Project Report - Group 22

Description of the Game:

"Deny and Conquer" is a multiplayer grid-based game where players compete to claim cells on a shared grid. The primary objective is to paint and claim as many cells as possible while preventing other players from doing the same. A player successfully claims a cell by painting more than 50% of it. The game concludes when all cells are claimed, and the player with the most cells claimed is declared the winner.

Design Overview:

1. Client-Server Architecture: The game is structured as a client-server application. One player acts as the server (host) while others join as clients. All participants, including the host, can play the game.

2. Server Role:

- The server plays a central role in managing the game's state.

- It keeps track of which cells are locked, claimed, and the scores of each player.

- It also broadcasts messages to synchronize game state across all clients.

- The server listens for incoming client connections on a specific port and handles each client in a separate thread.

3. Client Role:

- Clients are responsible for player interactions.

- They provide a GUI allowing players to lock, claim, or unlock cells by painting them.

- They send messages to the server based on player actions and update their GUI based on messages received from the server.

- On starting the game, players can choose to either host (acting as the server) or join as a client.

4. Communication: All players, including the server, communicate by sending messages over TCP sockets. This ensures reliable, ordered delivery of messages, which is crucial for the game's consistency.

Application-Layer Messaging Scheme:

1. Connection and Initialization:

- `YOUR_COLOR(COLOR)`: Upon a client's connection, the server sends this message to assign a unique color to the client.

2. Cell Operations:

- `LOCK(CELL_NUMBER)`: A client sends this message to the server to indicate the intent to start painting a cell.

- `UNLOCK(CELL_NUMBER)`: If a client releases a cell without claiming it, this message is sent to the server.

- `CLAIM(CELL_NUMBER)`: On successfully painting and claiming a cell, a client sends this message to the server.

- `LOCKED(CELL_NUMBER)`: The server broadcasts this message to inform all clients that a particular cell is currently under paint.

- `UNLOCKED(CELL_NUMBER)`: This broadcasted message from the server informs clients that a cell has been released without being claimed.

- `CLAIMED(CELL_NUMBER, COLOR)`: A server-broadcasted message notifying all clients of a cell's successful claim and the color (player) that claimed it.

3. Game Control Messages:

- `GAME_OVER(RESULT)`: Once all cells are claimed, the server sends this message to all clients, indicating the game's end and its outcome.

4. Message Format:

- Messages typically start with a keyword indicating the message's purpose, followed by relevant data enclosed in parentheses. This consistent format ensures easy parsing and interpretation of messages on both client and server sides.

(i) Opening Socket:

1. Server-side:

   A code snippet of opening socket on the Server side in TCPServer.java:

```
public class TCPServer {

// TCPServer class is responsible for handling server-side logic and
communication with multiple clients.



private ServerSocket serverSocket;

private List<ServerThread> clients = new ArrayList<>();

private AtomicInteger clientCounter = new AtomicInteger();

private String[] names = {"Red", "Blue", "Green", "Yellow"};

private String[] cellOwners = new String[64];

private int[] playerScores = new int[4]; // To keep track of scores for
each player

private boolean gameStarted = false;



// Constructor initializes the server by creating a ServerSocket
listening on port 7070.

public TCPServer() throws IOException {

serverSocket = new ServerSocket(7070);

}



// Starts the server and continuously accepts incoming client
connections.

public void start() throws IOException {

while (true) {
```

```
Socket socket = serverSocket.accept();



if (clientCounter.get() >= 4) {

// Reject the connection if there are already 4 clients connected.

socket.close();

} else {

// Create a new ServerThread for the client, start it, and add it to the
clients list.

ServerThread st = new ServerThread(socket, this,
names[clientCounter.get()]);

clients.add(st);

st.start();

clientCounter.incrementAndGet();

}

}

}

//Rest of the code



// ServerThread class handles communication with an individual client.

class ServerThread extends Thread {

private Socket socket;

private TCPServer server;

private PrintWriter writer;

private String username;

private Pattern pattern = Pattern.compile("\\(([^)]+)\\)");
```

```
// Constructor initializes the ServerThread with the client's socket, the
TCPServer instance, and the client's username.

public ServerThread(Socket socket, TCPServer server, String username) {

this.socket = socket;

this.server = server;

this.username = username;

}
```

2. Client-side:

   A code snippet of opening socket on the Server side in TCPClient.java:

```
// TCPClient class is responsible for handling client-side communication
with the server.



private Socket socket;

private PrintWriter writer;

public String userColor;

private InteractiveFillableColorGridGUI gameGui;



// Constructor initializes the client by establishing a connection to the
server and creating a PrintWriter.

public TCPClient() throws IOException {

this.socket = new Socket("localhost", 7070);

this.writer = new PrintWriter(socket.getOutputStream(), true);
```

```
}

//Rest of the code

// ReadThread class handles reading data from the server.

class ReadThread extends Thread {

private BufferedReader reader;

private TCPClient client; // added this to store reference to TCPClient



// Constructor initializes the ReadThread with the socket and a reference
to the TCPClient instance.

public ReadThread(Socket socket, TCPClient client) {

this.client = client; // store the TCPClient instance

try {

InputStream input = socket.getInputStream();

reader = new BufferedReader(new InputStreamReader(input));

} catch (IOException ex) {

System.out.println("Error getting input stream: " + ex.getMessage());

ex.printStackTrace();

}

}

//REST OF THE CODE

// WriteThread class handles writing data to the server.

class WriteThread extends Thread {

private PrintWriter writer;
```

```
// Constructor initializes the WriteThread with the socket.

public WriteThread(Socket socket) {

try {

OutputStream output = socket.getOutputStream();

writer = new PrintWriter(output, true);

} catch (IOException ex) {

System.out.println("Error getting output stream: " + ex.getMessage());

ex.printStackTrace();

}

}
```

(ii) Handling the shared object:

1. **Server-side (TCPServer.java):**

```
private ServerSocket serverSocket;

private List<ServerThread> clients = new ArrayList<>();

private AtomicInteger clientCounter = new AtomicInteger();

private String[] names = {"Red", "Blue", "Green", "Yellow"};

private String[] cellOwners = new String[64];

private int[] playerScores = new int[4]; // To keep track of scores for each player

private boolean gameStarted = false;

// Rest of the code

// The run method is the main logic for handling client-server communication.

public void run() {

try {

InputStream input = socket.getInputStream();
```

```
BufferedReader reader = new BufferedReader(new InputStreamReader(input));

OutputStream output = socket.getOutputStream();

writer = new PrintWriter(output, true);



// Send the client's color to identify them in the game.

sendMessage("YOUR_COLOR(" + username + ")");



while (true) {

String clientMessage = reader.readLine();

Matcher matcher = pattern.matcher(clientMessage);

String number = "";

int cellNumber = -1;



// Extract the cell number from the client's message (if any).

if (matcher.find()) {

number = matcher.group(1);

cellNumber = Integer.parseInt(number);

}



synchronized(cellOwners) {

// Handle different client messages based on their content.



if (clientMessage.startsWith("LOCK") && cellNumber != -1) {
```

```
// The client is requesting to lock a box. If the box is unclaimed, lock it and inform
all clients.

if (cellOwners[cellNumber] == null) {

cellOwners[cellNumber] = username;

server.broadcastMessage("LOCKED(" + number + ")");

}

} else if (clientMessage.startsWith("UNLOCK") && cellNumber != -1) {

// The client is requesting to unlock a box. If the box is owned by the client, unlock
it and inform all clients.

if (cellOwners[cellNumber].equals(username)) {

cellOwners[cellNumber] = null;

server.broadcastMessage("UNLOCKED(" + number + ")");

}

} else if (clientMessage.startsWith("CLAIM") && cellNumber != -1) {

// The client is requesting to claim a box. If the box is owned by the client, claim
it and inform all clients.

if (cellOwners[cellNumber].equals(username)) {

server.broadcastMessage("CLAIMED(" + number + ", " + username + ")");



// Increase the player's score by 1 for claiming the box.

playerScores[Arrays.asList(names).indexOf(username)] += 1;



// Check if all boxes are claimed, then determine the game winner.

if (Arrays.stream(cellOwners).allMatch(Objects::nonNull)) {

int maxScore = Arrays.stream(playerScores).max().getAsInt();
```

```
List<String> winners = new ArrayList<>();

for (int i = 0; i < 4; i++) {

if (playerScores[i] == maxScore) {

winners.add(names[i]);

}

}

if (winners.size() > 1) {

server.broadcastMessage("GAME_OVER(Tie between " + String.join(", ", winners) + ")");

} else {

server.broadcastMessage("GAME_OVER(" + winners.get(0) + " wins)");

}

}

}

} else if (clientMessage.equals("START") && username.equals("Red")) {

// The client with the username "Red" is requesting to start the game.

// Set the gameStarted flag to true and inform all clients to start the game.

gameStarted = true;

server.broadcastMessage("START_GAME");

} else {

// If the message doesn't match any specific command, treat it as a general chat
message

// and broadcast it to all clients.

server.broadcastMessage(username + ": " + clientMessage);

}
```

```
}

}

} catch (IOException ex) {

System.out.println("Server exception: " + ex.getMessage());

} finally {

try {

socket.close();

} catch (IOException e) {

e.printStackTrace();

}

clients.remove(this);

}

}
```

2. **Client Side (TCPClient.java):**

```
// Sends a message to the server to lock a specific box identified by its row and
column.

public void lockBox(int row, int col) {

sendMessage("LOCK(" + (row * 8 + col) + ")");

}



// Sends a message to the server to claim a specific box identified by its row and
column.

public void claimBox(int row, int col) {

sendMessage("CLAIM(" + (row * 8 + col) + ")");
```

```
}



// Sends a message to the server to unlock a specific box identified by its row and
column.

public void unlockBox(int row, int col) {

sendMessage("UNLOCK(" + (row * 8 + col) + ")");

}

// Updates the GUI to lock a specific cell identified by its cell number.

public void lockCellInGui(int cellNumber) {

gameGui.lockCell(cellNumber);

}



// Updates the GUI to unlock a specific cell identified by its cell number.

public void unlockCellInGui(int cellNumber) {

gameGui.unlockCell(cellNumber);

}



// Updates the GUI to claim a specific cell identified by its cell number with the
given owner color.

public void claimCellInGui(int cellNumber, String owner) {

gameGui.claimCell(cellNumber, owner);

}

//Rest of the code

// The run method listens for incoming messages from the server.

public void run() {
```

```
while (true) {

try {

String response = reader.readLine();

System.out.println(response);



// Parse the server response and take appropriate actions based on the message
received.



if (response.startsWith("YOUR_COLOR")) {

// The server informs the client about its color in the game.

// The color information is extracted from the message and used to initialize the GUI.

String color = response.substring(11, response.length() - 1);

javax.swing.SwingUtilities.invokeLater(() -> {

client.userColor = color;

client.initializeGameGui(color); // Initialize the GUI here

});

} else if (response.startsWith("LOCKED")) {

// The server informs that a box has been locked by another player.

// The cell number is extracted from the message and used to lock the cell in the GUI.

String cellStr = response.substring(7, response.length() - 1);

int cellNumber = Integer.parseInt(cellStr);

client.lockCellInGui(cellNumber);

} else if (response.startsWith("UNLOCKED")) {

// The server informs that a box has been unlocked by the previous owner.
```

```
// The cell number is extracted from the message and used to unlock the cell in the
GUI.

String cellStr = response.substring(9, response.length() - 1);

int cellNumber = Integer.parseInt(cellStr);

client.unlockCellInGui(cellNumber);

} else if (response.startsWith("CLAIMED")) {

// The server informs that a box has been claimed by a player.

// The cell number and owner information are extracted from the message and used to
update the GUI.

String[] parts = response.split(", ");

String cellStr = parts[0].substring(8);

String owner = parts[1].substring(0, parts[1].length() - 1);

int cellNumber = Integer.parseInt(cellStr);

client.claimCellInGui(cellNumber, owner);

} else if (response.startsWith("GAME_OVER")) {

// The server informs that the game is over, and the result is provided.

// A message dialog is shown with the result, and the client exits the application.

String result = response.substring(10, response.length() - 1);

javax.swing.SwingUtilities.invokeLater(() -> {

javax.swing.JOptionPane.showMessageDialog(null, result, "Game Over",
javax.swing.JOptionPane.INFORMATION_MESSAGE);

System.exit(0);

});

}
```

```
} catch (IOException ex) {

System.out.println("Error reading from server: " + ex.getMessage());

ex.printStackTrace();

break;

}

}

}
```

3. InteractiveFillableColorGridGUI.java:

```
//Rest of the code



// Method to update the GUI when a cell is locked by another player.

public void lockCell(int cellNumber) {

int row = cellNumber / GRID_SIZE;

int col = cellNumber % GRID_SIZE;

gridBoxes[row][col].setBackground(Color.GRAY);

}



// Method to update the GUI when a cell is unlocked by the previous owner.

public void unlockCell(int cellNumber) {

int row = cellNumber / GRID_SIZE;

int col = cellNumber % GRID_SIZE;

gridBoxes[row][col].setBackground(Color.WHITE);

}
```

```
// Method to update the GUI when a cell is claimed by a player.

public void claimCell(int cellNumber, String owner) {

int row = cellNumber / GRID_SIZE;

int col = cellNumber % GRID_SIZE;

Color ownerColor = getColorForOwner(owner);

gridBoxes[row][col].setBackground(ownerColor);

cellClaimed[row][col] = true; // Mark the cell as claimed

}
```

(b) List of Group Members:	**Participation:**
Laksh Agarwal	40%
Vishavjit Singh Harika	20%
Balraj Singh	20%
Ujjwal Maken	18%
Sepinoud Fasihimajd	2%

(c) Commented source code of the client and server and ServerAsClient:

GitHub link: https://github.com/lakshagarwal/DenyAndConquer

- **TCPClient.java:**

```
import java.io.*;
import java.net.*;
import javax.swing.*;
import java.util.*;

public class TCPClient {

// TCPClient class is responsible for handling client-side communication with the
server.
```

```
private Socket socket;
private PrintWriter writer;
public String userColor;
private InteractiveFillableColorGridGUI gameGui;

// Constructor initializes the client by establishing a connection to the server and
creating a PrintWriter.
public TCPClient() throws IOException {
this.socket = new Socket("localhost", 7070);
this.writer = new PrintWriter(socket.getOutputStream(), true);
}

// Starts the threads for reading and writing data to the server.
public void startThreads() {
new ReadThread(socket, this).start(); // Pass this TCPClient instance
new WriteThread(socket).start();
}

// Initializes the GUI for the game with the user's color.
public void initializeGameGui(String color) {
gameGui = new InteractiveFillableColorGridGUI(this, color);
}

// Sends a message to the server to lock a specific box identified by its row and
column.
public void lockBox(int row, int col) {
sendMessage("LOCK(" + (row * 8 + col) + ")");
}

// Sends a message to the server to claim a specific box identified by its row and
column.
public void claimBox(int row, int col) {
sendMessage("CLAIM(" + (row * 8 + col) + ")");
}

// Sends a message to the server to unlock a specific box identified by its row and
column.
public void unlockBox(int row, int col) {
sendMessage("UNLOCK(" + (row * 8 + col) + ")");
}
```

```
// Sends a message to the server.
public void sendMessage(String message) {
writer.println(message);
}

// Updates the GUI to lock a specific cell identified by its cell number.
public void lockCellInGui(int cellNumber) {
gameGui.lockCell(cellNumber);
}

// Updates the GUI to unlock a specific cell identified by its cell number.
public void unlockCellInGui(int cellNumber) {
gameGui.unlockCell(cellNumber);
}

// Updates the GUI to claim a specific cell identified by its cell number with the
given owner color.
public void claimCellInGui(int cellNumber, String owner) {
gameGui.claimCell(cellNumber, owner);
}

// Main method to create a new TCPClient instance and start the client.
public static void main(String[] args) throws IOException {
new TCPClient();
}
}

// ReadThread class handles reading data from the server.
class ReadThread extends Thread {
private BufferedReader reader;
private TCPClient client; // added this to store reference to TCPClient

// Constructor initializes the ReadThread with the socket and a reference to the
TCPClient instance.
public ReadThread(Socket socket, TCPClient client) {
this.client = client; // store the TCPClient instance
try {
InputStream input = socket.getInputStream();
reader = new BufferedReader(new InputStreamReader(input));
} catch (IOException ex) {
System.out.println("Error getting input stream: " + ex.getMessage());
ex.printStackTrace();
```

```
}
}

// The run method listens for incoming messages from the server.
public void run() {
while (true) {
try {
String response = reader.readLine();
System.out.println(response);

// Parse the server response and take appropriate actions based on the message
received.

if (response.startsWith("YOUR_COLOR")) {
// The server informs the client about its color in the game.
// The color information is extracted from the message and used to initialize the GUI.
String color = response.substring(11, response.length() - 1);
javax.swing.SwingUtilities.invokeLater(() -> {
client.userColor = color;
client.initializeGameGui(color); // Initialize the GUI here
});
} else if (response.startsWith("LOCKED")) {
// The server informs that a box has been locked by another player.
// The cell number is extracted from the message and used to lock the cell in the GUI.
String cellStr = response.substring(7, response.length() - 1);
int cellNumber = Integer.parseInt(cellStr);
client.lockCellInGui(cellNumber);
} else if (response.startsWith("UNLOCKED")) {
// The server informs that a box has been unlocked by the previous owner.
// The cell number is extracted from the message and used to unlock the cell in the
GUI.
String cellStr = response.substring(9, response.length() - 1);
int cellNumber = Integer.parseInt(cellStr);
client.unlockCellInGui(cellNumber);
} else if (response.startsWith("CLAIMED")) {
// The server informs that a box has been claimed by a player.
// The cell number and owner information are extracted from the message and used to
update the GUI.
String[] parts = response.split(", ");
String cellStr = parts[0].substring(8);
String owner = parts[1].substring(0, parts[1].length() - 1);
int cellNumber = Integer.parseInt(cellStr);
```

```
client.claimCellInGui(cellNumber, owner);
} else if (response.startsWith("GAME_OVER")) {
// The server informs that the game is over, and the result is provided.
// A message dialog is shown with the result, and the client exits the application.
String result = response.substring(10, response.length() - 1);
javax.swing.SwingUtilities.invokeLater(() -> {
javax.swing.JOptionPane.showMessageDialog(null, result, "Game Over",
javax.swing.JOptionPane.INFORMATION_MESSAGE);
System.exit(0);
});
}

} catch (IOException ex) {
System.out.println("Error reading from server: " + ex.getMessage());
ex.printStackTrace();
break;
}
}
}
}

// WriteThread class handles writing data to the server.
class WriteThread extends Thread {
private PrintWriter writer;

// Constructor initializes the WriteThread with the socket.
public WriteThread(Socket socket) {
try {
OutputStream output = socket.getOutputStream();
writer = new PrintWriter(output, true);
} catch (IOException ex) {
System.out.println("Error getting output stream: " + ex.getMessage());
ex.printStackTrace();
}
}

// The run method listens for input from the console and sends it to the server.
public void run() {
Console console = System.console();
String text;

while (true) {
```

```
text = console.readLine("");
writer.println(text);
}
}
}
```

- **TCPServer.java:**

```
import java.io.*;
import java.net.*;
import java.util.*;
import java.util.concurrent.atomic.AtomicInteger;
import java.util.regex.Matcher;
import java.util.regex.Pattern;

public class TCPServer {
// TCPServer class is responsible for handling server-side logic and communication
with multiple clients.

private ServerSocket serverSocket;
private List<ServerThread> clients = new ArrayList<>();
private AtomicInteger clientCounter = new AtomicInteger();
private String[] names = {"Red", "Blue", "Green", "Yellow"};
private String[] cellOwners = new String[64];
private int[] playerScores = new int[4]; // To keep track of scores for each player
private boolean gameStarted = false;

// Constructor initializes the server by creating a ServerSocket listening on port
7070.
public TCPServer() throws IOException {
serverSocket = new ServerSocket(7070);
}

// Starts the server and continuously accepts incoming client connections.
public void start() throws IOException {
while (true) {
Socket socket = serverSocket.accept();

if (clientCounter.get() >= 4) {
// Reject the connection if there are already 4 clients connected.
socket.close();
```

```
} else {
// Create a new ServerThread for the client, start it, and add it to the clients list.
ServerThread st = new ServerThread(socket, this, names[clientCounter.get()]);
clients.add(st);
st.start();
clientCounter.incrementAndGet();
}
}
}

// Broadcasts a message to all connected clients.
public void broadcastMessage(String message) {
for (ServerThread st : clients) {
st.sendMessage(message);
}
}

// ServerThread class handles communication with an individual client.
class ServerThread extends Thread {
private Socket socket;
private TCPServer server;
private PrintWriter writer;
private String username;
private Pattern pattern = Pattern.compile("\\(([^)]+)\\)");

// Constructor initializes the ServerThread with the client's socket, the TCPServer
instance, and the client's username.
public ServerThread(Socket socket, TCPServer server, String username) {
this.socket = socket;
this.server = server;
this.username = username;
}

// Sends a message to the client.
public void sendMessage(String message) {
writer.println(message);
}

// The run method is the main logic for handling client-server communication.
public void run() {
try {
InputStream input = socket.getInputStream();
```

```
BufferedReader reader = new BufferedReader(new InputStreamReader(input));
OutputStream output = socket.getOutputStream();
writer = new PrintWriter(output, true);

// Send the client's color to identify them in the game.
sendMessage("YOUR_COLOR(" + username + ")");

while (true) {
String clientMessage = reader.readLine();
Matcher matcher = pattern.matcher(clientMessage);
String number = "";
int cellNumber = -1;

// Extract the cell number from the client's message (if any).
if (matcher.find()) {
number = matcher.group(1);
cellNumber = Integer.parseInt(number);
}

synchronized(cellOwners) {
// Handle different client messages based on their content.

if (clientMessage.startsWith("LOCK") && cellNumber != -1) {
// The client is requesting to lock a box. If the box is unclaimed, lock it and inform
all clients.
if (cellOwners[cellNumber] == null) {
cellOwners[cellNumber] = username;
server.broadcastMessage("LOCKED(" + number + ")");
}
} else if (clientMessage.startsWith("UNLOCK") && cellNumber != -1) {
// The client is requesting to unlock a box. If the box is owned by the client, unlock
it and inform all clients.
if (cellOwners[cellNumber].equals(username)) {
cellOwners[cellNumber] = null;
server.broadcastMessage("UNLOCKED(" + number + ")");
}
} else if (clientMessage.startsWith("CLAIM") && cellNumber != -1) {
// The client is requesting to claim a box. If the box is owned by the client, claim
it and inform all clients.
if (cellOwners[cellNumber].equals(username)) {
server.broadcastMessage("CLAIMED(" + number + ", " + username + ")");
```

```
// Increase the player's score by 1 for claiming the box.
playerScores[Arrays.asList(names).indexOf(username)] += 1;

// Check if all boxes are claimed, then determine the game winner.
if (Arrays.stream(cellOwners).allMatch(Objects::nonNull)) {
int maxScore = Arrays.stream(playerScores).max().getAsInt();
List<String> winners = new ArrayList<>();
for (int i = 0; i < 4; i++) {
if (playerScores[i] == maxScore) {
winners.add(names[i]);
}
}
if (winners.size() > 1) {
server.broadcastMessage("GAME_OVER(Tie between " + String.join(", ", winners) + ")");
} else {
server.broadcastMessage("GAME_OVER(" + winners.get(0) + " wins)");
}
}
}
} else if (clientMessage.equals("START") && username.equals("Red")) {
// The client with the username "Red" is requesting to start the game.
// Set the gameStarted flag to true and inform all clients to start the game.
gameStarted = true;
server.broadcastMessage("START_GAME");
} else {
// If the message doesn't match any specific command, treat it as a general chat
message
// and broadcast it to all clients.
server.broadcastMessage(username + ": " + clientMessage);
}
}
}
} catch (IOException ex) {
System.out.println("Server exception: " + ex.getMessage());
} finally {
try {
socket.close();
} catch (IOException e) {
e.printStackTrace();
}
clients.remove(this);
}
```

```
}
}
}
```

- **ServerAsClient.java**

```
import java.io.IOException;

public class ServerAsClient {
private TCPClient client;

// The ServerAsClient class acts as a client to the TCPServer.

public ServerAsClient() throws IOException {
// Start the server in a new thread to handle incoming client connections.
new Thread(() -> {
try {
TCPServer server = new TCPServer();
server.start(); // Start the TCPServer and accept incoming client connections.
} catch (IOException e) {
e.printStackTrace();
System.out.println("Error starting server: " + e.getMessage());
}
}).start();

// Sleep for a short time to ensure the server has started.
try {
Thread.sleep(2000);
} catch (InterruptedException e) {
e.printStackTrace();
}

// Initialize the client for the server.
client = new TCPClient();
}

// This method is used to initialize the game GUI for the client.
public void initializeGameGui() {
client.startThreads(); // Start the client threads (ReadThread and WriteThread).
}
}
```

(d) Video of the working demo:
https://drive.google.com/file/d/1tplMDplmMR8tRXeduDhw-rhPGM_9g1xd/view?usp=sharing