Simple Instant Messenger with SSL Encryption in C#

TeapotDev

27 Jul 2012 CPOL

Instant messaging in C# .NET using TCP protocol with SSL encryption and authentication.

Download demo project - 17.1 KB

Download source - 29.1 KB

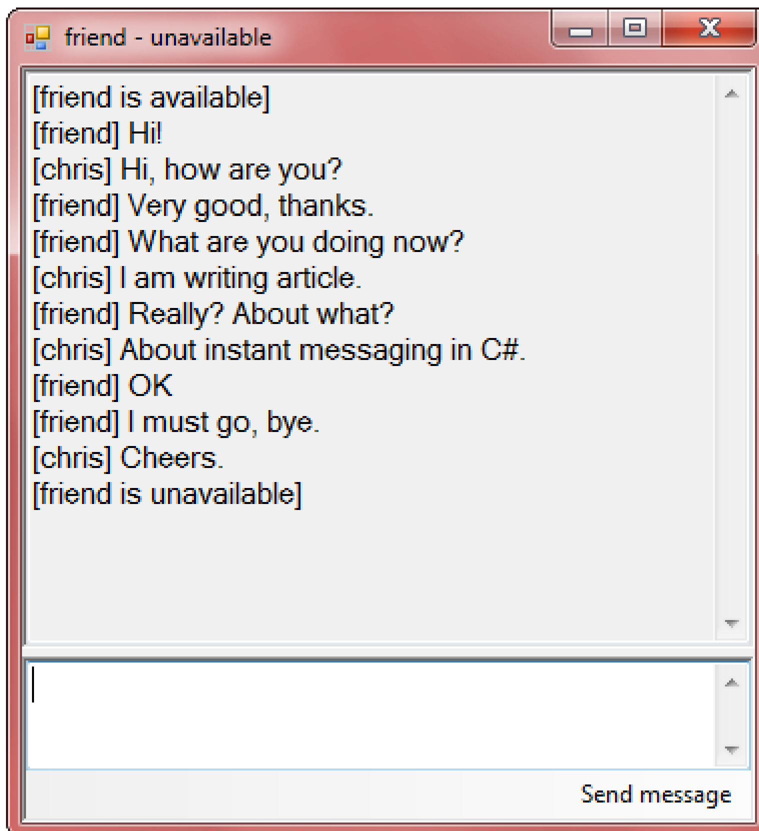


Table of contents

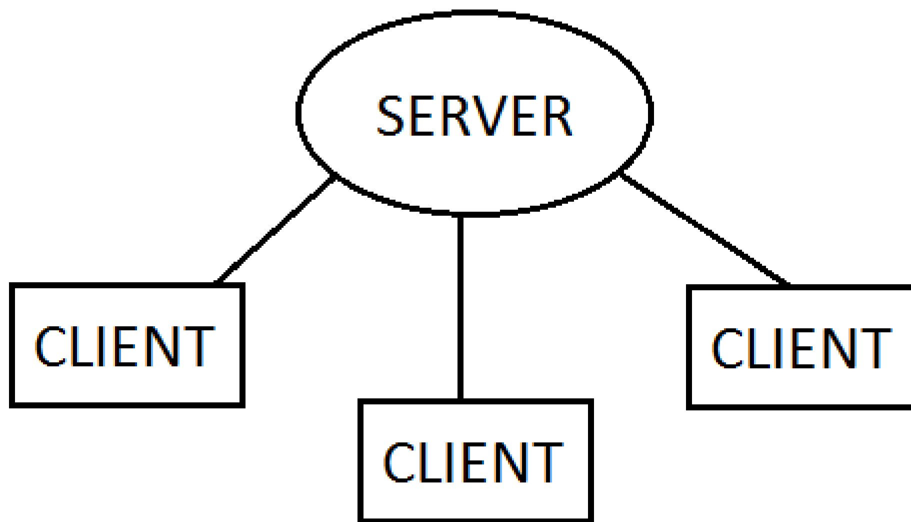
1. Introduction
2. Background
3. Preparing
4. Listen! Incoming connection!
5. Connection
6. Packet types
7. Say hello
8. Events
9. Register and login
10. Packets receiving loop

11. Save users information
12. Check other users availability
13. Culmination: Sending and receiving messages
14. User interface
15. Conclusion

Introduction

Did you ever want to write your own instant messenger program like Skype? OK, not so advanced... I will try to explain how to write a simple instant messenger (IM) in C#.NET.

First, some theory. Our instant messenger will work on a client-server model.



Users have client programs which connect to the server application. Client programs know the server's IP or hostname (e.g., example.com).

The most popular internet protocols are TCP and UDP. We will use TCP/IP, because it is reliable and it has established connection. .NET offers `TcpClient` and `TcpListener` classes for this protocol. TCP/IP doesn't offer encryption. It is possible to create own encryption protocol over TCP, but I recommend using SSL (used in HTTPS). It authenticates server (and optionally client) and encrypts connection.

SSL is using X.509 certificates for authenticating. You can buy real SSL certificate (trusted) or generate self-signed certificate (untrusted). Untrusted certificates allow encryption, but authentication isn't safe. We can use them for testing. I made batch script, which generates self-signed certificate in PFX package. My script requires *OpenSSL* installed in system. I included also one in server application project.

At the end there is your higher-level protocol, which sends messages to specified users and does other IM stuff. I will explain my protocol during article.

You can debug your server and client on the same computer: hostname of server will be *localhost* or *127.0.0.1* (local IP - same computer).

Background

You should know something about SSL protocol, certificates, and networking.

Preparing

Create two projects: server and client. Server will be a console application, client - Windows Forms (or WPF). You will need to debug two projects at once, so don't place them in one solution.

Server

We will write the main server code in non-static scope. Add these lines to `Main`:

```
Program p = new Program();
Console.WriteLine();
Console.WriteLine("Press enter to close program.");
Console.ReadLine();
```

It will create a new instance of the `Program` (class containing `Main`). The server code will be in the constructor.

```
public Program()
{
    Console.Title = "InstantMessenger Server";
    Console.WriteLine("----- InstantMessenger Server -----");
}
```

Client

Add class `IMClient` - it will process and send all packets of our protocol. Add basic variables:

```
Thread tcpThread;      // Connection thread
bool _conn = false;    // Is connected/connecting?
bool _logged = false;  // Is logged in?
string _user;          // Username
string _pass;          // Password
bool reg;              // Register mode
```

And some properties:

```
public string Server { get { return "localhost"; } }
public int Port { get { return 2000; } }

public bool IsLoggedIn { get { return _logged; } }
public string UserName { get { return _user; } }
public string Password { get { return _pass; } }
```

`Server` is the name or IP of the computer where the server software is running. We will test the IM on one computer so the address of the server is *localhost*. `Port` is the TCP port of the server. For example, HTTP default port is 80. These methods will be used to connect and disconnect:

```
void SetupConn()  // Setup connection and login
{
}
void CloseConn() // Close connection.
{
}
```

Finally public methods for login, register, and disconnect.

```
// Start connection thread and login or register.
void connect(string user, string password, bool register)
{
    if (!_conn)
    {
        _conn = true;
        _user = user;
        _pass = password;
        reg = register;

        // Connect and communicate to server in another thread.
        tcpThread = new Thread(new ThreadStart(SetupConn));
        tcpThread.Start();
    }
}
```

```
public void Login(string user, string password)
{
    connect(user, password, false);
}
public void Register(string user, string password)
{
    connect(user, password, true);
}
public void Disconnect()
{
    if (_conn)
        CloseConn();
}
```

Listen! Incoming connection!

The server will listening to incoming connections. Some variables at the start:

```
public IPAddress ip = IPAddress.Parse("127.0.0.1");
public int port = 2000;
public bool running = true;
public TcpListener server;  // TCP server
```

These lines in the constructor will create and start the server:

```
server = new TcpListener(ip, port);
server.Start();
```

The server is started. Now listen to incoming connections:

```
void Listen()  // Listen to incoming connections.
{
    while (running)
    {
        TcpClient tcpClient = server.AcceptTcpClient();
    }
}
```

`AcceptTcpClient` waits for incoming connections and then it returns it as a `TcpClient`. Now we have to handle the client. Create a class for this and name it `Client`. Then add a constructor and these variables:

```
public Client(Program p, TcpClient c)
{
    prog = p;
    client = c;
}

Program prog;
public TcpClient client;
```

In the method for listening, we are passing `tcpClient` and `Program` instances to the `Client` class:

```
Client client = new Client(this, tcpClient);
```

In the class for handling the client, add these functions (again):

```
void SetupConn()  // Setup connection
{
}
void CloseConn() // Close connection
{
}
```

And finally in the constructor (of the `Client` class), run code for preparing connection in another thread.

```
(new Thread(new ThreadStart(SetupConn))).Start();
```

It's time to setup a connection.

Connection

We have to establish a connection. The following code will be placed in the server's `Client` class and in the client's `IMClient`. We need the following variables:

```
public TcpClient client;
public NetworkStream netStream;  // Raw-data stream of connection.
public SslStream ssl;            // Encrypts connection using SSL.
public BinaryReader br;          // Read simple data
public BinaryWriter bw;          // Write simple data
```

Connect to the server (only at client):

```
client = new TcpClient(Server, Port);
```

Now let's get a stream of the connection. We can read and write raw data using this stream.

```
netStream = client.GetStream();
```

OK, we can read and write, but it isn't encrypted. Let's add SSL.

```
// At server side
ssl = new SslStream(netStream, false);
```

```
// At client side
ssl = new SslStream(netStream, false,
    new RemoteCertificateValidationCallback(ValidateCert));
```

When the server is authenticating, the client has to confirm the certificate. `SslStream` checks the certificate and then passes the results to `RemoteCertificateValidationCallback`. We need to confirm the certificate in the callback. Add this function (it is passed in the `SslStream` constructor):

```
public static bool ValidateCert(object sender, X509Certificate certificate,
              X509Chain chain, SslPolicyErrors sslPolicyErrors)
{
    return true; // Allow untrusted certificates.
}
```

For testing we are using an untrusted certificate, so we are ignoring policy errors and accepting all certificates. SSL needs a certificate. You have to generate one (I made a script for this) or you can use the certificate included in the source code. Let's load this in the `Program` class.

```
public X509Certificate2 cert = new X509Certificate2("server.pfx", "instant");
```

The second parameter is the password of the certificate. My batch script is automatically setting the password to "***instant***". Now authenticate the server and client:

```
// Server side
ssl.AuthenticateAsServer(prog.cert, false, SslProtocols.Tls, true);
```

```
// Server side
ssl.AuthenticateAsClient("InstantMessengerServer");
```

Now the connection is authenticated and encrypted. We need some reader and writer for simple data, such as integers, strings, etc.

```
br = new BinaryReader(ssl, Encoding.UTF8);
bw = new BinaryWriter(ssl, Encoding.UTF8);
```

There remains closing the connection.

```
void CloseConn()
{
    br.Close();
    bw.Close();
    ssl.Close();
    netStream.Close();
    client.Close();
}
```

```
CloseConn(); // At the end of SetupConn method
```

We are ready to communicate.

Packet types

In this instant messenger packets start from byte type and then there are other data written with `BinaryWriter`. These are the packet types used in my protocol (as yet):

```
public const int IM_Hello = 2012;      // Hello
public const byte IM_OK = 0;           // OK
public const byte IM_Login = 1;        // Login
public const byte IM_Register = 2;     // Register
public const byte IM_TooUsername = 3;  // Too long username
public const byte IM_TooPassword = 4;  // Too long password
public const byte IM_Exists = 5;       // Already exists
public const byte IM_NoExists = 6;     // Doesn't exists
public const byte IM_WrongPass = 7;    // Wrong password
public const byte IM_IsAvailable = 8;  // Is user available?
public const byte IM_Available = 9;    // User is available or not
public const byte IM_Send = 10;        // Send message
public const byte IM_Received = 11;    // Message received
```

Put them in the client and server.

Say hello

For courtesy reasons, the client and server should greet themselves... The server will do that first, then the client.

Server

First, write `IM_HELLO`.

```
bw.Write(IM_Hello);
bw.Flush();  // Clears buffer - sends all data to client.
```

Second, receive hello.

```
int hello = br.ReadInt32();
```

Third, check if hello is OK.

```
if (hello == IM_Hello)
{
    // Hello is OK, continue.
}
```

Client

First, receive hello.

```
int hello = br.ReadInt32();
```

Second, check if hello is OK.

```
if (hello == IM_Hello)
{
    // Hello is OK, continue.
}
```

Third, write `IM_HELLO`.

```
bw.Write(IM_Hello);
bw.Flush();  // Clears buffer - sends all data to client.
```

Events

Now we have to write our own event args, handlers, and define events for the `IMClient` class. First, an enumeration with errors. It will be used for error events (e.g., login failed).

```
public enum IMError : byte
{
    TooUserName = IMClient.IM_TooUsername,
    TooPassword = IMClient.IM_TooPassword,
    Exists = IMClient.IM_Exists,
    NoExists = IMClient.IM_NoExists,
    WrongPassword = IMClient.IM_WrongPass
}
```

It is created from packet types. It will contain all errors. Now event args for error events.

```
public class IMErrorEventArgs : EventArgs
{
    IMError err;

    public IMErrorEventArgs(IMError error)
    {
        this.err = error;
    }

    public IMError Error
    {
        get { return err; }
    }
}
```

And custom event handler:

```
public delegate void IMErrorEventHandler(object sender, IMErrorEventArgs e);
```

We can define other event args for future use now.

```
public class IMAvailEventArgs : EventArgs
{
    string user;
    bool avail;

    public IMAvailEventArgs(string user, bool avail)
    {
        this.user = user;
        this.avail = avail;
    }

    public string UserName
    {
        get { return user; }
    }
    public bool IsAvailable
    {
        get { return avail; }
    }
}
public class IMReceivedEventArgs : EventArgs
{
    string user;
    string msg;

    public IMReceivedEventArgs(string user, string msg)
    {
        this.user = user;
        this.msg = msg;
    }

    public string From
    {
        get { return user; }
    }
    public string Message
    {
        get { return msg; }
    }
}
```

Corresponding handlers:

```
public delegate void IMAvailEventHandler(object sender, IMAvailEventArgs e);
public delegate void IMReceivedEventHandler(object sender, IMReceivedEventArgs e);
```

Now events for the client class.

```
public event EventHandler LoginOK;
public event EventHandler RegisterOK;
public event IMErrorEventHandler LoginFailed;
public event IMErrorEventHandler RegisterFailed;
public event EventHandler Disconnected;
public event IMAvailEventHandler UserAvailable;
public event IMReceivedEventHandler MessageReceived;
```

Then you have only to write helpers for raising events.

Register and login

We are connected. Now it's time to login or register.

Client

First we have to send the packet type. This will be the register or login. We will use the previously defined variable `reg` (in section *Preparing*).

```
bw.Write(reg ? IM_Register : IM_Login);
```

Then the username and password. We are using only the username and password for register (and in order to login too).

```
bw.Write(UserName);
bw.Write(Password);
bw.Flush();  // Flush buffer
```

The server will process this and then answer. Let's read the packet type. Then check if login is OK and raise events.

```
if (ans == IM_OK)  // Login/register OK
{
    if (reg)
        OnRegisterOK();  // Register is OK.
    OnLoginOK();         // Login is OK, too
}
else  // Login/register failed
{
    IMErrorEventArgs err = new IMErrorEventArgs((IMError)ans);
    if (reg)
        OnRegisterFailed(err);
    else
        OnLoginFailed(err);
}
```

Server

The server has to store usernames and passwords.

```
public class UserInfo
{
    public string UserName;
    public string Password;
    public bool LoggedIn;      // Is logged in and connected?
    public Client Connection;  // Connection info

    public UserInfo(string user, string pass)
    {
        this.UserName = user;
        this.Password = pass;
        this.LoggedIn = false;
    }
    public UserInfo(string user, string pass, Client conn)
    {
        this.UserName = user;
        this.Password = pass;
        this.LoggedIn = true;
        this.Connection = conn;
    }
}
```

This simple class will contain information about the user. If the user is connected then it will contain the `Client` class too.

In `Program` there will be a dictionary of users (key - username, value - information).

In `Client` class define a variable for storing the associated `UserInfo` (after login).

```
UserInfo userInfo;
```

Now it's time to handle user login. Read information from stream.

```
byte logMode = br.ReadByte();
string userName = br.ReadString();
string password = br.ReadString();
```

Let's check if the values are not too long and answer if they are incorrect.

```
if (userName.Length < 10)
{
    if (password.Length < 20)
    {
    }
    else
        bw.Write(IM_TooPassword);  // Too long password
}
else
    bw.Write(IM_TooUsername);  // Too long username
```

If they are correct, check which mode is selected. If we are registering we have to check whether username is free, and if we are logging in we must check if the account is existing and if the password is correct.

```
if (logMode == IM_Register)  // Register mode
{
    if (!prog.users.ContainsKey(userName))
    {
        // Username is free
    }
    else
        bw.Write(IM_Exists);  // Already exists
}
else if (logMode == IM_Login)  // Login mode
{
    // If account exists, get information to userInfo.
    if (prog.users.TryGetValue(userName, out userInfo))
    {
        if (password == userInfo.Password)
        {
            // Password is OK.
        }
        else
            bw.Write(IM_WrongPass);  // Wrong password
    }
    else
        bw.Write(IM_NoExists);  // Doesn't exists
}
```

If register is OK, create and add `UserInfo`. If login is OK, get `UserInfo` and associate current connection with this. At the end, tell client OK.

```
// Register OK
userInfo = new UserInfo(userName, password, this);
prog.users.Add(userName, userInfo);
bw.Write(IM_OK);
bw.Flush();
```

```
// Login OK
// Disconnect other persons logged in this account.
if (userInfo.LoggedIn)
    userInfo.Connection.CloseConn();

// Associate connection
userInfo.Connection = this;
bw.Write(IM_OK);
bw.Flush();
```

Packets receiving loop

When we are logged in, we have to listen to incoming packets in loop. Define the method `Receiver` in the server and client.

```
void Receiver()  // Receive all incoming packets.
{
    _logged = true;

    try
    {
        while (client.Connected)  // While we are connected.
        {
            byte type = br.ReadByte();  // Get incoming packet type.
        }
    }
    catch (IOException) { } // Thrown, when reading from closed connection.

    _logged = false;
}
```

At server `logged` will be replaced with `userInfo.LoggedIn`. Call this method after successful login or register.

Save user information

Information about users are stored in a collection. If server closes then data would be lost.

It can be simply done using serialization. It's not recommended if there are millions of users (then use database), but for a simple messenger, we can use serialization.

As yet, we have to save only usernames and passwords. We have to add some attributes in the `UserInfo` class.

```
[Serializable] public class UserInfo
[NonSerialized] public bool LoggedIn;
[NonSerialized] public Client Connection;
```

The `Serializable` attribute makes a class serializable... We don't want to serialize connection data, so it has a `NonSerialized` attribute. Now functions to save and load users to file. A `Dictionary` is not serializable, but we don't need the keys. Before saving, values from dictionary are converted to an array. While loading array with users it is converted to dictionary using LINQ.

```
string usersFileName = Environment.CurrentDirectory + "\\users.dat";
public void SaveUsers()  // Save users data
{
    try
    {
        BinaryFormatter bf = new BinaryFormatter();
        FileStream file = new FileStream(usersFileName, FileMode.Create, FileAccess.Write);
        bf.Serialize(file, users.Values.ToArray());  // Serialize UserInfo array
        file.Close();
    }
    catch (Exception e)
    {
        Console.WriteLine(e.ToString());
    }
}
public void LoadUsers()  // Load users data
{
    try
    {
        BinaryFormatter bf = new BinaryFormatter();
        FileStream file = new FileStream(usersFileName, FileMode.Open, FileAccess.Read);
        UserInfo[] infos = (UserInfo[])bf.Deserialize(file);      // Deserialize UserInfo array
```

```
        file.Close();
        users = infos.ToDictionary((u) => u.UserName, (u) => u);  // Convert UserInfo array to
Dictionary
    }
    catch { }
}
```

Now before the server starts, put this line in order to load users:

```
LoadUsers();
```

As yet, we are changing user info only, while registering. After adding a new user to collection save users.

```
prog.SaveUsers();
```

Check other users availability

Before we implement sending messages we have to know if recipient is online.

Server

First, check type (obtained in the loop).

```
if (type == IM_IsAvailable)
{
}
```

Second, who has to be checked?

```
string who = br.ReadString();
```

Then begin packet:

```
bw.Write(IM_IsAvailable);  // Available or not
bw.Write(who);             // Who?
```

Now we have to check if user exists and then if user is connected:

```
if (prog.users.TryGetValue(who, out info))
{
    if (info.LoggedIn)
        bw.Write(true);   // Available
    else
        bw.Write(false);  // Unavailable
}
else
    bw.Write(false);  // Unavailable
```

It's writing the last part of the packet, too. Now we only have to flush the buffer.

```
bw.Flush();
```

Client

Client has to send request and then asynchronously receive answer in loop.

```
public void IsAvailable(string user)
{
```

```
    bw.Write(IM_IsAvailable);
    bw.Write(user);
    bw.Flush();
}
```

Now receive answer in receiver loop.

```
if (type == IM_IsAvailable)
{
    string user = br.ReadString();
    bool isAvail = br.ReadBoolean();
}
```

And invoke event:

```
OnUserAvail(new IMAvailEventArgs(user, isAvail));
```

Culmination: Sending and receiving messages

Finally, we have reached the destination of instant messaging.

Client

First, method for sending message.

```
public void SendMessage(string to, string msg)
{
}
```

We have to send packet type, recipient name, and message.

```
bw.Write(IM_Send);
bw.Write(to);
bw.Write(msg);
bw.Flush();
```

Now receiving. Check packet type and get additional data.

```
else if (type == IM_Received)
{
    string from = br.ReadString();
    string msg = br.ReadString();
}
```

And raise event.

```
OnMessageReceived(new IMReceivedEventArgs(from, msg));
```

Server

Server has to receive the *send* packet and send the *receive* packet with message to the recipient.

```
else if (type == IM_Send)
{
    string to = br.ReadString();
    string msg = br.ReadString();
}
```

We have all the needed data. Now let's try to get the user:

```
UserInfo recipient;
if (prog.users.TryGetValue(to, out recipient))
{
}
```

If recipient exists we must check if he is online.

```
if (recipient.LoggedIn)
{
}
```

Using associated connection we can access the `BinaryWriter` of recipient and write the *receive* packet.

```
recipient.Connection.bw.Write(IM_Received);
recipient.Connection.bw.Write(userInfo.UserName);  // From
recipient.Connection.bw.Write(msg);                // Message
recipient.Connection.bw.Flush();
```

It's done! Simple instant messenger **protocol** is ready to use!

User interface

Server and protocol are ready, but we haven't got a user interface. You have to design your Instant Messenger user interface. Then simply connect to the server using your `IMClient` and use its functions and events. You can download the source code and see how it is working.

Conclusion

It's the end of my sixth article (for the time being...). I tried to explain step by step how to write a simple instant messenger in C#.NET and I hope I succeeded.

License

This article, along with any associated source code and files, is licensed under The Code Project Open License (CPOL)

About the Author

TeapotDev

Poland **My homepage**

Comments and Discussions

18 messages have been posted for this article Visit **https://www.codeproject.com/Articles/429144/Simple-Instant-Messenger-with-SSL-Encryption-in-Cs** to post and view comments on this article, or click **here** to get a print view with messages.

Permalink
Advertise
Privacy
Cookies
Terms of Use

Article Copyright 2012 by TeapotDev
Everything else Copyright © CodeProject, 1999-2020

Web04 2.8.20201101.1