

Assignment 1 – Socket Programming



Student Name: Morgan Lloyd

Student Number: 296463

Contents

1. Chat Application & Requirements	3
2. User Interface (UI)	4
3. Client-server system – Flow	7
4. Data & Communication	9
5. Classes, controllers and methods	10

1. Chat Application & Requirements

This project is centred around developing a chat application for student communication within the college computing classes. Below are the requirements of the application:

- Allow users to enter their names for identification.
- Support multiple simultaneous users.
- Allow users to connect to the server.
- Allow users to leave the server.
- Allow users to send messages.
- Implement an interface for users to interact with.

Tools

For the development of this application, I will be using Java as the programming language in combination with JavaFX and scene builder. Java has been selected as it supports socket programming and has shown in research to be efficient in creating basic socket applications for communication compared to C. [1]

2. **User Interface (UI)**

The user interface for the application will be split into two scenes. The first is the login page. The second is the chat room.

Login scene

Entry fields for the user names are included as per requirements. Exit and start buttons are included for the actions they represent.

NPTC Computing Class Chat

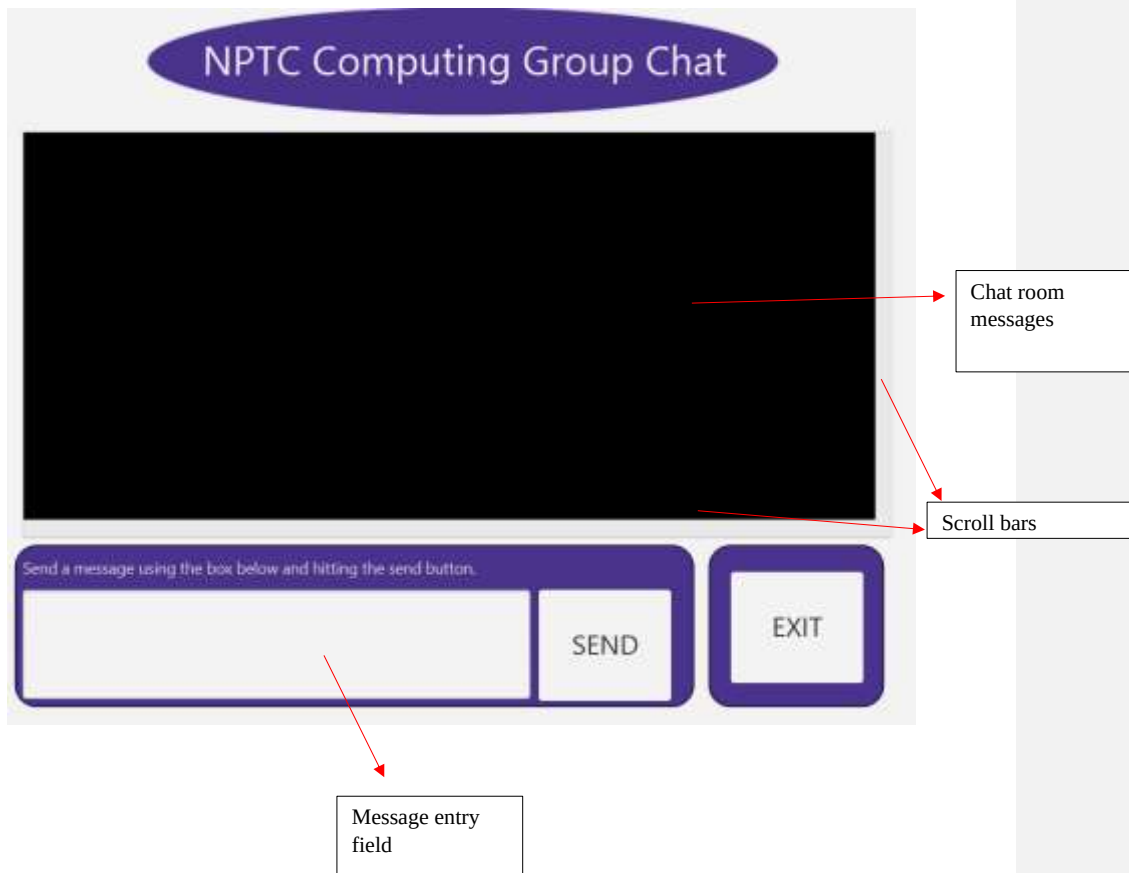
First Name:

Last Name:

EXIT START

Chat room scene

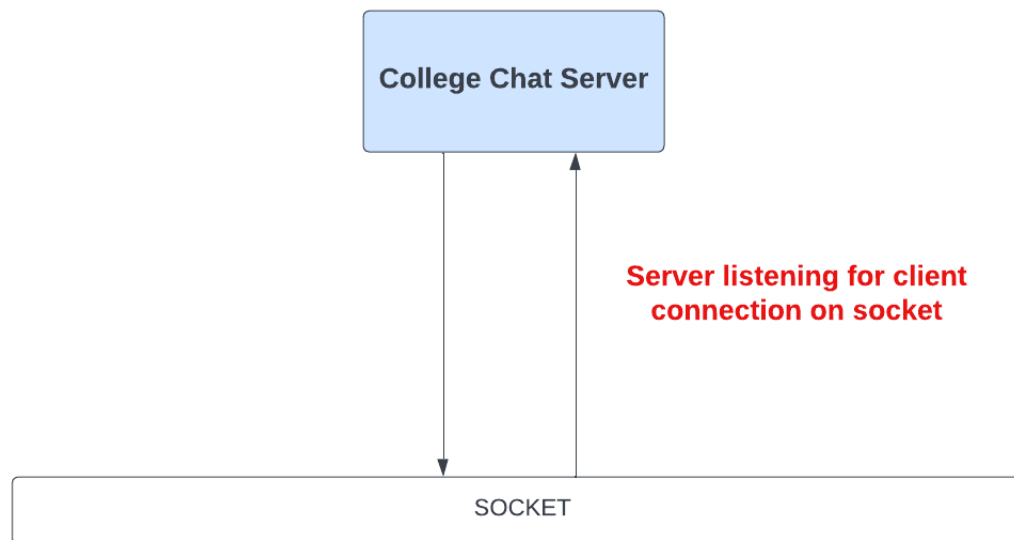
This scene includes a message entry field for the user to enter their message into the chat. Send button for broadcasting the message. Exit button to leave the server and application. Chat room message box for the conversation between the clients. Scroll bars for moving through the chat history.



3. Client-server system – Flow

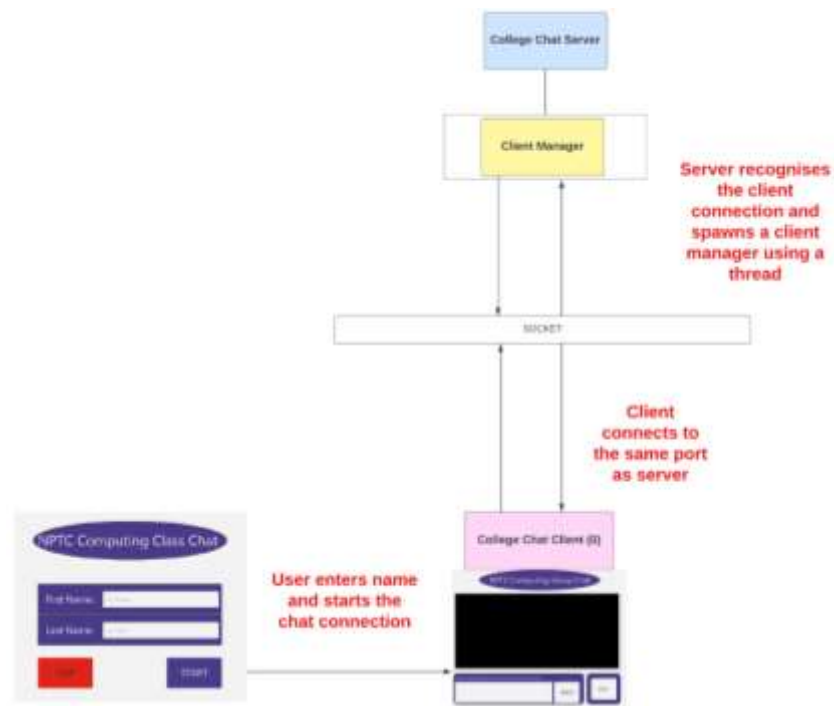
Step 1 - Server

First, the server will be running listening for a client connection on the socket at a specific port number.



Step 2 – Client connects

The client enters their name and clicks start to enter the chat room. This operation initialises the client connection to the socket at the same port number of the server. The server recognises the connection and creates a client manager and thread for the instance of this client connection. The use of the client manager and threads will enable multiple clients to join the chat room and send messages to each other. Because each client manager is an instance of a client connection, every time one gets started it gets added to a list to keep track of current connections within the program. This list can then be used to send messages from a client to other client connections.



4. Data & Communication

TCP has been selected as it ensures reliable transmission which is vital for a chat room where messages need to be complete and correct



Classes	
Class	Description
Main	Starts the application and loads the first scene
Controller 1	Responsible for controlling the login page
Controller 2	Responsible for the chat room UI and interacting with client class
Server	Listens for incoming client connections and spawns client manager class with a thread for handling passing messages to other clients
Client	Client connection and methods for sending to other clients and receiving messages to send to controller 2.
Client Manager	This class is used to handle incoming messages from the client to send other connected clients messages

5. Classes, controllers and methods

Main

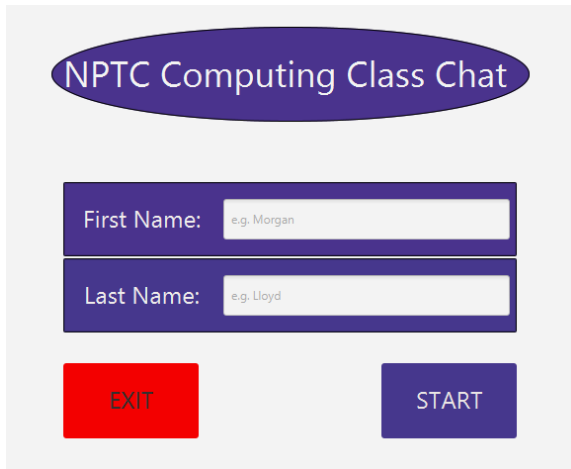
Below is the main class that is responsible for **loading** the starting login scene.

Main
<pre>public void start(Stage primaryStage) throws Exception { Parent root = FXMLLoader.load(getClass().getResource("/Scene1.fxml")); primaryStage.setScene(new Scene(root, 330.0, 560.0)); primaryStage.show(); public static void main(String[] args) {launch(args); } }</pre>

The screenshot shows a login window titled "NPTC Computing Class Chat" in a purple oval. Below the title, there are two input fields: "First Name:" with a placeholder "e.g. Morgan" and "Last Name:" with a placeholder "e.g. Lloyd". At the bottom, there are two buttons: a red "EXIT" button and a purple "START" button.

Controller 1 – Login page

This controller is responsible for the login page. Each button has a corresponding function for its action. The send button retrieves the user name using the 'getText()' method when pressed and sends it to the next controller using a setter. The send button also starts the chat room scene.



```
@FXML
void registrationStart(ActionEvent event) throws IOException {

    firstname = this.firstnameField.getText();
    lastname = this.lastnameField.getText();

    fullName = firstname + " " + lastname + " ";

    FXMLLoader loader = new FXMLLoader(getClass().getResource("Scene2.fxml"));
    root = loader.load();

    Controller2 controller2 = loader.getController();
    controller2.setName(fullName);
    controller2.sendMessage(fullName);

    stage = (Stage)((Node)event.getSource()).getScene().getWindow();
    scene = new Scene(root);
    stage.setScene(scene);
    stage.show();
}
```

Controller 2 – Chat Room – Part 1

This controller is responsible for the chat room. Upon initialising, it creates a client instance using its constructor in the client class. It then calls a **client** method to receive messages by passing in the vertical box (chatmessages). The send button retrieves the message from the entry field and combines it with the name to form a complete message to send. All of this gets inserted inside the vertical box using a variety of controls. In the end, the message is sent to the client function which will send it to the server and the message is cleared from the text field.



```
public void initialize(URL location, ResourceBundle resources) {  
  
    client = new Client();  
  
    client.receiveMessageFromOtherClients(chatmessages);  
  
    sendBtn.setOnAction(new EventHandler<ActionEvent>() {  
  
        public void handle(ActionEvent actionEvent) {  
            String messageForChat = clientmessage.getText();  
            String fullname = getName();  
            if (!messageForChat.isEmpty()) {  
                HBox horizontalChatRow = new HBox();  
                horizontalChatRow.setAlignment(Pos.CENTER_RIGHT);  
                Text text = new Text(" " + fullname + messageForChat);  
                horizontalChatRow.setStyle("-fx-background-color: WHITE;");  
                TextFlow textFlow = new TextFlow(text);  
                text.setFill(Color.NAVY);  
                text.setFont(Font.font("Arial", FontWeight.BOLD, 15));  
                horizontalChatRow.getChildren().add(textFlow);  
                chatmessages.getChildren().add(horizontalChatRow);  
  
                client.sendMessageToServer(fullname, messageForChat);  
                clientmessage.clear();  
            }  
        }  
    });  
}
```

Controller 2 – Chat Room – part 2

The exit button terminates the system and also the connection with the socket. The message from other clients function is used to insert messages into the vertical box from other clients. It uses horizontal boxes with text flows to achieve the desired effect on the screen. At the bottom of the function, the Platform.runLater() method is used because only a JavaFX application thread can update the GUI not a different thread like the one calling this function.

```
exitBtn.setOnAction(new EventHandler<ActionEvent>() {
    @FXML

    public void handle(ActionEvent actionEvent) {

        System.exit(0);

    }

});

// message
public static void messagesFromOtherClients(String messageFromOtherClients, VBox vbox){

    HBox horizontalChatRowResponses = new HBox();
    horizontalChatRowResponses.setAlignment(Pos.CENTER_LEFT);
    horizontalChatRowResponses.setStyle("-fx-background-color: rgb(78, 53, 144);");
    Text text = new Text(messageFromOtherClients);
    TextFlow textFlow = new TextFlow(text);
    text.setFill(Color.WHITE);
    text.setFont(Font.font("Arial", FontWeight.BOLD, 15));

    horizontalChatRowResponses.getChildren().add(textFlow);

    Platform.runLater(new Runnable() {

        public void run() {

            vbox.getChildren().add(horizontalChatRowResponses);

        }

    });

}
```

Client

This class is used when controller 2 is loaded. This causes the constructor function to connect to the socket for this particular client. A buffered reader and writer are also created. It is best practice to wrap an input stream reader or output stream writer in a buffer reader or writer if the operations are costly. [2] The send message to server function takes in the users name and message from controller 2 and writes them to the stream. The flush method is called as it is best practice to do so after writing to the stream as it avoids program hangs [3]

```
1 usage
public Client(){
    try {
        this.collegesocket = new Socket("localhost", 8080);

        this.bReader = new BufferedReader(new InputStreamReader(collegesocket.getInputStream()));
        this.bWriter = new BufferedWriter(new OutputStreamWriter(collegesocket.getOutputStream()));

    } catch (IOException e) {
        closeCommunication(collegesocket, bReader, bWriter);
    }
}

2 usage
public void sendMessageToServer(String finalMessage, String fullName){
    try{
        bWriter.write(finalMessage);
        bWriter.write(fullName);
        bWriter.newLine();
        bWriter.flush();
    } catch (IOException e) {
        closeCommunication(collegesocket, bReader, bWriter);
    }
}
```

Client – Part 2

The close communication method is called when exceptions happen such as the user closing the application. The reader, writer and finally the socket is closed. The receive message function is run on a thread as listening is a blocking operation. The function passes the messages from other clients into controller 2 for display.

```
public void closeCommunication(Socket collegesocket, BufferedReader bReader, BufferedWriter bWriter) {
    try{
        bReader.close();
        bWriter.close();
        collegesocket.close();
    } catch (IOException e) {
        e.printStackTrace();
    }
}

//usage
public void receiveMessageFromOtherClients(VBox vbox) {
    new Thread(() -> {
        while (collegesocket.isConnected()) {
            try{
                String messageFromOtherClients = bReader.readLine();
                Controller2.messagesFromOtherClients(messageFromOtherClients, vbox);
            } catch (IOException e){
                closeCommunication(collegesocket, bReader, bWriter);
                break;
            }
        }
    }).start();
}
```

Server

The main method creates a socket and passes it to the server constructor then calls the start server method. This method constantly listens for a connection from a client when the socket is not closed. If a client connects the server accepts the connection using the '.accept()' method. Then the server creates a new client manager using the socket. It then makes a new thread passing in that client manager and starts the thread. The use of threads enables the multiple connections required for this application.

```
private final ServerSocket collegesocket;

// usage
public Server(ServerSocket collegesocket) { this.collegesocket = collegesocket; }

// usage
public void startServer() {
    try {
        while (true) {
            Socket socket = collegesocket.accept();
            System.out.println("Client connected");
            ClientManager clientManager = new ClientManager(socket);
            Thread thread = new Thread(clientManager);
            thread.start();
        }
        while (!collegesocket.isClosed());
    } catch (IOException e) {
        closeServerSocket();
    }
}

// usage
public void closeServerSocket() {
    try {
        if (collegesocket != null) {
            collegesocket.close();
        }
    } catch (IOException e) {
        e.printStackTrace();
    }
}

public static void main(String[] args) throws IOException {
    ServerSocket collegesocket = new ServerSocket(8080);
    Server server = new Server(collegesocket);
    server.startServer();
}
```

Client Manager

When the server gets a client connection it spawns a thread of this runnable class to handle the listening for other client messages. This class also creates a socket, buffered reader, buffered writer and client name via the constructor method which adds these to the client manager list. The constructor also sends a message down the socket alerting others that this connection has joined. The run method is code that the thread runs. It is run on a separate thread as listening is a blocking operation.

```
public ClientManager(Socket collegesocket){
    try{
        this.collegesocket = collegesocket;
        this.bReader = new BufferedReader(new InputStreamReader(collegesocket.getInputStream()));
        this.bWriter = new BufferedWriter(new OutputStreamWriter(collegesocket.getOutputStream()));
        this.clientFullName = bReader.readLine();
        clientManager.add(this);
        propagateMessage( messageToSend clientFullName + "Joined the chat");
    } catch (IOException e) {
        closeEverything(collegesocket, bReader, bWriter);
    }
}

public void run() {
    String messageFromChat;
    do{
        try {
            messageFromChat = bReader.readLine();
            propagateMessage(messageFromChat);
        } catch (IOException e) {
            closeEverything(collegesocket, bReader, bWriter);
            break;
        }
    }
    while(collegesocket.isConnected());
}
```

Client manager

The close everything method closes the socket, reader and writer when an I/O exception occurs. It also removes the connection from the client manager list using the remove method which then alerts the remaining connection that particular client connection has left the room. The propagate message function takes in a message from the run function and gives it to the other clients by writing to the stream. It determines who to send the message to by looping through each connection in the list.

```
public void closeEverything(Socket colloguesocket, BufferedReader bReader, BufferedWriter bWriter) {
    remove();
    try {
        bReader.close();

        bWriter.close();

        colloguesocket.close();
    } catch (IOException e) {
    }
}

//=====
public void remove() {
    clientManager.remove(== this);
    propagateMessage( "messageSend:clientFullName + " - has left the college chatroom ");
}

//=====
public void propagateMessage(String messageToSend) {
    for (ClientManager clientManager : clientManager) {
        try {
            if (!clientManager.clientFullName.equals(clientFullName)) {
                clientManager.bWriter.write(messageToSend);
                clientManager.bWriter.newLine();
                clientManager.bWriter.flush();
            }
        } catch (IOException e) {
            closeEverything(colloguesocket, bReader, bWriter);
        }
    }
}
```

References

1. Sawant A, Meshram B. Network programming in Java using Socket. International Journal of Engineering Research and Applications [Internet]. 2013 Feb [cited 2022 May 20];3(1):1299–305. Available from: http://www.ijera.com/papers/Vol3_issue1/GU3112991305.pdf
2. BufferedReader (Java Platform SE 8) [Internet]. Oracle.com. 2019 [cited 2022 May 20]. Available from: <https://docs.oracle.com/javase/8/docs/api/java/io/BufferedReader.html>
3. Elliotte Rusty Harold. Java network programming. 4th ed. Beijing: O'reilly; 2014.