CSC 213 Final Project - Grinnell Doc

Hongyuan, Abyaya, Joshua and Anaan

Project Overview

Our project goal was to implement a real-time shared document editing platform, where multiple users can edit the same file simultaneously. The main areas of focus of our project are networking (the server-user system) and concurrency (multiple users editing simultaneously). The main components of the system include a central server and a network of users. Initially, when the server fires up, it opens the specified file and waits for users to join. Once a user makes a request, it authenticates the user and sends a copy of the file to the user for editing. Upon successful joining, the user can edit the document and see his/her own and other users' changes to the document. The server merges the changes and sends updated versions to every user. We support insertion and deletion of characters, including the newline character. The users' changes will be merged into the document in the order the users made them, if the users did not make changes at exactly the same time. As more users join, the server follows the same protocol, except it sends the latest version of the file for editing. We handle potential problems induced by concurrent editing by keeping a log of historical changes.

We evaluate our system by measuring the time between one user makes a change and this user receives the version of the document with that change merged. We vary the number of users in the system to evaluate our system's scalability. The results indicate that our system does become a little slower when the number of users in the system increases, but the performance is still within an acceptable range. There are some outliers in each experiment and we suspect that they are due to the unreliability of MathLAN.

Design and Implementation

Our implementation consists of two main pieces, the central server program and the user program. In general, the central server is in charge of accepting new users, listening for any update to the document by any joined user, merging that change into the file, and sending the updated version to every user. Users in the system can move their cursors to make changes to the document as wanted, while being able to see other users' changes. The user program sends every single character change made by a user to the server. The user program also adjusts a user's cursor upon reception of a new version from the server. In addition, to handle the potential problem caused by concurrent editing, we keep a log of historical changes so that when merging a change, the server is able to find the correct location of the change in its version of the document.

It is worth noticing that the server is the only component that has direct access to the shared text file that is to be edited. Therefore, the server is the only portion that can modify the real file. Upon reception of a change by any user, the server rewrites the file with the change merged and sends the new version to all the users.

Server

The server program has one main thread that should run forever, for accepting new users and one thread per user to listen for their changes to the shared document. We chose to use separate threads because we want to keep listening to each user while listening for new users who want to join. Before running the server, the file must be already created and has a newline character at the end of it. This is because our program assumes that there is a

newline at the end of the file, since many text editors, including Emacs, automatically add a newline to the end of a file.

The server accepts a connection request and then authenticates user validity using a password. If the password entered by a user on the command line matches the server's, the server adds this user to its user list by keeping track of this user's socket and unique id (assigned by the server). The server then tells this new user his/her unique id. Next, the server launches a thread for listening to this user's change to the shared document. On the other hand, if the password does not match the server's password, the server will immediately close the corresponding socket used to talk to this user.

In the thread for listening to a user, the server first sends the latest version of the document to the user for editing. Then the server keeps listening for any change by this user. When there is a change made by this user, this user program sends a struct that contains information about what this change is, the version he/she was at when he/she made this change; and the location of this change in his/her version. Once the server receives an update, it starts to process this change.

First, the server traverses the linked list of historical changes to find the location of the change in the server's version of the file. Although we allow for concurrent editing, the server merges one change at a time. After merging the first change, the server's version of the file will differ from the remaining changes'. This discrepancy occurs due to the fact the rest of the changes were made in a an earlier version of the document. Thus we need to keep a linked list of historical changes, with each node containing the version this change was merged based on, the type of the change (insertion or deletion), and the location of the change. The server traverses starting at the node that has the same version as the version sent by the user in the change struct. If a node is of insertion type and the location in that node is less than or equal to this user's location of change, the server increments this user's location of the change. If a node is of deletion type and the location in that node is less than or equal to this user's location of change, the server decrements the user's location of change. Then, we apply the changes described above when we go through the rest of the list. The server traverses each node that has version greater than or equal to this user's version in order to gradually update this user's location of change to the location in the server's version. After checking the last node of the linked list, this user's location of change should be in sync with the server. If this user's version is the same as the server's version, the server program will not traverse the linked list at all.

Second, the server program saves the current version of the document into an array of characters and reopens the document file in "w+" mode. Following this, it overwrites the file according to the type of the change made by the user. If the change is a deletion, the program first writes the part of the saved buffer before the deleted character and then appends the part of the file after the deleted character. If this change is an insertion, the program first writes the part of the saved buffer before the inserted character, then the inserted character, and the remaining part of the buffer. This is how the server merges each change by a user.

Third, the program adds this change to the tail of its linked list of historical changes and updates the version number (a global), which indicates the server's version of the document. When there are over 100 nodes in this linked list, the program will free the head node. The program then sends each user the updated version of the document with the change merged. Before sending the text, the server sends information regarding this version of the document, including the version number, length of this version of the document (so that a user can allocate a buffer of an appropriate size), the id of the user who made this

change, the location of this change in terms of the version when the server merged this change, and whether this change was a newline (regardless of newline deletion/insertion).

We use a mutex to lock the process of merging and sending a change. If multiple users are editing at exactly the same time, the order of merging the changes depends on which user grabs the mutex first. If a connection is cut, the user is removed from the global linked list, which uses a mutex as well. Since we check errors constantly in the read() while loop (for listening for changes), we don't do error checking for each read()/write().

User

Once the server is run, the users can be fired up as well. For any given user, the user program keeps track of the user's cursor position, the version of file this user is working on, the lengths of the previous version and the current version, user id, and the user's cursor position in terms of the original file. The user program first sets up a socket and connects to the server and then sends the password entered on the command line to the server. If the password is not correct, the socket will be closed by the server and then the program will exit. Otherwise, the user program reads from the socket for a unique id and then saves this id. The program then launches a thread to listen for new versions from the server. The main thread is dedicated to reading user input. We want to keep listening to both the user and the server, so we chose to run these tasks on separate threads.

In the main thread, the program keeps reading user input - keys that a user presses. We used functions in the ncurses library to manipulate the terminal screen. To navigate through the document, the user can use the left or the right arrow keys. When the program receives a left arrow key or right arrow key as an input, it checks whether the cursor is currently before/after a newline and whether it is at the border of the terminal. If the user presses the left arrow key and previous character before the cursor is a newline, the program moves the user's cursor to the position of this newline character. If the previous character is not a newline, the cursor shift to the left. If the cursor is at the border, it will end up at the end of the previous line. Upon reading a right arrow key input from a user, we have similar cases, except for we limit the cursor within the file length. Once we hit a newline, we simply move the cursor to the beginning of the next line.

If the user wants to make a change to the document, i.e. if the user presses keys in order to insert or delete characters (including newline), the change, the change's location and the user's version will be sent to the server.

In the thread for listening for newer version from the server, once we get some version's information, we keep reading for a new version and then update this user's version number and other globals. We can get this version's length from version info and then allocate a buffer of an appropriate size. Once we read the newer version, we update this user's terminal. Then we move the user's cursor accordingly. In general, we always let the cursor stay at the same character it was on before getting this newer version. We have 6 cases for moving cursor: 1) the first time this user gets the document, 2) newline insertion, 3) newline deletion, 4) normal character insertion, 5) normal character deletion and 6) no change in this version. In Case 1), we just place the cursor at the very first position in the terminal. In Case 6), we don't move. Finding the location of the previous newline before the cursor helps us determine where we should move in Cases 2) to 5), where we differentiate between whether this user has made the change merged in this newer version and whether the change is before or after where the cursor was, because if you are the user who made the change, you always move upon getting change-merged version back and if you are not, you move if the change is before your cursor and don't move otherwise. In Cases 4) to 5),

we also check if the change took place in a position between the cursor and a newline character, because we don't move the cursor if there is a newline between the cursor and the change. If the program ever stops listening to the server, we exit the program because the server is probably down.

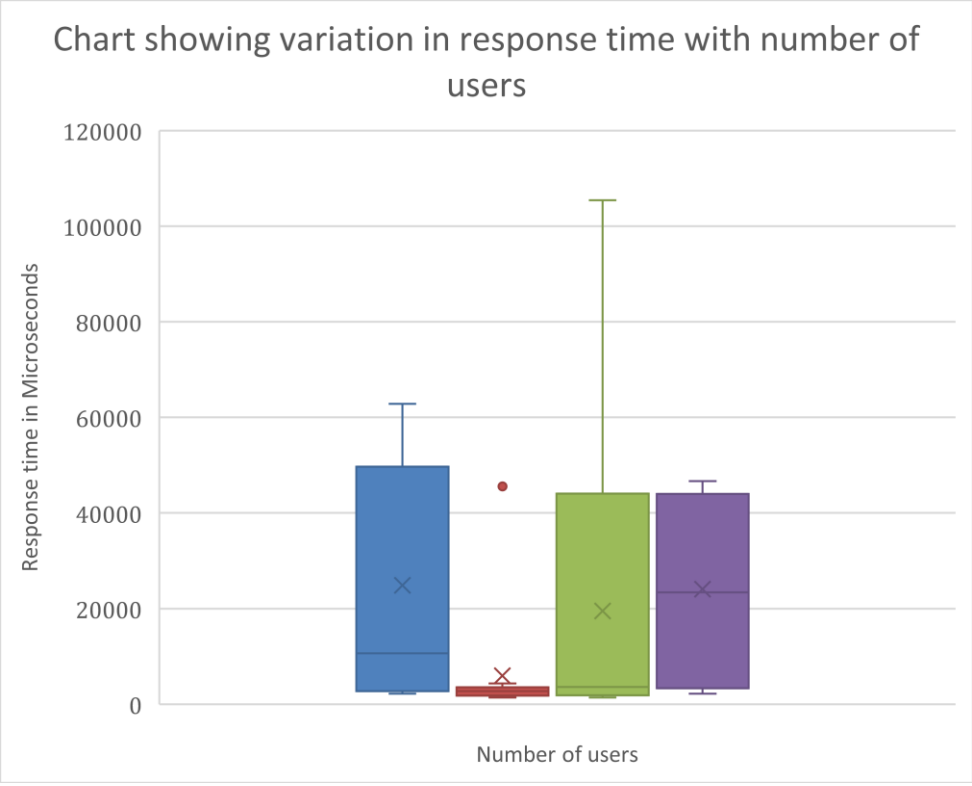
Evaluation

In order to formally evaluate our program, we decided to measure the difference in time between when a user sends a change and when the server sends back the version incorporating that particular change back to this user. Hence, we ignore any other versions not made by the particular user. We compared this difference for varied number of users, choosing up to 4 users at a time and recording their individual performance. We let each user in any experiment make 8 changes and then record the time. The changes include insertion and deletion (including newline insertion/deletion). We used time_us() function provided in the Virtual Memory lab to get the absolute microseconds.

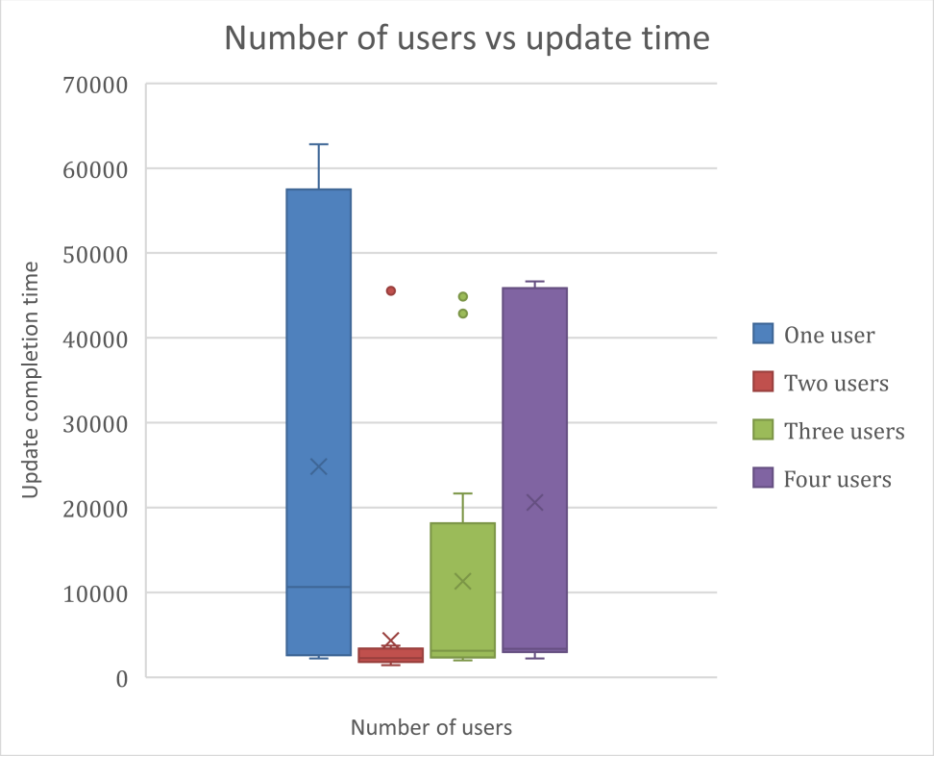
Experimental Setup

We ran the experiments on computers in 2401 Noyce on May 14, 2018. We ran our program with 1 user, 2 users, 3 users and then 4 users. For each user in each experiment, we record more than 8 data points and then randomly select 8 of them for each user. We are using Ncurses 6.0. We ran two times the whole test suite.

Experiment Results



Experiment 1



Experiment 2

As we can tell from the two boxplots, there are usually large outliers during any run. This is due to the unreliability of MathLAN. Somehow when there are two users, the speed of update is the fastest, but there still exist large outliers. Our performance did become slower from 3 to 4 users, but it is below 0.05 second for 4 users for both experiments. Thus, our system does have some scalability, since it does not become significantly slower when are there more users. The time it takes a change to be displayed is still within an acceptable range.

Unseen Complications

As with any project involving the development of a system, there were several complications and nuances we did not foresee when we began our project. The bulk of these came from the simple fact that text editing is a complicated process. We figured out a basic, working structure in the first week of our schedule, but we spent the bulk of rest of time on manipulating the cursor and newlines which was not on our list of expected complications.

We encountered another issue regarding cursor movement when we tried to test our text file with adding and deleting newlines into our file. Because we handled newlines by filling in an appropriate number of spaces, the behavior of newlines in our program was not the same as the real one. For example, if you edit on one line, the content of the next line will shift, because of the filled spaces. In order to solve this issue, we had to add a constraint of not being able to move up and down in the file while editing. Having this restraint, we are able to track a user's cursor position in terms of the real file and find the real location of change.