Rodney Harris

Class Chat System

Overview:

This project was created using the JetBrains Rider 2022.2.2 IDE and I used C# 10 and .NET 6.0.

To run the program with dotnet, you would need to install the proper dotnet SDK from the Microsoft website: https://dotnet.microsoft.com/download

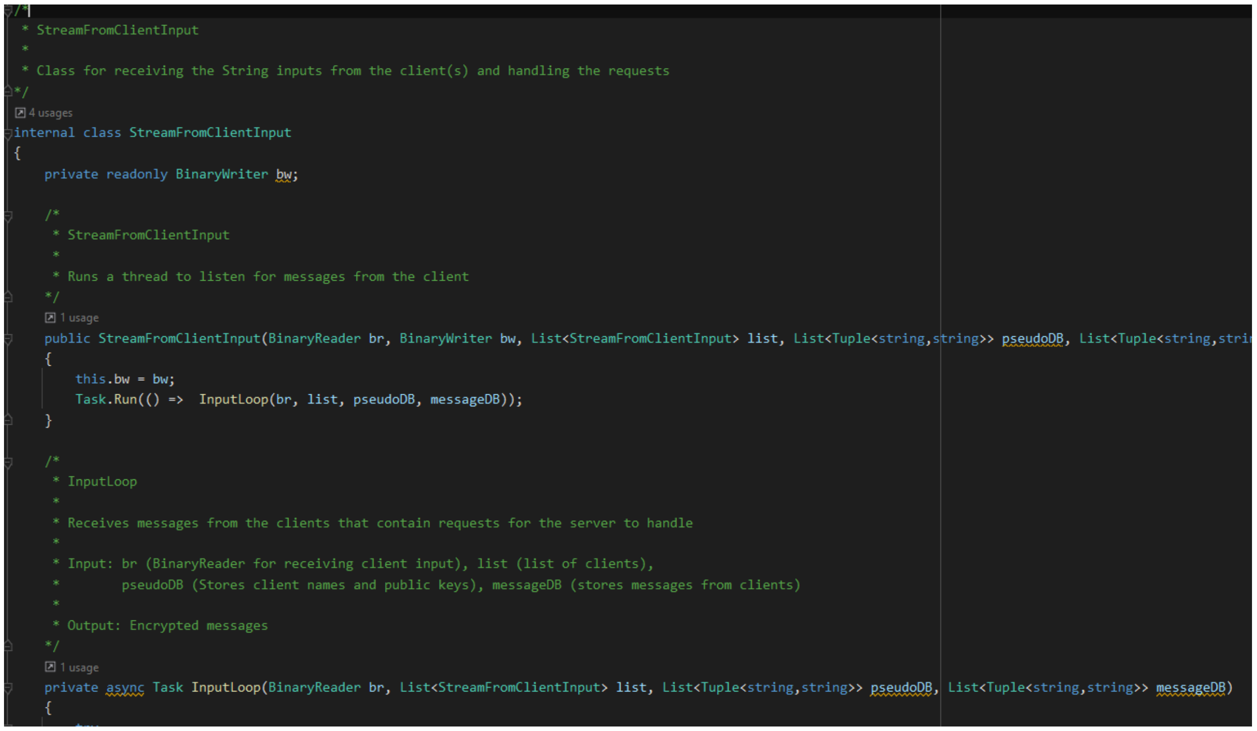
You would also need to install the Rider itself. Rider is paid service but there is a 30-day free trial and UL students have free accounts. https://www.jetbrains.com/rider/download/#section=windows

Assuming you have these installed correctly, you will be able to open the projects from their directories and run them in the IDE.

My class chat system was created using C#/dotnet 6.0 in JetBrains Rider. For the purposes of this project, I have created three different project instances in Rider: one server instance and two client instances.

The Server:

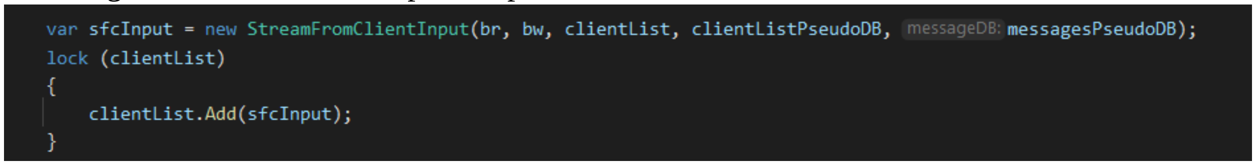
The server creates a socket for communication using methods from C# libraries. In an internal class labeled StreamFromClientInput, I can open the socket to listen for messages from the client and establish my binary writer which will be used to write messages back to the client. An asynchronous Task, Input Loop, will be used to receive messages back from the client.



```
/*
 * StreamFromClientInput
 *
 * Class for receiving the String inputs from the client(s) and handling the requests
*/
4 usages
internal class StreamFromClientInput
{
    private readonly BinaryWriter bw;

    /*
     * StreamFromClientInput
     *
     * Runs a thread to listen for messages from the client
    */
    1 usage
    public StreamFromClientInput(BinaryReader br, BinaryWriter bw, List<StreamFromClientInput> list, List<Tuple<string,string>> pseudoDB, List<Tuple<string,stri
    {
        this.bw = bw;
        Task.Run(() =>  InputLoop(br, list, pseudoDB, messageDB));
    }

    /*
     * InputLoop
     *
     * Receives messages from the clients that contain requests for the server to handle
     *
     * Input: br (BinaryReader for receiving client input), list (list of clients),
     *        pseudoDB (Stores client names and public keys), messageDB (stores messages from clients)
     *
     * Output: Encrypted messages
    */
    1 usage
    private async Task InputLoop(BinaryReader br, List<StreamFromClientInput> list, List<Tuple<string,string>> pseudoDB, List<Tuple<string,string>> messageDB)
    {
```

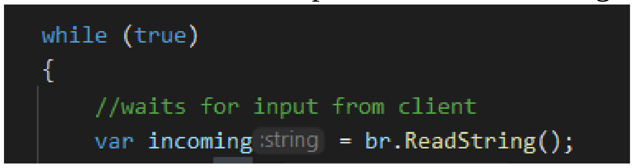
Multi-Thread Communication Server

And because the server needs to be able to interact with more than one client, I created a list for StreamFromClientInput that can each client every time a new one runs. With this list being used in the socket loop, multiple clients can interact with the server at once.



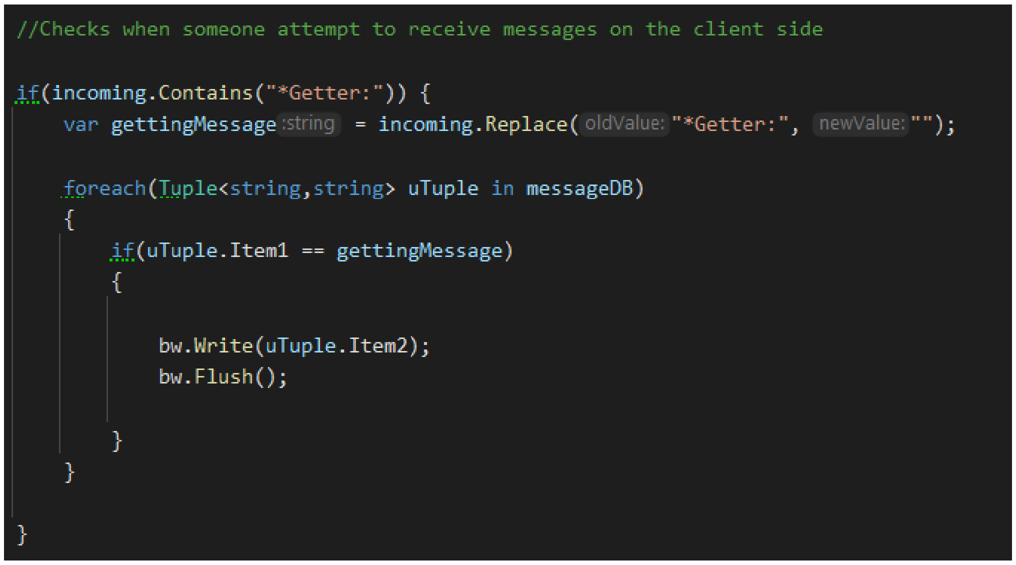
```
var sfcInput = new StreamFromClientInput(br, bw, clientList, clientListPseudoDB, messageDB: messagesPseudoDB);
lock (clientList)
{
    clientList.Add(sfcInput);
}
```

Meanwhile, my binary reader, which is responsible for reading messages from the server runs in a continuous loop as it waits for messages from the clients.

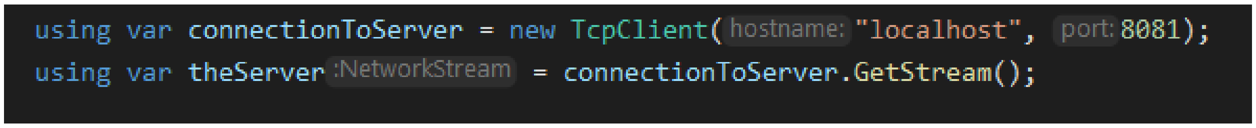


```
while (true)
{
    //waits for input from client
    var incoming :string = br.ReadString();
```

I sort through the messages based on the keywords included in the string received. For example, in the if statement below, I check to see if “*Getter:” is contained in a string so that I know which user I need to return the encrypted messages for.


```
//Checks when someone attempt to receive messages on the client side

if(incoming.Contains("*Getter:")) {
    var gettingMessage :string  = incoming.Replace( oldValue: "*Getter:",  newValue: "");

    foreach(Tuple<string,string> uTuple in messageDB)
    {
        if(uTuple.Item1 == gettingMessage)
        {

            bw.Write(uTuple.Item2);
            bw.Flush();

        }
    }

}
```

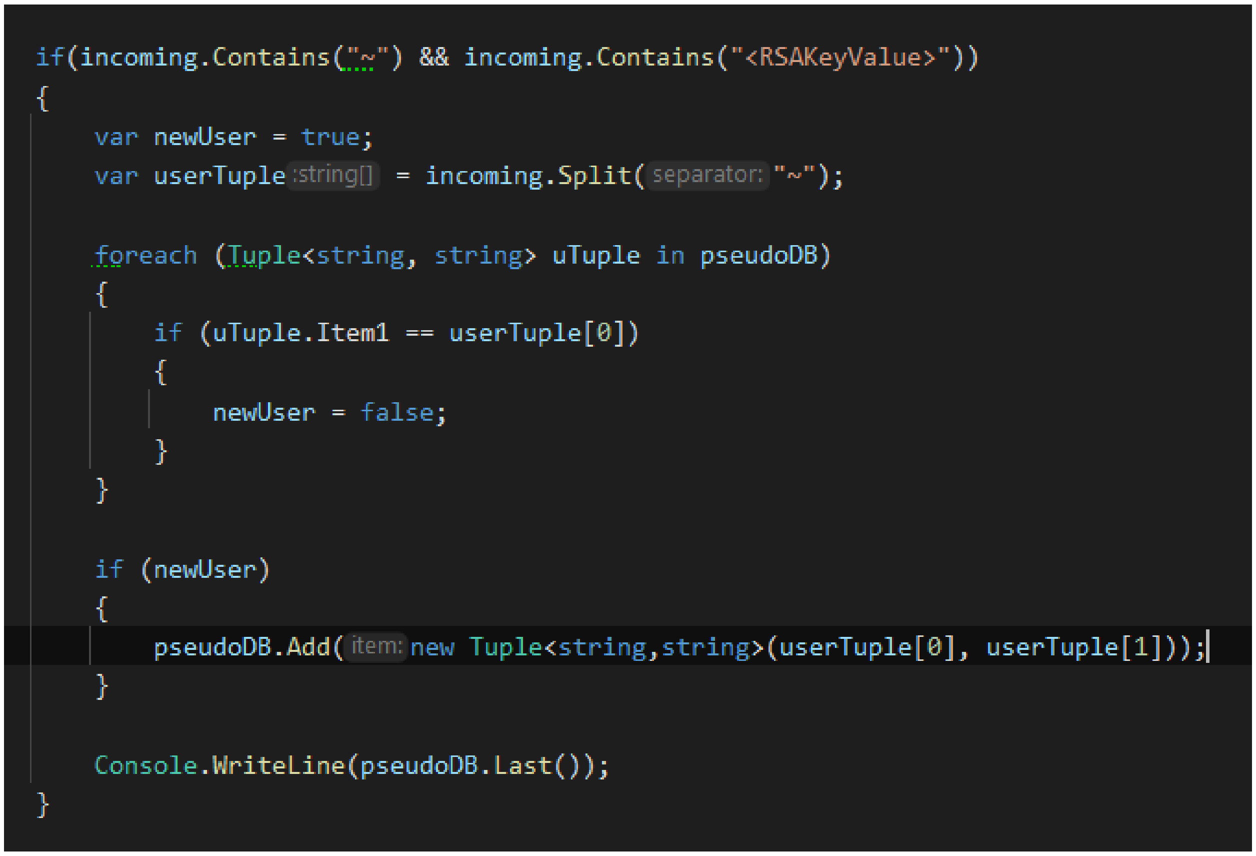
TCPListener allowed me to bind the local port and connection address needed for the server.


```
using var connectionToServer = new TcpClient( hostname: "localhost",  port: 8081);
using var theServer :NetworkStream  = connectionToServer.GetStream();
```

Encryption: Public Keys:

Because this system includes encryption, our server creates a list which is used to store user’s public keys so that when users attempt to send messages between each other, they can retrieve each other’s public keys from the server.

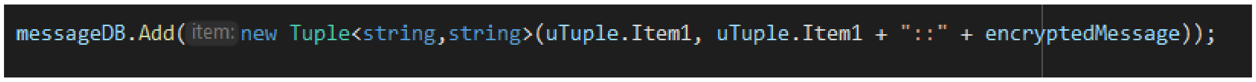
In the if check below, a user’s information is stored in the server list if the RSAKey string is detected.



```
if(incoming.Contains("~") && incoming.Contains("<RSAKeyValue>"))
{
    var newUser = true;
    var userTuple :string[] = incoming.Split( separator: "~");

    foreach (Tuple<string, string> uTuple in pseudoDB)
    {
        if (uTuple.Item1 == userTuple[0])
        {
            newUser = false;
        }
    }

    if (newUser)
    {
        pseudoDB.Add( item: new Tuple<string,string>(userTuple[0], userTuple[1]));
    }

    Console.WriteLine(pseudoDB.Last());
}
```

[Bonus: Offline Messaging] Message List:

If the server is running, all messaged will be stored in a list known as messageDB which stores a Tuple containing the name of the user the message is intended for as well as the encrypted message for that user.

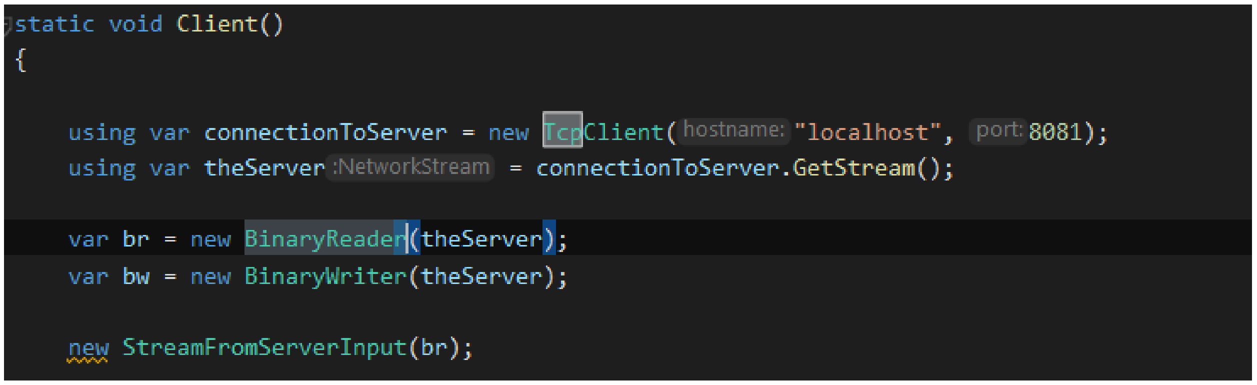


```
messageDB.Add( item: new Tuple<string,string>(uTuple.Item1, uTuple.Item1 + "::" + encryptedMessage));
```

So long as the client holds the private key within the correct directory, the user can restart their client as many times as they'd like and still retrieve their messages stored on the server.

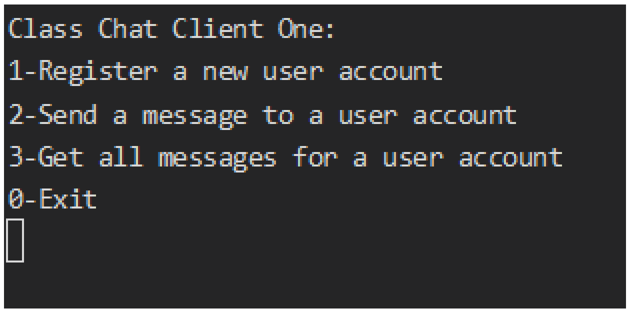
The Clients:

The programming for both clients is essentially identical. Much like the server side, the clients use the TCPClient to configure the TCP protocol as well as open a new stream by using the C# NetworkStream class.



```
static void Client()
{

    using var connectionToServer = new TcpClient( hostname: "localhost",  port: 8081);
    using var theServer :NetworkStream  = connectionToServer.GetStream();

    var br = new BinaryReader(theServer);
    var bw = new BinaryWriter(theServer);

    new StreamFromServerInput(br);
```

The interface for the clients is created using WriteLine statements which present command options for whomever is using the client. Commands are selected by entering a number.

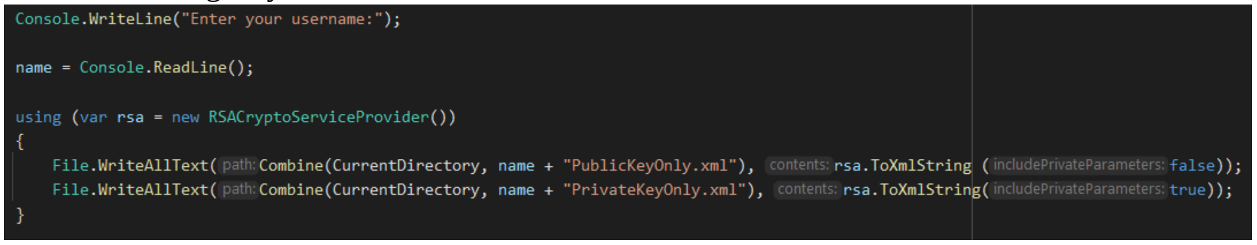


```
Class Chat Client One:
1-Register a new user account
2-Send a message to a user account
3-Get all messages for a user account
0-Exit
```

The first option allows someone to create a new user. Upon creating a new user, the client will generate a public and private key, both of which will be stored in the project directory. The client will also write to the server a message containing the users. username and public key so that it is stored in the Server's client list.

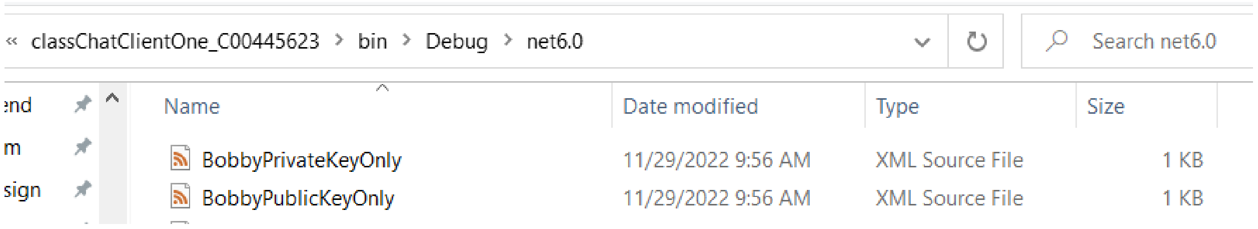
Encryption: Generating Keys:

Code Generating Keys:

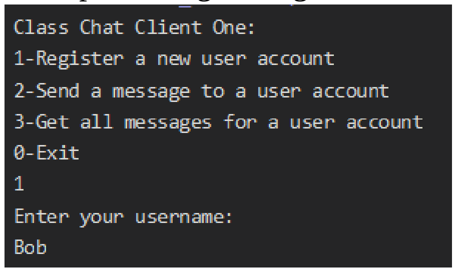


```
Console.WriteLine("Enter your username:");

name = Console.ReadLine();

using (var rsa = new RSACryptoServiceProvider())
{
    File.WriteAllText( path: Combine(CurrentDirectory, name + "PublicKeyOnly.xml"),  contents: rsa.ToXmlString ( includePrivateParameters: false));
    File.WriteAllText( path: Combine(CurrentDirectory, name + "PrivateKeyOnly.xml"),  contents: rsa.ToXmlString( includePrivateParameters: true));
}
```

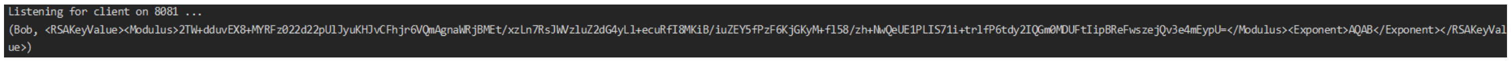
Files Generated:


```
« classChatClientOne_C00445623 › bin › Debug › net6.0

Name                      Date modified         Type              Size
BobbyPrivateKeyOnly       11/29/2022 9:56 AM    XML Source File   1 KB
BobbyPublicKeyOnly        11/29/2022 9:56 AM    XML Source File   1 KB
```

Example for registering user:


```
Class Chat Client One:
1-Register a new user account
2-Send a message to a user account
3-Get all messages for a user account
0-Exit
1
Enter your username:
Bob
```

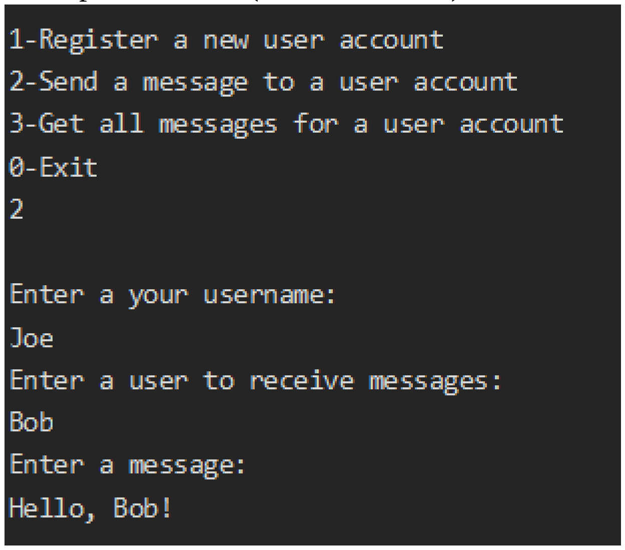
Server Confirmation that the username and public key have been received:


```
Listening for client on 8081 ...
(Bob, <RSAKeyValue><Modulus>2TW+dduvEX8+MYRFz022d22pUlJyuKHJvCFhjr6VQmAgnaWRjBMEt/xzLn7RsJWVzluZ2dG4yLl+ecuRfI8MKiB/iuZEY5fPzF6KjGKyM+fl58/zh+NwQeUE1PLIS71i+trlfP6tdy2IQGm0MDUFtIipBReFwszejQv3e4mEypU=</Modulus><Exponent>AQAB</Exponent></RSAKeyVal
ue>)
```

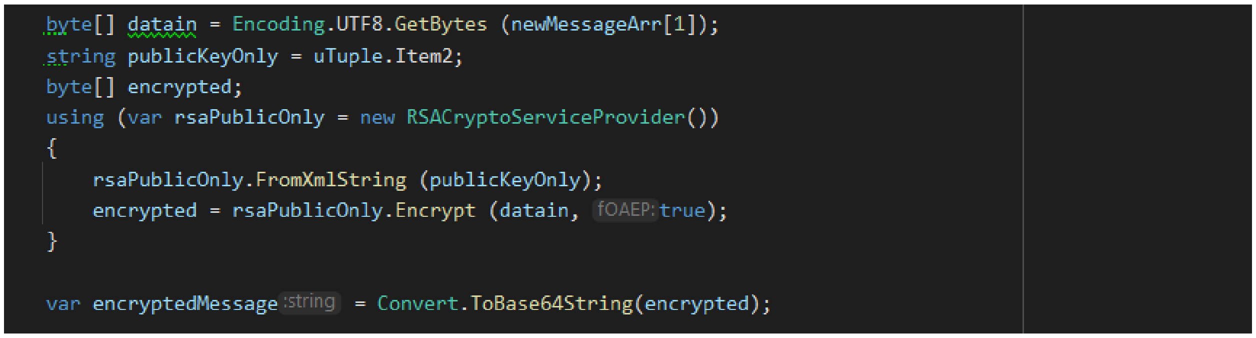
Client-to-Client Communication:

The second option allows users to send a message to other users through the server. After entering the necessary information through the interface, the message is then encrypted before it is sent to the next client.

Example Interface (On Client Two):


```
1-Register a new user account
2-Send a message to a user account
3-Get all messages for a user account
0-Exit
2

Enter a your username:
Joe
Enter a user to receive messages:
Bob
Enter a message:
Hello, Bob!
```

Code for encrypting message:

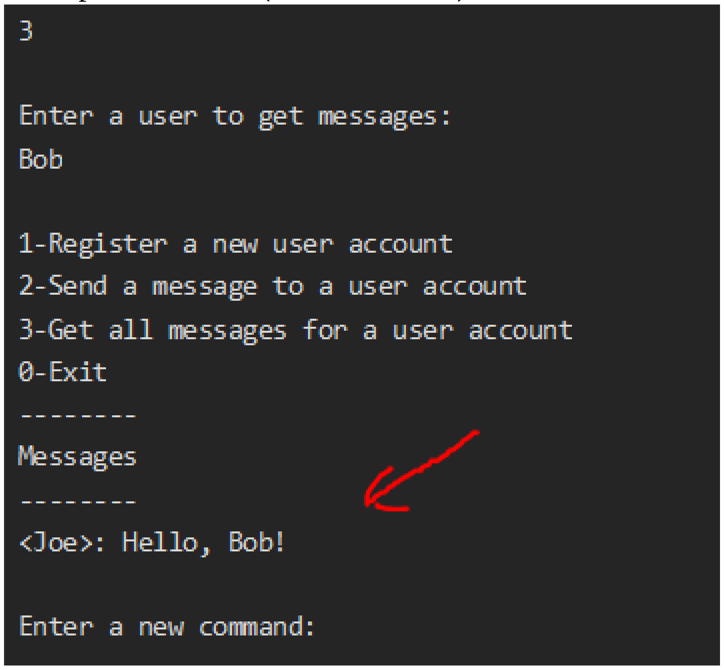


```
byte[] datain = Encoding.UTF8.GetBytes (newMessageArr[1]);
string publicKeyOnly = uTuple.Item2;
byte[] encrypted;
using (var rsaPublicOnly = new RSACryptoServiceProvider())
{
    rsaPublicOnly.FromXmlString (publicKeyOnly);
    encrypted = rsaPublicOnly.Encrypt (datain, fOAEP: true);
}

var encryptedMessage :string  = Convert.ToBase64String(encrypted);
```

Advanced Client (socket receives data from keyboard input):

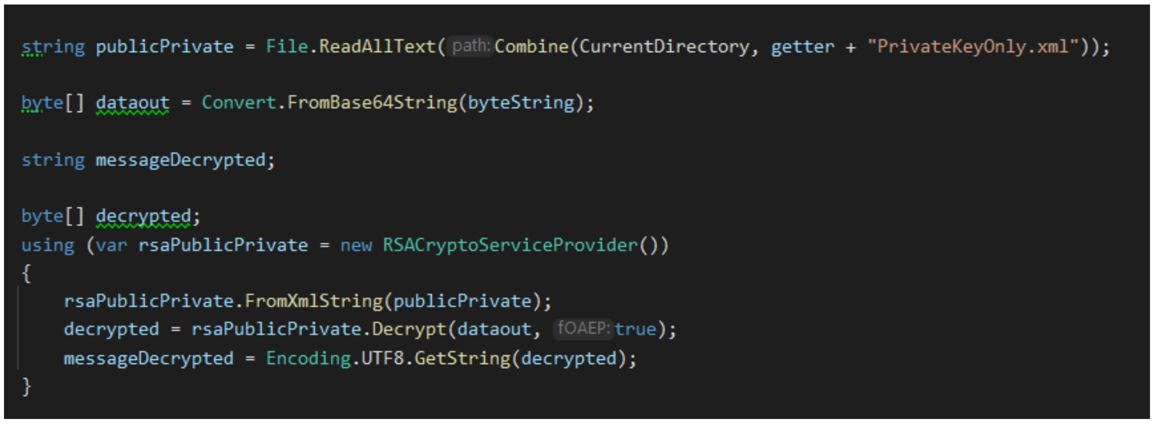
The third option allows users to retrieve their encrypted messages stored on the server side by entering their username. Once those encrypted messages are received, the client will then search for a locally stored private key containing the username the messages are intended for. If that private key is present, the new messages will be decrypted for the user to read.

Example continued (On Client One):



```
3

Enter a user to get messages:
Bob

1-Register a new user account
2-Send a message to a user account
3-Get all messages for a user account
0-Exit
--------
Messages
--------
<Joe>: Hello, Bob!

Enter a new command:
```

Code for decryption:



```
string publicPrivate = File.ReadAllText( path: Combine(CurrentDirectory, getter + "PrivateKeyOnly.xml"));

byte[] dataout = Convert.FromBase64String(byteString);

string messageDecrypted;

byte[] decrypted;
using (var rsaPublicPrivate = new RSACryptoServiceProvider())
{
    rsaPublicPrivate.FromXmlString(publicPrivate);
    decrypted = rsaPublicPrivate.Decrypt(dataout,  fOAEP: true);
    messageDecrypted = Encoding.UTF8.GetString(decrypted);
}
```