Collaborative Edit:

A Collaborative Package for the Atom Text Editor

Prepared For: Michel Barbeau, COMP 3203

Authors: Eric Adamski, 100873833 and James Roose, 100852693

Date: November 22, 2014

1. Introduction

1.1 Problem Statement

Creating a program of significant size often requires the work of a team. In order to work most efficiently, members of a team must be able to collaborate on tasks and issues, even when unable to physically gather all at once. Therefore, in order to increase the efficiency of the team, collaborative software is necessary to coordinate the asynchronous contributions of individual team members in an organized fashion. We aim to solve a small subset of this larger problem, by creating a collaborative environment in which multiple developers can simultaneously work on a task or issue within a document.

1.2 Project Description:

The goal of our project is to create a package for the Atom text editor that will allow the use of collaborative editing within a document. This package is intended to allow for fluid transfer between a local and collaborative editing environment that is capable of supporting any number of users while maintaining document structure. Therefore, features such as syntax highlighting and IDE quick editing capabilities should remain intact even while multiple users interact with a document. Additionally, should time allow, we aim to add several 'flavor' features to enhance the usability of our final product (for example, the ability to see each other's cursor within the shared document).

1.3 Background/Context:

Collaborative editing has played a large role in the success of the programming world through the use of version controls, such as Git, or the use of other online collaborative tools, such as Google Docs. These tools are sought out because they remove the need for a group to assemble and increase the time efficiency of a project by allowing individuals to asynchronously and seamlessly combine their efforts.

So, how does collaborative editing work? The implementation of collaborative software is actually very complicated due to factors involved in network communication (ex. latency) as well as local hardware communication (ex. read/write speed of a HDD or SDD). However, each of these issues have known solutions that fall under a set of algorithms know as Operational Transformation (OT) algorithms.

OT is a set of frameworks that encompass multiple algorithms that are meant to handle the quick storage and retrieval of information. These frameworks include schemas on data precedence, function precedence, permissions and much more.

Essentially, it can be represented as a Git or SVN repository that executes fetches, pulls, commits and pushes every few seconds. This process allows for a collaborative setting in which the most recent changes made to a document are regularly made available to all users.

1.4 Tools For The Job:

Programming today is rarely about building from the ground up; why reinvent something that has already been done? With this philosophy in mind, we wanted to create a meaningful solution to a relevant problem. In order to accomplish this, we made use of three important technologies.

1) NodeJS:

NodeJS is an open source and cross-platform runtime environment for server side and networking applications. It is a Javascript application that provides an event-driven architecture and a non-blocking I/O API aimed at optimizing an application's throughput and scalability. NodeJS serves as the foundation upon which ShareJS, our next important technology, builds upon

2) ShareJS:

ShareJS is a Javascript implementation of an OT Protocol as a simple server/client library. ShareJS takes advantage of the event-driven and non-blocking nature of NodeJS to allow users to collaboratively edit documents and arbitrary JSON data in real time.

3) Atom:

Atom is a completely open source and 'hack-able' text editor that allows the use of custom packages. With the implementation of ShareJS, we aim to create a custom package for Atom that will enable collaborative editing within a document.

In addition to the tools listed above, we utilize web sockets to connect each part and create a single usable package for local-collaborative editing. Lastly, the socket API we use is a javascript wrapper for C system sockets.

Result Summary:

In summary, we feel we have succeeded in achieving our goals. The Collaborative-Edit package for the Atom text editor allows multiple users to simultaneously interact with a document with minimal issues. However, some bugs do still exist, which we speak towards in section 2.3 of this document. We have also succeeded in implementing a few quality of life features, such as the ability to see other user's cursors in a document.

2. Project Analysis

2.1 Implementation:

In this section, we will discuss the overall architecture and structural components of our system. Additionally, this section will cover the important design choices we made and explain why we made these decisions.

The most important steps in creating working software are often the steps that come before writing a single line of code. NodeJS (and by extension, ShareJS) provided us with a solid architectural foundation to build upon. Our entire system is event driven and built upon a simple server-client architecture where a single user hosts a document as a server and the remaining users connect to the host as clients. Maintaining this structure allowed us to tie NodeJS into our system in a natural and seamless manner. In essence, we've created a simple distributive system where the clients do all of the operations and the server handles the message passing between clients. This design allows for the server to be single-threaded and lightweight.

It was important for us to fully integrate ShareJS into our system because it contains the OT frameworks we require for collaborative editing to be possible. However, ShareJS only enabled OT operations to be done on the server-side of our system. This is problematic because, although the server is primarily responsible for the synchronization of the documents, special cases may arise where the client must also be able to make OT-based decisions. In these cases, we had to decide ourselves which version of the document would be the newest. We implemented some simple rules to accommodate this and, throughout our system, made design choices that would make it easier for the client to make OT decisions.

All connected clients send messages to the server regarding changes made to the document in chunks, rather than on a per-character basis. This is an example of a design choice we made that helps simplify our system by decreasing the number of messages being passed. This reduces the number of OT decisions the client needs to make by eliminating many special cases that arise from simultaneous message sending/receiving. Additionally, each client has two sockets connecting to the server: a 'document' socket and a 'metadata' socket. This design choice is actually a work-around that arises from the lack of an API that can handle document data as well as additional metadata (such as cursor position in the document) within the same socket. This issue will be described more at length in section 2.3.

The server is responsible for the orchestration of the documents. With the OT operations provided by ShareJS, the host is what keeps the client documents synchronized. To help with this task, the host maintains a list of all connected clients and assigns each client an ID. These clients are then connected to each other on a per-document basis. As a result, the server knows which clients to contact when a document has been altered simply by going through its list of clients and checking which ones are subscribed to the changed document. The server is also capable of recognizing which client sent the document changes and therefore will not echo the changes back to the original author.

Overall, our system structure can be summarized as such. Clients are in charge of document modification and informing the server of changes it makes, meanwhile the server is in charge of informing other clients of changes and making OT decisions that keep the documents in sync. This structure is a simple, elegant and effective means of creative a collaborative editing space for multiple clients.

2.2 Analyses and Discussion:

The following section will analyze the effectiveness of our solution to the problem and make general discussion about the topic. As mentioned in our summary, we are confident that our product accomplishes our goals and makes a useful contribution to the problem of collaborative programming.

As mentioned previously, our system aims to solve a small subset of a larger problem: synchronizing the efforts of individuals for optimal performance of the team. Over recent years, this problem has been largely solved. Companies and teams of programmers already use tools, such as Github, that are very effective at coordinating their efforts. This is important to us because the tools we need to contribute to the problem already exist! We can assume the collaborative aspect of our system is relatively optimized because of the heavy optimization of the OT algorithms and ShareJS library that we use.

The remainder of our system is less optimized. The Atom text editor is another tool we make use of that we did not create. However, it is a relatively new and open source project that is still under development. Additionally, there are many improvements that can be made to the code we've written ourselves that will be discussed in section 2.3. Overall, our system is relatively lightweight and performs well.

A noteworthy topic of discussion is how much we learned about networking while working with the metadata socket. Unlike the document socket we worked with, the metadata socket was done from scratch, without the use of an API. In a sense, the addition of the event handler socket, though poor design, was beneficial to us because it displays an understanding of networking aspects that would have otherwise simply been implemented by an API.

TODO: I feel like this section is the weakest, if you have anything to add, please do

2.3 Problems and Improvements:

In this section, we will discuss the existing problems within our system, potential solutions to those problems and general improvements we would like to make to the package. Our biggest goal with this project is to release the collaborative-edit package as an Atom package on Github, where people can actually use the product. We want our project to be useful and relevant. We fully intend to improve upon our system because we believe that it can be useful in the real world.

The first, and most serious problem within our system is the inability to recover after the documents have fallen out sync. Theoretically, the documents should always be synchronized, however, in the rare case that they become unsynchronized, there is currently no way to recover. Users would have to disconnect from each other, discard the unwanted version, and reconnect with the proper version. This is very poor design, and in the future, we would like to add mechanisms that would be capable of recognizing when documents are out of sync and take the appropriate actions.

Another issue within our system is the inability to host and edit multiple documents at once. Although we have taken measures to support this feature, our current product cannot do this. This is because opening a new document currently does not open new sockets; instead, it re-uses the sockets of the older document. If the older document was still in use, its sockets will be stolen, therefore making it impossible share both documents at once. The reason we handle sockets in this way is because a single text buffer can contain multiple documents. If this is the case, returning the position of a client's cursor can no longer be done with a simple index. If we cannot properly track a client's cursor position in all cases, then we cannot properly track the modifications that that client is making to their respective document. As a result, hosting multiple documents is currently blocked due to the way we track document changes. In the future, we would like to find a workaround that will allow us to fully implement this feature.

The last major issue with our system relates the only way we currently know of to desynchronize client documents. Currently, if a user attempts to delete an entire document using the CTRL-A select-all shortcut, the other documents will remain unaltered. We believe this bug is due to CTRL-A deletions not properly updating the cursor position in the document. Since tracking cursor positions is essential to the way we detect document changes, this poses a problem. If the cursor never registers as being altered, then an event is never launched notifying the server of that document's changes. In future iterations of our system, we would like to fix this bug by implementing special case for these types of deletions.

In addition to fixing the problems in our system, we would also like to make improvements to the system in future iterations. The main improvement we would like to make to our system was previously mentioned in section 2.1. Currently, each client uses two sockets to communicate with the server: a 'document' socket and a ‘metadata’ socket. We did this because ShareJS reacted strangely when we included metadata with our document transformations. Unfortunately, we could not find an API that would allow us to send both types of information through the same socket. However, we realize that the use of two sockets is an awful workaround and in the future we would like to find an alternative.

Lastly, in addition to these larger problems, we would also like to make smaller improvements that speak toward the general usability of the product. For example, we would like to better present errors to our users and make it easier for users to get connected.

TODO: ERIC, DOES THIS MAKE SENSE??? Should it be added?

The first real issue with our system is related to the nature in which clients send messages to the server. There currently exists a certain degree of back propagation, where any time a client receives and implements a local change, the change is then echoed back to the client that originally created the change. This echo presents a problem because unless it is properly ignored, there is potential for an infinite loop of echoes.

3. Conclusions

In summary, our system is a collaborative package for the Atom text editor. It enables users to share a document over a local network and simultaneously make changes to it. The purpose of this software is to enable members of a team to synchronize their individual contributions to a project in an efficient manner. Our system is intended to tackle a smaller subset of this problem that deals with individual issues within a single document. This kind of software should appeal to developers who are fond of paired programming, which is becoming increasingly popular in the workplace.

4. References

- NodeJS. (2014, January 1). Retrieved November 22, 2014, from http://nodejs.org/
- Gentle, J. (2011, November 6). ShareJS - Live concurrent editing in your app. Retrieved November 22, 2014, from http://sharejs.org/
- Atom - A hackable text editor for the 21st Century. (2014, November 24). Retrieved December 1, 2014, from https://atom.io/

Finishing Notes:

- Do we want figures?
- Do we want to find an article/paper to site?
- Did I miss anything really important?
- Is anything I wrote flat out wrong?