CollabCanvas

Bri Binder and Zack Romrell
Williams College

1 Introduction

CollabCanvas is an interactive, collaborative canvas. Users can join CollabCanvas and change the color of any pixel they choose and see the changes done by their peers in real-time!

The idea for CollabCanvas came to us when Jeannie provided us with a list of possible ideas, and one of them was "drawing with many, many friends." This reminded us of what Reddit did on April Fools of 2022, which was launched r/place. This was a social experiment to see how people around the world would interact with each other through art. They provided some limitations that required every user only to be able to place a pixel every five minutes, which forced people to collaborate with each other to get anything done. This social experiment only lasted for three days. Thus, we wanted to challenge ourselves and create our own version, CollabCanvas.

The rest of this paper is structured as follows. Section 2 presents the design choices we made for CollabCanvas. Section 3 presents the implementation of CollabCanvas. Section 4 presents the way we evaluated CollabCanvas. Section 5 presents future work. Finally, we summarize our contributions in the conclusion, Section 6.

2 Design

There were many directions we could have taken CollabCanvas, but we decided to focus on four specific goals. The first is to make sure the app is easy to navigate. This was simple since we had access to Jim Bern's graphics API which you can see in Figure 1.

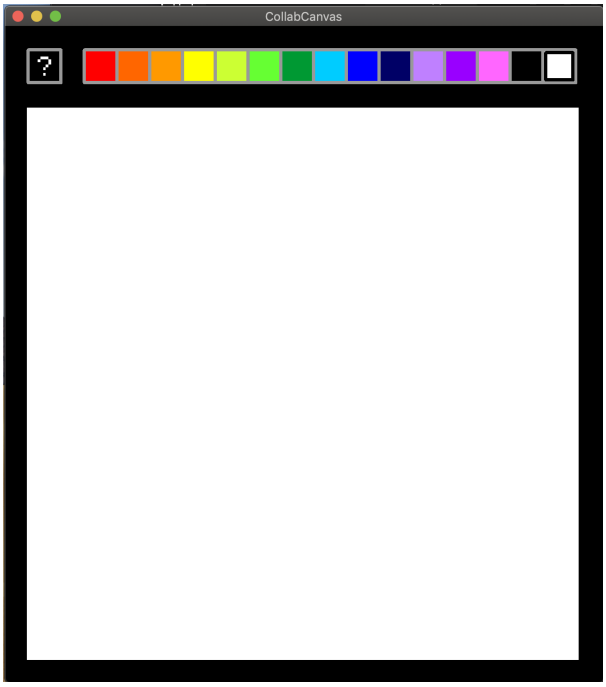

Figure 1: This is what the CollabCanvas interface looks like when you first run the server

Second, we wanted to ensure that every user can only change one pixel at a time. The reason why we wanted this feature is to put less emphasis on the individual and instead emphasize the collaboration aspect. This way, a single user cannot easily eliminate the works of others. Our third design choice is to make sure communication between the server and the client is simple. Since there are going to be lots of messages exchanged between the server and all

the clients, we wanted to ensure that messages were simple/short so that they could be sent quickly and efficiently. Our last design choice is canvas color consistency. Round-trip time to send messages may vary from client to client, so we wanted to ensure that every user will eventually see the changes on their canvas.

3 Implementation

In order to use Jim's graphics library, we decided to write CollabCanvas in C and C++. Although we already have implemented the basic functions of a server in C as our first coding assignment there were still some painful moments with creating CollabCanvas and creating evaluation tests also caused some headaches. CollabCanvas is a distributed system with a centralized server that stores the contents of the board. The centralized server waits for new clients to connect and immediately sends them the current contents of the board see Figure 2.

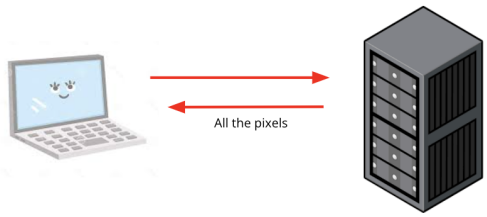

Figure 2: The client requests to join the server. The server will accept the request and then send the client the current state of the entire canvas.

Each time a client clicks a pixel the client sends a pixel update to the server and the server broadcasts this pixel update to each other client connected, see Figure 3.

We will go into more depth about the server in subsection 3.1 and the client in subsection 3.2.

3.1 Server

The server's main purpose is to maintain an accurate state of the board and manage each client's experience. The server maintains an active list of currently connected clients with a doubly linked

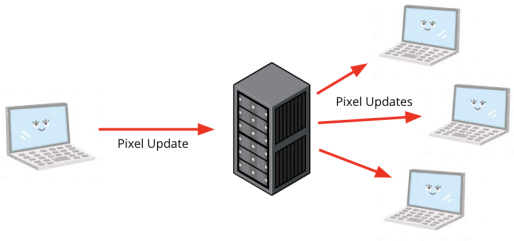

Figure 3: The client can change the color of a pixel by sending the server the pixel update. The server will then update the canvas and send every other client the pixel update.

list of clientInfo structs which contain the clients socket number and IP address. The server then allocates a thread for each client which oversees receiving pixel updates from that client and broadcasting the updates it receives. When a thread receives a pixel update from its respective client it changes the board state in a locking manner to avoid concurrent updates. We encoded the board state as a 1-dimensional list of length equal to the product of the canvas's dimensions. With each received pixel update, the thread loops through the doubly linked list of connected clients and sends a pixelUpdate struct which consists of two integers, the index of the updated pixel and an integer representing its new color. For simplicity we only support 15 colors (indexed 0-14), however, by sending two more integers we could represent the RGB values of each pixel and support more colors. While the client remains connected each thread waits to receive pixel updates and broadcast the change.

3.2 Client

The client's main purpose is to display the graphical interface of the canvas and send/receive pixel updates. When you run the client, it immediately connects to the server and receives the current state of the canvas. With the received pixel data, it initializes a canvas with its respective colors. The client is then free to make local changes to the canvas which are immediately sent to the server. The client polls messages in a non-blocking fashion which allows for them to continually receive messages while still

being able to click. In the case where the server goes down, the client's canvas still remains open, however, is now purely a local canvas. To display the interface, we utilized Jim Bern's graphics cow API, which is essentially a nicer version of OpenGL code.

4 Evaluation

Although our primary goal was color consistency we still wanted to evaluate and better understand how our server performed under different circumstances. To test our server's performance in the starting stages of running, we first created a program that connected a specified number of clients to our server and recorded the average time it took to receive the initial contents of the board. In Figure 4 we can see that after increasing the number of clients joining to above 3 the average time it took to receive the board contents levelled off.

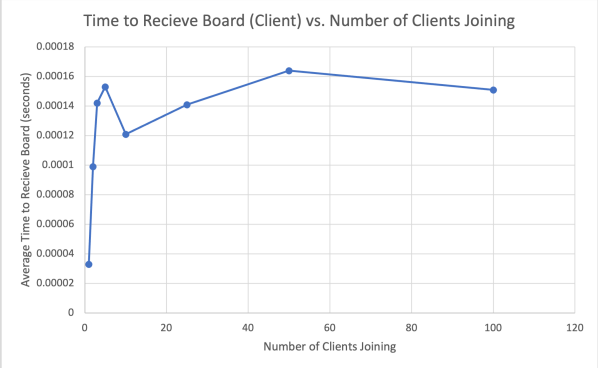

Figure 4: A graphical representation of how the average time it takes to receive the initial board contents change as the number of clients joining changes.

The reason for this had to do with the fact that the server was sending out the board contents quick enough that it was always at most dealing with around 3 clients. We saw this behavior by disconnecting each client after receiving the board contents and viewing the size of the doubly linked list as we stress tested our server. However, as you might imagine if we were to increase the dimensions of the canvas this could add much more stress to our server.

We next wanted to test how the time it took the server to receive and send pixel updates changed as we simultaneously added clients. This test was also meant to replicate and evaluate the performance during the early stages of the server. In order to determine these effects, we ran two separate test varying whether clients were joining simultaneously or not. In each case we recorded the total time it took the server to send out each pixel update after receiving it for 10 pixels, 100 pixels, 1,000 pixels, and 10,000 pixels. In Figure 5 on the x-axis we graph the log base 10 of the number of pixels being sent ($10^1, 10^2, 10^3$, and 10^4).

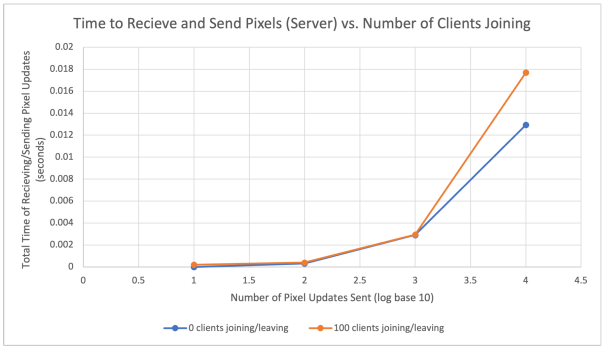

Figure 5: The client can change the color of a pixel by sending the server the pixel update. The server will then update the canvas and send every other client the pixel update.

On the y-axis we record the total time it took for the server to send out these pixel updates after receiving them. As we can see in the case where 100 clients are joining/leaving the total time it took the server to send out these pixel updates was slightly greater than the case where no clients were joining/leaving. It was really until the 10,000 pixel case until we truly saw a big jump in the total time it took to send these messages after receiving them. However, even in the 10,000-pixel case the total time was still relatively fast. In this test we only had a single other client connected which decreased the time it took to send each individual pixel update.

Lastly, we wanted to better understand how our server would behave once more clients were connected and interacting with their local canvas. We decided to record how the average time it took for the server to send out pixel updates changed as the number of clients connected increased. Since the

server is required to send each pixel update to all the connected clients, as the number of clients increased, we expected the average time to increase as well (ideally linearly). In Figure 6 we can see how the average time it took to send each pixel (in a batch of 10,000 pixels) changed as we added more clients to the server.

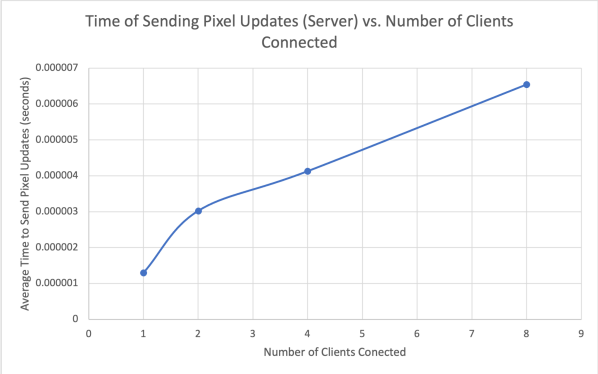

Figure 6: A graphical representation of how the average receive and send time changes as the number of connected clients increase.

From the 2 client case to 8 client case we can see this linear increase which make sense because the server only needs to loop through a larger doubly linked list when broadcasting a change.

In general, the performance of our CollabCanvas server was surprisingly resilient to many different situations. This is most likely due to the simplicity of our messaging protocol and small size of our messages being sent and received. If we were to incorporate 2 more integers to describe an RGB value, we would most likely see the overhead of our server increase by a factor of 2.

5 Future Work

When working on CollabCanvas we came across many possible extensions we would love to implement.

The first possible extension we thought of was to cache the contents of our canvas before shutting down the server. Currently, CollabCanvas is set up to start with a blank canvas every time we restart the server. However, by simply reading and writing to a file we could cache the contents of the canvas into a separate file. Personally, we enjoyed the fresh start each time a new server session started.

The next idea we wanted to implement, however, did not have enough time to be a private canvas feature. Such a feature would allow a client to decide they want to host their own canvas and provide their friends with a code so that they can join. This could be done quite easily by incorporating more of our server's code in the client. We would also have to add a text box on the CollabCanvas graphical interface to allow for a client to connect to their peer. By adding a list to store the contents of their private canvas to each server we could have a client run a separate thread to accept new clients to their private canvas and branch off an independent thread for each client connected. By typing in a code to the text box the main server could use a hash table to get the corresponding peer's IP address to the code and send that to the client wanting to connect.

We also wanted to make our app more accessible. The simplest solution is to access CollabCanvas through a web browser using a URL address. In order to run the app on your computer, it requires downloading an executable file, which is not the most trustworthy or safest option. Having access to CollabCanvas through a web browser makes it more accessible, however, this would require learning how to convert the interface we previously made into something that a web browser can run or rewriting the entire interface in WebGL. Alongside re implementing the graphical component we would also have to support the headers necessary for whichever web browsers we are interested in.

Another cool feature we thought of was to remove anonymity. The idea is that if you hover your mouse over a pixel, it will tell you who colored in that pixel. This could easily be done by adding more data about each client like a name they would like to be shared. Also, this would require us to add more data to each pixel on the canvas. Overall, this would increase the size of our messages, however, would hopefully make users behave better. This could provide more interaction with users, such as "Greg that was a sweet duck you drew on CollabCanvas!"

The last one is to improve scalability. Our project works extremely well for a smaller sample size, such as the size of our class. However, everything gets more complicated once we want to scale up.

How would CollabCanvas work if we wanted to take it worldwide? We would have to think of clever solutions on how to distribute the canvas more efficiently. One possible solution is to send the contents of the board in chunks instead of individual pixels. A user will tend to work in a section of the board; thus, after a certain amount of time, the changes made in this chunk will be sent to everyone else.

6 Conclusion

In the end, we had a lot of fun creating CollabCanvas. It was enjoyable to incorporate a graphics component into our project and successfully distribute canvases to multiple users.

CollabCanvas is a centralized distributed system. When a client joins CollabCanvas, they are sent the current state of the entire canvas. After this initiation, messages of pixel updates are sent back and forth between the clients and the server. Essentially a client requests to update a pixel, the server will receive the update, and then the server will broadcast this update to all the other clients.

We learned a lot about distributed systems this semester and while creating CollabCanvas. It was interesting to think about how we wanted to design CollabCanvas and how the server and the clients would interact with each other. Many questions arose such as how would messages be sent? What would happen when a client joins our server? How would we organize clients? Ultimately, we are very proud of what we created, and hope others can also enjoy drawing in CollabCanvas.