Computer Network Programming Final Homework Paper

Ergül FERİK 190315041 Burak Can ALTUNOĞLU 190315064
190315041@ogr.cbu.edu.tr 190315064@ogr.cbu.edu.tr

Manisa Celal Bayar University
Computer Engineering Department

Abstract

We have created a system for this project that will enable clients (as a tenant or landlord) to access the interface, read their cards as they enter the apartment, interact with the occupants of the other apartment, and ensure that other dwellings are not visible between the flats. Let's quickly go through the benefits of this method. The system keeps track of who enters and exits the apartment when the outer gate is opened, the time they enter and leave, the conversations taking place between the clients, and if they are taking part in the conversation. At the entrance and departure to the flat, it scans a card. This is how we offer a safe sitting option.

1. Introduction

Figure 1. These days, many door intercom systems look like the in the image.

The systems mentioned above have certain security and auditing problems. In systems without a camera, knowing the password is all that is required, however in systems with a camera, it will not be possible to identify the user if the camera is not functioning. The outer gate system that we'll use will also guarantee that communication and security are maintained at a high level between the apartments if we take into account that the information of who entered the apartment and when, thanks to the entrance and exit times of the apartment and card reading in these traditional home systems.

2. Our Work

This project was created using the C# language and the.NET 6.0 Framework. In a nutshell, our project is as follows:

a)Server

Interface:

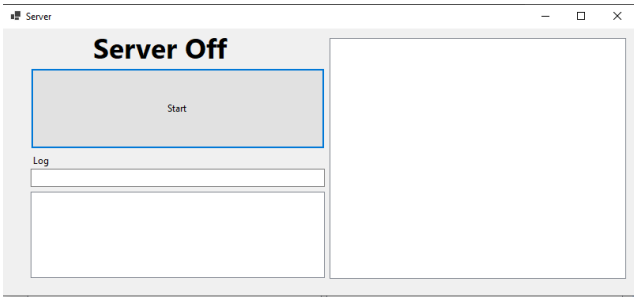


Figure 2. The server interface of our software is what you can see above.

Starting Server:

```
1 reference
private void button1_Click(object sender, EventArgs e)
{
    ///Starting server///

    server = new Socket(AddressFamily.InterNetwork,SocketType.Stream, ProtocolType.Tcp);
    IPEndPoint iep = new IPEndPoint(IPAddress.Any, 9050);
    server.Bind(iep);
    server.Listen(5);

    Thread threadBeginAccept = new Thread(beginAccept);
    threadBeginAccept.Start();

    label1.Text = "Server On";
}
```

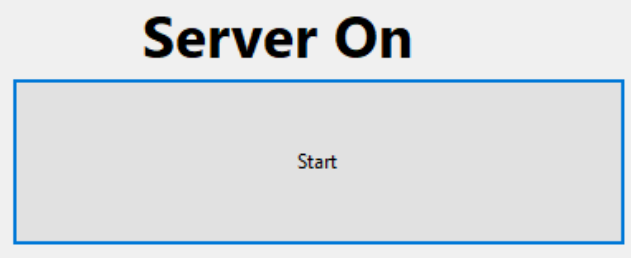
After our **TCP/IP** protocol code in the beginning server phase, we wished to configure a threaded server.

What is threaded server ?

A threaded server is a type of server that uses multiple threads to handle multiple client requests simultaneously. Each thread can handle a single client connection, allowing the server to handle multiple connections at the same time, improving performance and scalability. This is in contrast

to a single-threaded server, which can only handle one request at a time.

We begin the threaded function with start after threading our begin accept function. Then we print our label, "**label1**," with the phrase "**Server On**".



Imports:

```
using System;
using System.Drawing;
using System.Linq;
using System.Net;
using System.Net.Sockets;
using System.Text;
using System.Windows.Forms;
using System.Xml;
using System.Xml.Linq;
using Newtonsoft.Json;
using static System.Net.WebRequestMethods;
```

The above code imports various namespaces that are used in the rest of the code.

"**System**" is a namespace that contains basic types and base classes that are commonly used in the .NET Framework.

"**System.Drawing**" is a namespace that contains classes for drawing graphics and text, as well as providing basic imaging functionality.

"**System.Linq**" is a namespace that contains classes and interfaces that support Language-Integrated Query (LINQ).

"**System.Net**" is a namespace that contains classes for working with the network, including classes for sending and receiving data over the network, as well as classes for working with the Internet Protocol (IP) and Transmission Control Protocol (TCP) protocols.

"**System.Net.Sockets**" is a namespace that contains classes for working with the TCP and UDP protocols, including classes for creating and managing socket connections.

"**System.Text**" is a namespace that contains classes for working with text, including classes for encoding and decoding character data, as well as classes for working with strings and character arrays.

"**System.Windows.Forms**" is a namespace that contains classes for creating Windows-based applications, including classes for creating and managing forms, controls, and dialog boxes.

"**System.Xml**" is a namespace that contains classes for working with XML data, including classes for reading and writing XML documents, as well as classes for working with the XPath query language.

"**Newtonsoft.Json**" is a namespace that contains classes for working with JSON data, including classes for serializing and deserializing JSON data.

"**static System.Net.WebRequestMethods**" is a way of accessing the WebRequestMethods class and its members without having to create an instance of the class. It contains several constant fields which are used to specify the method of a WebRequest.

Initializing and Form Loading:

```
public partial class Form1 : Form
{
    private Socket server;
    private byte[] data = new byte[1024];
    private int size = 1024;
    Socket Tempclient;
    Socket client;
    bool flag = false;



    List <Socket> clients = new List<Socket>();

    Dictionary<string, Socket> users = new Dictionary<string, Socket>();

    XmlDocument xmlFile = new XmlDocument();

    1 reference
    public Form1()
    {
        InitializeComponent();
    }

    1 reference
    private void Form1_Load(object sender, EventArgs e)
    {
            ///Dictionary assignment///
        for (int i = 1; i < 9; i++)
        {
            users.Add("Family " + i.ToString(), Tempclient);
        }

    }
```

This C# code creates a form that may be used to launch a socket server. The form includes three variables: a size variable for the data, a byte array to store the data, and a private socket server variable. Additionally, it has a flag variable, a temporary client socket, and a client socket. A dictionary of users and a list of customers are also included. The Form1 Load method inserts "**Family 1**" through "**Family 8**" as keys to the dictionary and gives each key the value of the temporary client socket.

Threaded Server Function:

```
1 reference
public void beginAccept()
{
    for (int i = 0; i < 10; i++)
    {
        server.BeginAccept(new AsyncCallback(AcceptConn), server);
    }
}
```

We go on to our **AcceptConn** function with **AsyncCallback** for 9 clients in the **beginAccept** threaded function.

What does AsyncCallback funtion do?

An asynchronous callback function is a function that is passed as an argument to another function and is executed after the completion of some other operation. The function that is passed as an argument (the callback) is executed asynchronously, meaning that it does not block the execution of the rest of the program while it is running. Instead, it is executed at a later time, usually after an event or some other asynchronous operation has completed. This allows the program to continue executing other code while the callback function is running.

Connection Accepting:

```
void AcceptConn(IAsyncResult iar)
{
    Socket oldserver = (Socket)iar.AsyncState;
    client = oldserver.EndAccept(iar);

    clients.Add(client);

    listBox2.Items.Add(client.RemoteEndPoint.ToString()  + " Date & Time : " + DateTime.Now);

        ///Sending data to client///
    textBox1.Text = "Connected to: " + client.RemoteEndPoint.ToString();
    string stringData = "Welcome to Cins Apertment Management System";
    byte[] message1 = Encoding.ASCII.GetBytes(stringData);

        ///Begin send///

    client.BeginSend(message1, 0, message1.Length, SocketFlags.None,
                new AsyncCallback(SendData), client);
}
```

We add the clients we accept to the list of clients in our "**clients**" section after they have been approved. The listbox "**listbox2**" that will store our logs is where we are pulling the data from.

What then are these statistics?

*We record the client's **remote IP** and **login time** in this log section.*

The approved client is then given the string "Welcome to Genus Apertment Management System." The message is then converted to byte datatype and sent to the server.

Continue Sending:

```
void SendData(IAsyncResult iar)
{
    Socket client = (Socket)iar.AsyncState;
    int sent = client.EndSend(iar);
    client.BeginReceive(data, 0, size, SocketFlags.None,
                new AsyncCallback(ReceiveData), client);
}
```

This function is named "**SendData**" and it is passed an "**IAsyncResult**" object called "**iar**". The first line of the function casts the "**AsyncState**" property of the "**IAsyncResult**" object to a "**Socket**" object and assigns it to a variable named "**client**".

The next line calls the "**EndSend**" method of the "**client**" socket, passing in the "**iar**" object. This method completes a previously started asynchronous send operation and returns the number of bytes sent.The final line of the function initiates another asynchronous operation to receive data from the client. It calls the "**BeginReceive**" method of the "**client**" socket and passes in the "**data**" buffer, the starting offset and the size of the buffer, and a "**SocketFlags**" enumeration value of "**None**". It also passes in an "**AsyncCallback**" delegate that references the "**ReceiveData**" function and the "**client**" socket as the state object. This callback function is used to receive the data sent by the client and it will execute once the send operation is complete.

Receiving Data:

```
void ReceiveData(IAsyncResult iar)
{
    ///Receive Data///

    Socket client = (Socket)iar.AsyncState;
    int recv = client.EndReceive(iar);
    if (recv == 0)
    {
        client.Close();
        MessageBox.Show("No received message. Waiting to connect...");
        server.BeginAccept(new AsyncCallback(AcceptConn), server);
        return;
    }
```

The "**iar**" object is passed when the "**client**" socket's "**EndReceive**" method is called. The number of bytes received is returned once this method completes an earlier began asynchronous receive operation.

The client has disconnected and the socket has closed if the amount of bytes received is 0. Next, the server displays a notification "**No received message. Waiting to connect...**" which invokes the "**BeginAccept**" method of the "**server**" socket to accept new connections and passes a "**AsyncCallback**" delegate that makes use of the "**AcceptConn**" and "**server**" methods "as the state object, socket.

Data from the client is received, processed, and then sent back to the client using this callback function. Once the receiving operation is finished, it will go into action.

```
///Received data///

string receivedData = Encoding.ASCII.GetString(data, 0, recv);
String msg = DateTime.Now.ToString() + " -- ";
```

This code is from the "**ReceiveData**" asynchronous callback function. The next step is to analyze the data after it has been established that it has been received by determining whether or not recv is greater than 0. The first thing it does is use the ASCII encoding to transform the bytes of incoming data into a string. The current date and time are then appended to a newly created string called "**message**" before the punctuation " — ".

The message delivered by the client is contained in this "**receivedData**" variable, and the message with the sender's details and timestamp will be stored in the "**msg**" variable. The receivers of this message will either see it in the listbox or receive it.

```
///Substring for sender///

String[] messageMetas = receivedData.Split("*-*");
```

The subsequent step is to extract the essential data from the received message after converting the received data to a string and producing a new string named "**msg**" with a timestamp.

The client's message, which comprises the sender and the message itself, is included in the "**receivedData**" variable. The "***-***" delimiter is used to divide this message, and the pieces are then placed in a string array named "**messageMetas**." Both the message and the receiver are contained in each entry of the array.

The recipient will be included in the first member of the array "**messageMetas[0]**" and the message itself will be included in the second element. The receiver and the message that has to be sent may be identified using this information.

```
///Adding sender info in the message ///

foreach (KeyValuePair<string, Socket> item in users)
{
    if (item.Key.Equals(messageMetas[1]))
    {
        users[item.Key] = client;
        msg += item.Key;
    }
}

msg += " : " + messageMetas[2];
listBox1.Items.Add(msg);
```

It is used to update the user's socket information in the dictionary entry for "**users**" as well as to add the sender's information to the message.

It cycles over the list of connected clients in the "**users**" dictionary. Each entry in the dictionary is a KeyValuePair made up of a Socket value and a string key (*user name*) (the socket of the user).

It determines if the sender's name matches the key (user name) for each entry in the dictionary (**messageMetas[1]**). If so, it modifies the key value associated with the "client" socket to update the user's socket information in the dictionary. The "**msg**" option additionally adds the sender's name to the message.

```
///Echo server to all clients

byte[] message2 = Encoding.ASCII.GetBytes(msg);

if (messageMetas[0] == "Me")
{
    client.BeginSend(message2, 0, message2.Length, SocketFlags.None,
                                new AsyncCallback(SendData), client);
}

else if (messageMetas[0] == "Family 1")
{
    client.BeginSend(message2, 0, message2.Length, SocketFlags.None,
                    new AsyncCallback(SendData), client);
    users["Family 1"].BeginSend(message2, 0, message2.Length, SocketFlags.None,
                                            new AsyncCallback(SendData), client);
}

else if (messageMetas[0] == "Family 2")
{
    client.BeginSend(message2, 0, message2.Length, SocketFlags.None,
                    new AsyncCallback(SendData), client);
    users["Family 2"].BeginSend(message2, 0, message2.Length, SocketFlags.None,
                                            new AsyncCallback(SendData), client);
}

else if (messageMetas[0] == "Family 3")
{
    client.BeginSend(message2, 0, message2.Length, SocketFlags.None,
                                new AsyncCallback(SendData), client);
    users["Family 3"].BeginSend(message2, 0, message2.Length, SocketFlags.None,
                                            new AsyncCallback(SendData), client);
}
```

The message is prepared to be delivered back to the customers after it has been processed and the recipient identified.

The message "**msg**" is encoded as bytes using the ASCII character set and is then converted to a new byte array called "**message2**"

The recipient indicated in the "**messageMetas**" array's first entry is then verified, and the message is then sent appropriately.

The message is only sent to the sender if "**Me**" is the receiver (client).

The message is delivered to the sender (client) and the designated recipient if the recipient is "**Family 1**" to "**Family 8**" (users["Family x"]).

The message is broadcast to all connected clients if the recipient is set to "**All Family**" (clients[i]).

Through a call to the relevant object's "**BeginSend**" function, it initiates an asynchronous send process.

By calling the "**BeginSend**" method of the relevant socket, passing in the "**message2**" byte array, the starting offset, the size of the buffer, a "**SocketFlags**" enumeration value of "**None**" and a "**AsyncCallback**" delegate that uses the "**SendData**" function and the "client" socket as the state object, it initiates an asynchronous send operation.

When the send operation is finished, this callback function, which is used to transmit the data to the appropriate customers, will run.

Exchange & Weather Conditions:

```
1 reference
private void timer1_Tick_1(object sender, EventArgs e)
{

    string today = "https://www.tcmb.gov.tr/kurlar/today.xml";
    xmlFile.Load(today);

    string dolar = xmlFile.SelectSingleNode("Tarih_Date/Currency[@Kod='USD']/BanknoteSelling").InnerXml;
    string euro = xmlFile.SelectSingleNode("Tarih_Date/Currency[@Kod='EUR']/BanknoteSelling").InnerXml;

    byte[] currency = Encoding.ASCII.GetBytes("$/$" + dolar + "$/$" + euro);

    for (int i = 0; i < clients.Count; i++)
    {
        clients[i].BeginSend(currency, 0, currency.Length, SocketFlags.None,
                new AsyncCallback(SendData), client);
    }
```

It is used to periodically communicate all connected customers the current USD and EUR exchange rate.

The first thing it does is load an XML file from a specified URL that contains the current exchange rates for various currencies.

Then, using the "**SelectSingleNode**" method of the "**xmlFile**" object and an XPath expression to identify the precise nodes holding the USD and EUR exchange rate, it extracts the USD and EUR exchange rate from the XML file.

The exchange rate data is then translated into bytes using the ASCII encoding and placed in a newly created byte array called "**currency**".

The "**BeginSend**" method of the clients[i] socket is then called, passing the "**currency**" byte array, the starting offset, the size of the buffer, the "**SocketFlags**" enumeration value of "**None**" and a "**AsyncCallback**" delegate that refers to the "SendData" function and the "**client**" socket as the state object, after iterating through the list of connected clients.

In this manner, the exchange rate data is broadcast to all connected customers at predetermined intervals.

```
string APIKey = "a1e27152d07c199e4dbf29b0827e119d";
WebClient web = new WebClient();
string url = string.Format("https://api.openweathermap.org/data/2.5/weather?q=manisa&units=metric&appid={0}", APIKey);

var json = web.DownloadString(url);
WeatherInfo.root Info = JsonConvert.DeserializeObject<WeatherInfo.root>(json);

byte[] message1 = Encoding.ASCII.GetBytes("!$#$!" + Info.weather[0].main + "!$#$!" + Info.main.temp.ToString());
for (int i = 0; i < clients.Count; i++)
{
    clients[i].BeginSend(message1, 0, message1.Length, SocketFlags.None,
                                            new AsyncCallback(SendData), client);
}
```

All linked clients are informed via code about the current weather conditions at "**Manisa**" in this example.

It first makes an instance of the "**WebClient**" class and sets the URL of the API request by using the

"**string.Format()**" function to insert the API key into the URL. Next, it establishes a variable called "**APIKey**" that holds the OpenWeatherMap API key.

The JSON data is then downloaded from the API as a string using the "**DownloadString**" method of the "**web**" object, and it is converted into a C# object of type "**WeatherInfo.root**" using the "**JsonConvert.DeserializeObject**" function (which must be a class that matches the structure of the JSON data, and is used to deserialize the Json string).

Weather Info:

```
internal class WeatherInfo
{
    1 reference
    public class coord
    {
        0 references
        public double lon { get; set; }
        0 references
        public double lat { get; set; }
    }

    1 reference
    public class weather
    {
        1 reference
        public string main { get; set; }
        0 references
        public string description { get; set; }
        0 references
        public string icon { get; set; }

    }

    1 reference
    public class main
    {
        1 reference
        public double temp { get; set; }

        0 references
        public double pressure { get; set; }
        0 references
        public double humidity { get; set; }

    }

    1 reference
    public class wind
    {
        0 references
        public double speed { get; set; }

    }

    1 reference
    public class sys
    {
        0 references
        public long sunrise { get; set; }
        0 references
        public long sunset { get; set; }
    }
    2 references
    public class root
    {
        0 references
        public coord coord { get; set; }
        1 reference
        public List<weather> weather { get; set; }
        1 reference
        public main main { get; set; }

        0 references
        public wind wind { get; set; }
        0 references
        public sys sys { get; set; }

    }
```

This is a class called "**WeatherInfo**" that defines the structure of the JSON data that is returned by the OpenWeatherMap API.

It contains several internal classes, each of which corresponds to a JSON object in the API response. Each class contains properties that match the names and data types of the corresponding JSON properties.

The class "**coord**" contains two properties "**lon**" and "**lat**" of type double, which represent the longitude and latitude of the location.

The class "**weather**" contains three properties "main", "**description**" and "**icon**" of type string, which respectively represent the main weather condition, the detailed weather condition, and the icon id.

The class "**main**" contains three properties "**temp**", "**pressure**" and "**humidity**" of type double, which respectively represent the temperature, pressure and humidity of the location.

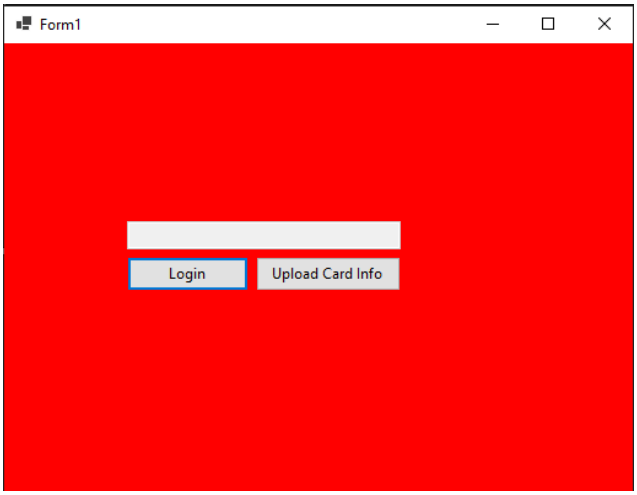
The class "**wind**" contains one property "**speed**" of type double, which represents the wind speed of the location.

The class "**sys**" contains two properties "sunrise" and "**sunset**" of type long, which respectively represent the Unix timestamp of the sunrise and sunset of the location.

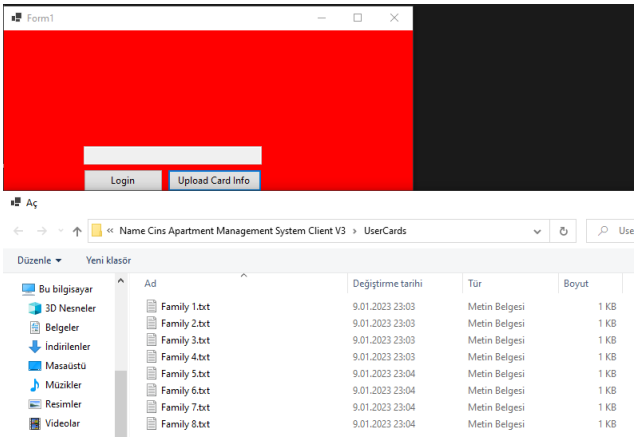
The class "**root**" contains all the classes as properties, and it is used to deserialize the Json string.

By using this class structure, we can easily access and use the data returned by the API in our code.

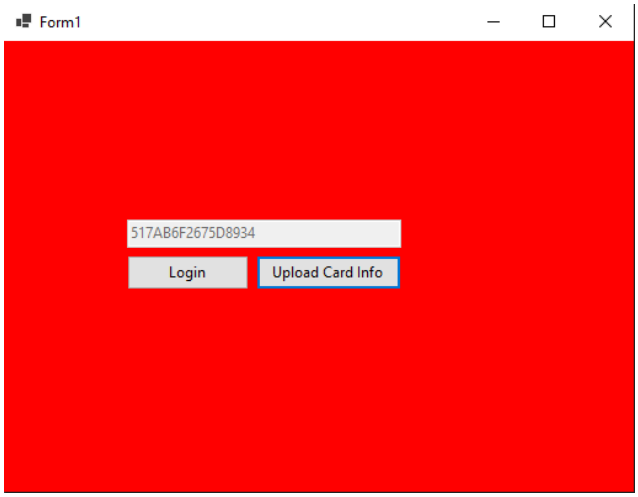
b)Client
Interface:



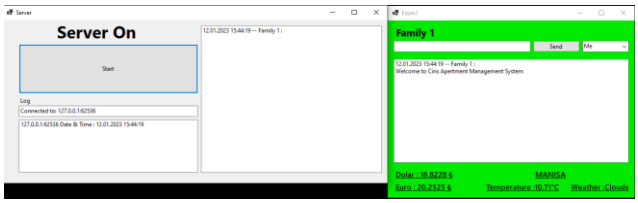
This section disables the user to input text into the textbox but enabled upload card information by selecting from a file.



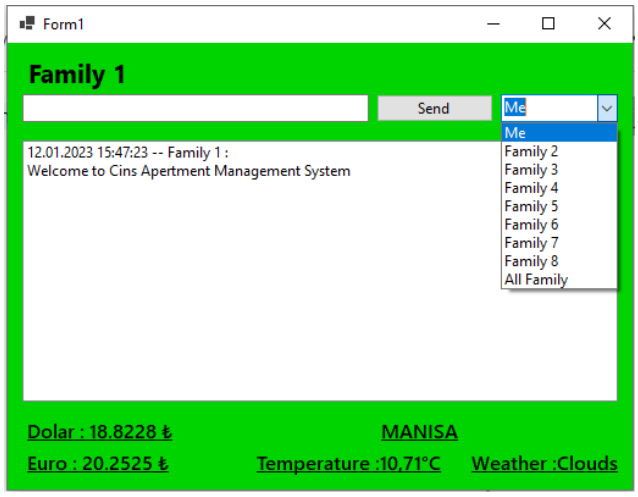
The user selects which family to upload the file to in this part.



Here, the key information comes to the textbox.



If the server is accessible in this case, the family may easily gather there.



You may decide who to speak with from here. "Me" will be the default choice. Additionally, as you can see below, information for dollar and euro will be updated every minute, and the weather data will also automatically change.

Imports:

```
using System;
using System.Diagnostics.Metrics;
using System.Drawing;
using System.Net;
using System.Net.Sockets;
using System.Text;
using System.Threading.Tasks;
using System.Windows.Forms;
```

These are using statements that are bringing in various namespaces to the code file.

- **System**: the fundamental namespace for the C# programming language that includes basic types and commonly used types
- **System.Diagnostics.Metrics:** Provides types for working with performance metrics
- **System.Drawing:** Provides access to GDI+ basic graphics functionality
- **System.Net:** Provides a simple programming interface for a number of network protocols
- **System.Net.Sockets:** Provides a managed implementation of the Berkeley sockets interface for developers who need to tightly control access to the network
- **System.Text:** Provides classes for encoding and decoding characters and strings
- **System.Threading.Tasks:** Provides types that simplify task-based asynchronous programming
- **System.Windows.Forms:** Provides classes for creating Windows-based applications that take full advantage of the rich user interface features available in the Microsoft Windows operating system.

Initializing:

```
public partial class Form1 : Form
{
    private Socket client;
    private byte[] data = new byte[1024];
    private int size = 1024;

    private String[] usersInfo = new String[9];

    1 reference
    public Form1()
    {
        InitializeComponent();
        comboBox1.SelectedIndex = 0;
        for (int i = 1; i < 9; i++)
        {
            usersInfo[i] = "Family " + i.ToString();
        }
    }
```

The form being created by this code includes a private socket variable named "**client**" a private byte array named "**data**" and a private int variable named "**size**". It also features a 9-length private text array named "**usersInfo**"

The form's constructor uses the InitializeComponent() function to initialize the form's components. A comboBox1's chosen index is also set to 0, and the usersInfo array is given values. Every entry in the array is given the value "**Family**" concatenated with the iteration's index number in the for loop.

Send Button:

```
private void button1_Click(object sender, EventArgs e)
{
    ButtonSendOnClick();
}
```

This code is the event handler for the button1_Click event, which is triggered when the button1 is clicked by the user. When this event is triggered, it calls the method ButtonSendOnClick(). The message is constructed by concatenating the selected item from the comboBox1, a separator string "*-*", the text from label2 and another separator string "*-*", and the text from the textBox1.

Function of ButtonSendOnClick:

```
void ButtonSendOnClick()
{
    byte[] message = Encoding.ASCII.GetBytes(comboBox1.SelectedItem.ToString() + "•-•" + label2.Text + "•-•" + text
    textBox1.Clear();
    client.BeginSend(message, 0, message.Length, SocketFlags.None,
                new AsyncCallback(SendData), client);
}
```

When the button is clicked, the method performs the following actions:

1. It creates a byte array called "message" by encoding a string that is composed of the selected item in a combobox (comboBox1), the text in a label (label2), and the text in a textbox (textBox1).
2. It clears the text in the textbox (textBox1)
3. It uses the BeginSend method from the client object to send the message asynchronously using the specified socket flags, and an async callback function called " " using the client object.

Function of ButtonConnectOnClick:

```
void ButtonConnectOnClick()
{
    textBox1.Text = "Connecting...";
    Socket newsock = new Socket(AddressFamily.InterNetwork,
                        SocketType.Stream, ProtocolType.Tcp);
    IPEndPoint iep = new IPEndPoint(IPAddress.Parse("127.0.0.1"), 9050);
    newsock.BeginConnect(iep, new AsyncCallback(Connected), newsock);
}
```

This is a method that is called when a button labeled "**Connect**" is clicked. When the button is clicked, the method performs the following actions:

1. It sets the text of a textbox (textBox1) to "**Connecting...**"
2. It creates a new socket called "**newsock**" using the AddressFamily.InterNetwork, SocketType.Stream, and ProtocolType.Tcp.
3. It creates a new IPEndPoint object called "iep" with the IP address "**127.0.0.1**" and the port number 9050.
4. It uses the BeginConnect method of the "**newsock**" object to asynchronously connect to the specified IPEndPoint, using an async callback function called "**Connected**" and passing "**newsock**" as a parameter.

Asynchronous Connected Function:

```
void Connected(IAsyncResult iar)
{
    textBox1.Text = "";
    client = (Socket)iar.AsyncState;
    try
    {
        Thread receiver = new Thread(new ThreadStart(ReceiveData));
        receiver.Start();

        Thread sender = new Thread(new ThreadStart(sendData));
        sender.Start();

        Thread colorThread = new Thread(color);
        colorThread.Start();

        client.EndConnect(iar);
        client.BeginReceive(data, 0, size, SocketFlags.None,
                    new AsyncCallback(ReceiveData), client);
    }
    catch (SocketException)
    {
        textBox1.Text = "Error connecting";
    }
}
```

When the asynchronous connection attempt started by the "**ButtonConnectOnClick**" method has finished, this code is a callback function that is invoked. The following things happen when the function is called:

1. It replaces the text in **textBox1** with an empty string.
2. The socket object that was sent in along with the IAsyncResult object is set as the client variable.
3. Three new threads are started, one for the "**ReceiveData**" method, one for the "sendData" method, and one for the "**color**" method.
4. The asynchronous connection is finished by using the client socket's EndConnect function.
5. It starts asynchronously receiving data using the client socket's BeginReceive method, supplying the parameters size, data, and ReceiveData callback function.
6. It catches any SocketExceptions and if one occurs, it sets the text of the textbox (textBox1) to "**Error connecting**".

Interface Coloring:

```
void color()
{
    while (true)
    {

        for (int i = 0; i < 200; i++)
        {
            Thread.Sleep(5);
            this.BackColor = Color.FromArgb(0, 50+i, 0);
        }
        for (int i = 0; i < 200; i++)
        {
            Thread.Sleep(5);
            this.BackColor = Color.FromArgb(0, 250 - i, 0);
        }

    }
}
```

The thread begins by going into an endless loop that does the following tasks:

1. It cycles 200 times in a loop, sleeping for 5 ms between iterations while changing the form's background color (according to the RGB color model) by increasing the value of the green channel.
2. Another loop is iterated through 200 times while sleeping for 5 milliseconds each time. The RGB color model is used to change the form's background color by decrementing the value of the green channel.
3. The result is a green color effect on the form that flashes.

It should be warned that this code will continue to execute forever and might eventually lead to performance problems.

Not Delegated sendData Function:

```
void sendData()
{
    byte[] message = Encoding.ASCII.GetBytes(comboBox1.SelectedItem.ToString() + "+-+" + label2.Text + "+-+");
    client.BeginSend(message, 0, message.Length, SocketFlags.None,
     new AsyncCallback(SendData), client);
}
```

The thread begins by doing the following things:

1. By encoding a string made up of the text from a label and the selected item in a combobox (comboBox1), it generates a byte array called "message" (label2).
2. It asynchronously sends the message using the BeginSend method of the "client" socket object, supplying the message's byte array, length, and "SendData" callback function as parameters.

The content of the textbox (textBox1) that the "**ButtonSendOnClick**" function provides is not included in this method, which may cause problems if the textBox1 is intended to be delivered with the message.

Not Delegated ReceiveData Function:

```
void ReceiveData()
{
    int recv;
    string stringData;
    string[] currency;
    string[] weather;
    while (true)
    {
        recv = client.Receive(data);
        stringData = Encoding.ASCII.GetString(data, 0, recv);

        if (stringData.StartsWith("$/$"))
        {
            currency = stringData.Split("$/$");
            label1.Text = "Dolar : " + currency[1] + " ₺";
            label3.Text = "Euro : " + currency[2] + " ₺";
        }
        else if (stringData.StartsWith("!$#$!"))
        {
            weather = stringData.Split("!$#$!");
            label7.Text = "Weather :" + weather[1];
            label6.Text = "Temperature :" + weather[2] + "°C";
        }
        else
        {
            listBox1.Items.Add(stringData);
        }
    }
}
```

The thread begins by going into an endless loop that does the following tasks:

1. The "**recv**" variable holds the number of bytes that were received from the "**client**" socket object.
2. The result is saved in the "**stringData**" variable once the received bytes are converted to a string using the ASCII encoding.
3. The uses to determine whether the "**stringData**" begins with the "**$/$**".
   - If true for the StartsWith method:
   - The "**stringData**" is divided using the "**$/$**" delimiter, and the resulting strings are added to a string array called "**currency**".
   - it changes the label's text to "**Dolar:**" + currency[1] + " ₺"
   - it modifies a label's text to read "**Euro:**" + currency[2] + " ₺"
4. It uses the to determine whether the "stringData" begins with the "**!$#$!**". If true for the StartsWith method:
   - Using the "**!$#$!**" delimiter, it divides the "stringData" and assigns the resulting strings to the "**weather**" string array.
   - It modifies a label's text to read "**Weather:**" + weather[1].

- It modifies a label's text to read "Temperature:" + weather[2] + "**°C**"

5. It adds the "**stringData**" to a listbox if neither of the aforementioned statements is true (listBox1)
6. In an endless loop, it continues to take in and analyze input.

This function probably receives data from a server, then depending on the incoming data's format, changes several labels with the data.

Asynchronous SendData Function:

```
void SendData(IAsyncResult iar)
{
    Socket remote = (Socket)iar.AsyncState;
    int sent = remote.EndSend(iar);
    remote.BeginReceive(data, 0, size, SocketFlags.None,
                new AsyncCallback(ReceiveData), remote);
}
```

When the asynchronous transmit operation started by the "**ButtonSendOnClick**" method or "**sendData**" method is finished, that is when it is called. The following things happen when the function is called:

1. The socket object used for the transmit operation is taken from the IAsyncResult object and assigned to the "**remote**" variable.

2. To finish the asynchronous transmit operation, it calls the EndSend method of the "**remote**" socket and records the total amount of bytes transferred in the "sent" variable.

3. It begins asynchronously receiving data using the BeginReceive method of the "**remote**" connection, supplying the parameters size, data, and ReceiveData callback function. In order to be prepared to receive any new messages, this method is probably called after the message has been delivered.

Login Button:

```
private void button2_Click(object sender, EventArgs e)
{
    String cardId = textBox2.Text;

    for (int i = 0; i < 9; i++)
    {
        if ("Family "+cardId[5].ToString() == usersInfo[i])
        {
            label1.Visible = true;
            label2.Visible = true;
            label3.Visible = true;
            textBox1.Visible = true;
            listBox1.Visible = true;
            comboBox1.Visible = true;
            button1.Visible = true;
            label6.Visible = true;
            label7.Visible = true;
            label8.Visible = true;

            textBox2.Visible = false;
            button2.Visible = false;
            button3.Visible = false;

            label2.Text = "Family " + cardId[5].ToString();
            comboBox1.Items.Remove("Family " + cardId[5].ToString());
            ButtonConnectOnClick();
        }
    }
}
```

When the button is pressed, the following things happen:

1. It creates a variable called "**cardId**" and sets its value to the content contained in a textbox (textBox2).
2. It then begins a 9-iteration for loop.
3. It determines if an element of an array called usersInfo at the current position of the loop matches the text "**Family**" plus the sixth letter of cardId (cardId[5]) within the loop.
4. Labels, textboxes, listboxes, combo boxes, buttons, and other form components become visible if the check is true. Additionally, textBox 2, Button 2, and Button 3 become invisible.
5. The text of label2 is changed to "**Family**" and the sixth character of cardId.
6. It eliminates "**Family**" plus the sixth character of cardId from the combobox1 items.
7. The ButtonConnectOnClick function is invoked.
8. This function appears to be used to verify a user's cardId supplied by the user; it determines if the cardId is legitimate by comparing it to potential family IDs, and if the card is valid, it provides the user access to the form's elements and the network connection.

Send with Enter Key:

```
private void textBox1_KeyPress(object sender, KeyPressEventArgs e)
{
    if (e.KeyChar == (char)Keys.Enter)
    {
        ButtonSendOnClick();
    }
}
```

The following things happen when the enter key is hit with the textbox in focus:

The function "**ButtonSendOnClick()**" is called.

This function responds to the textbox1's "**KeyPress**" event and determines if the key being pressed is the "**Enter**" key before invoking the "**ButtonSendOnClick()**" method. As a result, entering text into the textbox1 while pressing the enter key causes the message to be sent in the same way as hitting the "**Send**" button would.

Uploading Button For User Cards:

```
private void button3_Click(object sender, EventArgs e)
{
    try
    {
        openFileDialog1.FileName = "*";
        openFileDialog1.Filter= "Text File (*txt)|*.txt";
        openFileDialog1.ShowDialog();
        StreamReader read = new StreamReader(openFileDialog1.FileName);
        textBox2.Text = read.ReadToEnd();
        read.Close();
    }
    catch {

    }
}
```

The following things happen when the button is pressed:

1. An attempt-catch block is produced.
2. It changes the Filter property to "**Text File (txt)|.txt**" and the FileName property of an openFileDialog object (openFileDialog1) to "*****" in the try block so that the user may only select.txt files.
3. The file dialog box is opened for the user to choose a file once it calls the ShowDialog method of the openFileDialog object.
4. The user-selected file name (openFileDialog1.FileName) is sent as a parameter when a new StreamReader object is created.
5. Using the ReadToEnd function of the StreamReader object, it reads the whole contents of the chosen file and assigns it to the textBox2.
6. The StreamReader object is shut off.
7. There is nothing it does in the catch block.

This technique makes it possible for the user to choose a.txt file and read its content into the textBox 2. It also makes use of a try-catch block to deal with any potential exceptions that could arise.

Form Loading:

```
1 reference
private void Form1_Load(object sender, EventArgs e)
{
    textBox2.Enabled = false;
    this.BackColor = Color.Red;
}
```

The following activities happen when the form loads:

1. The textbox with the label "**textBox2**" is disabled.
2. Using the BackColor attribute, it changes the form's background color to red.

This function is used to set the initial state of the form and its elements. It disables the textBox2 and changes the form's background color to red when the form is loaded.

Conclusion

A C# Windows Forms application uses the code that is given. To build a networked chat application, it has a number of methods and event handlers. The code contains methods for establishing connections to servers, transmitting and receiving data, validating user cardIds, managing file operations, and handling GUI activities. The program has a number of features, including the ability to send and receive messages, handle weather and currency data, show conversation history, and manage user access by verifying their card ID. The.NET framework and its classes are well utilized by the code to handle network and GUI tasks, and it is well organized, clear to read, and comprehend.

References

https://www.w3schools.com/cs/index.php
https://learn.microsoft.com/en-us/dotnet/csharp/
https://stackoverflow.com/
C# Network Programming by Richard Blum