CS494 - INTERNETWORKING PROTOCOL

LAB 01: SOCKET PROGRAMMING

Self Evaluation

1. Team Information

No.	Name	Student ID	Contribution (%)
1	Trần Bảo Lợi	20125010	50%
2	Huỳnh Bá Đông Cát	20125087	50%

2. Score Sheet

No.	Requirements	Score	Evaluate
1	Use C/C++, Java, C#	2	2
2	Implement whole gameplay properly	3	3
3	Socket Non-blocking	2	2
4	Have a good GUI (MFC, WPF, Swing, etc.)	3	3
	Total	**10**	**10**

Socket Programming Report

Our project is made in C# using WPF for making a beautiful UI, easy to use, interactive display.

Game Story

In this version of "*Who Wants to Be a Millionaire*" players compete against each other in a quiz game where they must answer questions correctly to earn points. The game story begins as the players log in and join the game room.

Each player must select their own unique username to distinguish themselves from the others. The game then begins, and the first question appears on the screen. The players must select the correct answer from four choices within a set time limit, but they also have the option to skip a question if they are unsure of the answer.

If a player answers a question incorrectly, they lose the game, and if they skip a question, the next player will answer the question. Each game the player can only skip one time

The remaining players continue to compete against each other until there is only one player left standing or they run out of questions, who will take home the grand prize of flex privilege.

Packets Client send to Server

```
5 references
public enum ClientPackets
{
    ResendUsername = 1,
    GiveAnswer,
}
```

Name	Description	Parameters	Server action
ResendUsername	Send username from the client to server	PacketID ClientID Name	Server Receive the name and check if it is available or not. -> Server sends Packet “RegistrationSuccessful” to client and does other work Else send packet “RegistrationFailed”.

GiveAnswer	Send answers to the server from 0 to 3 for A/B/C/D and 5 for skipping this quiz.	PacketID ClientID Answer RoundID	Receive the answer give to the MatchManager to handle

Packets Server send to Client

```
28 references
public enum ServerPackets
{
    WelcomePlayer = 1,

    RegistrationFailed,
    RegistrationSuccessful,

    UpdatePlayerOrder,
    CountdownStartGame,

    SetupGame,
    StartRound,
    SendQuestion,
    SendAnswer,
    SkipQuiz,
    EndRound,
    EndGame,

    NumberOfQuestion,
    YOUWIN,
}
```

Name	Description	Parameters	Client action
WelcomePlayer	Send ID to the client	PacketID ClientID	Client know it ID to use when send back packet
RegistrationFailed	Announce that the name is invalid	PacketID ClientID	Client try again at the username

RegistrationSuccessful	Announce that the name is valid	PacketID ClientID	Enter the game
UpdatePlayerOrder	Send The list of current player to client	PacketID Number of element List of {ID, name, order, iskilled}	Receive the player list then know who is next, the status of each player, is it my turn
CountdownStartGame	Announce that the game is going to start after time	PacketID	Countdown the timer
SetupGame	Announce the client to ready for game	PacketID	Reset all the status of the game to ready for next game
StartRound	Send the round number to the client to know which round it is	PacketID RoundID	Receive the Round number
SendQuestion	Send question to client	PacketID String Question String choice A String choice B String choice C String choice D	Receive the question and update the UI
SendAnswer	Send the answer of question to client	PacketID Int answer	Receive the answer and then update the UI

SkipQuiz	Announce that there is one player skip	PacketID	Know that someone is skip quiz
EndRound	Announce that the round is end	PacketID RoundID	Set timer to 0 and wait
EndGame	Announce that the game is end	PacketID	Announce lose if the client is a loser
NumberOfQuestion	Send the number of question in list	PacketID Int number of question	Receive number of question
YOUWIN	Announce that “you are the winner”	PacketID	Announce wins

Packet

```
public class Packet : IDisposable
{
    private List<Byte> _buffer;
    private Byte[] _readableBuffer;
    private int _readPointer;
    private bool _disposed = false;
```

The Packet class essentially serves as encapsulating the message, sent between clients and server, and easily disposable to not put heavy load onto memory. The _buffer field is used to store the bytes that are being read or written, and the _readableBuffer field is used to provide a way for the user of the class to read the bytes that have been received so far. The _readPointer field keeps track of where the next read operation should occur, and the _disposed field is used to prevent access to the object after it has been disposed.

The class also comes with many methods to insert various data types or be read by other objects.

In order to distinguish each packet, we incorporate the subsequent details in every transmission:

- Length: A 4-byte integer data type positioned at the beginning of each packet, indicating its length. Any data beyond that length belongs to a separate packet.
- Packet ID: This specifies the packet's intended use, allowing us to identify the corresponding function to process the packet.

We have integrated "write" and "read" functions for various primitive data types in our packet system. For instance, we use PutInt to serialize an integer into the packet and ReadInt to deserialize an integer from the byte list. To maintain the current position within the packet, we utilize a pointer, and all reading operations are conducted relative to the pointer. In the event that we try to read an integer from a position intended for a string, the data will become corrupted, and the pointer will be misaligned. Additionally, there are structures in place for handling Short, Long, Float, Bool, and other data types.

```
0 references
public void PutByte(Byte value) => _buffer.Add(value);

1 reference
public void PutBytes(Byte[] value) => _buffer.AddRange(value);

0 references
public void PutShort(short value) => _buffer.AddRange(BitConverter.GetBytes(value));

19 references
public void PutInt(int value) => _buffer.AddRange(BitConverter.GetBytes(value));

0 references
public void PutLong(long value) => _buffer.AddRange(BitConverter.GetBytes(value));

0 references
public void PutFloat(float value) => _buffer.AddRange(BitConverter.GetBytes(value));

1 reference
public void PutBool(bool value) => _buffer.AddRange(BitConverter.GetBytes(value));

8 references
public void PutString(string value)
{
    // a string does not have a fixed size, so we write the length before every string
    PutInt(value.Length);
    _buffer.AddRange(Encoding.ASCII.GetBytes(value));
}
```

```
0 references
public Byte ReadByte(bool moveNext = true)
{
    if (_buffer.Count > _readPointer)
    {
        Byte value = _readableBuffer[_readPointer];
        if (moveNext) _readPointer += sizeof(Byte);
        return value;
    }
    else throw new Exception("Read byte error!");
}

2 references
public Byte[] ReadBytes(int length, bool moveNext = true)
{
    if (_buffer.Count > _readPointer)
    {
        Byte[] value = _buffer.GetRange(_readPointer, length).ToArray();
        if (moveNext) _readPointer += length;
        return value;
    }
    else throw new Exception("Read byte array error!");
}
```

And many more read/ write functions to handle various types of variables. This offers great flexibility in structuring the packet which helps avoiding misinterpretation of the data structure that can lead to fatal errors.

TCP

```
internal class TCP
{
    public TcpClient socket;
    private NetworkStream _stream;
    private Packet _data;
    private byte[] _buffer;
    private readonly int _id;
```

This class serves as the delivery man that handles sending and receiving packets, while also performing checks to indicate whether the packets needs to be re-sent or handed off to the recipient.

Client

```
internal class Client
{
    private static TcpClient client;
    private static NetworkStream stream;
    private static Packet _data;
    private static byte[] buffer;

    public static int ID = 0;
    public static string nickname;
    public static bool isRegSuccess = false;

    public delegate void PacketHandler(Packet _packet);

    public static Dictionary<int, PacketHandler> packetHandlers;
    public static List<Player> playerList;

    public static QuizQuestion question;
    public static int _round = -1;

    1 reference
    public static void HandlePacket(int id, Packet packet) => packetHandlers[id](packet);

    public static int currentNumberOfQuestion = 0;
    public static int NumberOfQuestion = 0;
```

While the game runs on the surface, there's an active TCP client running underneath to handle sending answers to the server and receiving new questions from the server. When receiving a packet, the client needs to identify the type of message that the server is trying to send. That's why in our packet data, we put an integer at the head so when reading the packet, the client can take that integer and determine what's inside the packet. A dictionary is prepared beforehand that rules out the corresponding packet handler function. These functions helps read packet content and passing values into variables inside Client class that can be then used in the main game

```
packetHandlers = new Dictionary<int, PacketHandler>()
{
    {(int)ServerPackets.WelcomePlayer, Client.RecieveID },

    {(int)ServerPackets.RegistrationFailed, Client.RegistrationFailed },
    {(int)ServerPackets.RegistrationSuccessful, Client.RegistrationSuccessful },

    {(int)ServerPackets.UpdatePlayerOrder, Client.UpdatePlayerOrder },
    {(int)ServerPackets.CountdownStartGame, Client.CountdownStartGame },

    {(int)ServerPackets.SetupGame, Client.SetupGame },
    {(int)ServerPackets.StartRound, Client.StartRound },
    {(int)ServerPackets.SendQuestion, Client.ReceiveSendQuestion },
    {(int)ServerPackets.SendAnswer, Client.ReceiveSendAnswer },
    {(int)ServerPackets.SkipQuiz, Client.ReceiveSkipQuiz },
    {(int)ServerPackets.EndRound, Client.EndRound },
    {(int)ServerPackets.EndGame, Client.ReceiveEndGame },

    {(int)ServerPackets.NumberOfQuestion, Client.ReceiveNumberOfQuestion },
    {(int)ServerPackets.YOUWIN, Client.YOUWIN },
};
```

Server

```
internal class Server
{
    public static Dictionary<int, ClientItem> clients = new Dictionary<int, ClientItem>();
    public delegate void PacketHandler(int _fromClient, Packet _packet);
    public static Dictionary<int, PacketHandler> packetHandlers;

    private static TcpListener _tcpListener;
```

```
packetHandlers = new Dictionary<int, PacketHandler>()
{
    { (int)ClientPackets.ResendUsername, ServerHandler.ResendUsername },
    { (int)ClientPackets.GiveAnswer, ServerHandler.GiveAnswer }
};
```

There is a slight difference in structure between the Server and Client. The Client manages handle and sending packets within the same class, while the Server separates these tasks into ServerHandle and ServerSender. However, the ideas and structures are the same for both the Server and Client.

Packet Sender

```
public static void sendDataToServer(Packet packet)
{
    try
    {
        if (client != null) stream.BeginWrite(packet.Array, 0, packet.Length, null, null);
    }
    catch (Exception e)
    {
        // Die
    }
}
```

```
1 reference
public static void SendUsername()
{
    using (Packet packet = new Packet((int)ClientPackets.ResendUsername))
    {
        packet.PutInt(ID);
        packet.PutString(nickname);
        packet.InsertLength();

        sendDataToServer(packet);
    }
}
```

As I mentioned earlier, the send structure of both the Server and Client is nearly the same. The packet that is sent is created with an ID, followed by its content, and finally, its length. The length is always inserted at the beginning of the packet to keep track of reading the packet.

Packet Handler

This is a critical component to ensure the non-blocking socket of the game. Whenever a packet is handled, "OnReceivedData" will be triggered to try to read the data via "HandleData".

```
3 references
private static void OnReceivedData(IAsyncResult result)
{
    try
    {
        int byteLength = stream.EndRead(result);
        if (byteLength <= 0)
        {
            // Connection closed by server
            Disconnect();
            return;
        }

        byte[] data = new byte[byteLength];
        Array.Copy(buffer, data, byteLength);

        // Handle the incoming data using the same packet format as the server
        if (HandleData(data)) _data.Reset();
        else _data.Revert();

        // Read the next packet
        stream.BeginRead(buffer, 0, Constants.DATA_BUFFER_SIZE, OnReceivedData, null);
    }
    catch (Exception e)
    {
        Disconnect();
    }
}
```

```
1 reference
private static bool HandleData(byte[] data)
{
    // Parse the incoming data using the same packet format as the server
    int packetLength = 0;
    _data.AssignBytes(data);

    if (_data.UnreadLength >= 4)
    {
        packetLength = _data.ReadInt();
        if (packetLength <= 0) return true; // The packet is empty
    }

    while (packetLength > 0 && packetLength <= _data.UnreadLength)
    {
        byte[] packetBytes = _data.ReadBytes(packetLength);
        ThreadManager.ExecuteWithPriority(() =>
        {
            using (Packet pt = new Packet(packetBytes))
            {
                int packetId = pt.ReadInt(); // The first integer of the packet indicates the enum of the method to call
                Client.HandlePacket(packetId, pt); // Call appropriate method to handle the packet
            }
        });

        // Read the next packet
        packetLength = 0;
        if (_data.UnreadLength >= 4)
        {
            packetLength = _data.ReadInt();
            if (packetLength <= 0) return true; // The packet is empty
        }
    }

    if (packetLength <= 0) return true; // Recycle packets
    return false;
}
```

If the reading process is successful, we will clear the _data; otherwise, we return the pointer and save the data until it is complete to read. Once a data is read, we can determine its packet type. Instead of executing it immediately, we put it into the "ThreadManager". Immediate execution can interrupt the packet handling process. "ThreadManager" serves as a tool to communicate between the UI thread, the Handle Packet thread, and the game logic processing thread.

```
5 references
internal class ThreadManager
{
    private static readonly List<Action> _executePriority = new List<Action>();
    private static readonly List<Action> _executeDuplicatePriority = new List<Action>();
    private static bool _newPriorityAdded = false;

    3 references
    public static void Update()
    {
        if (_newPriorityAdded)
        {
            _executeDuplicatePriority.Clear();
            lock (_executePriority)
            {
                _executeDuplicatePriority.AddRange(_executePriority);
                _executePriority.Clear();
                _newPriorityAdded = false;
            }

            for (int i = 0; i < _executeDuplicatePriority.Count; i++)
            {
                _executeDuplicatePriority[i]();
            }
        }
    }

    2 references
    public static void ExecuteWithPriority(Action _action)
    {
        lock (_executePriority)
        {
            _executePriority.Add(_action);
            _newPriorityAdded = true;
        }
    }
}
```

Every Update on the UI and execution of the packet comes in ThreadManager will trigger in:

```
1 reference
private void MainThread()
{
    DateTime nextLoop = DateTime.Now;

    while (isInGame)
    {
        while (nextLoop < DateTime.Now)
        {
            ThreadManager.Update();
            Dispatcher.Invoke(() =>
            {
                UpdatePlayerList();
                if (!gameEnd)
                {
                    UpdateQuizQuestion();
                    UpdateChoicesColor();
                    UpdateTurnDisplay();
                    UpdateTimer();
                    if (isSetup)
                    {
                        QuestionBlock.Text = "READY TO START NEW GAME";
                        isSetup = false;
                    }
                }
                else
                {
                    CheckWinCondition();
                }
            });

            nextLoop = nextLoop.AddMilliseconds(Constants.MS_PER_TICK); // Calculate at what point in time the next tick should be executed

            if (nextLoop > DateTime.Now)
            {
                // If the execution time for the next tick is in the future, aka the server is NOT running behind
                Thread.Sleep(nextLoop - DateTime.Now); // Let the thread sleep until it's needed again.
            }
        }
    }
}
```

Same idea between Client and Server, continuously update.

Game Logic

When first Connect:

Step	Server	Client
1	Open the port to listen	
2		Connect to the port
3	Check If there is a free slot. If there is free slot, store the connection in that slot	
4	Send The ID to The client	
5		Receive the ID

6		The Client send The Name To the server
7	If the Client sent the wrong name format -> back to step 6 Else go to step 8	
8	Check if there >= 2 Client is connected, start a countdown timer before starting the game	

After having enough clients connected, the game will start counting down and transition to the playing phase.

Step	**Server**	**Client**
1	Server Announce ready to start the game	
2	Server load the question from database	
3	Server Send the number of question will have in this game	
4	Server check the condition of is the time to stop game	
5	Server send the round ID to all client	
6		Client receive RoundID
7	Server send the player list include the order to all client	
8		Client receive and know that is my turn or not

9	Server Send the Question	
10		Client Receive the question and update the UI
11	Server change to state wait for the answer	
12		If there are an answer client will send it to Server
13	Server receive the answer or it out of time	
14	Server check the answer	
15	Server Update the Player List	
16	Return to step 3	
17	Server Send the "YOUWIN" to the Client which is win	
18		The client update the UI to announce victory
19	Return to step 1	

References

We are using WPF C# for our project, and we would like to express our gratitude to ChatGPT for the valuable assistance provided.

For the UI, we have opted for Material Design In XAML. Additionally, we have gained a lot of knowledge on how to build a socket project through Old C# Networking Tutorials - YouTube, where we learned about TCP setup. The setup offered by the channel was great, and although we encountered some difficulties in the beginning, we managed to overcome them after about a week.

While a part of our project was based on the tutorial, we have thoroughly studied the code, experimented with it in various ways, and fully grasped its workings. This helped us to improve our project significantly.

Would like to thank the artist for the artwork but it’s taken randomly on google image so we can’t trace the source