Gollaborate

Real-time collaborative text editor utilizing conflict-free data type (CRDT) to impart consistency among collaborators in a peer-2-peer network

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

1 - Introduction

Groupware is a type of software that coordinates actions from multiple clients. Collaborative document editing is one manifestation of groupware in which multiple users are able to concurrently view and edit documents in real-time. Examples of real-time collaborative editors include Google Docs and Quip. Maintaining a consistent view when users are making concurrent edits to a shared document is challenging. Several solutions have been proposed to solve this challenge such as operational transformation [1] and conflict-free replicated data types (CRDT) [2]. Gollaborate aims to resolve concurrent editing amongst collaborators by achieving eventual consistency via CRDT.

2 - System Overview

2a - Network Architecture

There are 2 nodes in the Gollaborate system - the *server* node and the *client* node (Figure 1). The server node exposes a simple interface to enable clients to register with the server. The server maintains a mapping between document ID to the list of active clients editing the document with the aforementioned ID. 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 that are actively editing the document to which the new client should connect to. Thus, collaborators editing the same document form a strongly connected graph (Figure 1). Operations performed by one client will be broadcast to all of its peers who are editing the same document and consistent state can be negotiated amongst peers. This network topology is suitable for a constrained system in which no more than 4 - 5 users concurrently edit a single document.

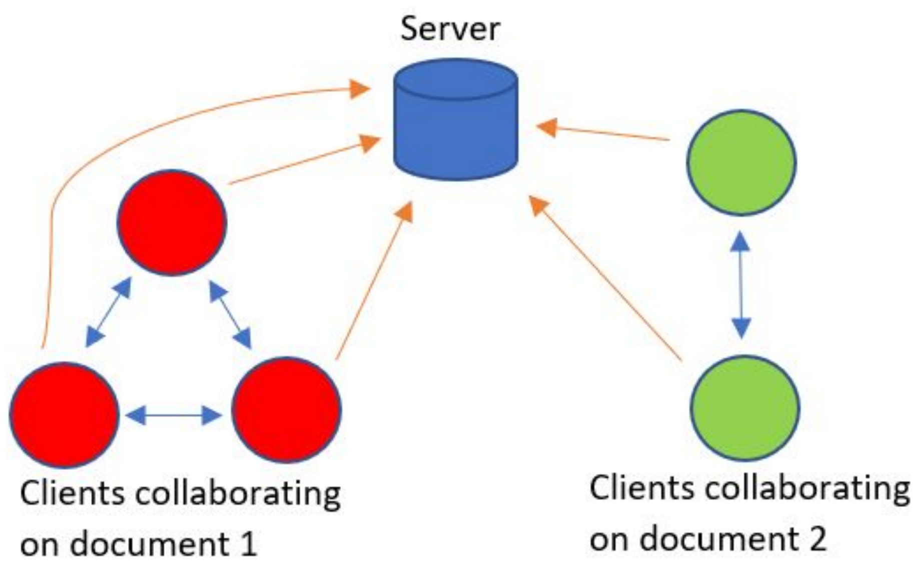

Server
Clients collaborating on document 1
Clients collaborating on document 2

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.

The client exposes an interface to a text editor application to perform insertion and deletion operations. The Gollaborate system enforces a 1-to-1 mapping between text editor to client (Figure 2). 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 from the client node to receive operations made by other client nodes in the network and display the inserted character or remove the deleted character.

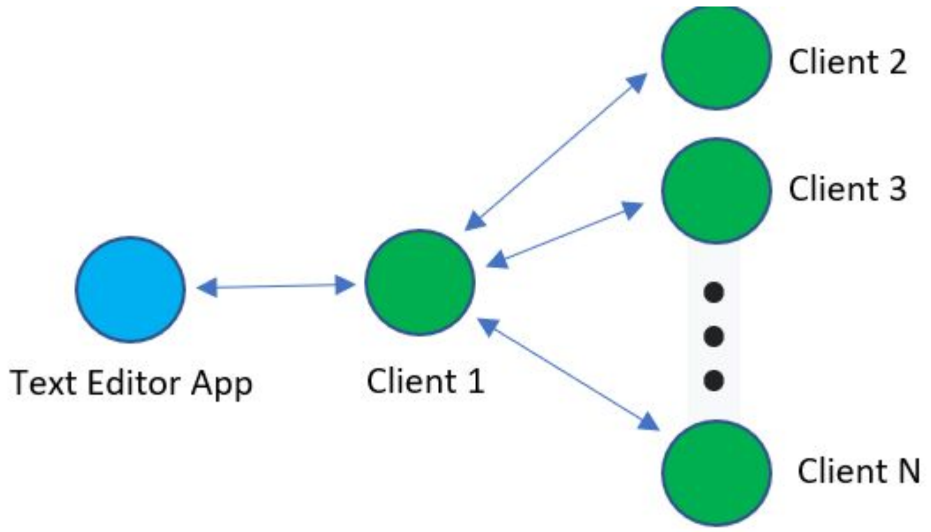


Figure 2 – One text editor connects to one client. The Gollaborate system enforces a 1-to-1 mapping between a text editor application and a client node. All clients 1 to N maintain a connection to a distinct text editor application, but only the connection between an app and client 1 is shown in this diagram for clarity. If a user wishes to edit more than 1 document on a given computer, they must initiate a new instance of a client-app pair for each additional document.

2b - Consistency Model

When dealing with a groupware system, one of the key challenges comes in the form of providing consistency across all nodes in the network. In our application, this means that when an edit is made to a document, making sure that changes are properly applied given that every user could see a slightly different version due to propagation and network delay. Broadly there are two approaches to handling this: Operational Transformation and Conflict-Free Replicated Data Types (CRDTs).

Both of these paradigms are ways of implementing *eventual consistency* which is the idea that if no updates to a document take place for a long time, all the replicas of the document will eventually become consistent. CRDTs perform operations on the local version of a CRDT whose state is then sent to the other nodes of the network and is eventually merged with the state of a copy. For our system, we propose using CRDTs due to empirical research [3] that has shown that in practice, many OT algorithms do not satisfy the convergence properties stated by their authors. The two figures below highlight two common situations arising from collaborative editing and how they are mitigated by use of the CRDT data structure.

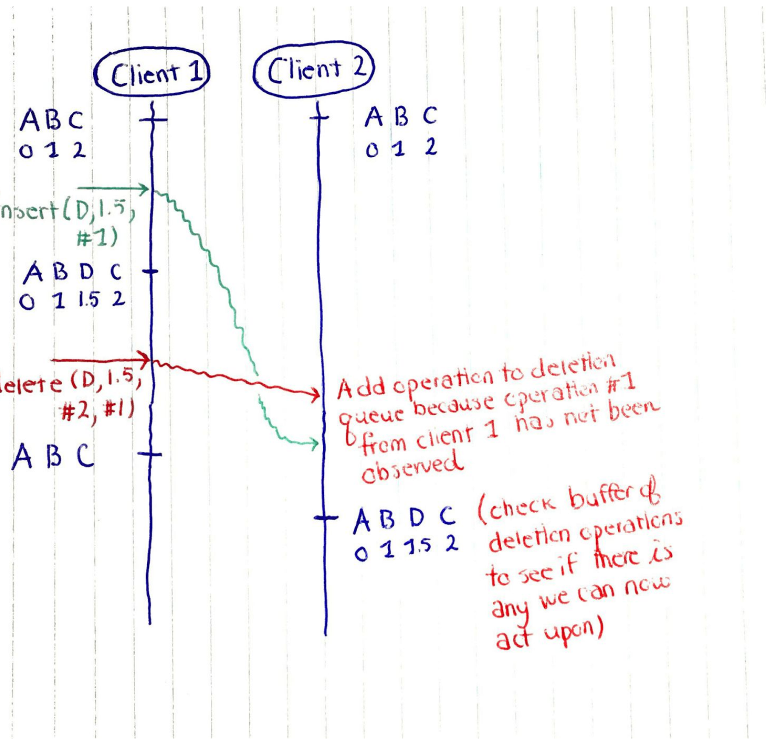
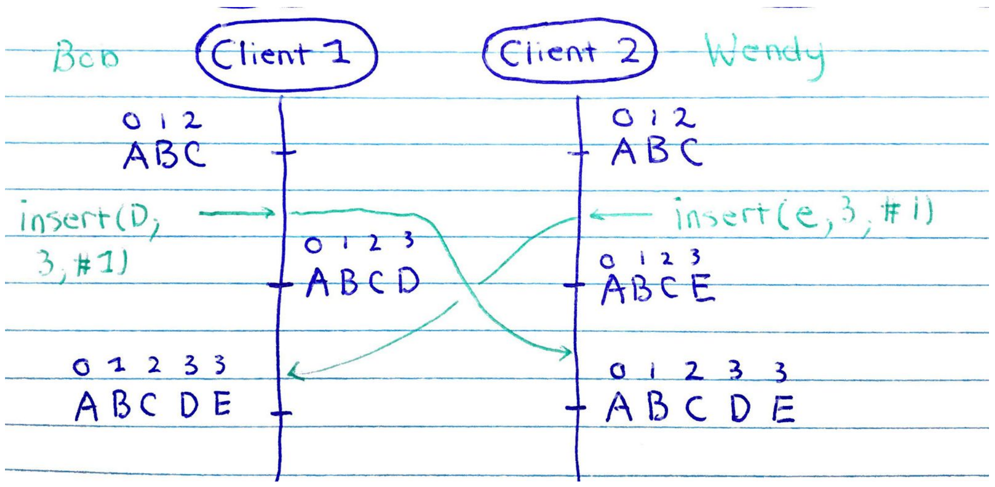

Client 1
Client 2
ABC
0 1 2
A B C
0 1 2
insert(D, 1.5, #1)
A B D C
0 1 1.5 2
delete (D, 1.5, #2, #1)
Add operation to deletion queue because operation #1 from client 1 has not been observed
A B C
A B D C
0 1 1.5 2
(check buffer of deletion operations to see if there is any we can now act upon)


Figure 3: CRDT ensures eventual consistency of state among clients.
Top: To handle out-of-order operations, we include the count of the number of operations performed by a client along with the CRDT data object that is sent to peers. If a request is observed to delete operation #7, but it hasn't been seen yet, we'll add it to a deletion buffer and check the buffer upon future operations to see if any item in the deletion buffer can be acted upon.
Bottom: When two clients add a character with the same relative positioning, we'll further sort alphabetically based on the author name. Thus, Bob's insertion of the letter "D" precedes Wendy's insertion of the letter "E".

2c - Authenticity/Threat Model

Collaboration between peers is built on trust. A peer cannot join a subnet to edit a specific document unless provided the unique document ID which was decided by the document creator. Moreover, there is no incentive to replay operations because operations neither consume nor generate currency/resources or otherwise provide any form of reward.

However, a malicious client could potentially assume the identity of another node and perform edits on their behalf. They could accomplish identity spoofing by substituting the *author* field of the character object 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 both the *author* and *message* field to be digitally signed with the client's private key. Thus, a CRDT packet is only considered valid if its author and message parameters 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 contains a digitally signed author and payload field.

2d - Fault Tolerance

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 will build 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. However, new clients are unable to join an existing cohort to begin collaboration. To mitigate this failure path, the recovery mechanism is that: (1) a new instance of the server with the same IP:port pair as the previous server will be instantiated and (2) the client nodes will detect a lack of heartbeat replies and attempt to reconnect to the server using the same IP:port combination. In this case, the first client to reconnect to the server will continue working with the state of its data structures. Any further clients that re-connect to the server will drop their own state and receive the state of the 1st client and proceed using that state. If the *server* experiences transient disconnection, then some clients may detect disconnection and attempt to reconnect. When they receive a non-empty set of peers to connect to, they will connect and request a copy of the document from a random peer and drop their own copy.

On the other hand, if any *clients* fail, there are two situations that we need to consider. First, if we have *n* clients connected to a document and one client fails, then that client will lose its state, connect to the server for a given document ID, and request the document from some peer. Due to the eventual consistency of CRDTs this is a trivial case. The second case to consider is the one in which all clients for a particular document fail. In this case, when a client fails it will reconnect to the server which will return a set of clients connected to this document, which in this case will be an empty set. When the client node sees the empty set, it will check its local cache to see if it has a stored version of the document or not. If so, that cached version becomes the most recent version of the document. If there is no cached version of the document, a new document will be created. Caching a local copy of state will occur on a timed interval for each client independently of other clients.

3 - Miscellany

3.1 - Utilizing Azure

All components of the Gollaborate system will be deployed on Azure. The system is composed of a single *server*, multiple *client* nodes, and multiple *text editor applications*. The server and each client-text editor pair will live on its own VM instance. For example, 1 server and 4 clients+text editor apps requires a total of 5 Azure VM instances. Other than the GOCUI library, no additional software, plugins, or tools need to be installed on the VM instances.

3.2 - System Testing

Each component of the Gollaborate system (client node, server node, and text editor application) will be tested as they are implemented. All API and internal helper functions will be unit-tested to ensure that they conform to the specifications. There are two important milestones in the project dedicated to testing:

- March 30th: Bugs could arise when integrating the front-end (terminal text editor) with the back-end (client & server). During the week between March 26th to March 30th, we will be dedicating full group resources to testing the system and ensuring that the two halves interoperate smoothly and correctly.
- April 4th: During this week, we will further test deployment of the Gollaborate system on Azure and resolve any issues as they arise during the demo process before the project deadline.

Testing of our system is simplified because all components are written in Go. There is minimal dependency on external frameworks, libraries, and tools.

4 - Risk Analysis

4.1 - SWOT Analysis

Internal	
Strengths	**Weaknesses**
● Team members meet at least twice per week (usually 3x/wk) in person. ● Team members have worked with each other on project 1 of this course. ● Each team member has well-defined tasks to be completed at certain dates. Each member's role and responsibility is clearly delineated.	● All team members began learning Go this semester. ● Haotian is taking CPSC 411 - Compiler Construction, which has weekly deadlines. He will be constrained during certain weekdays to make progress on deliverables.

External	
Opportunities	**Threats**
● Go Console User Interface (GOCUI) library enables efficient implementation of a front-end console-based user interface. ● Collaborative editors are a mature system with well-understood algorithms, data structures, and solution approaches	● Conflict-free replicated data types are straightforward to understand conceptually, but may be difficult to implement practically ● Real-time editing in a peer-2-peer fashion increases project difficulty with regards to consistency ● Azure funds may be insufficient given that we expended $70 over a week of testing. In this project, we anticipate testing over a longer period of time.

4.2 - Project Timeline

Deadline	**Description of Deliverable**
March 9th	Final project proposal completed by group
March 12th	● Andrew has partially completed the server code, implementing the following RPC calls: connectDoc(docID string), heartbeats() ● Haotian has partially completed client code, implementing connection to server and establishing bi-directional RPC calls to peers ● Mengyu has completed console-based UI for text editor application using GOCUI library ● Yiyun has completed functions to send pending characters that need to be sent to other clients
March 19th	● Mengyu completes the data structures and helper functions on text editor application to send and receive CRDTs to client node. ● Haotian completes client logic to handle remote insertion of a character. ● Yiyun completes client logic to handle local insertion of a character. ● Andrew completes server recovery mechanism after server node failure
March 23rd	● Setup project status meeting with TA supervisor
March 26th	● Mengyu and Yiyun complete client logic for handling local deletion of a character. ● Andrew and Haotian complete client logic for handling remote deletion of a character.
March 30th	● Integration of front-end (GUI) and back-end (client node) by group to ensure full functionality of local and remote insertions & deletions. ● Group has completed testing of application to ensure full functionality in the presence and absence of node failures
March 30th	Final report draft completed completed by group
April 4th	All group members will contribute to integrating the project with Azure. All necessary VMs must be configured appropriately with Go 1.9.2, the GOCUI library must be installed, and the system should be

	bug-free. Additionally, the group will formulate and practice a demo of the system in Azure.
April 6th	Project code & final report completed

**course-defined deadlines are highlighted in blue*

References:

[1] Agarwal, Srijan. *Building a real-time collaborative editor using Operational Transformation*. https://medium.com/@srijancse/how-real-time-collaborative-editing-work-operational-transformation-ac4902d75682 [Accessed February 26th, 2018]

[2] Yigitbasi, Nezih. *A Look at Conflict-Free Replicated Data Types (CRDT)* https://medium.com/@istanbul_techie/a-look-at-conflict-free-replicated-data-types-crdt-221a5f629e7e [Accessed February 26th, 2018]

[3] Imine, A., Rusinowitch, M., Oster, G., & Molli, P. (2006). Formal design and verification of operational transformation algorithms for copies convergence. In *Theoretical Computer Science* (Vol. 351, pp. 167–183). https://doi.org/10.1016/j.tcs.2005.09.066 [Accessed February 26th, 2018]