TCP-SOCKETS

CLIENT-SERVER CHAT APPLICATION

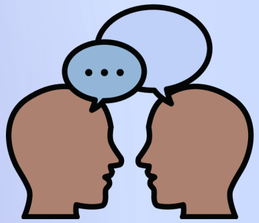
Prsented by: Ziad Ghoraba

AGENDA

- Application Overview.
- Server Side.
- Client Side.
- Live Demo.

OVERVIEW

This is a TCP client-server chat application. The main purpose of this application is to demonstrate the concept of sockets in operating systems, specifically how we use TCP sockets for communication between a client and a server.



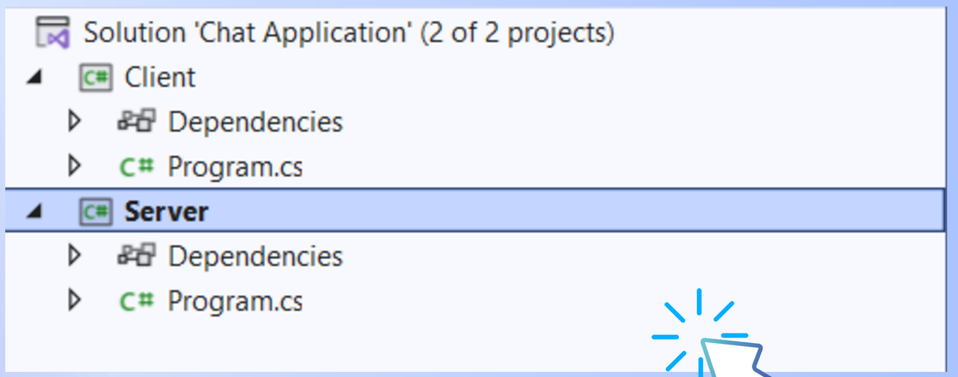
The client sends messages to the server, and the server responds back, allowing us to have a simple conversation. This is done using the TCP protocol, which ensures reliable data transmission.

OVERVIEW

Here, we have a screenshot of our solution in Visual Studio.

"**The Server** code listens for incoming connections, handles the receiving and sending of messages, and manages the connected client."

The solution is divided into two main parts: the Client and the Server.



"**The Client** code is where the user interacts with the application. It connects to the server, sends messages, and waits for responses."

"Both the client and server use the SimpleTCP library to simplify working with TCP sockets."

OVERVIEW

Purpose in Operating System Context:

At the operating system level, sockets provide a mechanism for applications to communicate with each other over a network.

The TCP protocol ensures that messages sent from the client arrive at the server in the correct order and without errors, as it handles retransmission and message integrity.

Functionality Recap:

To summarize, the Server listens for a connection, accepts it, and then waits for messages from the Client. When the server receives a message, it processes it and can send a response back.

The Client connects to the server, sends messages, and displays any responses from the server.

SERVER SIDE

```
var server = new SimpleTcpServer();

// Start the server on port 5000
server.Start(5000);
Console.WriteLine("Server started on port 5000...");
Console.WriteLine("Press Ctrl+C to stop the server.");
```

Server Initialization: Creates a SimpleTcpServer instance to manage client connections using TCP sockets.
Start Method: Binds the server to port 5000, enabling it to listen for incoming connections.
Console Messages: Provides feedback to confirm the server is running and explains how to terminate it..

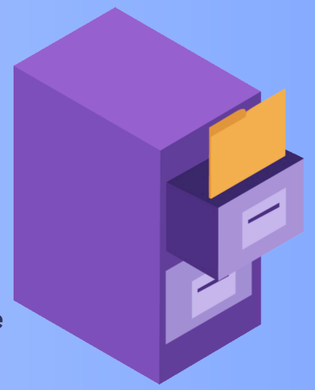
Client Connected: Triggered when a client connects, storing their socket and logging the connection.
Client Disconnected: Clears the connected Client variable if the client disconnects, ensuring no invalid references.
Data Received: Processes incoming data by decoding it to a string and displaying it on the server console.
Purpose: Manages the full client lifecycle, including connecting, disconnecting, and exchanging messages.

```
// Handle when a client connects
server.ClientConnected += (sender, e) =>
{
    connectedClient = e.Client; // Store the connected client Socket
    Console.WriteLine($"Client ({e.Client.RemoteEndPoint}) connected!");
};

// Handle when a client disconnects
server.ClientDisconnected += (sender, e) =>
{
    if (e.Client == connectedClient)
    {
        connectedClient = null; // Clear the connected client when disconnected
        Console.WriteLine($"Client ({e.Client.RemoteEndPoint}) disconnected!");
    }
};

// Handle when data is received from the client
server.DataReceived += (sender, e) =>
{
    var msg = Encoding.UTF8.GetString(e.Data); // Convert received data to string
    Console.WriteLine($"Client: {msg}"); // Display message from client
};
```

SERVER SIDE



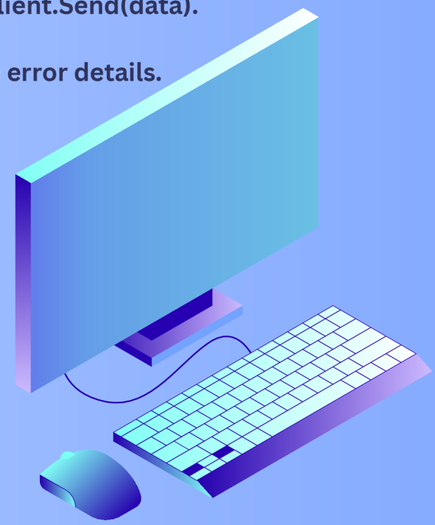
```
// Server input loop
while (true)
{
    var input = Console.ReadLine(); // Read server input
    if (string.IsNullOrEmpty(input)) continue; // Skip empty input

    if (connectedClient == null)
    {
        Console.WriteLine("No clients connected.");
        continue;
    }

    var data = Encoding.UTF8.GetBytes($"Server: {input}");

    try
    {
        connectedClient.Send(data); // Send data to the single conne
    }
    catch (Exception ex)
    {
        Console.WriteLine($"Error sending to client: {ex.Message}");
    }
}
```

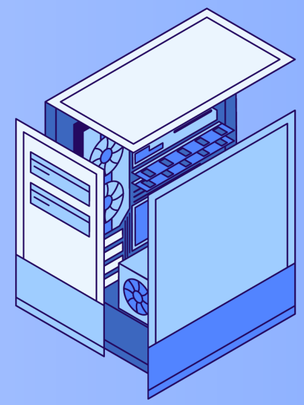
1. **Infinite Loop:**
   - **Continuously waits for the server user to input messages to send to the connected client.**
2. **Skip Empty Input:**
   - **Uses string.IsNullOrEmpty(input) to ensure blank inputs are ignored.**
3. **Client Connection Check:**
   - **If no client is connected (connectedClient == null), displays a message:**
   - **"No clients connected."**
   - **Skips further processing in this iteration.**
4. **Data Conversion:**
   - **Converts the input message into a byte array using Encoding.UTF8.GetBytes() for transmission.**
5. **Message Sending:**
   - **Sends the message to the connected client using connectedClient.Send(data).**
6. **Error Handling:**
   - **Catches exceptions during message transmission and logs the error details.**



Purpose

Facilitates two-way communication by allowing the server user to send messages to the client.
Ensures robustness by handling missing clients and transmission errors gracefully.

CLIENT SIDE



Client Initialization and Data Handling

```
var client = new SimpleTcpClient();

// Handle incoming data from the server
client.DataReceived += (sender, e) =>
{
    var msg = Encoding.UTF8.GetString(e.Data);
    Console.WriteLine(msg); // Display the mess
};
```

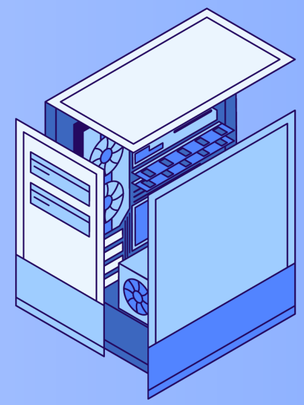
Client Setup: Creates a SimpleTcpClient to connect to the server.
DataReceived Event: Triggers when the server sends data to the client.
Data Conversion: Converts the received byte array to a string using UTF-8 encoding.
Output: Displays the server message in the client console for real-time feedback.

Connect Method: Establishes a connection to the server using the specified IP address and port.
Success Feedback: Logs a success message when the connection is established.
Error Handling: Catches connection errors, displays the error message, and stops further execution.

Client Connection

```
try
{
    // Connect to the server
    client.Connect("192.168.1.14", 5000); // Replace with your server's IP
    Console.WriteLine("Connected to the server.");
}
catch (Exception ex)
{
    Console.WriteLine($"Error connecting to the server: {ex.Message}");
    return;
}
```

CLIENT SIDE



Client Chat Loop

```
// Chat loop for client to send messages to the server
while (true)
{
    var input = Console.ReadLine(); // Read client input
    if (string.IsNullOrEmpty(input)) continue; // Skip empty input

    try
    {
        client.Write(input); // Send message to the server
        //! This is how it works explicitly -> client.Write(Encoding.UTF8.GetByte

         // After sending, display an acknowledgment message on the client side
        //Console.WriteLine("Message sent successfully!");
    }
    catch (Exception ex)
    {
        Console.WriteLine($"Error sending message: {ex.Message}");
    }
```

Message Input: Continuously reads user input from the console.
Message Sending: Sends the input to the server via the Write method.
Error Handling: Displays an error message if sending fails, ensuring the loop continues.

Going Live!
Watch as the code comes to life, live and in action!

THANK YOU!

LinkedIn

Mail

GitHub