VaporPad: A Web-Based Collaborative Notebook

Authors: Lisa Agastine, Josh Caldwell, Mark Ferrall

Abstract

Real time collaborative text editors allow many clients to participate in the joint creation of a shared document. The immediate collaboration offer many benefits to users as the creation of a shared state simplifies the editing and revision process and offers opportunity for immediate feedback during document development. This document details the specifications for, design decisions behind, and operations of VaporPad, a lightweight collaborative editor. The document includes a discussion of the client-server model that the program uses and the Operational Transformation algorithm as implemented.

Keywords

Real time text editor, Collaborative editor, Operational transformation, Client-server architecture

Table of Contents

Project Objective **2**
Demonstration Video **2**
Acceptance Criteria and Status **2**
Development Team Skills Overview **3**
Technical Challenges **4**
Technology Specifications **4**
Program Architecture **5**
Front End 5
Server 5
Database 6
Workflow **6**
Technology Stack in Detail **10**
Front End 10
Server 12
Database 15
Development Environment 19
Version Control 20
Instructions for running the project **20**
Maintaining Consensus **21**
Operational Transformation 21
Future Improvements **27**
References **28**

Project Objective

The project objective was to build a real time collaborative editor modeled on EtherPad. This collaborative text editor permits multiple users to work on a single document simultaneously via the network in real time. To achieve consistency across all users, we utilize an Operational Transformation algorithm, alongside a client-server communication model.

Demonstration Video

A video demonstrating the functionality of the project is available at the following link.
https://youtu.be/Kwa39vGP9fA

Acceptance Criteria and Status

Given the objective to develop a real time collaborative text editor, the team identified the following acceptance criteria.

Requirement	**Acceptance Criteria**
User Account	1. The user account must include 1.1. The ability to create a new user account 1.2. Password protection of the account 1.3. Access to all the notes created by or shared with a user 1.4. The ability to sign out of an account
Database	2. The notes database must support 2.1. Creation of a new note 2.2. Storage of the full text of each note 2.3. Retrieval of notes by user 2.4. Storage of note history 2.5. Saving new copies of the note as they are updated 2.6. Addition of new users and passwords
Notes	3. A client facing note must support 3.1. The ability to edit a note all times 3.2. Distribution of edits to the server if the network is available 3.3. If network is unavailable, maintain local edits and the ability to edit until the connection is restored or times out 3.4. Incorporation of external edits upon receipt,

	while maintaining the integrity and consistency of those edits across all clients and the server 3.5. Maintaining cursor position as new edits are incorporated
Server	4. The server must support 4.1. Creation of a new note 4.2. Retrieval of current note state from the database 4.3. Distribution of a current note state to a client 4.4. The ability to share notes 4.5. Maintenance of all edits to a note during an editing session 4.6. Incorporation of edits from a client in a way that preserves the client's intent 4.7. Designation of the total order of edits by all clients 4.8. The ability to transform an edit to account for any previous edits 4.9. The ability to broadcast edits to clients in a way that guarantees convergence of document states 4.10. The ability to write notes to the database when all clients have finished editing a note

Project Status

The project is fully functional and addresses all requirements outlined above. Several issues related to workflow and error handling still exist, and the opportunity for functional and nonfunctional improvements remain. These are detailed in Future Improvements.

Development Team Skills Overview

The skills of each team member contributed to the design choices that were made throughout the project. Development time was reduced by by choosing technologies that team members were already comfortable with. This required integrating several technology stacks into the final project, dependent on each team member's area of responsibility.

- Lisa Agastine - Experience in PHP, JS, HTML, CSS, Jquery and with backend MySQL
- Joshua Caldwell - Experience in C++, HTML, CSS, JavaScript, MySQL, Qt, and PHP
- Mark Ferrall - Experienced with C++, Python, some HTML/CSS, and limited Javascript. Experienced with relational databases.

Technical Challenges

As the project team designed and developed the application, several technical challenges became apparent:

- Operational Transformation (OT) Algorithm: The OT algorithms identified in the academic literature were more generic in nature, and did not clearly identify how much of the implementation of the algorithm is application dependent. Even after a model of the OT algorithm was identified, it was extremely sensitive and error prone, requiring substantial debugging.
- Our team had no previous experience with either Node.js or MongoDB. We had some experience with similar technologies but learning these new technologies required a substantial time investment.
- Our application expects the server to handle the set of changesets of notebook content, combining it into the document revision history before broadcasting to other clients. This required use of the server's memory, however, the team was not experienced in managing server side memory objects. The team identified Redis, an in memory key value pair database, to fill this role, but the implementation was challenging.
- Implementing socket.io in our application was challenging as we had no prior experience on sockets and room concept.

Technology Specifications

Frontend:	HTML, JS, CSS, Jquery, Ajax, Bootstrap
Server:	Node JS
Backend:	Mongo DB
Data Store:	Redis

Program Architecture

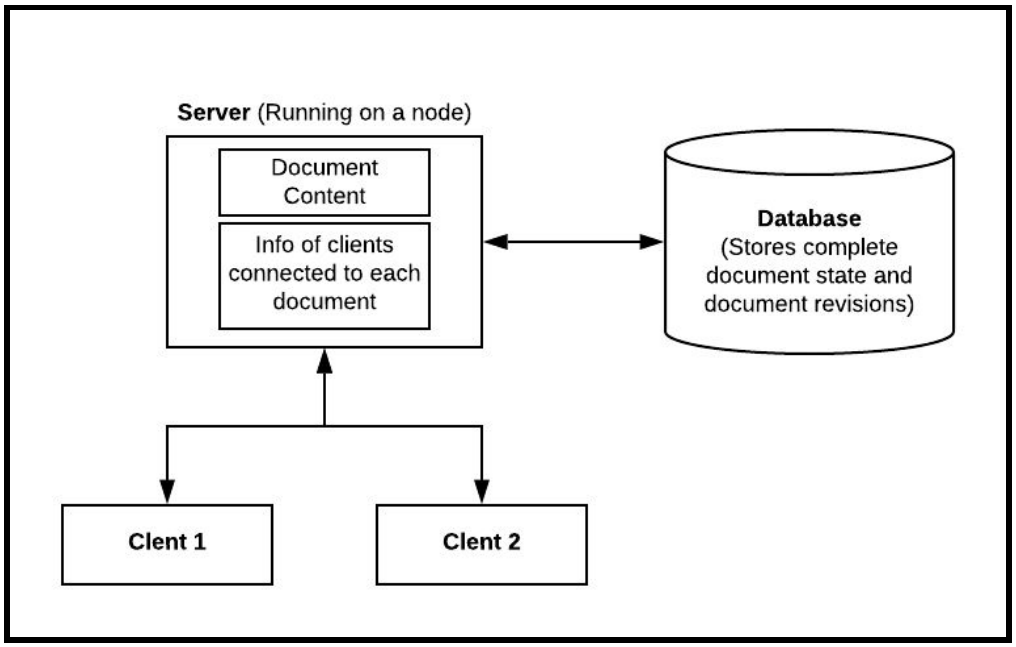

Figure 1: VaporPad Architecture Diagram

Our high level architecture is structured as follows:

Front End

The front end is responsible for presenting the user with a text based note input as well as a login page and ability to select and share notebooks. We developed this with HTML, CSS, Javascript, Jquery and Ajax.

Server

The server is responsible for communicating data to and from the front end and the database. The server reconciles changes between each user editing a notebook and is responsible for communicating the shared notebook state to both the front end and database. The server was developed using Node.js. The server utilizes socket connections for bidirectional communication between client and server and Redis for in memory storage of document edits. We utilize many other Node.js libraries including Express, Body-parser, Redis, MongoDB, and Socket.io. These are discussed further in the technology stack section.

Database

The database is responsible for storing the begin and end states of a shared notebook as well as storing iterative updates that would allow the notebook to be permanently saved periodically in case of a server failure. We used MongoDB as our database of choice as it is a distributed database which could scale horizontally to large amount of data as the VaporPad user accounts and data grow.

Workflow

In determining how we would program and setup the application, we first determined what a typical user workflow should look like. Ultimately we wanted a user to be able to share and collaboratively edit a note, but to build a working application that supports this objective required development of additional features. The section below discusses some of the additional features necessary to support our main objective of collaborative note editing.

1) As a part of our initial objective of building a note application similar to EtherPad, we needed a way for a note to persist after editing the note. A note application likely would not be useful if the note was deleted after exiting the application. This required some way for the note to be stored persistently in memory when the user exited the application. For this we decided to use a database as it would allow us to structure the information and query the data easily. While there are other ways to store information persistently on disk, such as in files, a database seemed to be the best option as it is designed for storing large amounts of data and supports queries. For details please see the database architecture section.
2) The user would expect to create a note, edit it and re-open that note at a later time. Based on the collaborative editing functions, they would also expect to share that note with other users. Due to these requirements, we needed some user login functionality for a user to be able to create an account, log in to view and edit their notes and also share their notes with other active users. This user data would need to be stored persistently similar to the note data. For this we again needed to use the database. For details please see the database architecture section.
3) The user would need a way to pass their login information to open, view and edit their notes. For this we needed some sort of user interface. We decided to implement a web-based user interface as our team had some experience in web development. We could have instead used a PC based tool such as Visual Studio but this seemed more limiting than a web based environment. For details please see the front end architecture section.
4) Finally, to achieve our goal of simultaneous, collaborative, editing of a note we needed a centralized processing point that would receive updates from all users currently editing a note and combining those edits into an “agreed” state that accounted for all user edits of

that note. This required a way for user edits on the front end interface to be communicated to the central processing server. For this we decided to utilize a web server. For details please see the server architecture section.

Once we determined the general architecture necessary to support the basic workflow of our objective, we developed a more detailed workflow to understand communication flow between the architectural pieces of our project, specifically the front end, server and database. This was important as each member of our team was working on different pieces of the architecture and it helped each of us understand the communication and functions needed to support each workflow item. The workflow below is a detailed look at the actions a user has available at each step in the workflow, and the information and communication necessary to support each action.

1) User enters website
   a) Need a webserver with URL
2) Once at the main page of the website the user will have the options to:
   a) Create a new account. If the user selects to create a new account they will be taken to the account creation page.
      - On the new account creation page the user will be asked for a username and password to create a new account. The data will be sent to the server which will query the database to determine if the username is unique. If it is a unique username and valid password, a new account will be created.
   b) Log in.
      - The user will enter their username and password and this information will be communicated to the server. The server will check if the username and password exist in the database and, if so, the database will return the list of notebook IDs shared with that user. The server will send this information back to the front end which will log the user in.
      - If the database does not have that username and password, it will return an error to the server which will communicate that to the front end interface. The front end will communicate to the user that the username and password combination was not found.
3) Once the user has logged in, either as a new user or an existing user:
   a) A list of notebooks shared with the user will be shown as a list of names. From the step above, when the user logs in, a list of notebook IDs shared with them will be transmitted to the server. The server can then request the name of each notebook ID from the database and transmit this back to the front end to be displayed to the user.
   b) If there are no notebooks currently shared with the user or they would like to create a new note, the user can create a new notebook. When the user selects to create a new notebook they will be prompted to enter a notebook name. Once entered, the notebook name will be communicated to the server which will query the database for the next available notebook ID. Each notebook ID must be unique in the database as a new table will be created for each notebook, based on the unique notebook ID. The database will return to the server the next available notebook ID and the server will use that ID to create a new notebook table in the database with the user specified note name.

4) When either a new notebook has been created or an existing notebook has been selected a text box will be opened in the front end which will be used to display and modify the text of the note. The user will also have the option to share the note currently opened with another existing username.
   a) If the user selects an existing note name to edit:
      i) The note ID corresponding to that note name will be communicated to the server. The server will check if that notebook ID is currently being edited on by another user with whom note is shared.
         (1) When a notebook is currently being edited by any user, a variable with server wide scope will be created relating to that note ID. This variable will keep track of the current state of the note on the server, ideally in RAM memory, which will be much faster than using disk storage. The server can check if another user is currently editing a note by checking if a variable corresponding to the note ID has been created. The server will also create a list of all users currently connected and working on that note.
      ii) If the server determines the note ID is being edited by another user, the server will communicate the current state of the note to the front end by reading the server scoped variable corresponding to the note ID described above. The front end HTML text box will be loaded based with the current note state.
      iii) If the server determines the note ID is not currently being edited by another user, the server will request the last saved state of the note from the database. This will be referred to as the HEAD state. The database will respond with the HEAD state of that note and the server will return this to the front end as well as create appropriate variables corresponding to the note ID with server wide scope. These variables will be used in case a new user wants to edit the same note. The HTML text box will then be loaded based on the last saved (HEAD) state of the note in the database.
   b) If the user chose to create a new note in the last step there will be no HEAD state so the HTML test box will be loaded with no text. The server will create appropriate variables corresponding to the note ID with server wide scope. It will also create a list of all users currently connected and working on that note.
   c) If the user selects to share the currently opened notebook with another user, they will have the option to enter a username with whom they would like to share the note. Once a username is entered, the front end will communicate both the username and note ID to the server. The server will communicate to the database that the note ID should be shared with that username. The database will add that note ID to the username, if valid, or return an error if no username matches the one entered. If this is the case, the server should communicate this back to the front end to provide feedback to the user. If successful, the note ID that was shared will populate for the user it was shared with on the next login.
5) When the text box is loaded with the up to date note state based on the user selection in the step above, the user may modify the note utilizing the HTML text box on the front end. When the user modifies a note:
   a) The front end will, on a timed basis, compare the previous state of the note to the current state and produce an operational transformation describing the edit that was completed by

the user on the text. It will save that transformation along with a value to identify the relative time that operation took place. It will then send that operational transformation and relative time to the server.

b) The server will receive that operational transformation and relative time from all users currently editing the same shared note. Based on this data it will build a consensus state of the note with all user edits taken into account.

c) The server will store this consensus state as well as a list of the previous operational transformations in variables with server wide scope so these variables can be shared with all users editing the shared note as described in the sections above.

d) The server will, as edits are received, send the consensus state of the note back to the front end user interface.

e) The server will periodically send “checkpoint” transformations to the database. These checkpoint transformations are operation transformation data comparing the last check pointed state to the current state. This checkpoint information can be used to rebuild the last checkpointed state of the note in case of a server or database failure assuming the failure is ultimately recoverable.

f) The front end will take the consensus state data sent from the server and display the consensus state of the note to the user in the HTML text box.

g) The user can continue to edit the note and this cycle starts over until completion.

6) When a user has completed editing a note and either logs out or exits the web page the server will be aware that the user has exited the application.

a) If there are currently other users editing the same note, the server wide variables will be kept alive and the user will be removed from the list of users currently working on that note.

b) If there are no other users currently working on that note, the server will:

i) Save the last determined consensus state of the note into the database as the new HEAD state for persistent storage until the note is edited again.

ii) Delete all server wide variables relating to that note ID.

7) When a user logs back in this process is repeated.

Figure 2 graphically represents of the workflow described in detail above.

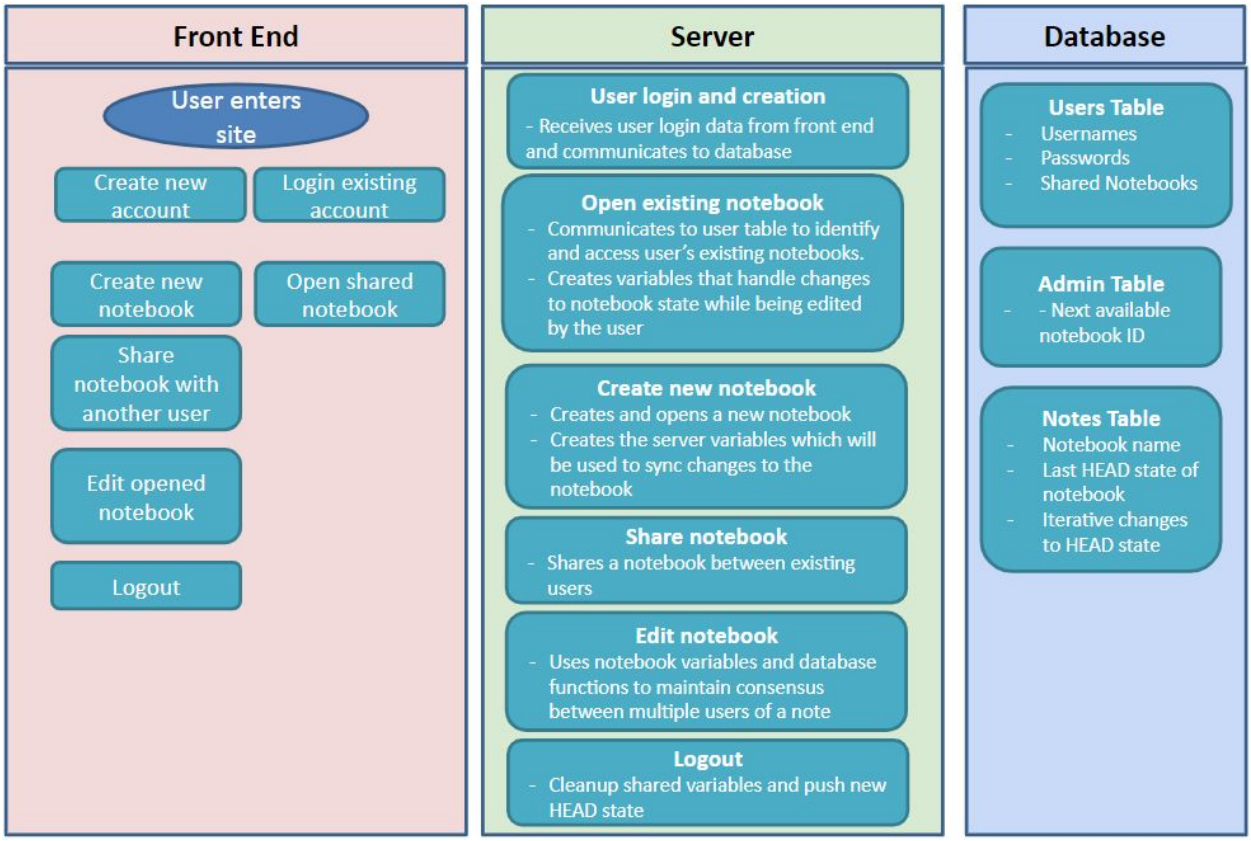

Figure 2: Application Workflow Diagram

Technology Stack in Detail

Front End

The application uses a simple front end that primarily serves as a demonstration of the capabilities of the application's implementation of the Operational Transformation algorithm and account management.

As discussed in the program architecture section, the front end uses a combination of HTML, CSS, Javascript (JS), Jquery and Ajax. Hypertext Markup Language is the standard markup language for creating web pages and web applications. Cascading Style Sheets(CSS) and JS are used to make the interface more interactive and user friendly. The Jquery Library in Javascript is utilized to simplify HTML DOM tree traversal and manipulation, as well as event handling, CSS animation, and Ajax. With Ajax, web applications can send and retrieve data from a server asynchronously without interfering with the display and behavior of the existing page. We have also utilized Bootstrap CSS Framework in our application for responsiveness and for JS based design templates.

Below is are descriptions of the front end pages available to the user.

Account Login Page

The account login page allows the user to login to an existing account. If the user does not have an account, they are provided a link to the account registration page. On submission of a valid username and password combination, the user is taken to the Note Selection Page. The submission of the password calls the database's logIn function.

Account Registration Page

The account registration page allows a new user to sign up for a password protected account. The database's insertUser function is called on clicking the Register button.

Note Selection Page

The note selection page displays the notes available for a given user account and provides the ability to create a new note. The list of notes is generated by a call to the databases' getNoteIDs function. If the user chooses to add a new note by adding a new note name, the databases' createNotebook function is called. Both opening an existing and creating a new notebook take the user to the Note Page.

Note Page

The note page displays all information for the current note and handles the sending of edits to the server and the incorporation of edits from the server into the note. The page uses the socket.io library to handle incoming events and send scheduled outgoing messages. The messages that note.html awaits and sends to the server are detailed below.

Messages Between the Note Page and Server

Initial Page Content

This message is an incoming event from the server contains the initial document contents and metadata. It uses the following socket.io message:

paddetails

Message Contents:

- Content - the current text of the document
- ClientID - the ID number the client will use for the remainder of the session
- ServerPosition - the index in the server's revision history

Events Triggered:

- This triggers the scheduling of the SendText Function, which repeats every 500ms while the document is open

Getting New Changesets from the Server

This is an incoming event that triggers the incorporation of new edits from the server into the local client document. It using the following socket.io message:

changesetFromServer

Message Contents:

- serverPos - the new top index in the server's revision history
- Content - the edits that must be incorporated into the clients state

Events Triggered:

- The doGetCaretPosition and setCaretPosition functions, which identify the current caret position and update it to account for the new edits, so that the user experiences the update fluidly
- The transformAndGetViewChangeset function, which transforms the edit to be relative to clients local state
- The merge function, which merges the changes into the local view

Submitting Edits from the Client to the Server

This is an outgoing event that sends local edits to the server. It uses the following socket.io message.

changesetFromClient

Message Contents:

- serverState - the most recent revision history position in the server that the client is aware of
- Content - the edits that must be incorporated into the clients state
- ClientNum - the clients ID number

Callback Confirmation

- The client awaits a callback confirmation that contains the new server position on incorporation of the submitted edit

Server

As mentioned in the program architecture, we decided to utilize Node.js for server development. While we had experience in PHP, Node.js seemed like a better choice primarily due to the real time, two way, communication available through the socket.io library discussed below. Node.js applications are written in JavaScript, which we were familiar with, and are used to transfer data to and from the front end. Node.js also provides a rich library of various JavaScript modules which simplifies development.

The server for a real-time collaborative editor is expected to take the following responsibilities:

1. Send most updated content of a notepad to the client
2. Send updated content for writing into DB for permanent storage
3. Maintain consistency across all users who are working on shared notepad content

These responsibilities require server to:

1. Send messages to and from the client and database.

2. Store edits to the notepad content in memory of the server.

For these requirements the server utilizes two main concepts:

1. Socket.io, library for real time, bidirectional communication between client and server
2. Redis an in-memory data structure

Below is a discussion of the main concepts and libraries utilized on the server.

Socket.io

Our application uses socket connections for sending data to and from the client and server. For creating and maintaining these socket connections it uses socket.io, which is a popular open source library. Socket.io enables real-time, bidirectional and event-based communication between the browser and the server. Sockets are typically used in applications which need instant, real time, messaging similar to the needs of the collaborative editor we have developed.

Socket.io consists of:

- a Node.js server
- a Javascript client library for the browser (which can be also run from Node.js)

Sequence of events:

1. When a client opens a notepad it sends socket connection request message to server with notepad Id.
2. On receiving the request, server creates a socket connection between the client and server
3. Once the connection is created, the server helps the client to join a room with corresponding to the notepad Id.

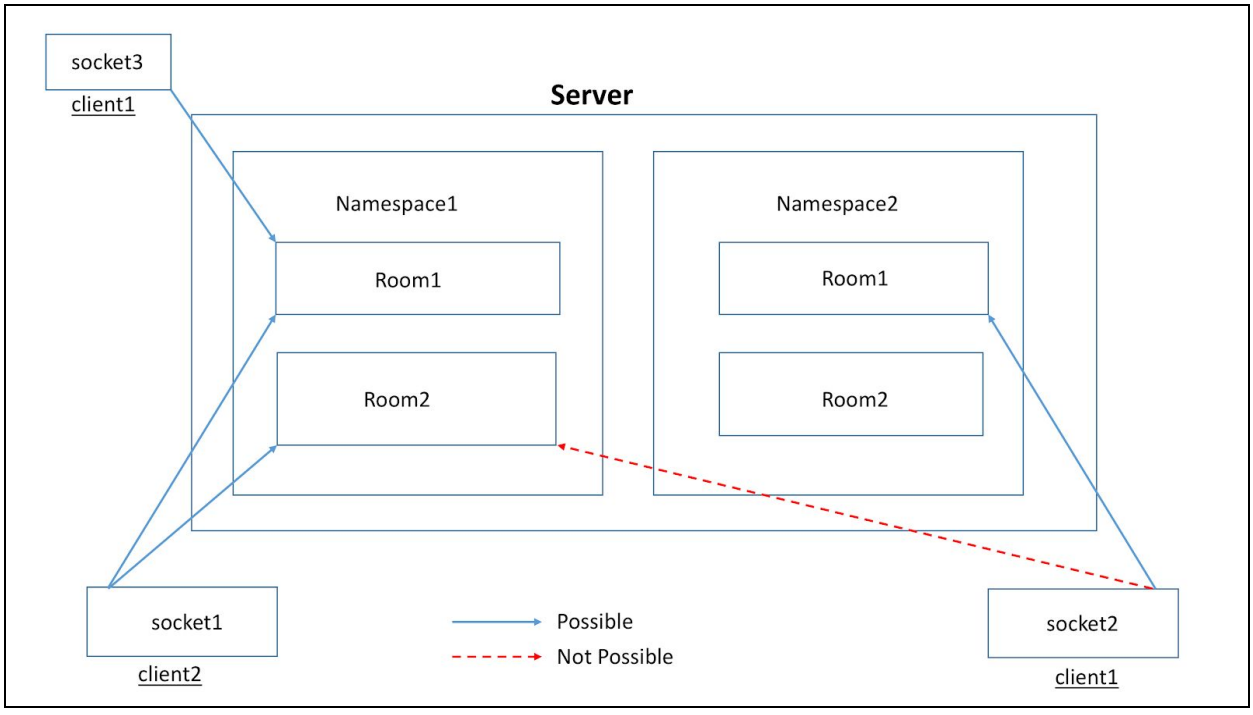

Figure 3: Server Namespace and Room Diagram

In Figure 3, you can see Namespaces and rooms in sockets. A namespace is a concept used in sockets to provide a high level separation layer between sockets. A socket can be a part of one namespace only. By default all the sockets connect to default namespace ('/') if a custom namespace is not provided.

Rooms can be created within a namespace, and a socket can be the part of multiple rooms. Each socket in socket.io is identified by a unique identifier SocketID. When a client opens a notepad, a message is sent to server with the notepadID. This notepadID is the room number used by the server and allows the client to join the room.

When all the clients leave a room, the socket connection is closed and the document state is written to the database.

Redis

Redis is an open source, in-memory data store, used as a database, cache and message broker. It supports data structures such as strings, hashes, lists, sets, sorted sets with range queries, bitmaps, hyperloglogs, geospatial indexes with radius queries and streams. Redis has built-in replication, Lua scripting, LRU eviction, transactions and different levels of on-disk persistence, and provides high availability via Redis Sentinel and automatic partitioning with Redis Cluster.

In our application, after a time interval of 500ms, the client collects all the changes made on a notepad and creates a changeset. It sends this changeset to the server via the socket.io connection. For maintaining consistency between clients of the same notepad the application stores the changesets in server memory. Redis is used to save these revisions and other metadata in memory.

Redis maintains a key-value pair to store data, where the NotepadID is the key and the revisions and metadata are the values.

When a specific user, for example user1, opens a notepad, Redis is used to save the content details of that particular notepad. When the next user, user2, opens the same notepad, the content of that notepad is loaded to user2 with latest the changes from user1. This avoids the cost of multiple reads from database to get the latest notepad details. When all users have left a room, ie the notepad on which they were working on, the data saved in Redis for that NotepadID is deleted.

Express Nodejs

Express is a minimal and flexible Node.js web application framework that provides a robust set of features to develop web and mobile applications. It facilitates the rapid development of Node based Web applications.

Following are some of the core features:

1) Allows creation of middleware to respond to HTTP Requests
2) Defines a routing table which is used to perform different actions based on HTTP Method and URL.
3) Allows dynamic rendering of HTML Pages based on the arguments passed to templates.

Body-parser

Body-parser is Node.js body parsing middleware that parses incoming request bodies prior to application handlers, available under the 'req.body' property. This handles JSON, Raw, Text and URL encoded form data.

Express-session

This is a Node.js module available through the npm registry. It helps create session middleware. To store or access session data, use the request property 'req.session'.

Mongodb

Used for creating mongodb middleware and connecting to Mongo DB and database functions described in the database section.

Database

Means of persistent storage was a required element of our project as our application required that a note state be saved from one note editing session to the next. A note editor would not be very useful if that note was deleted after a user finished editing it.

There are a few ways we could have achieved persistent storage but we decided to use a database for the following reasons:

- Data consistency and integrity
- Ability to quickly and easily query data
- Scalable data storage
- Fast data access
- Integration with web server

Once we had decided on using a database we needed to determine the database model and the specific implementation of that model we would utilize. We evaluated using:

- SQL or Relational database models: relational databases utilize a more traditional database structure where keys are used to relate data in one table to data in another table. Data types are typically pre-defined into a schema and need to adhere to a more rigid structure. Our team had experience with MySQL, a popular open-source database implementation of a relational database model. We considered using MySQL but

ultimately decided the NoSQL database model was more advantageous for our application as discussed below.

- NoSQL database models: NoSQL or a distributed database model offered some advantages over traditional relational databases. In our case we felt that a NoSQL model offered the following benefits:
    - While SQL databases rely on vertical scaling, increasing processing power on a single server or node to scale the database to handle more data, NoSQL databases are distributed and can horizontally scale by increasing the number of servers or nodes in the resource pool. For our application it would be important to easily scale the database as our user and note count grow.
    - SQL databases have a predefined schema which data must adhere to whereas NoSQL is document based and has no such predefined schema. While we may have been able to adopt our application to a more rigid schema view of the data, this would not be ideal. We knew for a given note we would need to store the name of the note, checkpoint data and HEAD data as described in the workflow description. While we could have split this data up into separate tables and linked them with keys, as would have been required by an SQL database, this was not as logical as a single note “collection” in a NoSQL database storing all relevant information for that note.

Relating to our application, based on the two main advantages of NoSQL databases described above, we decided to utilize a NoSQL database model. There are many different implementations of this database model such as BigTable, Cassandra, HBase, CouchDb and MongoDB, however, we ultimately selected MongoDB as it was supported in Cloud9, our development server.

In analyzing the workflow we determined the database needed to accomplish the following responsibilities:

- Store data persistently on disk
- Keep track of User data
- Keep track of Note data
- Keep track of the next available note ID

Database Structure

Based on these responsibilities the following describes the high level structure of the NoSQL database into collections or tables.

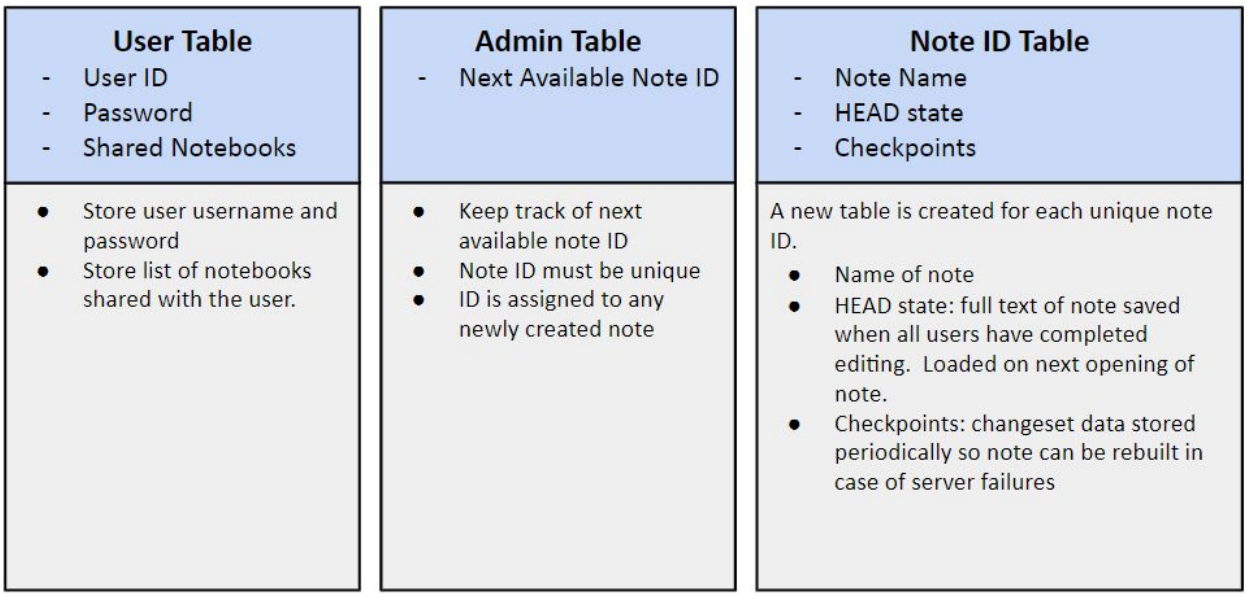

Figure 4: Database Tables and Contents

Database functions

Once we defined the MongoDB database structure, we needed to provide a list of functions to easily allow communication between the database and the server. This communication facilitated the workflow described in the workflow section. Below is a description of the database functions that are exposed to the server to realize the steps in the workflow.

Finding Notebook Names

When a user logs in, a list of notebook IDs that are shared with the user are returned. The user can provide the notebook a name when it is created and the name is what will be shown to the user(s) on subsequent logins. To get the notebook name from a notebook ID use :

findNoteName(notebookID, callback)

notebookID is a string type, not an integer

Callback will return one of the following:

- Success:<Notebook Name>
  A string of containing the name of the notebook. For example, if note ID 2 is named "Test Note", the callback will respond:
  Success:Test Note
- Error
  An error occurred. Check console log output on server.

Creating a new notebook

The user has the option to create a new notebook if they would like. In this case the user can select to create a new notebook from the front end HTML in which case this is communicated to the server which uses the following function to create a new notebook in the database :

createNotebook(noteName, callback)

The noteName is the notebook name the user selects for that notebook. That name will be displayed on future visits.
Callback will return one of the following:

- Success:<Notebook ID>
  If successfully created, a success will be returned from the database with the noteID attached for use by the server. If the notebook is successfully created the server should add the notebook ID to the username described in the Add Shared Notebook to Username below.
- Error
  An error occurred. Check console log output on server.

Add shared notebook to username
When a user successfully creates a new notebook or shares an existing notebook with another user, that notebook should be added to the username so it is shared with the creator and all designated users on future logins :
addSharedNotebook(username, noteID, callback)
The username is the username of the user you want to share the notebook (noteID) with.
Callback will return one of the following:

- Success
  If the notebook ID has been successfully added to the username. This notebook ID will be returned in the LogIn callback as described in the Log In section below.
- Error
  An error occurred. Check console log output on server.

Log In
User enters in a username and password in the UI text box in the HTML front end and this is transmitted to the server through socket connection.
Server uses database function:
logIn(username, password, callback)
Callback will return one of the following:

- Success: <list of shared notebook IDs>
  List of shared notebook IDs will be integer values separated by commas. Eg. if a user has two shared notebooks of ID 2 and 21 it will respond the following message
  Success:2,21
- NoMatch
  No username and password match in DB
- Error
  An error occurred. Check console log output on server.

New Account Creation
User enters in a username and password in the UI text box in the HTML front end and this is transmitted to the server through socket connection.
Server uses database function:

insertUser(username, password, callback)
Callback will return one of the following:

- Success
  proposed username successfully submitted
- Duplicate
  A duplicate username was already created
- Error
  An error occurred. Check console log output on server.

Pull Head Note State

When all users sharing a notebook exit the system, the full text of that note will be saved in the database as the HEAD state. If a user logs back in to work on the notebook at a later time and no other users are currently editing that note, the latest HEAD state of the notebook will be pulled from the database and returned to the server as a starting point for that note.
Server uses database function:
pullHead(noteID, callback)
Callback will return one of the following:

- Success:<Note Text>
  The last inserted HEAD state of the notebook will be returned as the notes full text
- Error
  An error occurred. Check console log output on server.

Pushing HEAD state

When the last user currently editing a note has completed, the final text of the note will be pushed to the database as the HEAD and will be used on future loads of that note.
Server uses database function:
pushHEAD(noteID, HeadText, callback)
HeadText will be the full text of the note.
Callback will return one of the following:

- Success
  Successfully inserted the HEAD
- Error
  An error occurred. Check console log output on server.

We successfully deployed MongoDB onto our development server and using the database structure and functions described above, communicated with the server as described in the workflow section. Full implementation of the functions described above can be found in the source code.

Development Environment

We decided to use the online development environment, Cloud9. When working as a team of three people on the project we had the following desires in a development environment:

- Easily collaborate on code
- Easily deploy code to a running server to test application

While there were other options to achieve these requirements, such as collaboration using Git and code deploying with Heroku, Cloud9 integrated the above features and was free to use. We also had previous experience with Cloud9.

Version Control

We used Git and Bitbucket, a free Git repository for version control.

Instructions for running the project

To run the project:

1) Navigate to https://c9.io/login in your web browser
2) Login to Cloud9 with the following credentials:
   Username: jkcaldwe
   Password: throwaway123.
3) Open the 'cis579_finalproject' workspace
4) Using a 'bash' terminal window near the bottom of the screen type './mongod &' and then press enter to start the mongoDB database.
5) In another bash terminal, type 'redis-server' to start Redis Server
6) Navigate to the 'server.js' file in the Workspace on the left of the page and double click it.
7) Click the 'Run' button at the top of the screen. A new terminal window should open at the bottom and report the code is running on a specific IP address. You can click on that or copy it into the browser IP address line.

8) This will start at the main 'server.js'. You may need to logout and log back in. You may create a username and password and then create a shared notebook. If instead you would like to test with usernames we have already setup whom share a notebook you can use the following:
   a) Username: mark, Password: pass1
   b) Username: lisa, Password: test
9) Once testing has been completed, you may stop the server by pressing the 'Stop' button on the 'server.js' terminal window.
10) After the server has been stopped, you may stop the database from running by navigating back to the terminal and typing './mongod --shutdown'
11) Shutdown Redis Server by typing 'redis-cli' in a bash terminal. Type 'shutdown' and then 'quit'

Maintaining Consensus

The primary technical challenge of the project was the implementation of an algorithm that maintains consensus between the clients and the server when collaborating on a document. To meet the acceptance criteria of the project, the application must support:

- Clients ability to edit their local copy of a document at all times. At no point should they wait or be blocked while a message from the server is being incorporated
- Distribution of a clients edits to all other clients in real time if the network is available
- Maintaining local edits until the connection is restored or times out if the network is not available
- Incorporation of external edits upon receipt, while maintaining the integrity and consistency of those edits across all clients and the server

Numerous models of consistency exist, not all of which guarantee convergence between copies of a document in all cases. Based on review of these models and review of the EtherPad application strategy, this application uses the Causality Admissibility(CA) model. The CA model has two properties:

- Causality: defines requirements on the order of operations for documents, classifying edits as either causal or concurrent. Causal edits are dependent on an order, and must be implemented in that order to guarantee consistency. Concurrent edits are independent of order, and can be implemented in any order on any client. [1]
- Admissibility: Admissibility applies a total order to the edits as they are submitted to some log, in this case, the log being maintained by the server. Admissibility states that all edits can be implemented only their execution state. This requires that clients receive and incorporate edits as defined by their total ordering. [2]

Operational Transformation

This project uses Operational Transformation as the foundation for its approach to maintaining consensus. Operational transformation supports concurrency control between copies of a document on multiple client machines. The messages passed between clients contain the instructions to alter the document from the most recent state available to the sender, in this application, the sender is the client to the server, or the server to the clients. Other applications of operational transformation enable the use of a fully distributed document state, and though the implementation and order of operations differ from the scheme used here, the general principles remain the same. [3]

The figure below provides a high level overview of operational transformation. In this case, both User 1 and User 2 start with the same word, ‘HAT’. User 1 modifies the word by inserting ‘C’ at position 0, creating the word ‘CHAT’, and User 2 makes a concurrent edit, deleting ‘H’ in the 0 position, creating the word chat. Both users send their respective edits to each other, but now

the local state of the text has changed, and the operation must be transformed. To account for its insertion of ‘C’ at position 0, User 1 transforms User 2’s edit to delete the letter at the 1 position. User 2 does not transform User 1’s edit, as the previous delete operation did not cause a shift in the 0 position. Applying operational transformation, both users end with the word ‘CAT’.

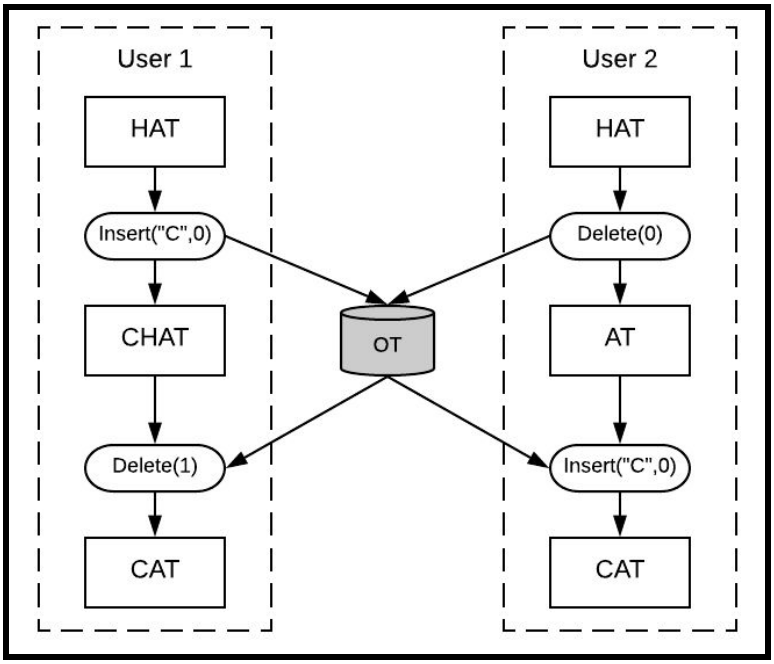

Figure 5: Simple Operational Transformation Illustration

Changesets

The “unit” of document change in VaporPad is a simplified version of the EtherPad changeset. It allows description of the changes to a document without transferring the full document text. The diagram below illustrates the three components of a sample changeset.[4]

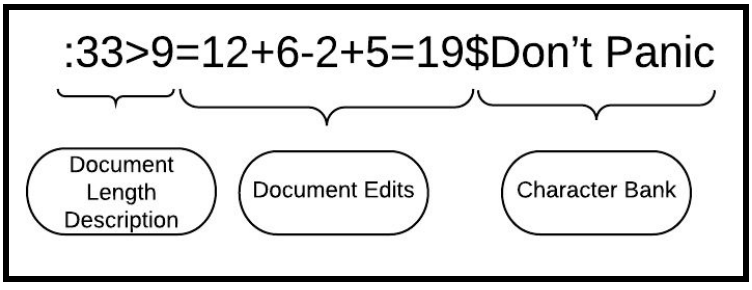

Figure 6: Changeset component illustration

- **Document Length Description:** The first character of the Document Length description is signified by a ‘:’, and is followed by an integer describing the initial length of the document prior to any edits, in this case 33. The integer is followed by a ‘>’ or ‘<’, indicating that the changeset adds or removes characters. That operator is followed by

an integer denoting the number of characters added or removed, in this case, 9 additoinal characters.[4]

- **Document Edits:** The document edits contain the operations applied to the document. There are three potential operators: '=' directs the retention of characters, '+' the insertion of characters drawn in order from the character bank, and '-' the deletion of characters. In this example, 12 characters are retained, followed by an insertion of 6 characters drawn in order from the char bank ("Don't "), 2 characters are deleted, 5 characters are inserted ("Panic"), and finally the 19 remaining characters are retained.[4]
- **Character Bank:** This is denoted by a '$', and represents all characters that are inserted into the text. Discrete insertion operations simply append the inserted characters together in the charbank.[4]

Changeset Transformation

Given the requirement that the application guarantee convergence of edits despite clients whose awareness of edits may differ due to unreliable network connections, changesets must be transformed to account for previous edits as they are incorporated into the server or into the client state. The function used for this is called the Following Function, or f(A,B), which transforms changeset B to account for changeset A, producing new changeset B'. This relationship is described in the formula below, given some initial shared text X.[5]

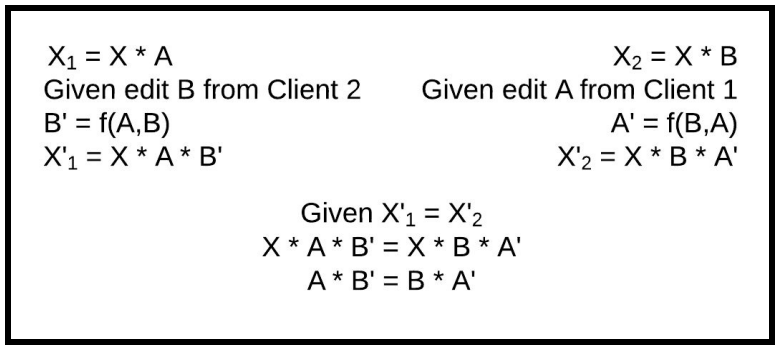

Figure 7: Following function formula demonstration

For the Following Function to operate correctly to produce the changeset B' = f(A,B), the principals below must be followed in order.

1. Character additions from changeset B remain additions in changeset B'
2. Character additions from changeset A become retentions in changeset B'
3. Characters retained in both changeset A and changeset B are retained in changeset B'
4. Characters deleted in changeset A are 'skipped' in changeset B' (if those same characters are retentions or deletions in changeset B, those characters are no longer accounted for in B', as they have been removed by A)
5. Character deleted in changeset B remain deleted in changeset B'

Client Document State

The client maintains three separate components that represent the full document state. Those states and the overview of how they are processed and how they transition between components are illustrated below. This client document state is modeled on the EtherPad system.[5]

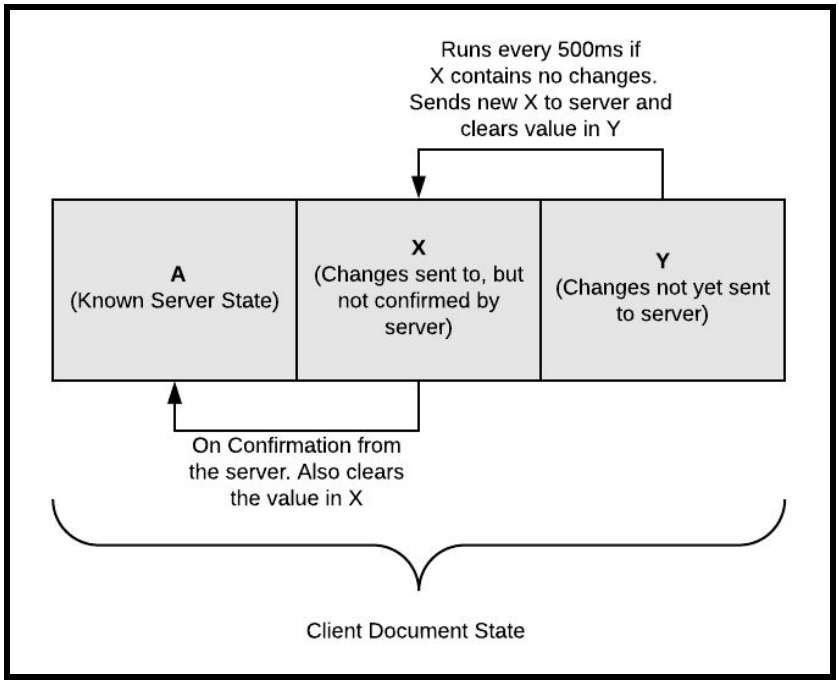

Figure 8: Client document state illustration

- **Known Server State** - This contains the most up to date information the client has regarding the server state. This is stored as the text of the server state.
- **X - Uncommitted Changes** - These are changes that have been submitted to the server, but have yet to be confirmed for incorporation. This is stored as a changeset.
- **Y - Unsent Changes** - These are changes that the client has created, but have not been sent to the server. This is updated every 500ms, and is the difference between the merger of A and X and the current screen view. If the X value contains no changes from A, then Y is sent to X, Y is cleared, and the new X is sent to the server.

Sending Changesets to the Server

As discussed, every 500ms the client checks if new information is available to send to the server. If a new changeset is available, the message is sent to and incorporated into the server prior to being sent to other clients. The diagram below illustrates a simplified version of the incorporation process.

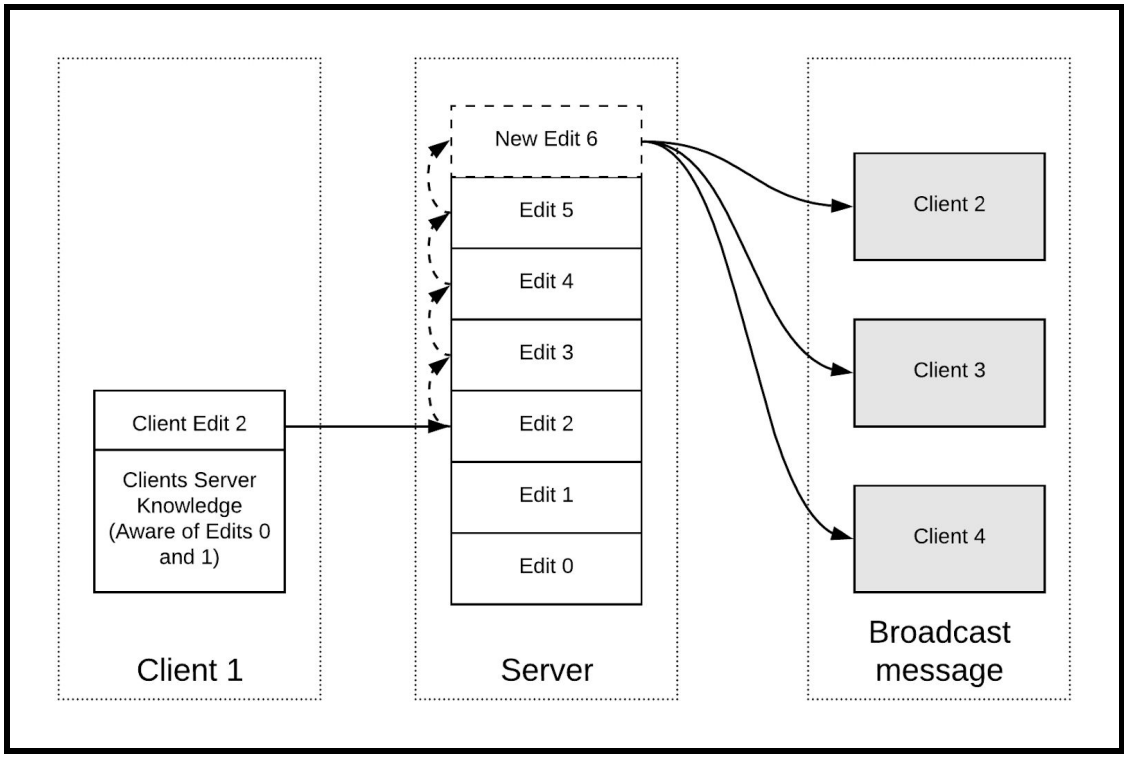

Figure 9: Server changeset incorporation and broadcast

- **Client generates a new changeset to send** - The client tracks its known the server revision number, and sends alongside its changeset to the server
- **Server adapts changeset on receipt -** The server checks the revision that the client is currently aware against the total revisions the server has commited. In the example above, since Edit 2 is taken on the server, the server calculates f = f(Edit 2, Client Edit); f' = (Edit 3, f); f'' = f(Edit 4, f'); etc; until reaching an open position.
- **Broadcasting changesets** - On incorporation of a new changeset, the server broadcasts all outstanding changesets to the remaining clients. It tracks the current position of each client, and sends them the combination of all changesets beyond the client's current state through the present server state. This still meets the total order requirement, but reduces network congestion by sending one changeset only, rather than multiple if the client is several revisions behind the server.

Incorporating Changesets from the Server

On receipt of a new changeset from the server, the clients local state must be transformed to account for the new server state. Since any information in X or Y have not yet been committed to the server, any message B coming from the server comes earlier in the documents edit order. Those states must be transformed as illustrated below. Both X and Y are transformed into X' and Y' to account for changeset B, while the server state simply merges B to produce A'.[5]

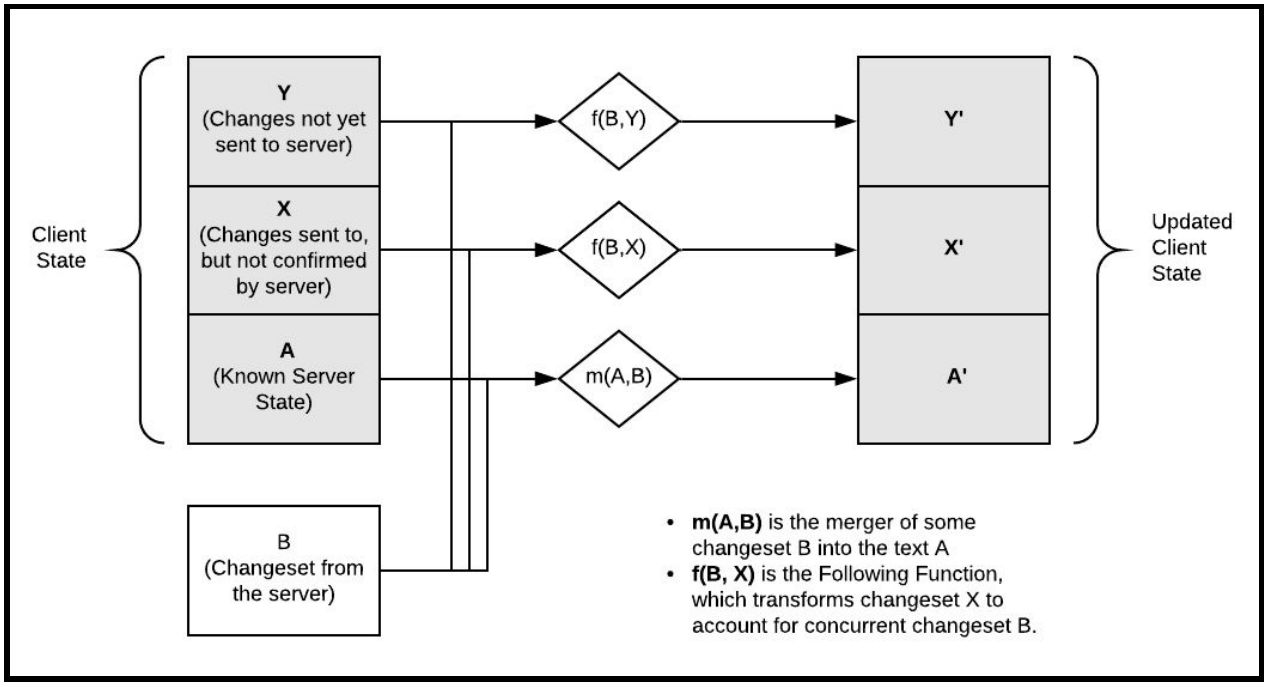

Figure 10: Client incorporation of changeset from server

Updating the Client's View

To preserve the clients caret position in the text, the updated view of the text cannot simply be a the calculation of A' * X' * Y', instead, a new changeset D is required, which is the changeset that would transform the current view, V, into A' * X' * Y'. By applying D to V, the current caret position can be tracked and updated based on D.[5] The derivation of D is shown below.

$$
\begin{aligned}
A'*X'*Y' &= A*B*f(B,X)*f(f(X,B),Y) \\
&= A*X*f(X,B)*f(f(X,B),Y) \quad \textit{because } X*f(X,B) = B*f(B,X) \\
&= A*X*Y*f(Y,f(X,B)) \\
&= A*X*Y*D \\
&= V*D
\end{aligned}
$$

Figure 11: Derivation of D, the view update changeset

Future Improvements

While the project addresses the acceptance criteria laid out by the project team, opportunities for improvement remain. Potential improvements include:

- Support for text formatting
- Image and table support
- Improved generation of changesets. The current algorithm scans the full text for changes, an improved algorithm could track cursor position and keystrokes for more efficient tracking.
- Improved text data structure. By storing the text as a tree of sections of the document, edits could be made only to the affected portions of the document without scanning the full text. This would improve handling of large documents.
- Compression of integers in changeset messages into base 16 or base 32.
- Display of other client's cursor positions
- Display of all the users currently working on the notepad.
- Display of all the users,with whom notepad has been shared.
- Improving User Interface of the application.
- Authenticating users by sending authentication link to their email while registering and captcha before login.
- Next note ID handling: next note ID in the database doesn't currently account for deleted notes.
- Note table: The note table in the database currently stores all previous states of a note no matter how far back in time. We could remove old note edits to save storage space.

References

1. Ellis, Clarence A., and Simon J. Gibbs. "Concurrency control in groupware systems." *Acm Sigmod Record*. Vol. 18. No. 2. ACM, 1989.
2. Li, Rui & Li, Du. (2005). Commutativity-based concurrency control in groupware. 2005. 10 pp.. 10.1109/COLCOM.2005.1651251.
3. Ellis, Clarence A., and C. Sun. "Operational transformation in real-time group editors: issues, algorithms, and achievements." *Proceedings of the 1998 ACM conference on Computer supported cooperative work*. 1998.
4. Ether. “Easysync Protocol.” *GitHub/ether/etherpad-lite/*. 2011. https://github.com/ether/etherpad-lite/blob/master/doc/easysync/easysync-notes.pdf
5. Ether. “Etherpad and EasySync Technical Manual.” *GitHub/ether/etherpad-lite/*. 2018. https://github.com/ether/etherpad-lite/blob/master/doc/easysync/easysync-full-description.pdf