CMPT 371: DATA COMMUNICATIONS AND NETWORKING

Final Project Report

by

Group 13

Jiadi Luo (jiadil@sfu.ca)
Adrian Marin Estrada (ama269@sfu.ca)
Jay Esquivel (bde@sfu.ca)
Kayla Lee (ke7@sfu.ca)
Harpreet Dubb (hdubb@sfu.ca)

Content

Game Description

Divide and Conquer is a real-time multiplayer game written in Java and features an interactive nxn grid board (by default 4x4) where players compete to competitively conquer lockable cells via colouring. The objective of the players is to fill as many cells as possible with their respective colours (Client 1 - Pink and Client 2 - Gray) while preventing the opponent from claiming more cells. The player with the most filled cells at the end wins.

Each of the 16 cells on the board can be coloured by players using their mouse through a graphical user interface. The game follows a client-server model, with two clients connected to a server. Each client controls in a different colour and competes by colouring the cells on the board. While a client is colouring a particular cell, the cell will be locked and other players will be unable to draw within that same cell. The clients use a simple GUI to draw on the grid, and the server updates the game state based on their actions.

In order to conquer a free box/cell, a player must click-and-hold inside the box while scribbling in it. Upon releasing the click button, the game calculates the percentage of the area that is coloured. If the coloured area is more than the set threshold, the player successfully takes over the box, and the entire box changes its colour to the colour of the conquering player. Once conquered, no other player can take over that box anymore.

However, if the coloured area is less than the threshold and the player’s attempt to take over the box fails the game immediately clears the box, changing its appearance back to pure white and making the box available for other players to conquer. A player may fail to take over the box by either releasing the click button before colouring more than the threshold amount, or by drawing outside of the box boundaries. Should a player draw outside of the boundaries of the box, then the capturing of the box fails regardless of how much of the box has been coloured.

Game Design

The game consists of two main components: the client-side (**Client.java**) and the server-side (**Server.java**). The Client is responsible for handling user input through the GUI and sending updates to the Server. The Server manages the game state, communicates with both clients and broadcasts updates to keep the clients in sync.

Client

The Client application is implemented in the "**Client.java**" file. It is a graphical application that displays a nxn (by default 4x4) grid on the screen. The grid represents the game board, and each cell can be coloured by the two players. The Client application has the following main functionalities:

- **Initialization:** The application initializes the game board, coloured areas, and coloured pixels in each cell. It creates a GUI with a nxn (by default 4x4) grid layout based on the "***NUM_CELLS***" and "***BOARD_SIZE***", where each cell is represented by a "***JPanel***".
- **Mouse interaction:** The player interacts with the game by clicking and dragging the mouse on the cells. The Client application responds to mouse events such as mouse presses, releases, and mouse dragging. When the player performs an action, the application updates the local game state and sends the information to the Server for synchronization with the other player.
- **Cell colouring**: The Client application supports cell colouring by each player using a brush tool. The brush size and colours are predefined for each player (e.g., "***CLIENT1_colour = PINK***" for player 1 and "***CLIENT2_colour = GREY***" for player 2).
- **Threshold check**: When the player finishes drawing in a cell, the application checks whether the cell is filled above a certain threshold (defined by "***colour_THRESHOLD***"). If the threshold is reached, the cell is considered claimed by the respective player.
- **Communication with the server:** Communication with the server in the game involves the Client application establishing a socket connection with the Server. The Client sends updates to the Server regarding the actions taken by the player, such as clicking and colouring cells. Additionally, it receives updates from the Server about the actions of the other player and promptly updates the game board based on these updates. To support multiple players, the Server utilizes multi-client threads. Each connected Client is handled by a separate thread on the Server, enabling concurrent communication with all players. This multi-threaded approach ensures that each player's actions are processed independently and in real-time, providing a seamless and interactive multiplayer experience.
- **Game result**: When the game is over and all the cells on the board are claimed or filled, the Client application receives the game result from the Server and displays a message to the player.

Server

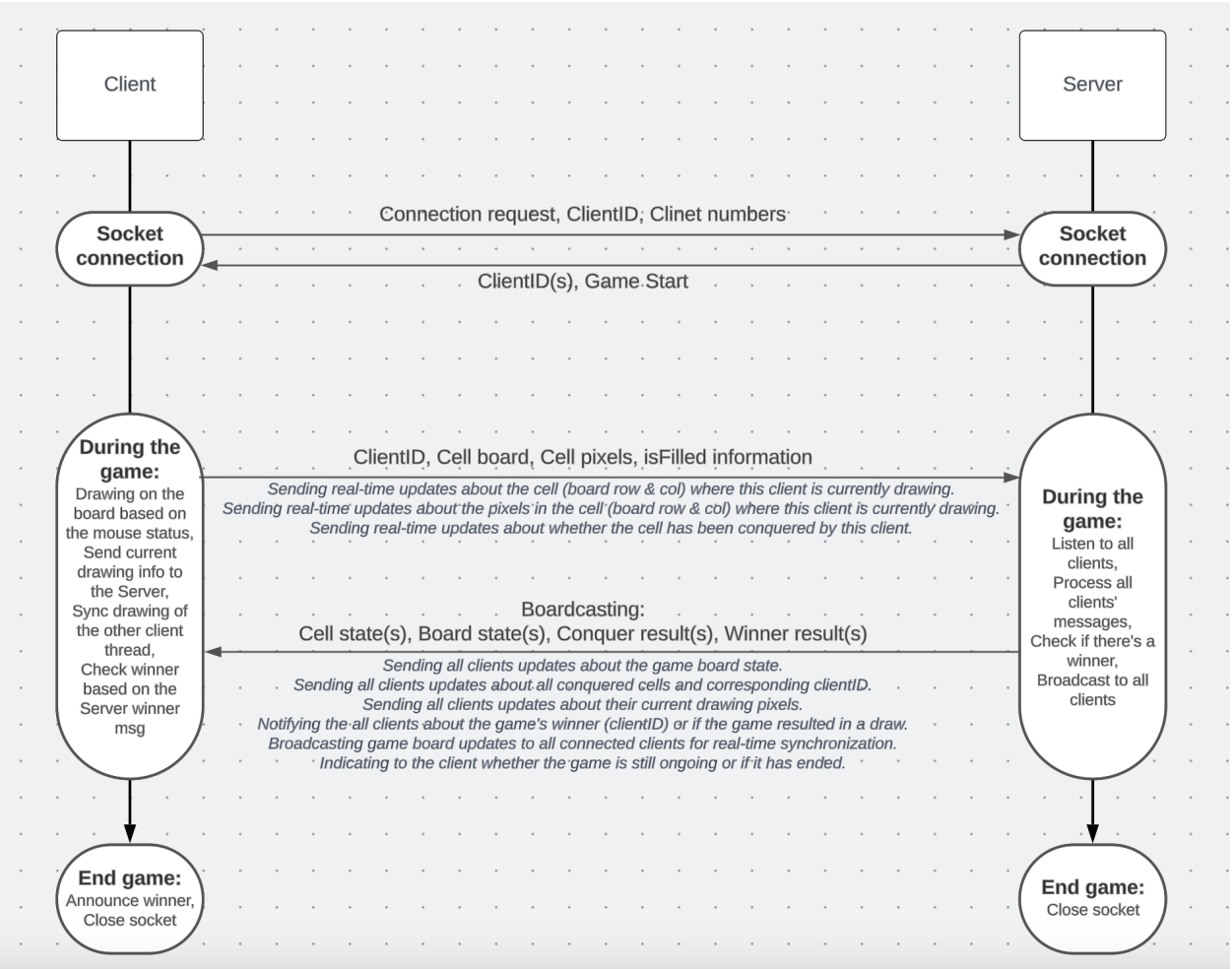
The Server application is implemented in the **"Server.java"** file. It manages the game state and communicates between the two clients. The Server application has the following main functionalities:

- **Initialization**: The Server initializes the game state by creating a nxn (by default 4x4) "*board*" array and starting a server socket to listen for incoming client connections.
- **Client connection**: The server waits for n (set to 2 in this project by using **"*MAX_CLIENTS*"**) clients to connect and assigns them unique client IDs (1 or 2). The clients are managed in separate threads, enabling concurrent communication.
- **Synchronization**: The Server receives updates from each client regarding the cells they have coloured and broadcasts these updates to the client. The server ensures that both clients see the same game state and remain in sync throughout the game.
- **Game result**: The Server continuously checks the game board for a winner or a draw when all cells are claimed or filled. When the game is over, the server broadcasts the results to both clients, indicating whether they won, lost, or the game ended in a draw.
- **Broadcasting**: The Server broadcasts the game state updates to both clients, so they can update their local representation of the game board.

Application Layer Messaging Scheme

The application layer messaging scheme is based on a protocol where the clients and the server exchange information in a predefined format. The messaging protocol involves sending and receiving data using sockets over a TCP connection. The data is transmitted using "**DataInputStream**" and "**DataOutputStream**" and messages exchanged between the client and server have fixed formats to ensure compatibility.

Graph:



Client Messaging

When a client sends a message to the server, it includes the row, column, and clientID for each cell drawn.

- **"*row*" (int)**: The row index of the cell being drawn on.
- **"*col*" (int)**: The column index of the cell being drawn on.
- **"*clientID*" (int)**: The ID of the client performing the action (1 or 2).When a client connects to the server, it receives a client ID from the server.
- **"*x*" (int)**: The x-coordinate of the pixel of "***board[row][col]***" being drawn in the cell by the current player.

- **“*y*” (int)**: The y-coordinate of the pixel of “***board[row][col]”*** being drawn in the cell by the current player.
- **“*isFilled*” (int)**: Indicates whether or not the cell (**“*board[row][col]”***) has been filled by the current player.

The client sends these messages by wrapping them into one single token (details will be presented in the “Code Snippets” section) to inform the server about the drawing actions taken by the player. The server then uses this information to update the game board and synchronize the state with the other player. When a cell drawing is completed, the client sends a message to the server to check the threshold and claim the cell if necessary.

Server Messaging

- **"*row*" (int)**: The row index of the cell being drawn on.
- **"*col*" (int)**: The column index of the cell being drawn on.
- **"*clientID*" (int)**: The ID of the client performing the action (1 or 2).
- **“*x*” (int)**: The x-coordinate of the pixel of “***board[row][col]****”* being drawn in the cell by the player.
- **“*y*” (int)**: The y-coordinate of the pixel of “***board[row][col]”*** being drawn in the cell by the player.
- **“*isFilled*” (int)**: Indicates whether or not the cell (“***board[row][col]****”*) has been filled by the player.
- **"*winner*" (int)**: The ID of the winning client (1 or 2), or -1 for a draw.

By broadcasting these messages to all clients, the server ensures that all clients receive real-time updates about the game state, including the actions of the current player and other players. The broadcast updates enable synchronized gameplay across all connected clients in the "Divide and Conquer" game.

Game Result Message

When the game is over and a winner or a draw is called, the server sends a special message to both clients indicating the game result:

- **"*winner*" (int):** The ID of the winning client (1 or 2), or -1 for a draw

When the game result message is received, the clients display their outcome to the player in a popup message. This can be win, lose, or draw.

Code Snippets

Opening Sockets

Opening sockets - Client

In the "**Client.java**" file, the "***connectServer()***" method is responsible for opening a socket and connecting to the server.

1. The **"*Client*"** class initializes a **"*Socket*"** object named **"*client*"**, which represents the client-side of the connection to the server.
2. The client tries to establish a connection to the server by specifying the server's address (in this case, “localhost”) and the port number (7070) in the **"*Socket*"** constructor. The **"*client = new Socket(“localhost”, 7070)”***; line accomplishes this.
3. Once the connection is established, the client obtains the input and output streams of the socket ("***DataInputStream***" and "***DataOutputStream***") to send and receive data to and from the server.
4. The client also reads its unique client ID from the server using "***in.readInt()*****".** The server assigns clientIDs (1 or 2) to the connected clients, and each client receives its ID upon connection. After receiving the clientID, the method prints a message indicating the successful connection to the server and the assigned clientID. If the clientID is 1, it also prints a message indicating that the client is waiting for another player to connect.
5. In case of any connection issues or errors, the method catches the IOException, prints an error message, and terminates the application with an exit code of -1.

The "***connectServer()***" method establishes a connection to the server, initializes input and output streams for communication, and receives a clientID from the server, allowing the client to participate in the multiplayer game:

```
public void connectServer() {
    try {
        client = new Socket("localhost", 7070);

        out = new DataOutputStream(client.getOutputStream());
        in = new DataInputStream(client.getInputStream());

        clientID = in.readInt();
        System.out.println("You are client #" + clientID + ", you are connected to the server!");
        if (clientID == 1) {
            System.out.println("Waiting for another client to connect...");
            showStartScreen();
        }
    } catch (IOException e) {
        System.out.println("Could not connect to server");
        System.exit(-1);
    }
}
```

The "***main***" method in the Client class sets up the client for the game, connects it to the server, initializes the GUI, starts a thread to synchronize with the server, and periodically sends pixel information to the

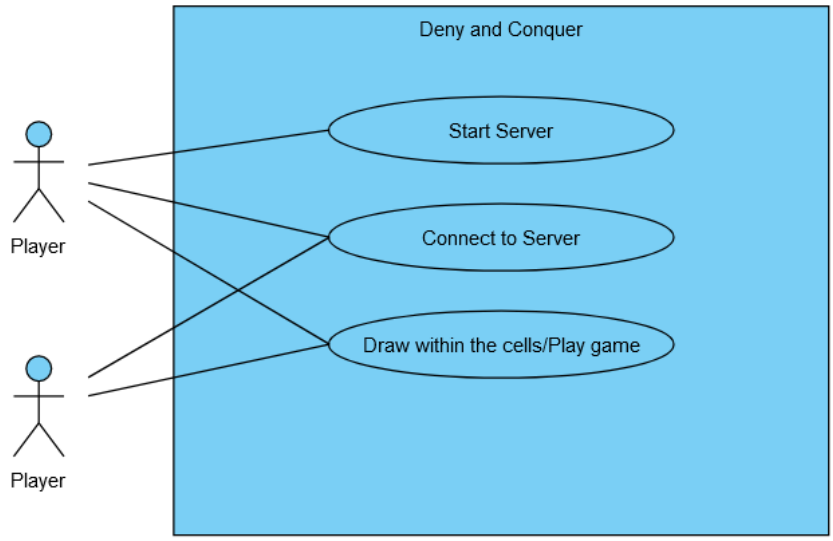
server for real-time gameplay updates:

```
// Main
Run | Debug
public static void main(String[] args) throws IOException {
    Client client = new Client(); // Create a client object
    client.connectServer(); // Connect to the server

    new Thread(client.new SyncServer()).start(); // Start a thread for receiving messages from the server

    // Periodically send pixelInfoList to server
    Timer timer = new Timer(delay:10, e -> {
        client.sendPixelInfoListToServer();
    });
    timer.start();

    // close connection
    Runtime.getRuntime().addShutdownHook(new Thread(() -> {
        client.closeSocket();
    }));
}
```



Opening sockets - Server

In the "**Server.java**" file, the "***newServer()***" method is responsible for initializing the server side of the connection.

1. The "***Server***" class initializes a "***ServerSocket***" object named "***server***", which represents the server's endpoint that waits for client connections on a specified port (in this case, 7070).
2. Once the server socket is created, it waits for incoming connections using the "***accept()***" method. When a client connects to the server, the "***accept()***" method returns a new "***Socket***" object representing the client-side of the connection.

3. The server assigns a unique client ID (1 or 2) to each connected client and manages them in separate threads for concurrent communication by creating a new thread (***SyncClients***) to handle the communication with that specific client independently. The thread takes the client's clientID and input stream (in) as parameters. This "***SyncClients***" thread is responsible for managing updates and interactions from the client, allowing the server to handle multiple client connections simultaneously.
4. In case of any connection issues or errors during the process, the method catches the IOException, prints an error message, and terminates the server application with an exit code of -1.

The "***connectClients()***" method in the server class sets up the server to accept and connect multiple clients for the game, assigns unique clientIDs to each client, creates separate threads for handling communication with each client, and prints messages indicating client connections:

```
public void connectClients() {
    try {
        while (numClients < MAX_CLIENTS) {
            // accept the client connection
            Socket client = server.accept();

            // increment the number of clients
            numClients++;

            // create the input and output streams
            DataOutputStream out = new DataOutputStream(client.getOutputStream());
            DataInputStream in = new DataInputStream(client.getInputStream());

            // send the client number to the client
            out.writeInt(numClients);
            System.out.println("Client #" + numClients + " has connected to the server!");

            if (numClients == 1) { // first client
                client1 = client;
                out1 = out;
            } else { // second client
                client2 = client;
                out2 = out;

                // send the game start message to both clients: initialize the start screen
                sendGameStartMessage();
            }

            // create a new thread to handle the client
            new Thread(new SyncClients(in)).start();
        }
    } catch (IOException e) {
        System.out.println(x:"Accept failed: 7070");
        System.exit(-1);
    }
}
```

The "***main***" method in the Server class initializes the server for the game, creates a new server instance, sets up the server to listen for incoming connections, and connects multiple clients to the server for real-time multiplayer gameplay. Additionally, it ensures that the server's resources are properly released and any cleanup is performed when the program is terminated.

```
Run | Debug
public static void main(String[] args) throws IOException {
    Server server = new Server(); // create a new server
    server.newServer(); // start the server
    server.connectClients(); // connect the clients

    // close the server
    Runtime.getRuntime().addShutdownHook(new Thread(() -> {
        server.closeServer();
    }));
}
```

Handling the shared object

In the game, the game board is a shared object between the clients and the server. The game board is represented by a 2D array called "***board***", which keeps track of the coloured cells and their ownership.

Handling the shared object - Client

When a player interacts with the GUI and colours a cell, the "***paintCell()***" method is called.

1. The client updates its local copy of the game board ("***board***") to represent the colour changes made by the player.
Update board information based on client's behaviour:

```
private void updateBoard() {
    for (int i = 0; i < NUM_CELLS; i++) {
        for (int j = 0; j < NUM_CELLS; j++) {
            // Create a JPanel for each cell
            JPanel cell = new JPanel();
            cell.setBorder(BorderFactory.createLineBorder(Color.BLACK, 1));
            final int row = i;
            final int col = j;
```

Different mouse motion listeners to fulfill the drawing and "let go" feature:

```
// Add a MouseListener to the JPanel (for clicking, releasing, and exiting)
cell.addMouseListener(new MouseAdapter() {
    @Override
    public void mousePressed(MouseEvent e) {
        paintCell(cell, row, col, e.getX(), e.getY());
        isMouseInsideCell = true;
    }

    @Override
    public void mouseReleased(MouseEvent e) {
        checkThreshold(cell, row, col, e.getX(), e.getY(), false);
        isMouseInsideCell = false;
    }

    public void mouseExited(MouseEvent e) {
        isMouseInsideCell = false;
    }
});

// Add a MouseMotionListener to the JPanel (for dragging)
cell.addMouseMotionListener(new MouseMotionAdapter() {
    @Override
    public void mouseDragged(MouseEvent e) {
        if (isMouseInsideCell) {
            paintCell(cell, row, col, e.getX(), e.getY());
        } else {
            checkThreshold(cell, row, col, e.getX(), e.getY(), true);
        }
    }
});

// Add the JPanel to the board
boardPanel.add(cell);
```

The normal drawing behaviour will call the "***paintCell***" method, which colours a cell on the game board based on the player's brush movement, tracks the coloured pixels for each cell, and updates the pixel information list for the server based on the cell's filing status:

```
private void paintCell(JPanel cell, int row, int col, int x, int y) {
    if ((board[row][col] == 0 || board[row][col] == -1) && (boardCurrentStatus[row][col] == clientID || boardCurrentStatus[row][col] == 0)) {
        // System.out.println("boardCurrentStatus[" + row + "][" + col + "] = " + boardCurrentStatus[row][col]);
        int cellWidth = cell.getWidth();
        int cellHeight = cell.getHeight();
        int cellArea = cellWidth * cellHeight;

        Color brushColor = clientID == 1 ? CLIENT1_COLOR : CLIENT2_COLOR;

        // Draw on the JPanel (for display purposes)
        Graphics boardImage = cell.getGraphics();
        boardImage.setColor(brushColor);
        boardImage.fillRect(x, y, BRUSH_SIZE, BRUSH_SIZE);

        // Create a BufferedImage to store the drawing (for tracking purposes)
        BufferedImage virtualImage = new BufferedImage(cellWidth, cellHeight, BufferedImage.TYPE_INT_RGB);
        Graphics2D virtualImageCanvas = virtualImage.createGraphics();
        cell.paint(virtualImageCanvas);
        virtualImageCanvas.setColor(brushColor);
        virtualImageCanvas.fillRect(x, y, cellWidth / 10, cellHeight / 10);

        // Check if the pixel is already colored, if not, color it
        for (int i = 0; i < cellWidth; i++) {
            for (int j = 0; j < cellHeight; j++) {
                if (!coloredPixels[row][col][i][j]) {
                    if (virtualImage.getRGB(i, j) == brushColor.getRGB()) {
                        coloredPixels[row][col][i][j] = true;
                        coloredArea[row][col]++;
                    }
                }
            }
        }

        // System.out.println("coloredArea: " + coloredArea[row][col] + ", cellArea: " + cellArea);

        // // Check if the cell is filled >= threshold
        int isFilled = 0;

        // Add pixel information to the list, to be sent to the server
        pixelInfoList.add(new int[]{row, col, clientID, x, y, isFilled});
    }
}
```

However, if the mouse is released by the player before it reaches the cell threshold or if the player draws outside the cell boundaries, a different action will be taken. Refer to the section below about "***checkThreshold***" for more details.

2. Additionally, it also keeps track of the number of coloured pixels in each cell using the "***colouredArea***" array and the "**colouredPixels**" 2D boolean array. These arrays store information about the coloured pixels in each cell and help in calculating when a cell is considered claimed (i.e., when the threshold is reached).

The "***checkThreshold***" method verifies whether a cell has legally reached the filling threshold based on the coloured area before the player releases the mouse. If yes, the cell will be assigned "***isFilled = clientID***", which means it has been occupied by the current player. If the threshold has not been reached upon the releasing of the mouse or if a player illegally draws outside of the bounds of the box, the

method clears the cell and resets it to its initial state, making it available for others to draw upon:

```
// Operation: let go feature
private void checkThreshold(JPanel cell, int row, int col, int x, int y, boolean mouseExitedCell) {
    int cellWidth = cell.getWidth();
    int cellHeight = cell.getHeight();
    int cellArea = cellWidth * cellHeight;

    // Check if the cell is filled >= threshold
    if ((board[row][col] == 0 || board[row][col] == -1) && (boardCurrentStatus[row][col] == clientID || boardCurrentStatus[row][col] == 0)) {
        int isFilled = 0;
        int belongsTo = clientID;
        if ((coloredArea[row][col] >= cellArea * COLOR_THRESHOLD) && !mouseExitedCell) { // if legally filled
            isFilled = clientID;
        } else { // if not filled, clear cell
            coloredArea[row][col] = 0;
            cell.removeAll();
            cell.revalidate();
            cell.repaint();
            isFilled = -1;
            belongsTo = 0;
            for (int i = 0; i < cellWidth; i++) { // flush coloredPixels
                for (int j = 0; j < cellHeight; j++) {
                    coloredPixels[row][col][i][j] = false;
                }
            }
        }
        pixelInfoList.add(new int[]{row, col, belongsTo, x, y, isFilled}); // add pixel info to list
    }
}
```

3. After a cell drawing is completed or undone, the client sends pixel information (row, column, client ID) to the server for synchronization. The pixel information is stored in the **"*pixelInfoList*"**, which is periodically sent to the server.
Defined **"*pixelInfoList*"** list**:**

```
private final List<int[]> pixelInfoList = new ArrayList<>();
```

The "***sendPixelInfoListToServer()***" method in the Client class plays a role in exchanging pixel-related data with the server. Pixel information, encompassing row, column, client ID, pixel coordinated, and cell filling status, is extracted from the “**pixelInfoList" contained** and compiled into a structured string, “**tokenizedMessage"**. This string is then transmitted to the server through an output stream (out). This process ensures real-time synchronization with the server, facilitating multiplayer gameplay updates and interactions between clients:

```
private void sendPixelInfoListToServer() {
    try {
        // save each pixel info to a string
        StringBuilder tokenizedMessage = new StringBuilder();
        // Send each pixel information to the server
        for (int[] pixelInfo : pixelInfoList) {
            tokenizedMessage.append(pixelInfo[0]).append(";"); // row
            tokenizedMessage.append(pixelInfo[1]).append(";"); // col
            tokenizedMessage.append(pixelInfo[2]).append(";"); // clientID
            tokenizedMessage.append(pixelInfo[3]).append(";"); // x
            tokenizedMessage.append(pixelInfo[4]).append(";"); // y
            tokenizedMessage.append(pixelInfo[5]).append("#"); // isFilled
        }

        // send the string to server
        out.writeUTF(tokenizedMessage.toString());

        out.flush(); // Flush the output stream to ensure all data is sent
        pixelInfoList.clear(); // Clear the pixelInfoList
    } catch (IOException ex) {
        ex.printStackTrace();
    }
}
```

Periodically send this list to server:

```
// Periodically send pixelInfoList to server
Timer timer = new Timer(10, e -> {
    client.sendPixelInfoListToServer();
});
timer.start();
```

4. After sending the pixel information to the server, the client application reads messages from the server, synchronizing the drawing actions performed by other players. Upon receiving updates, the client updates the game board accordingly, displaying the drawings made by other players in real-time, facilitating a shared and interactive gaming experience.

The "***SyncServer***" method receives all messages from the server and syncs the drawing behaviour:

```
private class SyncServer implements Runnable {
    public void run() {
        try {
            while (true) {
                // read the string from server
                String tokenizedMessage = in.readUTF();

                // if the game is not started and the message is "start", start the game
                if (tokenizedMessage.equals("start") && !isGameStarted) {
                    if (startScreen != null) {
                        startScreen.dispose();
                    }
                    clientGUI(clientID);
                    isGameStarted = true;
                }
```

```
// if the game is started, read the tokenized message
else {
    // tokenize the string
    String[] tokens = tokenizedMessage.split("#");
    String[] lastToken = tokens[tokens.length - 1].split(";");

    // read the token if it is valid
    if (lastToken.length == 6) {
        // System.out.println("lastToken: " + lastToken[0] + ", " + lastToken[1] + ", " + lastToken[2] + ", " + lastToken[3] + ", " + lastToken
        int currentRow = Integer.parseInt(extractNumericDigits(lastToken[0]));
        int currentCol = Integer.parseInt(extractNumericDigits(lastToken[1]));
        int currentClientID = Integer.parseInt(extractNumericDigits(lastToken[2]));
        int currentX = Integer.parseInt(extractNumericDigits(lastToken[3]));
        int currentY = Integer.parseInt(extractNumericDigits(lastToken[4]));
        int currentIsFilled = Integer.parseInt(lastToken[5]);

        System.out.println("currentClientID: " + currentClientID + " is drawing on " + currentRow + ", " + currentCol + " at " + currentX + ",

        board[currentRow][currentCol] = currentIsFilled;
        boardCurrentStatus[currentRow][currentCol] = currentClientID;
        winner = checkWinner();
```

```
SwingUtilities.invokeLater(() -> {
    // draw on board/cell
    JPanel cell = (JPanel) boardPanel.getComponent(currentRow * NUM_CELLS + currentCol);
    Graphics boardImage = cell.getGraphics();
    boardImage.setColor(currentClientID == 1 ? CLIENT1_COLOR : CLIENT2_COLOR);
    boardImage.fillRect(currentX, currentY, BRUSH_SIZE, BRUSH_SIZE);


// if player released before threshold, clear cell
if (currentIsFilled == -1) {
    System.out.println("SYNC Removing drawnlines");
    cell.removeAll();
    cell.revalidate();
    cell.repaint();
}

// fill cell if passed threshold
if (currentIsFilled != 0 && currentIsFilled != -1) {
    boardImage.fillRect(0, 0, cell.getWidth(), cell.getHeight());
}
```

It also announces the winner if a winning condition is met:

```
// announce winner
if ((winner == -1 || winner == 1 || winner == 2) && !isGameTerminated) {
    isGameTerminated = true;
    if (winner == clientID) {
        printGameResult("You win!");
    } else if (winner == -1) {
        printGameResult("Draw!");
    } else {
        printGameResult("You lose!");
    }
}
```

The "***checkWinner***" method determines the winner based on the filled cells on the game board, returning 0 for no winner, 1 for client 1, 2 for client 2, and -1 for a draw.

```
private int checkWinner() {
    // check if the board is full
    for (int row = 0; row < NUM_CELLS; row++) {
        for (int col = 0; col < NUM_CELLS; col++) {
            if (board[row][col] == 0 || board[row][col] == -1) { // the board is not full, no winner yet
                return 0;
            }
        }
    }

    // check if there is a winner
    int client1Count = 0;
    int client2Count = 0;
    for (int row = 0; row < NUM_CELLS; row++) {
        for (int col = 0; col < NUM_CELLS; col++) {
            if (board[row][col] == 1) {
                client1Count++;
            } else if (board[row][col] == 2) {
                client2Count++;
            }
        }
    }

    if (client1Count > client2Count) {
        return 1;
    } else if (client1Count < client2Count) {
        return 2;
    } else { // draw
        return -1;
    }
}
```

5. Once the game result is declared, the game ends and the socket is closed.

```
private void closeSocket() {
    try {
        if (client != null) {
            System.out.println("Closing client socket...");
            client.close();
            in.close();
            out.close();
        }
    } catch (IOException e) {
        e.printStackTrace();
    }
}
```

Handling the shared object - Server

The server maintains the gameboard and ensures that both clients see the same game state. It broadcasts updates to keep the clients in sync.

1. When the server receives pixel information from a client, it updates its copy of the game board (**"board"**) with the new information. This way, the server maintains the current state of the game.

The "***SyncClients***" class implements a thread that continuously reads messages from a client, updating the game board by broadcasting the received pixel information to all clients.

```
private class SyncClients implements Runnable {
    private DataInputStream in;

    public SyncClients(DataInputStream in) {
        this.in = in;
    }

    public void run() {
        try {
            while (true) {
                // receive the string
                String str = in.readUTF();
                // if string is empty
                if (!str.equals("")) {
                    System.out.println("Received: " + str);
                    broadcastUpdate(str);
                }
            }
        } catch (IOException ex) {
            ex.printStackTrace();
        }
    }
}
```

2. After updating the game board, the server broadcasts the updated cell information (row, column, client ID) to both clients. It does this by sending the information through their respective output streams (**"out1"** and **"out2"**).

3. The "***broadcastUpdate***" method updates the game board by sending pixel information to both connected clients.

```
private void broadcastUpdate(String str) {
    if (client1 != null && client2 != null) {
        try {
            //write back
            out1.writeUTF(str);
            out2.writeUTF(str);

            out1.flush();
            out2.flush();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

4. Upon receiving the updated cell information from the server, both clients update their local game boards to match the server"s state (done on the client side). This ensures that both players see the same board with synchronized colours.

5. Close the server once done.

```
private void closeServer() {
    try {
        if (server != null) {
            out1.close();
            out2.close();
            client1.close();
            client2.close();
            server.close();
            System.out.println("Server closed.");
        }
    } catch (IOException e) {
        System.out.println("Could not close server.");
        e.printStackTrace();
    }
}
```

Group Contribution

Group Member	% Contribution
Jiadi Luo	20
Adrian Marin Estrada	20
Jay Esquivel	20
Kayla Lee	20
Harpreet Dubb	20

Github Repository

https://github.com/MarinEstrada/cmpt371_Divide_and_Conquer

Demo Link

https://drive.google.com/file/d/1V-U8bwtCIGmnlPYW5cCZNUvsP1jTNSAc/view?usp=sharing