Select Squares

CMPT 371 – *Networking I*

Group 22

Patrick Burns, Hanjie Liu, Erik Hu, Lingjie Li, Xiaohong Xu

Game & Design Description

Our game is a fun and simple design where up to four players are presented with an array of squares before them and the object of the game is to click as many squares as possible before your opponent can select them. The winner is the person who selects the most squares when the last square has been selected. In our design, we used a client server design where each player will run an instance of a client which transmits player requests to the server. The server handles backend jobs such as win condition, player to player interaction, color assignment and keeping track of which squares have been selected by each player. The objects which must be locked for concurrency in our game are the squares on the board, and each square is a shared object. Both players may attempt to lock the square at the same time, this is not a turn-based game, but once the square is locked, it cannot be locked by the other player. To ensure we have no race conditions, we have employed semaphores on our server for data integrity and accuracy with our shared objects by recommendation of our Amandeep Dania, our TA who helped us do the best job we could on this project. The general concept of how our game works technically is the clients who are to participate in the game connect to the server and then the player who was first to connect, who is the controlling player, starts the game. The players are then presented with the array of squares and the players frantically click the squares as quickly as possible to try and select more squares than their opponents. Each click of a square sends off a packet from our application layer messaging scheme which I will detail shortly, and the server then parses the packets and makes the required changes to the data structure if the request is valid and ignores it otherwise. This continues for some time until the end condition is met, where all of the squares have been filled. The server then does a simple check to see who has selected the most squares and determines the winner, and then communicates to the clients who has won and the clients display the winner information on the screen.

Application-Layer Messaging Scheme

Our servers application layer messaging scheme is kept as simple as possible for efficiency. It is kept in a plaintext readable human format to ease troubleshooting and diagnostic. Further optimizations could be made in our messaging scheme in the future if it were to be deployed on a larger scale for data transmission efficiency, but we chose to forgo this and select instead for ease of use and diagnostics because the packets are already relatively small, so the additional overhead incurred for making them easy to understand was a small price to pay for the large gains in human readability. Our server is setup to detect the clients connecting IP and Port automatically. If this server were to be deployed on a larger scale on a static IP dedicated remote server, we would simply put the servers IP in to the client, much like how large scale applications like WhatsApp deal with client server traffic. The server side code would handle the rest with its automatic IP and Port detection.

There are only a few main packet types sent between the client and server and I will detail them here.

- **“require_color”** Is the “Hello” packet, from client to server. It’s the client connecting for the first time and letting the server know that it needs to be assigned a color so it knows what color to show its selected squares as on the client side screen.
- **“@color”** The word Color in this packet is just a placeholder, for example, the packet could be “@Blue” or “@Green”, this packet is the server acknowledging that the client wishes to connect and then replying with the color the client has been assigned.
- **“Player_start”** Is then sent from the clients to the server, letting the server know that the clients wish to start the game.

- **“All_start”** Is then sent from the server to all of the clients, letting the clients know the game has now begun, so the client can relay this to the player and the player can start to select their squares.
- **“!x:y”** Is the general format for selecting a square that the client will send to the server. X and Y in this example are placeholders for the x and y coordinate that the player is asserting that they wish to select. For example, if a player wanted to select the square at (X,Y) = (1,1), they would send the packet “!1:1”. This is the main packet sent many times during a game from each client to the server to play the game.
- **“Yes !” + <coordinate> + “is filled by the client” + client_ip_port + “which color is @” + currColor** this packet is sent from the server to each client to let the client know which color to fill a square that has been selected by any player. The “Yes !” indicates that the message is respective to a packet fill, the “<coordinate>” is transmitting the (X,Y) pair of the respective square to be filled, the “client_ip_port” is the client and IP port that will be auto filled by the server for each client its sending to, and the “which color is @” is letting the client know that the next word, “currColor” will be the color it should shade the square. This is the main packet for communicating from the server to the client which color it should shade a selected square. It does most of the work for server to client communication while the game is running.
- **“$$$” + “@” + “color”** Is the win condition packet sent to all clients from the server to indicate that the player with the sent color is the winner of the game. Once the client receives this packet, they indicate to the player which player has won, and then close connection with the server.

Code Snippets

- **Client opening socket** to communicate with server. Here we have coded in 127.0.0.1, however at the time of deployment the server IP would be set to whatever the static IP of the host was when hosting was arranged.

```
//Opening the socket, Listening data from server, get stream into
Recdata
        try {
            byte[] ipAddr = new byte[]{127, 0, 0, 1};
            InetAddress addr = InetAddress.getByAddress(ipAddr);
            Socket MySocket = new Socket(addr, 10101);
            OutputStream os = MySocket.getOutputStream();
            InputStream is = MySocket.getInputStream();
            out = new PrintWriter(os, true);
            in = new BufferedReader(new InputStreamReader(is));
            System.out.println("connection successful");
            RecData = "1";
            out.println("require_color");
            RecColor = in.readLine();
        }
```

- **Server opening socket** to accept connection from Clients and awaiting communication.

```
//The server is opening by Socket 10101, all Clients will be stored
for later on.
public class new_game_server {
    public static Socket [] client_list = new Socket[5];
    public static String[][][] server_inf = new String[10][10][2];
    public static String[][] client_port_list = new String[5][3];
    public static void main(String[] args) throws Exception
    {
        ServerSocket server = new ServerSocket(10101);
        Socket client;
        Thread client_handler;
        int player_count=0;
        while(true) {
            if (player_count >= 5)
                break;
            client_list[player_count] = server.accept();
            player_count++;
            game_thread_handler task = new
game_thread_handler(client_list,
player_count,server_inf,client_port_list);
            client_handler = new Thread(task);
            client_handler.start();
// These threads is to work on sending message and receiving message
        }
    }
}
```

- **Server accessing the shared object** using semaphores for safety and data accuracy to set the squares to selected on receiving information from the clients with regards to which square they are attempting to select. Using semaphores here was important to avoid race conditions and other data integrity issues. Amandeep pointed us in the right direction here and we are grateful for that.

```
//The server fills the square within a 3D array with a semaphore
// check for 3D array in [x,y], whether it is filled.
// if [x,y] is empty, send "Yes" to allow the player_X to fill box and
bookkeeping the 3D array
            if (recv_msg.charAt(0) == '!') {
                String coordinateXY=recv_msg.substring(1);
                System.out.println(coordinateXY);
                int index = coordinateXY.indexOf(":");
                String tempX=coordinateXY.substring(0,index);
                int X = Integer.parseInt(tempX);
                String tempY = coordinateXY.substring(index+1);
                int Y = Integer.parseInt(tempY);
                int newX = (X - 2) / 98;
                int newY = Y / 90;
                boolean isFill = false;
//able to fill the grid
                if (server_inf[newX][newY][0] != "fill"){
                    try {
                        semaphore.acquire();
                        fill_number ++;
                        server_inf[newX][newY][0] = "fill";
                        server_inf[newX][newY][1] = client_ip_port;
                        System.out.println( "new coordinate : " + newX
+ " " + newY);
                        System.out.println("able to fill
!!!!!!!!!!!!!!!!!!!!!!!!!!!!");
                        isFill = true;
                        semaphore.release();
                    } catch (InterruptedException e) {
                        throw new RuntimeException(e);
                    }
                }
//the grid is  filled
             else{
                    System.out.println("fail to fill
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!");
                }
// sending back the packet if it is able to fill, tell client to
change the color of squre
                if(isFill){
                    String currColor = null;
                    for(int i = 0; i < player_count ; i++)
                    {
                        if(client_list[i].equals(client))
                        {
                            currColor = color[i];
                        }
                    }
                    System.out.println("player_count: " +
player_count);
                    String reply_msg = "Yes !"+recv_msg.substring(1)+"
is filled by the client" + client_ip_port + " which color is
@"+currColor;
                    try {
                        semaphore.acquire();
                    } catch (InterruptedException e) {
                        throw new RuntimeException(e);
                    }
                    for(int i = 0; i < player_count ; i++) {
                        OutputStream os_temp = null;
                        InputStream is_temp = null;
                        PrintWriter out_temp = null;
                        try {
                            os_temp =
client_list[i].getOutputStream();
                            out_temp = new PrintWriter(os_temp, true);
                        } catch (IOException e) {
                            throw new RuntimeException(e);
                        }
                        String tmp_client_ip_port =
client_list[i].getRemoteSocketAddress().toString();
                        System.out.println("i=" + i + " sending: " +
reply_msg + " to " + tmp_client_ip_port);
                        out_temp.println(reply_msg);
                    }
                    semaphore.release();
                }
```

Group Member Contributions –

All group members contributed equally to the project

Closing remarks –

- **Commented source code** has been included with the report submission as requested.
- **A video demonstrating the function** of our game has been included with the report submission as requested.

Special thanks to TA Amandeep Dania for help understanding and fulfilling requirements over email! Look forward to having her as a TA again in the future.