Spring 2012

Online Collaborative Editor

Aditya Rao
San Jose State University

Online Collaborative Editor

A Writing Project

Presented to

The Faculty of the Department of Computer Science

San José State University

In Partial Fulfillment of the

Requirements for the

Degree Master of Computer Science

By

Aditya Rao

Spring 2012

SAN JOSÉ STATE UNIVERSITY

The Undersigned Writing Project Committee Approves the Writing Project Titled

Online Collaborative Editor

By

Aditya Rao

APPROVED FOR THE DEPARTMENT OF COMPUTER SCIENCE

_____

Dr. Robert Chun, Department of Computer Science

05/02/2012

_____

Dr. Chris Pollett, Department of Computer Science

05/02/2012

_____

Mr. Mayuresh Hajirnis, Software Engineer, Yahoo! Inc

05/02/2012

Abstract

"Online collaborative editor" is a node.js based browser application that provides real time collaborative editing of files and improves pair programming. Current real time editors fail to provide simultaneous viewing and editing of files within the server and results in a complex version controlling system. Such systems are also vulnerable to deadlocks and race conditions. This project provides a platform for real time collaborative editors, which can support simultaneous editing and viewing of files and handle concurrency problems by using locking mechanism. The experiment results showed that node.js platform provides good performance for collaborative editing.

Acknowledgements

I would like to thank my project advisor, Dr. Robert Chun, for providing his constant guidance; support and direction that helped me achieve my goals on this thesis project. I appreciate my committee members, Dr. Chris Pollett and Mr. Mayuresh Hajirnis for their valuable feedback and guidance.

I would also take this opportunity to thank my classmate Mr. Shailesh Sawant to guide me during the tricky phases of this project.

TABLE OF CONTENTS

LIST OF TABLES

TABLE OF FIGURES

1 INTRODUCTION

1.1 Problem Statement

It's been a common practice where team members working on a common project share their documents and text files electronically as an email attachment or storing the files on external devices (such as USB hard drive and flash-drive) and physically sending these devices to team members. When any member of the team updates the files then the entire process of distribution has to be repeated so that everyone has an updated copy. This process of distribution is slow and time consuming.

The primary goal of my project is to design a platform that is simple yet a practical solution that supports multi-user file editing over a distributed environment.
One way of solving the above problem is to create a web server that will maintain all the files being shared by group of users. These users will then use their client machines to request particular files from the web server. The web server will maintain files in its centralized repository along with the users information. When any client updates the shared file, the server synchronously updates the changed file to maintain a consistent state of the file.

1.2 What is collaborative Editor?

Online collaborative editing is a concept, where group of users geographically dispersed over a network are simultaneously working with the same set of files. As defined by Wilson (1991) [1] Computer Supported Cooperative Work is the way

people work in groups with enabling technologies of computer networking and associated software, hardware, services and techniques.

Developing a collaborative system involves people from geographically different locations working on a common project. Collaboration can be achieved in many ways but the simplest form of collaboration is through file editing and sharing. That is, files are retrieved from the web server, edited by a group of people and then saved on the web server in an asynchronous mode.

Computer software designers have constantly developed groupware (collaborative software) [2] that help users complete their projects collaboratively.

1.3 Design Overview

The project platform is designed in Client-Server styled architecture. The shared file is stored on a Back-End database that can be accessed by the server. Team members in the project query the server from client side. A key mechanism in this platform is the locking mechanism that will provide concurrency control. The different functionalities within this platform can be underlined as follows:

1.3.1 Editing functionalities:

Editing a file, with other users being able to view the file (and the changes) simultaneously. Lock or unlock the textarea block being edited to provide consistency control.

1.3.2 File system functionalities:

Create new files, open existing files and save modified files back to the server.

1.3.3 Account functionalities:

Secure Login (Username and Password), session management. If the client accidently gets disconnected from the server or closes his browsing session, the server will maintain his session until he reconnects or until the default time for reconnection expires.

2 Related work

Existing collaborative editors have also adopted the Client-Server architecture where the server holds a persisted copy of the shared document. Clients will have their own local copy of this shared document. Any changes to this document is synchronized to the server and then propagated to other clients by the server. Below I describe some collaborative editors with their restrictiveness on collaboration.

SASSE collaborative editor [3] uses the concept of shared workspace for every user working on the document concurrently. A unique color is assigned to every user of the document to represent changes in the document. This system however fails to provide any concurrency control.

DUPLEX distributed editor [4] splits the single document into individual modules maintained separately by different team members within the group. Kernel replicates updates to the server and reduces any communication overhead. This system failed to provide any simultaneous editing.

RCS [20] is a version control system where a user modifies the document by an external checkout step and finishes any updates by performing the check-in step. Multiple users can perform the same set of actions to reflect changes however exceptions are thrown if the user checks-in an update on an old copy of the document. The system uses locking mechanism to detect any editing conflicts and

provides diagnostic messages to the affected users. In this system users modify their local copy and save changes on central copy for consistency however the system fails to provide synchronous editing or viewing of a file.

Google Docs [18] provides capability to share documents collaboratively with different users. With Google docs we can share excel sheet documents with another user and collaboratively work to update any block within excel. However real time collaborative editing within the excel block is limited. User A can edit one block and user B can edit and see the changes only after user A has left that block. No real time editing content can be seen synchronously. Google doc also provides a document editor that displays real-time changes synchronously. This feature is advantageous where multiple users need to make changes simultaneously, however, it fails to provide any locking mechanism to the document where only a particular user needs to update the file. This project (inspired by Google Docs) provides both locking as well as synchronous editing functionalities and attempts to produce a simple and robust platform for real-time collaborative editing.

This project aims to reduce the traditional system limitations by providing a platform that enables us to have simultaneous read writes on a file and also provide effective consistency control on the file being edited.

3 Technology Stack

3.1 Node.js

Node.js [13] is a platform built on top of chrome's JavaScript runtime that is capable of building scalable and fast network applications. It uses an event driven and non-blocking I/O model that is lightweight and efficient. It's a technology ideal for real-time data-intensive applications that run across distributed devices. Node's provides memory efficiency under high loads for each connection and prevents deadlocks as no function of Node directly performs input/output operations.

3.1.1 Why node.js

Problem Statement:

Traditional web servers have always been thread-based models. We need to launch an Apache server to accept connections. The web server on receiving a connection will keep that connection open until it services the request for a page or it sends any information. All this time the server is blocking on that input/output operation.

In order to scale this type of web server additional copies of web servers are required. This is referred as 'thread-based' because every copy of the web server will require another operating system thread.

Solution:

In contrast to the above problem statement, Node.js is an event-based model where a web server accepts the request and then passes this request to handlers. It then continues to service the next request. Once the previous request is completed, it goes back to the process queue and on reaching the front of queue the results are returned to the requesting client.

This model is scalable and highly efficient as the web server always accepts requests and not blocking on any read/write operation. This is termed as 'event-driven' or 'non-blocking Input/Output'. The process can be given in following steps:

1. Web browser will make a request for "/MainPage.html" with the node.js server.
2. The node.js server will accept the browser's request and will call a function to retrieve that file from the disk.
3. While the node.js server is waiting for the file to be retrieved, it will service the next web request if any.
4. When the server retrieves this file from the disk, a callback function is added to the server's queue.
5. The node.js server will execute that callback which in this scenario will render "/MainPage.html" and then return the requested page back to the client's web browser.

This process may take milliseconds to service the request, but this would count when many requests are being processed.

3.2 Express framework

Express.js [14] is a high-class web development framework built on top of node.js that helps developers builds complex distributed web applications easily and efficiently. It extends the methods provided by standard node.js toolkit. This framework can be used along with connect framework to create http web server, sessions and cookie parser. An advantage of using such a framework is that it provides an efficient source code management. Files are managed in different modules where server files are placed at root level, view files are placed in view module and node specific modules and dependencies are placed in separate sections. Following figure shows the basic layout of express framework (opened with a text editor).

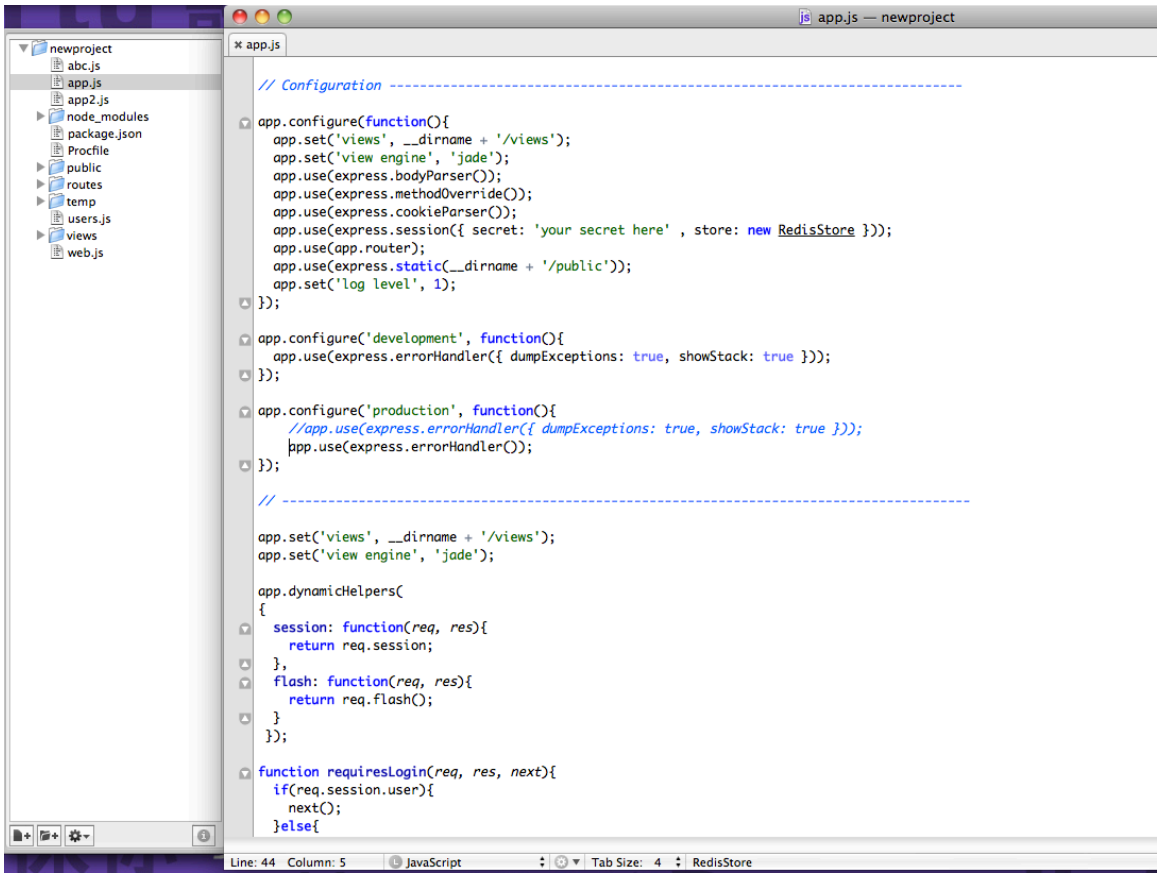

Figure 1: Expressjs framework

3.3 Redis Database

Redis [15] is open source key value pair storage. Since the keys within redis can contain Strings, lists, hashes, sets and ordered sets, its consider more of a data structure server. It supports Master-slave replication and so data from any redis server can be replicated to any number of slaves. Redis provides persistency of data by asynchronously transferring data from memory to disk at regular intervals. Redis also contains commands to lock any keys defined within its data structure. This project is using a redis client 'node-redis' to leverage all redis capabilities.

3.3.1 Why Redis?

Advantages:

- Redis provides fast key-value storage. It can perform a quick read and write operation on a database by using the GET, SET operations.
- Its collection types and the atomic operation on those types allow us to handle complex data scenarios.
- It can be used for data persistence.

Because of these advantages I chose redis to be my back-end database that can provide simple key-value operations.

3.3.2 Who uses Redis:

Craigslist, GitHub, guardian.co.uk, flickr by Yahoo and stackoverflow are some of the real world examples where redis is being used as a persistent storage.

3.4 Socket IO

Socket IO [16] is used to make real time applications on every browsers or mobile devices overcoming any differences between different transport mechanisms. This project uses Socket IO to create different client server connections.

3.5 jQuery framework

jQuery is a fast and efficient JavaScript library [17] that provides event handling, Ajax interaction and simplified HTML traversing for rapid web development. This project uses jQuery version 1.7.2.

4 Project Design

The main objective of this project is to design a browser-based editor that will be easy to use and help users share their work collaboratively over a geographically distributed network.

4.1 Back-End Server

Back-End of this project consists of two parts. One part is to serve regular http requests and other part to serve web socket requests. For serving http requests this project uses a node.js web framework called 'Express'. A simple http request server can be created as follows:

Figure 2: Creating a node.js app server

The above figure describes the process of creating http request server. We create a 'createServer' object and reference it using app. All corresponding calls to http server can be handled by using the reference variable app. When a client connects to the server, the requests are handled using the 'req' object and the response from server is initiated using the 'res' object. The server listens on a unique port '3000' for any connections.

The second part of the Back-End server is to provide WebSocket requests. WebSocket's provide us with methods to send client messages to the server efficiently using simple syntax. Socket IO is a WebSocket API [16]. Socket IO will decide based on feature detection if connection can be established with Ajax long polling, WebSocket or flash. Most real time applications that can be run on different browsers use Socket IO for client server communication.

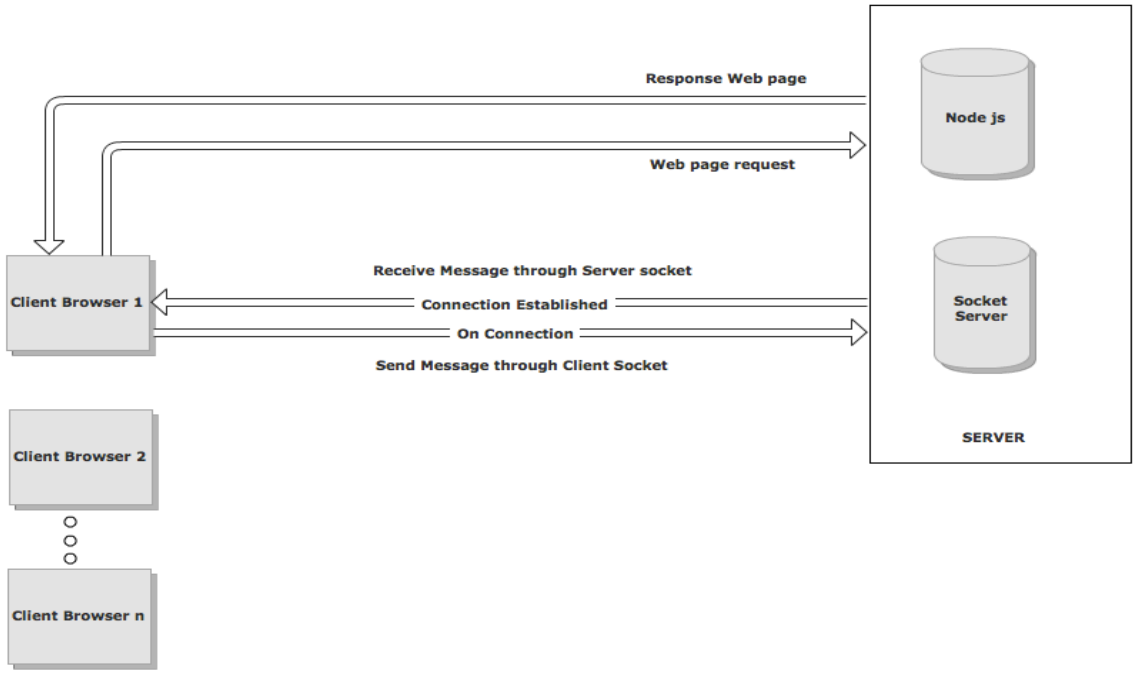

Figure 3: Establishing Client-Server connection

The figure above describes the Socket IO connectivity between client and server. Socket IO consists of two parts:

1. A server program that sends and receives data from client.

2. A client script that connects to the Socket IO server and then send and receive data.

Multiple client browsers can establish Socket IO connection and communicate with each other using the common Socket IO connection.

Client browser can request for webpages irrespective of using Socket IO, where the request response objects of node.js are used to send and receive data. This project establishes the connection with Socket IO at the same time when connection with

node js server is established through login. Only data send and receive is done through Socket IO and any web page requests are handled directly by node js server. Socket IO is attached to the node.js server, enhancing WebSocket capabilities to provide horizontal scalability, built-in multiplexing and automatic JSON encoding/decoding. This can be achieved using the code below:

```
var app = express.createServer();

var io = io.listen(app);

app.listen(3000);
```

Figure 4: node.js and Socket IO create server

4.2 Methods used by Socket IO:

4.2.1 Server Side:

Socket on connection: This script on the server listens for connections by clients and provides a handle that can be used for request response objects.

```
io.socket.on('connection', function(socket) {
        socket.emit('Welcome Client');
});
```

Figure 5: Socket IO on connection request

Socket on Disconnect: This script on the server is triggered when a client disconnects or closes his browser session. Any code related to disconnect event is placed within this section.

```
socket.on('disconnect', function() {
        io.sockets.emit('Client disconnected');
});
```

Figure 6: Socket IO on disconnect request

Socket on Message, join room, broadcast: Server side script contains a 'on message' method that accepts messages or data from client and then processes that message. Server script can also create multiple rooms and then add clients to them using the 'join' method. The server can then send messages only to clients within that room by using 'broadcast.to(room)' method. A broadcast in general would send messages to all clients connected to that socket session. The code below illustrates the desired functionality.

```
io.sockets.on('connection', function(socket) {
        socket.join('Room1');
        socket.broadcast.to('Room1').emit('Welcome to Room1');
        socket.broadcast('New Room created');
});
```

Figure 7: Socket IO broadcast message to clients

4.2.2 Client side

Socket on connect: The client side script establishes connection with the socket server. Client sends a connection request to the server listening on a particular port. The following code describes the connection procedure.

```
var socket = io.connect('http://localhost:3000');
socket.on('msg', function(data) {
        socket.emit(' Sending message');
});
```

Figure 8: Socket IO client Script accepting Server message

4.3 Back-End Database:

This project uses redis as the Back-End data store. Redis can be used for providing two important functionalities: redis as a persistent store and message delivery using redis publish/subscribe.

Redis as persistent store: This project uses redis as a persistent data store. Its key-value store provides fast in-memory access to data. It periodically writes the data to a disk. This mechanism is important to this project where integrity of data and faster access matters. Redis also supports different data types. At times the messages from chat sessions or files can be stored as lists within the redis data structure.

Message delivery using redis publish/subscribe: When the client edits any text within the editor, the data (text) needs to be propagated correctly to other clients. Redis publish/subscribe methods implement this functionality. Whenever clients are collaboratively using a single file, two redis client instances are created for them. One client is used for publishing the message and other client is used to listen for incoming messages from subscribed channels. So when one client writes (publishes) to the file, other clients are listening (subscribed) to that client. We need two redis-client instances per user because in redis when we subscribe to a channel we cannot run any other redis commands. So by using another instance of redis-client the user can run any other redis commands. A client can subscribe and

publish to multiple channels in redis. Following figure describes the architecture of redis publish/subscribe.

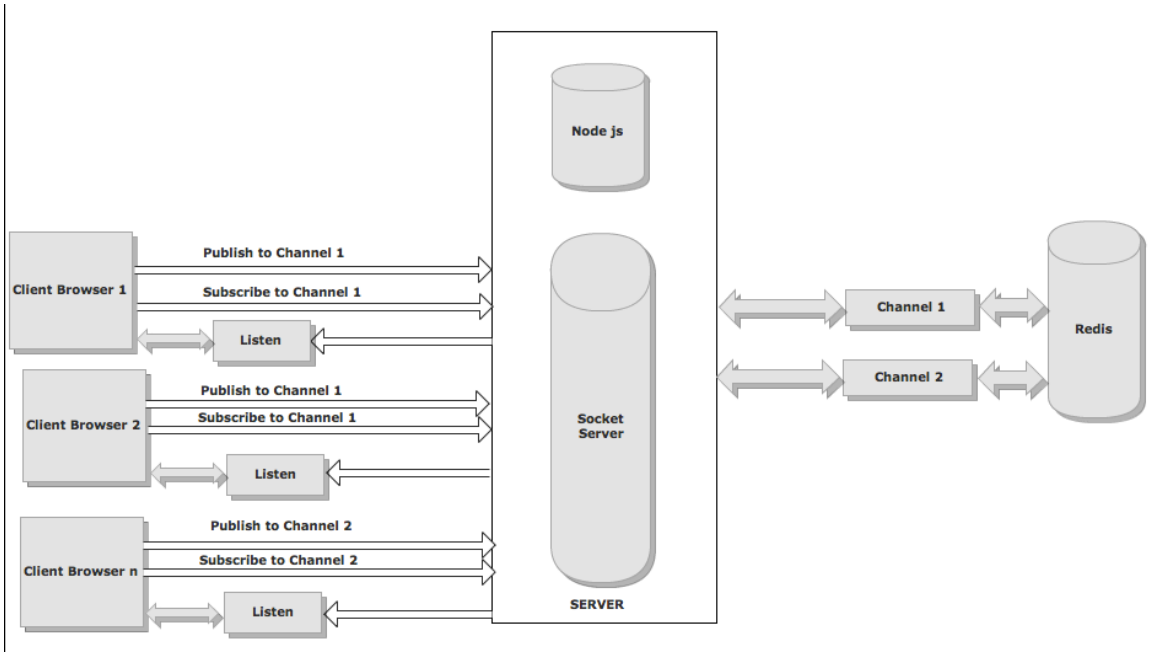

Figure 9: Redis Publish/Subscribe - Send/Receive updates

In above figure, Client 1 and Client 2 have two redis client instances out of which one redis-client is used for publishing to channel1 and other is used to subscribe to channel1. Similarly Client 2 subscribes and publishes to channel1. Any message published by Client 1 to channel1 will be sent to clients subscribed to channel1. When any client subscribes to a channel, a subsequent listener function within the subscribe method will listen to that channel and display the received message to the clients browser.

4.4 Session Management:

In this project, when a client wants to access the collaborative editor he has to login to the application. When a client has successfully logged in, a session is created for that client and stored within redis. This session will be persisted within redis until the client log's out. If the user is disconnected without logging out, his session will be maintained within the redis data store. This session can then be accessed for reconnection. If a client logs out of the application, then his session will be removed from redis store.

4.5 Account Management:

Every successfully logged in client is saved within an array list of users. This user list will be updated (increment or decrement) when any client log's in or log's out of the application.

4.6 Chat application:

This project also provides a chat application, where connected users (clients) can communicate. This application also displays list of users currently using the chat session. Every client is notified when any client joins the chat room or leaves the chat room. Subsequent users list with updated users are displayed. Following displays the chat UI.

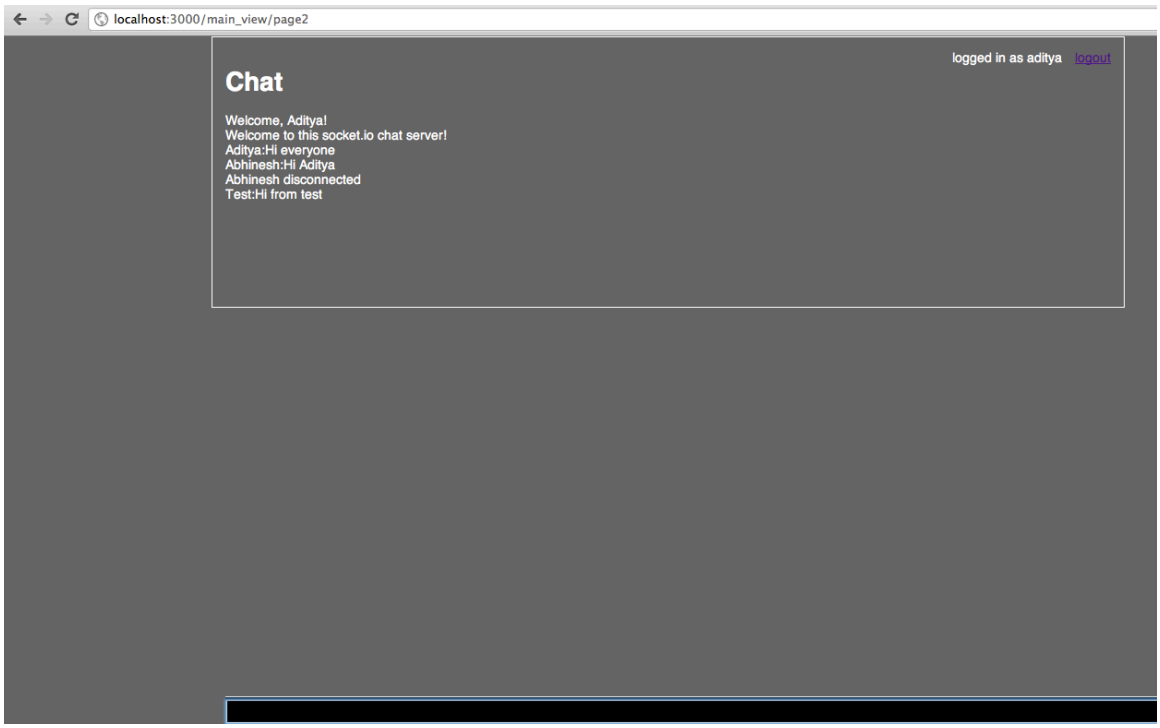

Figure 10: Chat room for connected clients

5 Project Flow

5.1 General Flow:

The general flow of the collaborative editor can be displayed as follows:

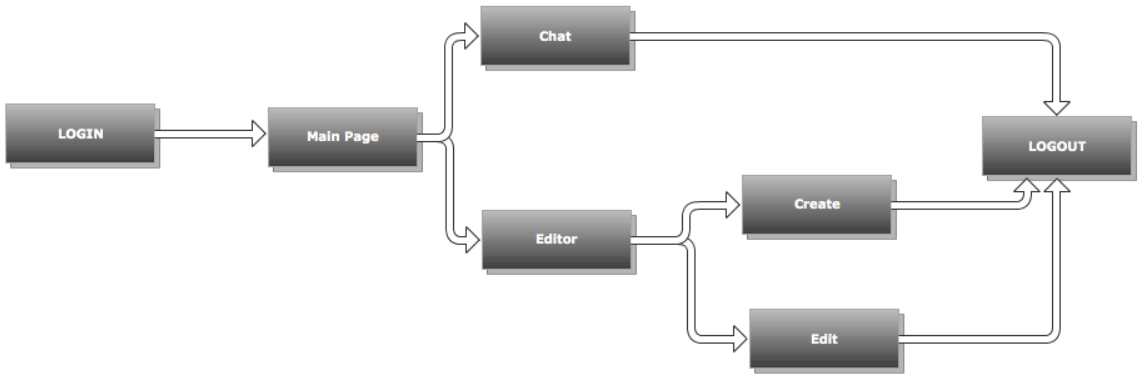

Figure 11: General design flow of collaborative editor

1) Client connects to the server and provides the login credentials.
2) On successful login the client is redirected to the main page of the application.
3) The Main page has link to two URL's. The Chat application and the Editor application.
4) Within Editor application, the client has an option to create a file or edit a file.
5) Client log's out of the application and the connection between client and server terminates.

5.2 Detailed Flow:

Login: The client browser sends a connection request to the node.js server listening on port 3000. On failure the client receives an error message else the client-server connection is established and the client is redirected to the login page. A list of valid usernames and their passwords are maintained within the node.js server. On successful login, a session is created for the respective client, which is persisted within the redis data store. A nickname corresponding to the client's login ID is used to name the session for that client. This name is then added to the list of users connected to the node js server. When the client successfully establishes the connection with node.js server another connection with Socket IO server is established automatically. Socket IO server references this client by his nickname, which is same as his login name saved by the node.js server.

Main Page: On successful login the client is redirected to the main home page of the application. This page has links for 'Chat' and 'Editor' applications. Clicking the chat URL redirects client to the chat application whereas clicking on the editor URL redirects client to the editor application.

Chat application: When the client is redirected to the chat application, the nickname as identified by the Socket IO server will be used for display. Alternatively the client can create a separate nickname for this chat session. The client is able to view subsequent users joining and leaving the chat room. Messages sent to this chat room will be broadcasted to every client connected to the chat room. That is, Socket

IO server broadcasts every message in this chat room to every client connected to the Socket IO server.

Editor application: The main feature of this project is the editor application. It has two features implemented within it. The create file and edit file.

Create File: The client writes the file name within the text box and enters the contents of the text within the textarea of the editor application and then saves the file. On clicking the save button, a key value pair is saved within redis data store. The key will be the name of the file and the value will be the text written inside the textarea. This is achieved using the redis LPUSH key [value] command. For the scope of this project all created files are shared between clients connected to the node.js server.

Edit File: The client opens a file for editing. The file stored within the redis data store is fetched and its contents are displayed within the textarea. All clients currently on the editor page can view the contents of the file within their editor textarea. The client who initiates the edit session by clicking on the textarea box becomes the writer and all other clients become viewers of the file. The textarea of all viewers are disabled and so contents of the file cannot be edited. Changes made to this file by the writer can be seen simultaneously by the viewer's. This functionality is achieved by using the redis publish/subscribe methods. Client making the changes becomes the publisher and the viewer's become the

subscribers. Any changes made to that file is published to the channel and subscribers listening to that channel get the changes made to that file.

The updates made to the file are synchronously being appended to the value attribute of the key within redis data store where the key is the name of the file.
When the writer finishes editing the file, he clicks on the release button that fires a jQuery event that releases the locks on all the viewer's textarea. The client who then initiates the edit event will own the lock for that file. Redis also provides SETNX command that can be used to lock a particular key within the data store and only the client who owns the lock is able to append the file. If other clients attempt to access the key they are unable to acquire the lock as its already being held by a different client. The only way to release the lock is when the time set for the key expires or if the client releases the lock held for the key. In this way deadlocks and race conditions are avoided using redis.

6 Experiment

After designing the platform for collaborative editor, experiments were done to test the performance and functionality of editor when multiple clients were involved.

Test 1: Basic collaborative Editor functional test.

Three browsers with multiple tabs (5 each) were used to simulate 15 clients. Before this experiment some files were added to the redis data store, which were shared between all the connected clients. After every client logged in to the editor client1 selected the file test1 to edit and then the contents of this file was retrieved in the text area box of all the clients. When client1 clicked on the text area to edit the file, all other users became readers and their text area was locked. All other users were able to successfully view the changes simultaneously made by client1. After cleint1 left the editing session other users were able to acquire the lock and continue with edit session.

Contents of file 'test1', was synchronously updated at the redis data store.

In this experiment, the platform successfully helped all the clients to edit the same file collaboratively. Concurrency control by using locks prevented any deadlock or race conditions. The only glitch in the platform is due to users having to wait for a long time to acquire the lock.

Test 2: Google Docs vs. Collaborative editor:

Case 1: Google Docs:

A document file (*.doc) and a spreadsheet file were shared between two users. Simultaneous read writes were performed on the document to analyze real-time editing and locking.

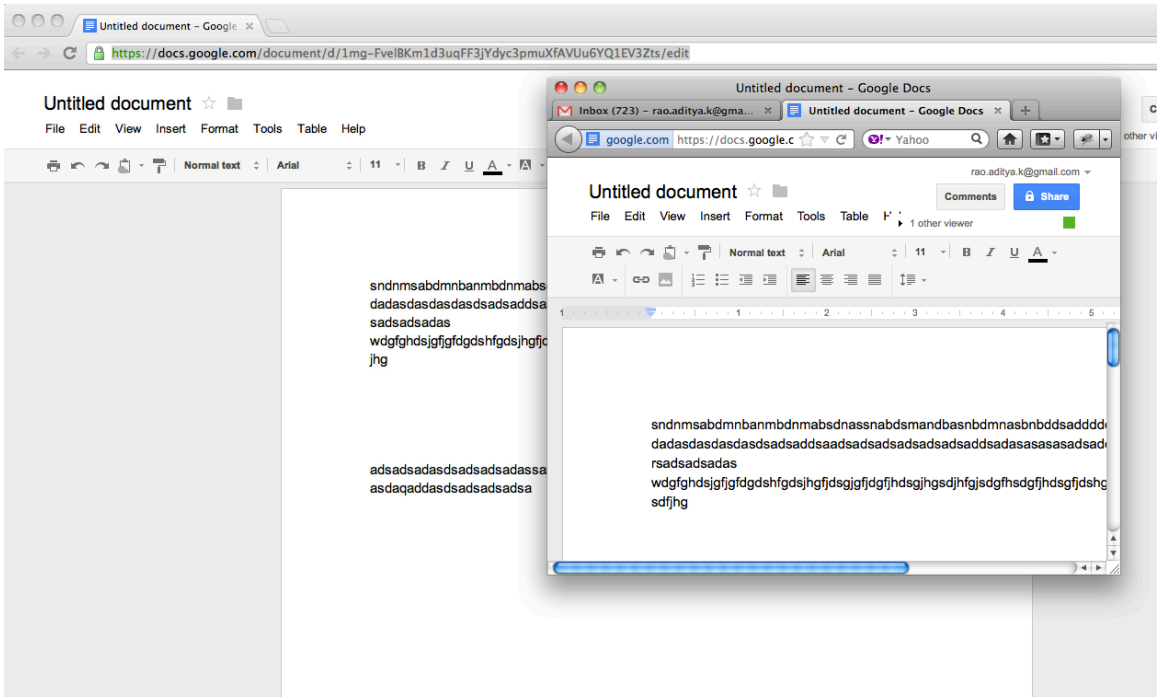

Figure 12: Google Docs sharing a doc file

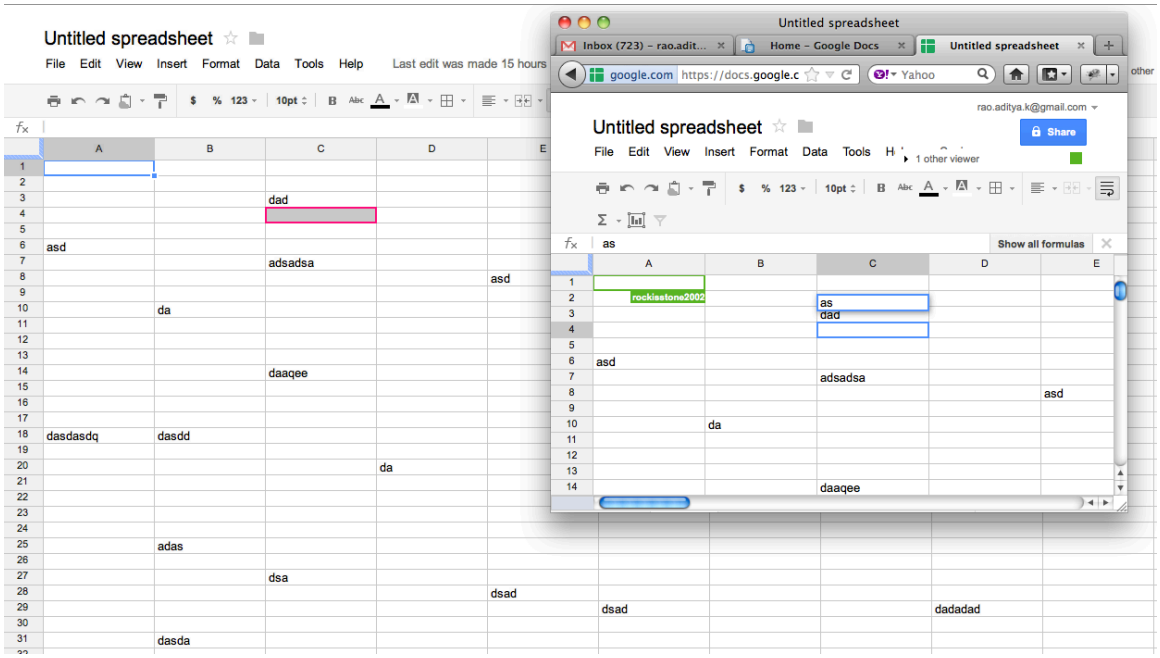

Figure 13: Google Docs - Sharing a spreadsheet

7 Analysis

Case 1 and Case 2 compared with Collaborative editor:

Using Google Docs, the two users were able to simultaneously update/edit a shared document. The files were saved on the fly and the simultaneous editing was indicated with a different color code, one for each user. The document (*.doc) file was unable to provide any locking mechanism to prevent one user from editing the document. This is feature is highly desired when one user does wants to limit access of the file to shared users.

For case 2, the users were able to share a spreadsheet and collaboratively edit the blocks within spreadsheet. Different color code was used to indicate individual user editing the block. However, the updates/edits were not displayed simultaneously

and the edit/updated block was displayed only after one user left the block after performing update. Other user was unable to edit the block until the user holding block released edit section.

Compared with the above two cases, collaborative editor was able to provide simultaneous update/edit the editor. The changes to the file was displayed in real-time to other user and also provided the feature to lock the edit area and prevent other user from editing the file. Thus locking mechanism implemented a semaphore on the textarea with one user being the writer and other users being the readers of the file. The files were saved upon clicking the save button which made sure the latest contents of the edit session being saved to the file. Since the edit session were synchronous and in real-time, the project successfully implemented updated collaborative editing enhancing collaborative work.

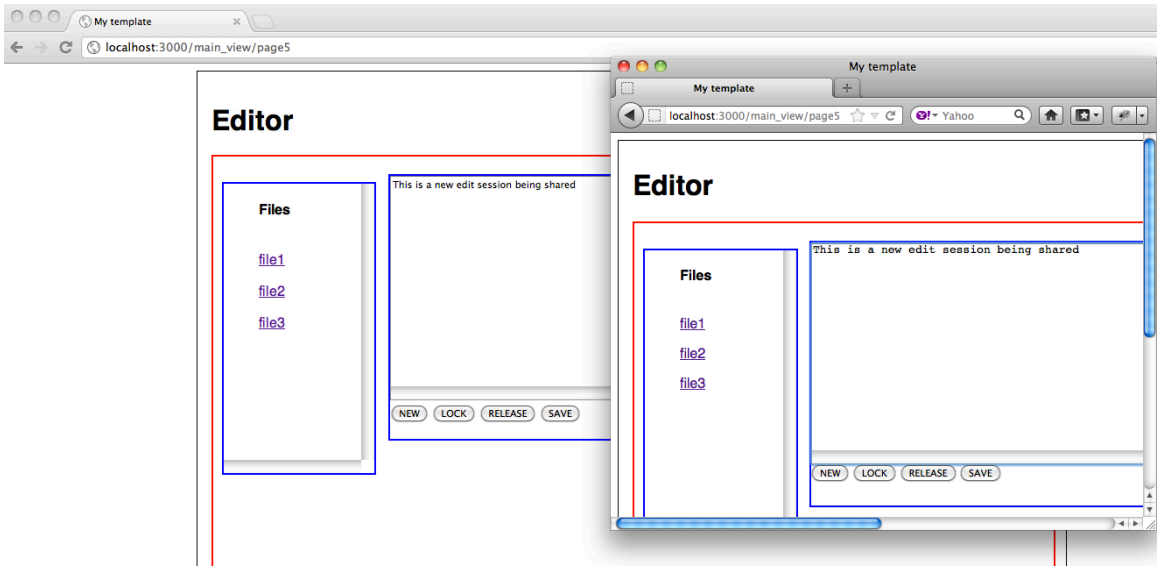

Figure 14: Two users sharing a document with collaborative editor

Statistical comparison with Collabedit:

Compared with CollabED [21], although our platform sometimes increases the wait time for users during synchronous editing, it has advantages because the performance in terms of CPU usage, no of clients that can be supported and performance of system with all the users are significantly better than the compared system. A table representing the observations is given below.

No of Browser clients	CollabED	Collaborative Editor
3	Performance: Fast. Concurrency: No CPU Usage: ~40%	Performance: Fast. Concurrency: Yes CPU Usage: ~20%
10	Performance: very slow. Concurrency: No CPU Usage: ~70%	Performance: Fast. Concurrency: Yes CPU Usage: ~40%
1000	Do not support	Performance: Fast. Concurrency: Yes CPU Usage: ~50%

Table 1: Statistical Analysis - Collabedit vs Collaborative editor

Maximum of 5000 clients can be supported by collaborative editor due to redis publish subscribe capability which is stable with a maximum of 5000 clients.

8 Problems

Technical Challenges:

The main challenge involved with this project was to select the right technology. Significant time was invested in researching different technologies like PHP, jQuery, MySQL etc. Most of these technologies did not have the required library to implement certain functionalities (like session creation and locks). Finally, node.js was selected along with redis data store for faster development of this collaborative editing application.

Design Challenges:

The main design challenge for this project was to implement concurrency control. Since redis is an upcoming technology certain commands are yet to be implemented completely. Redis SETNX command provided ways to acquire lock for a file however there are ways where this feature can be broken. Designing this semaphore such that readers can only read and writer only writes was the main challenge. The SETNX of redis along with ajax events in jQuery helped provide a stable solution to this design.

Another design challenge was to have multiple clients publish and subscribe to different channels and get updates from any channel they have subscribed. Many conflicting issues were seen when multiple channels were used along with Socket IO.

Hardware/Software Challenges:

Each version of the node.js server is compatible with certain stable releases of connect middleware; redis data store and Socket IO. All configurations needed to be compatible with each other.

9 Conclusion

This project designed and implemented a general collaborative editing platform. The platform was designed using node.js and Socket IO for providing real time sending and receiving data. A single file was shared by multiple clients and synchronously updated at real time within the redis data store. This project created a platform to provide concurrency control. File locking mechanism was used to help provide concurrency control. The experiment analysis showed that the designed platform made a good performance for collaborative real time editing but users had to wait for a long time before another user quitted editing a file.

10 Future work

In future work, we should be designing a more effective method to improve concurrency control, which would decrease users waiting time. For example redis provides ways to lock a key (file) for a specified amount of time by including the time to expire for the lock, however the user may increase the lock time by issuing another SETNX command with additional time to expire. If the user holding the lock gets disconnected then other users will have to wait for a longer time (till the lock time expires) before getting a new lock to the file.

In addition, other databases like MongoDB, CouchDB can be used as an experiment instead of redis data store.

11 References

[1] Wilson, P. Computer Supported Cooperative Work: An Introduction. Oxford, UK: Intellect Books, 1991

[2] Carstensen, P. H. Schmidt, K. Computer Supported Cooperative Work: new challenges to systems design. To appears in Handbook of Human Factors, Kenji Itoh, Tokio, 1999 (23p)

[3] Dewan, P., and Hegde, R. Semi-Synchronous Conflict Detection and Resolution in Asynchronous Software Development. Proc. ECSCW 2007, 159-178

[4] Kenroy G. Granville and Timothy J. Hickey. 2009. CollabEd: A Platform for Collaboratizing Existing Editors. In Proceedings of the 2009 International Conference on Mobile, Hybrid, and On-line Learning (ELML '09). IEEE Computer Society, Washington, DC, USA, 90-96.

[5] Ellis, C.A., Gibbs, S.J., and Rein, G.L. Design and use of a group editor. In Enmne~mr for Human- Computer Interaction. G. Cockton, Ed., North-Holland, Amsterdam, 1990, 13-25

[6] Du Li, Rui Li, Yingwei Yu, and Yi Yang, Using Familiar Single- Users Editors for Collaborative Editing. Proc. of the 36th Hawaii International Conference on System Sciences (HICS). Jan. 2003. 10p.

[7] Knister, M. J. and Prakash, A., "DistEdit: A distributed toolkit for supporting multiple group editors," *Proceedings of ACM Conference on Computer- Supported Cooperative Work*, pages 343-355, 1990.

[8] Knister M. J. and Prakash,A. "Issues in the design of a toolkit for supporting multiple group editors," *Journal of the Usenix Association*, Vol. 6, No. 2, pp. 135-166, Spring 1993.

[9] Pacull, F., Sandoz, A. and Schiper, A., "Duplex:a distributed collaborative editing environment in large scale," *Proceedings of ACM Conference on Computer- Supported Cooperative Work*, pages 165-173, 1994.

[10] Baecker, R. M., Glass, G., Mitchell, A. and Posner, I. "SASSE: the collaborative editor," *ACM Conference on Human Factors in Computing Systems*, pages 459 – 462, 1994.

[11] J. Begole, M.B. Rosson and C.A. Shaffer, Flexible Collaboration Transparency: Supporting Worker Inde- pendence in Replicated Application-Sharing Systems, *ACM Transactions on Computer-Human Interaction 6*, 2(June, 1999), 95-132.

[12] C. Gutwin, S. Greenberg, and M. Roseman, A usabil- ity study of awareness widgets in a shared workspace groupware system, *Proceedings of ACM CSCW'96*, November 1996, 258–267.

[13] nodejs.org, Joyent Inc!

[14] expressjs.com.

[15] redis.io, Cytrusbyte Inc!

[16] socket.io, Guillermo Rauch, gradebook learnboost labs.

[17] jquery.org, jQuery foundation.

[18] Google docs basics. http://docs.google.comlsupport/bin/static.py?hl=en&page=guide.cs&guide=20322, 2010.

[19] Qinyi Wu, Calton Pu, “Modeling and Implementing Collaborative Editing Systems with Transactional Techniques”, *Proceedings of ACM Conference on Computer- Supported Cooperative Work, 2010.*

[20] Walter F. Tichy, “Revision Control System”, GNU Project, Version No. 5.8, August 30, 2011.

[21] Ben Noland, “Collabedit – simple collaborative text”, 2010.