editor will sort the overlapping characters (1) alphabetically by author name and (2) in ascending numerical order by operation count. Thus, both clients will resolve to the same state. The diagram depicts that, post-resolution, the letter “b” has the CRDT position [0, -10]. The -10 indicates a “new level” in the array of CRDT structs, and helps us to resolve further conflicts such as when other characters arrived after “a” was inserted by editor1 but before “b” arrived at editor1. For a hypothetical 3rd client, they will receive two *RPCcharObjs* from client1 and client2, and resolve them in the same manner.

** as an aside, (x) indicates editor terminal GUI position, and [x] indicates CRDT position*

Scenario 2: Two users simultaneously delete the same character

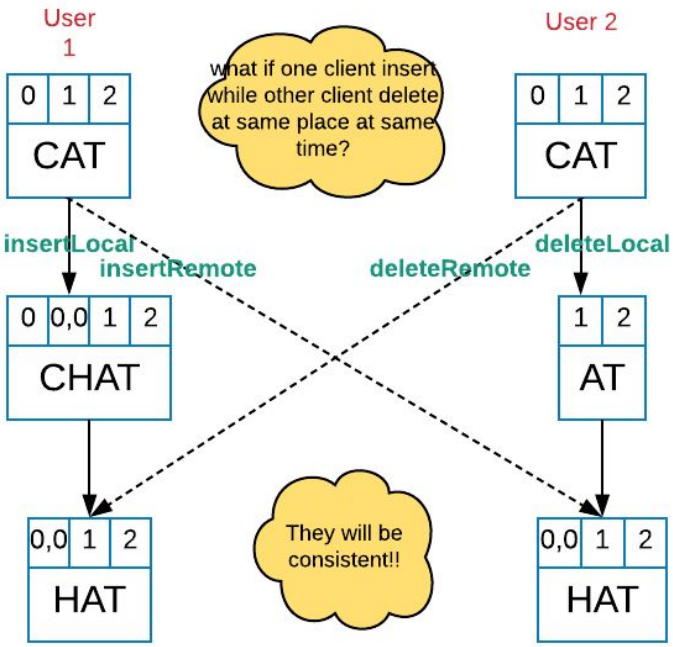
The second scenario is depicted in the diagram below. When two clients simultaneously delete the same character (or different characters in the same position), the operation is idempotent. Idempotency is enforced by the finite state machine described in section 2a, figure 3. When user1 deletes a character "x", it will package and send an *RPCcharObj* to user2, who has also deleted the same character. User2 will drop the *RPCcharObj* because it will first check the state of the character in the *RPCcharObj* by indexing into a locally-stored map based on the author of the character and its operation count (e.g. it was the 7-th operation by user XYZ). Because user2 already deleted the character, the map will return S3, indicating that the character was previously observed and deleted. Likewise, user2 will have sent an *RPCcharObj* to user1 instructing it to delete the character "x", and user1 will drop the data structure after consulting its locally-maintained map.



Scenario 3: Insertion and deletion operations are commutative

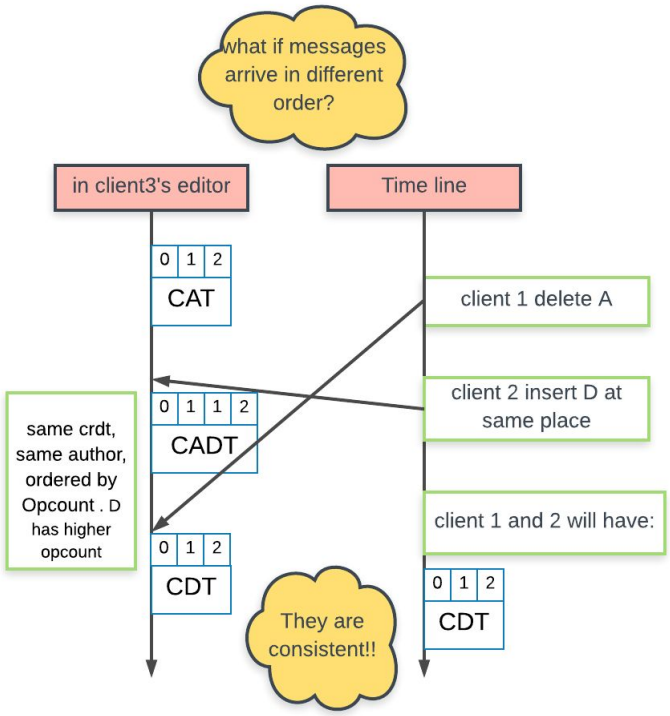
The 3rd scenario confirms that insertion and deletion operations are commutative. In the two diagrams below, we observe 2 users sharing one document. User1 inserts the letter "h" while user2 concurrently deletes "c". Because it takes time for each user to package the operation into an *RPCcharObj* and send it across the network, user1 will observe the insertion before the deletion and user2 will observe the deletion before the insertion. Because "h" is inserted between two existing characters, our logic will generate a *charObj (and RPCcharObj)* where the CRDT position has a new level indicated by the second 0 in the slice: [0, 0]. The first 0 indicates it's relative position to the characters A & T (i.e. the characters

that follow after), but the second zero indicates its relative position to the character “c” (i.e. the characters that go before). When the deletion arrives, user1 consults its map to make sure that the character “c” was observed, deletes it, and updates the map. Likewise, user2 deletes the character “c”, sends out the aforementioned deletion, and receives the insertion of “h” by user1. The letter “h” has CRDT position [0,0]; therefore, user2 knows to insert the character “h” before “a”. Thus, both users observe the consistent state of “HAT”.



Scenario 4: Out-of-order messages

This scenario concerns out of order messages. In this case, we focus on user3 who receives a request to delete a character before it receives the character from the network. Per the finite state machine in section 2a - figure 3, the client will consult its map of 2-tuples (author and operation number) to the state of the character. Because user3 has not yet seen the character, it will transition the character in the map to state S4 (which indicates that we need to delete the character, but we haven’t seen the character yet). Additionally, we’ll add the deletion *RPCcharObj* to a deletion buffer. At each subsequent insertion, we’ll check the deletion buffer to see if the character needs to be deleted. When the insertion finally arrives, we’ll check the map and discover the character is at state S4. Hence we’ll update the deletion buffer by removing the *RPCcharObj*, and we’ll transition the character in the map to S3 which indicates the character has been observed and deleted. Note that in this sequence of events, we do not insert the character onto the screen for user3 because we knew that we needed to delete it.



Scenario 5 - Client Initialization

In this simple scenario, we imagine a client joining an existing network of collaborators who are busy typing away. The client will initialize by requesting a copy of the document (linear slice of *charObj*) and the map from 2-tuple (author & operation number) to character state. But there can be operations from neighbours that it missed in-between the time that it requests the document to when it finally connects to all neighbours. It will still receive these operations because all neighbours are responsible for propagating them multiple times in the network based on the TTL field. So if the new client missed the operation the first time around, it will receive them later.

3 - Fault Tolerance

Mode of Failure	**Semantics**	**Action**
All clients fail stop	-If all the clients collaborating on a document fail, the document is lost	-No persistence so clients must reconnect and a new document with the given document ID is created

Some clients fail stop	-If a portion of clients collaborating on a document fail, they request the document from a peer upon reconnect	-Reconnect to server with the appropriate document ID and grab document from a client
Server fail stops	-Clients can continue to collaborate on a document -No new clients can join a document	-New server with same IP:Port must be instantiated - When server is up again, clients will automatically reconnect

In our system, we rely on a central server to tell a client which peers to connect to in the process of opening a document to edit. A characteristic feature of distributed systems is the idea that failure does not affect operation of other components. As such, we built our system to handle the failure of both *clients* and failure of the *server*. When the *server* node fail stops, clients are able to continue collaborating with their peers without issue as the server plays no role in mediating the communication between collaborators. When all client nodes fail, the document will be lost since we do not save state on the server. When some of the clients fail, each client can simply reconnect to the server and get the document from one of its neighbours. The clients that did not fail are not affected.

4 - Discussion

4.1 - Application Limitation

There are a number of limitations in our implementation of Gollaborate, which are enumerated below:

- If you hold down a key to insert or delete characters, the application will generally become inconsistent. This is due to the way characters are inserted into our CRDT and we would likely have to test different CRDT implementations to fix this.
- Wrap-around (when reaching end of terminal) is not implemented in the application to reduce application complexity and focus on distributed complexity.
- Gocui, a library we used for terminal rendering, is not compatible with all terminal programs. If using XShell, please upgrade to the latest version. A free alternative is MobaXterm.
- Undo/Redo is a core feature in any type of text editor, but in our application it is not included.
- Our application poorly scales because each node is strongly connected to its peers on a per-document basis.
- Right now, if all clients disconnect from a document you lose the document. Ideally, when all clients disconnect from a document, the document would persist on a server.

- Ideally we would have some sort of paragraph level locking mechanism so a document would be composed of a bunch of paragraphs and each of these can either be locked to edit and unlocked to edit.
- In our system, document names need to be unique because we assume that if you have a document name, then you are trusted to be an editor by someone currently editing the document.

4.2 - Future Work

Gollaborate is a terminal-based collaborative document editor. However, in order to have a full implementation of a collaborative document editor we would need to focus on the following four areas. First, you would need the ability to support undo/redo so that collaborators can have the ability to undo or redo any changes that have been applied to the entire document. Second, we would also want the server to save a copy of the document so that if everyone is disconnected from a document we can access it later. Third, we would want the ability to save and load documents from a local file store. Finally, we would want the ability to scale to more than 10 collaborators which would mean using a network topology different from the strongly connected graph we currently use.

References:

[1] Han, Mengyu; Thappa, Andrew; Xiao, Haotian; Xie, Yiyun. "*Gollaborate"* CPSC 416 Project Proposal.
[2] Beatteay, Sun-Li; Savant, Nitin; Olivares, Elise. *"Conclave Case Study".* Last accessed on April 6th, 2018: https://conclave-team.github.io/conclave-site/
[3] Martin, Roi. "*gocui: Minimalist Go package aimed at creating Console User Interfaces".* Last accessed on April 6th, 2018: https://github.com/jroimartin/gocui