Drawing Guessing Game
Martin Engström

1 Summary

The Drawing Guessing Game is a game where you try to guess what a person is drawing with a nice but simple interface. The clients connect to a central server which handles the clients guesses, and relays chat messages between the clients through a TCP connection. The server also relays the painter's drawing to other clients through unicast UDP traffic. The client can choose a username, receive points for guessing correctly, and the server deals with client connection and disconnection seamlessly to make the program easy to use. The architecture of the system is fairly robust, making it capable to host many clients over the internet.

2 Development Process

The idea of the project was conceived when I finished assignment 2.2.1 and realized that it'd be fun to create the classic drawing guessing game, with inspiration taken from other games such as Drawize.

2.1 Initial Idea

Initially, I explored the possibility of using UDP multicasting to handle the game mechanics entirely on the client side. However, multicast traffic would likely be blocked across different networks so it wasn't used since the game would be more useful if it could be played over the internet.

To improve performance, I considered replacing the Swing-based Paper class with a framework like JavaFX, as drawing is the system's bottleneck, but this seemed too complex for the scope of this project after further inspection.

2.2 Development Plan

The goal for this project was to create a multi-threaded program where players can chat with each other, get points for guessing the secret word correctly, be able to paint the secret word, and have rounds of game-play with some time limit in place.

I created a simple development plan to implement a server that:

- uses TCP connections, ensuring reliable transmission of important messages, such as guesses and system updates, which must not be lost or arrive out of order.
- uses unicast UDP to handle drawing data, offloading traffic from the TCP socket.
- chooses a client to be the painter and sends them a secret word to paint.
- gives other clients points if they guess the secret word correctly.

Additionally, a client would be implemented that:

- communicates with the server and sends drawn points and messages to be broadcasted.
- allows for guessing/chatting and displays messages, usernames, and player points.
- uses the Swing based drawing framework.

The variables to be used would be decided at a later stage, but a simple sketch of the classes I had in mind at this point can be seen in Figure 1.

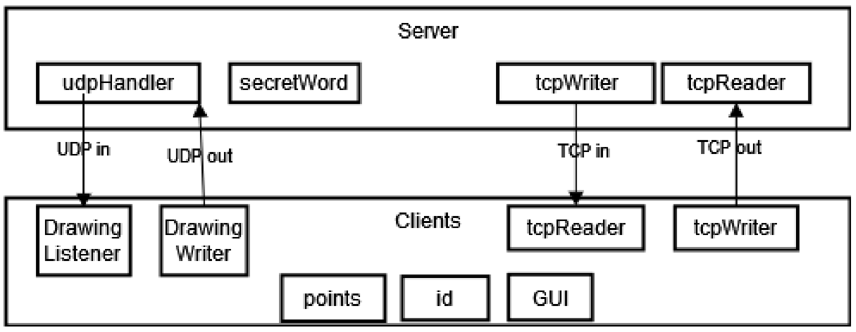


Figure 1: Initial Idea of the server-client system

2.3 Resolving Network Issues

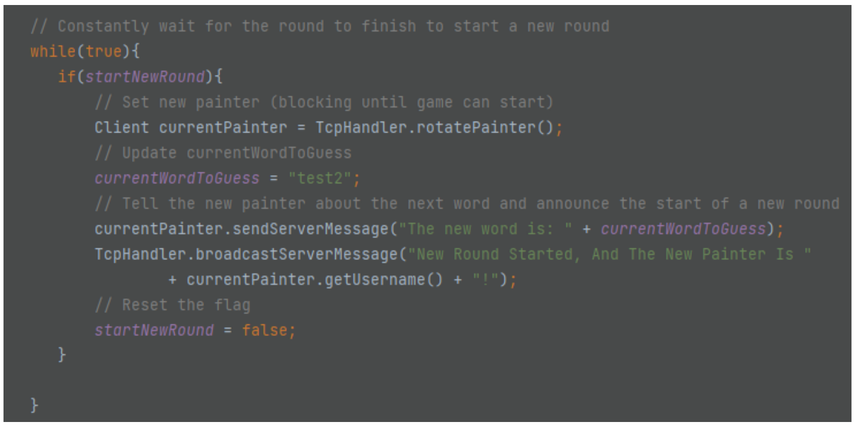
A critical component of the development was to ensure that the server could receive messages from both local and remote clients, so I started with a basic server that listened for clients sending dummy TCP and UDP messages at regular intervals.

The server was tested on a local machine, with one client on a different local computer and another on a remote desktop. While the local client's messages arrived, the remote client's did not. This was fixed by forwarding the ports 5000 and 5001 in my router, for TCP and UDP traffic respectively, allowing me to proceed with the main game development.

2.4 Minimal Server

To start, I developed a minimal server solution with the following classes:

- Server - Initialized server and contained the barely implemented game loop, where a new painter was chosen and sent the secret word. This would then loop if the secret word was guessed correctly, see Figure 2.
- Client - Contained all client information and the TCP socket to allow other classes to send messages through it.
- TcpHandler - Created clients, handled the list of clients, and creating a new thread for each client to listen for messages that they sent and then handle those messages.
- UdpHandler - Simple handler that merely echoed the traffic to all clients.



```
// Constantly wait for the round to finish to start a new round
while(true){
    if(startNewRound){
        // Set new painter (blocking until game can start)
        Client currentPainter = TcpHandler.rotatePainter();
        // Update currentWordToGuess
        currentWordToGuess = "test2";
        // Tell the new painter about the next word and announce the start of a new round
        currentPainter.sendServerMessage("The new word is: " + currentWordToGuess);
        TcpHandler.broadcastServerMessage("New Round Started, And The New Painter Is "
                + currentPainter.getUsername() + "!");
        // Reset the flag
        startNewRound = false;
    }

}
```

Figure 2: Initial game loop

In the TcpHandler, the messages were divided into specific categories: "/g " for guesses, "/m " for regular messages, and "/s " for critical system messages that required client system action. For instance, the username command was used during the initial handshake that enabled clients to set their username. Since the TcpHandler handled all the clients, it also contained the functions to broadcast messages to all clients and to rotate the painter role.

The client program at this point was barely functional and only sent hard coded responses to the server during debugging, making it possibly to verify that the communication was functioning correctly.

The game was almost working at this point and only a few additions had to be made to get a complete game, but I was unhappy with the system architecture as it provided poor separation of responsibility which both looked bad and bogged the system down as the threads in the TcpHandler and Server classes competed to make use of the socket in the Client class. A better solution for this was implemented at a later stage, but it was functional enough to test the game.

2.5 Minimal Client

Next, I extended the client primarily to finish the message logic, such as sending and receiving correctly formatted drawn points. To ensure that only the current painter could draw, I added checks on both the client and server sides to verify that the client is the current painter before allowing them to draw.

The client could also categorize the message types to handle them separately: Regular messages would be printed out, and some system messages/commands were implemented, namely the initial handshake for setting a username and accepting an ID, the handshake to accept the painter role, the start of new rounds, and the receiving of the secret word to draw.

As for the interface, a minimal GUI with only a drawing area was implemented. It was also verified that a client could send messages and guesses through the terminal and receive points from the server for correct guesses.

2.6 Final Solution

All the features in the development plan were now technically implemented, so it was time to refine the system for a better user experience and improved architecture. Development became more iterative during this phase as I regularly switched between server and client side development, fixing and improving one feature at a time.

2.6.1 Communication

I implemented additional system commands to handle starting and ending drawing rounds in the correct sequence, as well as allowing for new clients to join games mid-round. Previously I had used hard coded strings for all the system messages, but at this time I created the SystemUtility class that defined system message formats to bridge the logic between the client and server side code and make it easier for future modification.

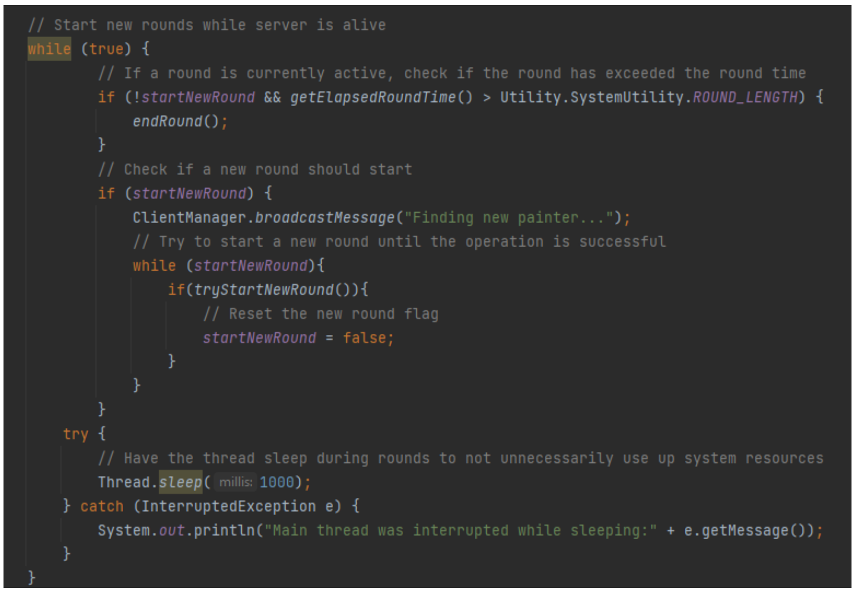
2.6.2 Server

In a failed attempt to improve thread safety and responsibility separation, I created separate TcpClient and UdpClient classes to replace the Client class to separate the network responsibilities for each client. This seemed like a good approach at first, but I realized that it didn't properly address the core issues. The threads were still doing too much work and competing for resources, even if it was slightly better separated.

To fix this, I removed all network logic from the Client class and gave them unique message buffers that could be used for adding messages that were to be sent to the client, and appending received ACK-responses from each client. This way, instead of having unrelated threads directly handling network logic, they would instead push messages to the buffers which would speed them up and better separate the responsibilities. This greatly simplified the Client class structure, reduced network error handling, and reduced resource contention.

I then tried to move the list of clients to the Server (now called DrawServer) class which made slightly more sense, but in the end opted for creating a ClientManager that has all the clients and the logic for interacting with them that was previously located in the TcpHandler, as previously mentioned in Section 2.4. By putting the ClientManager in the same package as the DrawServer, I could also protect the rotate painter logic that was only to be used by that class.

Previously, the selection of the next painter contained a loop that blocked access to the clients list so clients couldn't be added or removed during this process which I realized would break the program in some circumstances. To fix this I changed the logic so that if the next painter could not be found within a few seconds it wouldn't instantly try again, this way it would allow for other threads waiting for the clients list to be freed up to gain access to it in between attempts. The game loop of the server was now completed, see Figure 3.



```
// Start new rounds while server is alive
while (true) {
        // If a round is currently active, check if the round has exceeded the round time
        if (!startNewRound && getElapsedRoundTime() > Utility.SystemUtility.ROUND_LENGTH) {
            endRound();
        }
        // Check if a new round should start
        if (startNewRound) {
            ClientManager.broadcastMessage("Finding new painter...");
            // Try to start a new round until the operation is successful
            while (startNewRound){
                if(tryStartNewRound()){
                    // Reset the new round flag
                    startNewRound = false;
                }
            }
        }
    try {
        // Have the thread sleep during rounds to not unnecessarily use up system resources
        Thread.sleep( millis: 1000);
    } catch (InterruptedException e) {
        System.out.println("Main thread was interrupted while sleeping:" + e.getMessage());
    }
}
```

Figure 3: Final game loop

Finally, I created the WordGenerator class to randomly select the next secret word to use for a new round. There was also more refactoring done that does not fit into this report, for example the TcpHandler was expanded into three different classes: TcpServerController which accepts connections, ClientConnectionHandler that further accepts clients and handles messages from them, and ClientMessageSender which is responsible for sending the messages in the previously mentioned message buffer, but a bit of this will be explained in the program description.

2.6.3 Client

I finished the GUI by implementing the chat window with an input field, placed them next to the drawing canvas, implemented a header that describes the game state with text, and created the CircularTimer that is a simple timer that shows roughly how much time there's left of the round.

In the middle of the project, the TCP and UDP writers were created inside the reader classes, and the GUI was created in the UDP reader class. This was refactored so that they were all created in the main class instead, which made it easier to release system resources before exiting the program. This was necessary in case the server for some reason closes the connection to the client, for example if they're rejected due to bad arguments, in which case the client will inform the user and shut down after 5 seconds.

I initially had all clients use port 5001 for UDP traffic since it's a program meant to be used on different computers, but I changed it to use a random available port instead, and then stored the port numbers in the client's information on the server side to allow for easier local testing. Before this, if one attempted to create two clients on the same computer, one would fail because the UDP port was unavailable.

3 Program Description

This section will describe the program from a high-level perspective. The system is quite large so most of the code will not be explained, but we will delve slightly deeper into the more important pieces of code with help of pseudo-code.

3.1 Server

The main thread tries to select a new painter from the connected clients once two or more clients are connected, and then starts a new round that runs for 120 seconds or until someone guesses the secret word correctly, this process is then repeated indefinitely. This logic is described in the pseudocode of Algorithm 1 and 2. Note that the startNewRound variable can be set to true elsewhere in the code when a round should end, for example when a client guesses the correct word, but we will not go into further detail on that here.

Algorithm 1 drawServerMainLoop

```
tcpServerController(TCP_PORT)&
udpHandler(UDP_PORT)&
while $true$ do
    if round is active and has exceeded round time then
        tell clients that the round has ended&
        startNewRound = true
    while $startNewRound == true$ do
        if tryStartNewRound() == true then
            startNewRound = false
return
```

Algorithm 2 tryStartNewRound

```
nextPainter = tryGetNextPainter()
if nextPainter != null then
    currentPainter = nextPainter
    tell clients that a new round is about to start&
    secretWord = getNextWord()
    tell currentPainter what secretWord is%
    tell clients who the currentPainter is and announce start of round%
    set start of round time to current time
    return true
else
    return false
```

Once a round is set up and running, the flow of messages occur the way it's depicted in Figure 4. All guesser-clients are able to send guesses on what the secret word is, only the painter-client is able to relay drawn data to all the guessers, and all clients are able to relay normal chat messages.

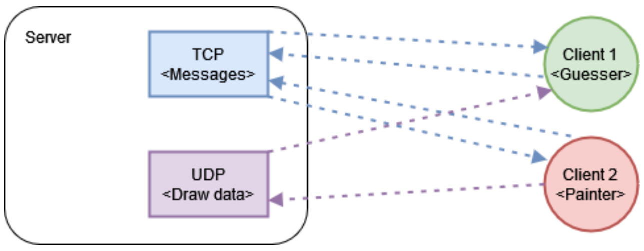


Figure 4: Diagram showing the flow of messages between server and clients during a round of game-play.

3.1.1 TCP controller

When the server is started up, it starts a controller for TCP connections. The TCP server controller accepts new clients and starts a new ClientConnectionHandler thread that expects the new client's first message to contain a username and the UDP port they will use for the session before they're added to the game. If this succeeds it continues to handle their communication, which involves dealing with system messages, relaying chat messages, and verifying the client's guesses. A thread that is dedicated to sending messages to the client that are added to their message buffer is also started here, see Algorithms 3, 4, and 5.

Algorithm 3 clientConnectionHandler

```
split client's first message into components and use it as arguments to tryAcceptClient
client = tryAcceptClient(clientArgs)
if client == null then return "IllegalArgumentException"
clientMessageSender(client)%
while new message from client can be read do
    handleIncomingClientMessage(message, client)
return
```

Algorithm 4 handleIncomingClientMessage

```
message = handleIncomingClientMessage_args(0)
client = handleIncomingClientMessage_args(1)
if message is a guess of the secret word then
    if client is not the current painter then
        if guess is correct then
            reward the client points and move into the end round phase
        send their guess to all clients&
    else
        tell the client they can't guess because they're painting&
if message is a chat message then
    send message to all clients&
if message is a system message && message confirms painter role then
    add message to client's ACK response buffer for tryGetNextPainter to deal with
return
```

Algorithm 5 clientMessageSender

```
client = clientMessageSender_args(0)
while this thread is alive do
    get message from client's buffer
    if message != null then
        Send message to client
return
```

3.1.2 UDP handler

When the server is started up, it starts a simple UDP handler that relays all data sent by the painter. The logic can be seen in Algorithm 6

Algorithm 6 udpHandler

```
while true do
    get next message from socket and put it in receivedMessage
    extract id, x-coordinate, and y-coordinate from receivedMessage
    if currentPainter has the same id as the received id then
        send the x- and y-coordinates to all other clients
return
```

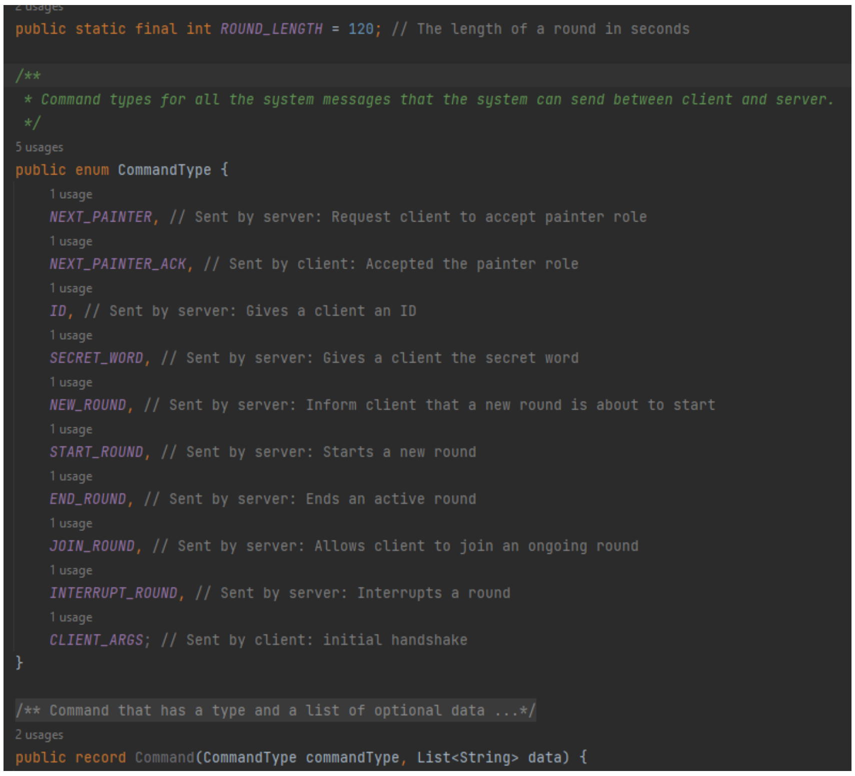
4 Clients

The main thread initiates the client with reader and writer threads for both TCP and UDP traffic, initializes the GUI, and then gets a username and server address from the program arguments which is used to establish a TCP connection to the server. The TCP reader thread then waits for the server to tell it what to do, such as accept the new painter role, start, join, and end rounds until the server closes the connection or the program is exited, see Algorithm 7.

Algorithm 7 drawClientMainLoop

```
username = drawClientMainLoop_args(0)
serverAddress = drawClientMainLoop_args(1)
start GUI
start TCP reader and writer&
send username and the used UDP port number to server to get accepted
start UDP reader and writer&
sleep until server connection closes
if server connection closed then
    release system resources
    Tell the client in the header that the system will exit in 5 seconds and count down
    close GUI
return
```

The more interesting networking part of the client is the handleMessage method in MessageReader that contains a switch case for each of the commands that the server can send to it; this is where the SystemUtility class really shines, see Figure 5.



```java
public static final int ROUND_LENGTH = 120; // The length of a round in seconds

/**
 * Command types for all the system messages that the system can send between client and server.
 */
5 usages
public enum CommandType {
    1 usage
    NEXT_PAINTER, // Sent by server: Request client to accept painter role
    1 usage
    NEXT_PAINTER_ACK, // Sent by client: Accepted the painter role
    1 usage
    ID, // Sent by server: Gives a client an ID
    1 usage
    SECRET_WORD, // Sent by server: Gives a client the secret word
    1 usage
    NEW_ROUND, // Sent by server: Inform client that a new round is about to start
    1 usage
    START_ROUND, // Sent by server: Starts a new round
    1 usage
    END_ROUND, // Sent by server: Ends an active round
    1 usage
    JOIN_ROUND, // Sent by server: Allows client to join an ongoing round
    1 usage
    INTERRUPT_ROUND, // Sent by server: Interrupts a round
    1 usage
    CLIENT_ARGS; // Sent by client: initial handshake
}

/** Command that has a type and a list of optional data ...*/
2 usages
public record Command(CommandType commandType, List<String> data) {
```

Figure 5: Utility class that creates and parses system commands.

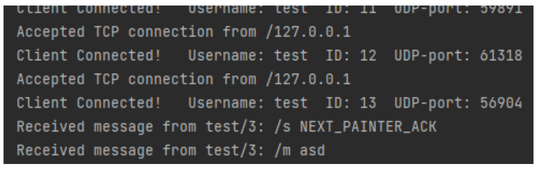
This class sets a standard for how system messages, or commands, are constructed and helps keeping the code clean by having methods to go from a string to a command and vice versa. The way it's used can be seen in Algorithm 8.

Algorithm 8 handleMessage

```
message = handleMessage_args(0)
if message is system message then
    try to convert message to system utility command
    if command == null then
        return
    if command matches any known commandType then
        Perform some useful function depending on the commandType
if message is a chat message then
    Print message to chat area in GUI
return
```

5 Functionality

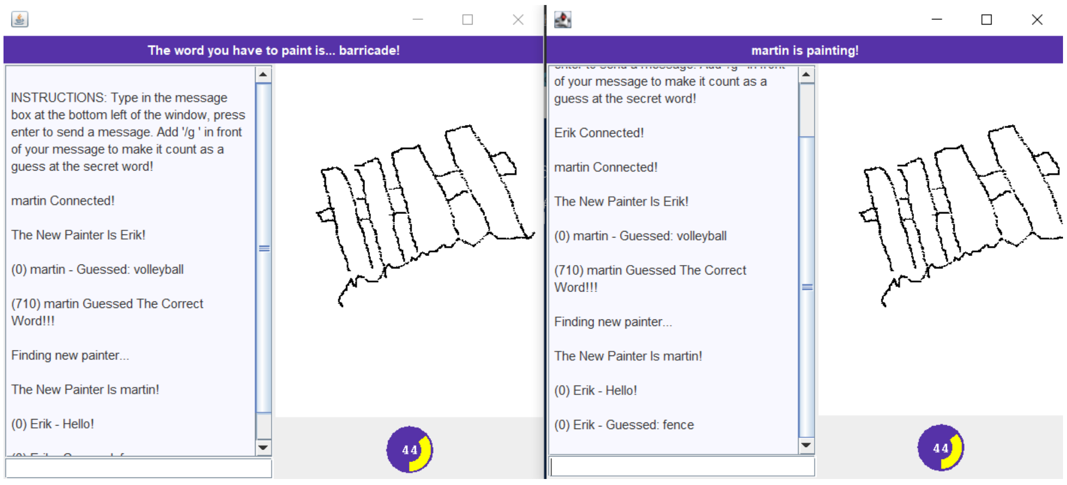
An important aspect of this system is that one game instance is capable of being used across the internet concurrently by as many users as the network and processing power of the system that the server is running on will allow for, which I suspect would be many clients. I tested running the program with 14 clients on a single computer and the system worked fine, although the drawing slowed down due to how all the messages have to be sent, received, and drawn on the same computer for each client. A small excerpt of the logs produced at the server side during this testing can be seen in Figure 6, where we can see the clients connecting, client with ID 3 accepting the painter role, and then writing a message. I don't have access to a lot of computers so it's difficult to test this in a more meaningful way.



```
Client Connected!   Username: test   ID: 11   UDP-port: 59891
Accepted TCP connection from /127.0.0.1
Client Connected!   Username: test   ID: 12   UDP-port: 61318
Accepted TCP connection from /127.0.0.1
Client Connected!   Username: test   ID: 13   UDP-port: 56904
Received message from test/3: /s NEXT_PAINTER_ACK
Received message from test/3: /m asd
```

Figure 6: Server logs when running test with 14 clients.

The server has no visual aspects to it besides the logs it produces, but the client displays a nice looking GUI to the client with a chat window, canvas to draw on, a header for messages displayed by the system, and a fun looking timer that counts down the rest of the round and changes color depending on the time, see Figure 7.



The word you have to paint is... barricade!

INSTRUCTIONS: Type in the message box at the bottom left of the window, press enter to send a message. Add '/g ' in front of your message to make it count as a guess at the secret word!

martin Connected!

The New Painter Is Erik!

(0) martin - Guessed: volleyball

(710) martin Guessed The Correct Word!!!

Finding new painter...

The New Painter Is martin!

(0) Erik - Hello!

martin is painting!

of your message to make it count as a guess at the secret word!

Erik Connected!

martin Connected!

The New Painter Is Erik!

(0) martin - Guessed: volleyball

(710) martin Guessed The Correct Word!!!

Finding new painter...

The New Painter Is martin!

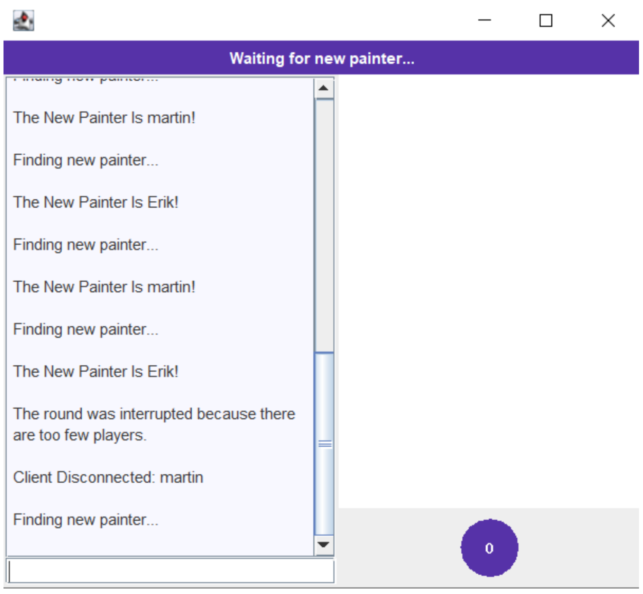
(0) Erik - Hello!

(0) Erik - Guessed: fence

Figure 7: GUIs of two clients connected to the server during a round of game play

In the figure, we can see the instructions for how messages are sent and that two players have connected, the first round then started with Erik being the painter. Martin then guessed the secret word volleyball correctly which ends the round and was awarded 710 points for it which is because there was 71 seconds left on the clock when the guess was verified. The new amount of points for Martin will then be seen in future references to them, as show in the victory message. The next round then starts by selecting the next painter to be Martin, and the new round continues where Martin attempts to draw the secret word "barricade". Erik finally says "Hello!" to Martin, and then proceeds to guess the secret word incorrectly as "fence" and is awarded no points for it, hence the round continues with 44 seconds left on the clock. Additionally, the system keeps informing the user about the changes in the game state, and shows who is painting and what word they have to paint in the purple header of the interface.

The program also smoothly handles cases where the painter disconnects or too few clients are connected to keep running the round by interrupting the round, as can be seen in Figure 8.


Waiting for new painter...

The New Painter Is martin!

Finding new painter...

The New Painter Is Erik!

Finding new painter...

The New Painter Is martin!

Finding new painter...

The New Painter Is Erik!

The round was interrupted because there are too few players.

Client Disconnected: martin

Finding new painter...

Figure 8: Round is interrupted if the painter leaves or if there are too few clients connected

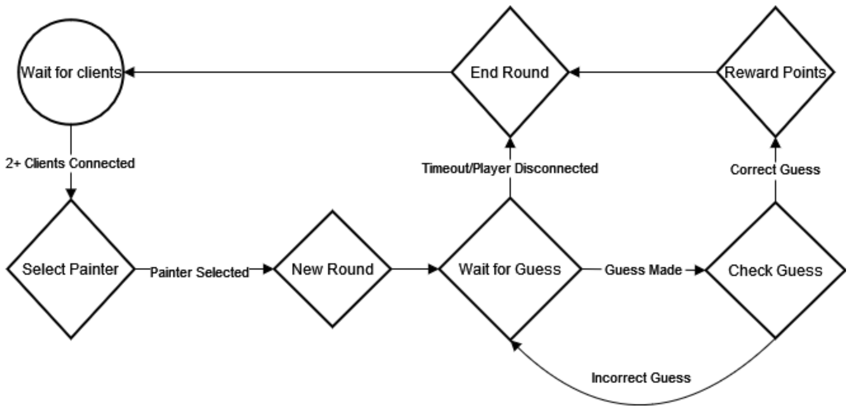


Figure 9: Flow chart of game states as seen by the client.

The entire process for a game as seen by a client can be gleaned from the flowchart in Figure 9. The server waits for enough clients to connect to select a painter, then it stays in a loop waiting for the correct word to be guessed to reward them points and end the round. The round can also time out, or a critical player can disconnect which causes the round to end before anyone can guess the correct word. The server then checks if there's enough players to select a new painter and start a new game.