

Programowanie Aplikacji Sieciowych

raport

Komunikator sieciowy klient- serwer

Opracował:

Konrad Taczak

Wydział Elektroniki i Telekomunikacji

specjalność: Sieci Komputerowe

1. Pełna treść zadania

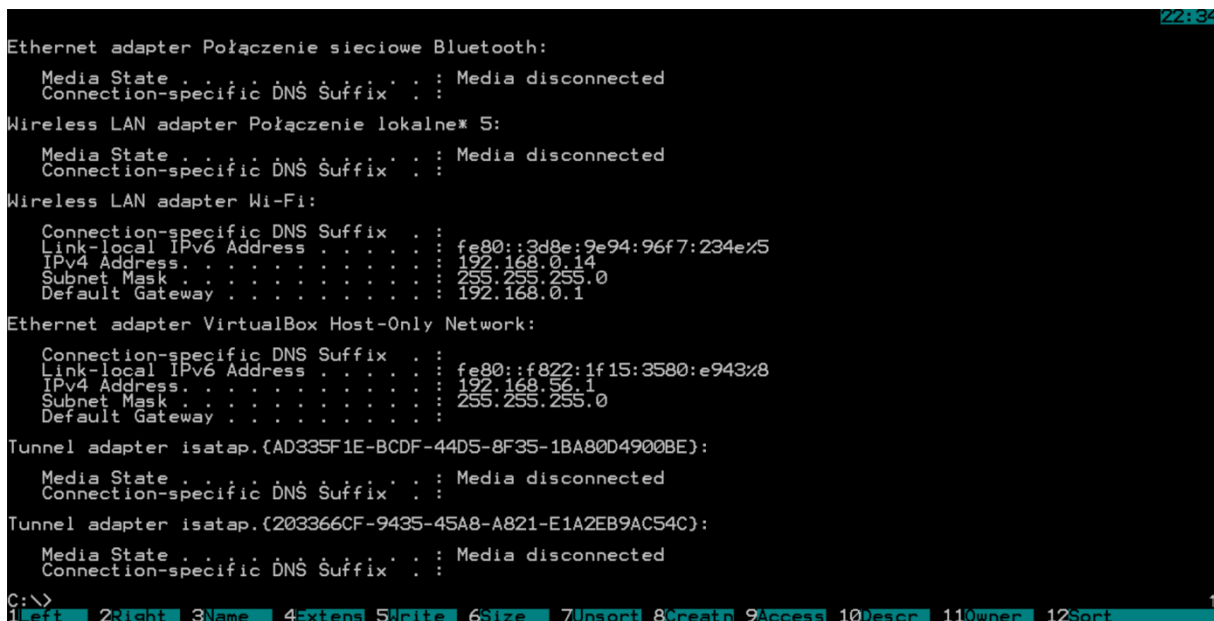
Napisz komunikator sieciowy. Program ma działać w formie klient-serwer. Graficzny interfejs użytkownika (GUI, *ang. Graphical User Interface*) nie jest wymagany – interakcja z użytkownikiem może odbywać się w wierszu poleceń (CLI – *ang. Command Line Interface*). Tabela 1 wyszczególnia podzadania i liczba punktów jakie można za nie uzyskać – dla danej grupy tylko jedno zadanie może być zrealizowane i ocenione. Grupy 1 i 3 zaznaczone na czerwono są obowiązkowe. Tabela 2 zawiera liczbę punktów jaką trzeba zbierać na każdą z ocen. Proponowanym protokołem transportowym jest TCP, jednak użycie UDP nie jest zabronione. Komendy wymyśla autor programu – te podane w zadaniach są tylko przykładami. Projekt można napisać w: Javie/C/C++/C#/Python'ie. Jeśli jednak chce się wybrać inny język – proszę się ze mną skontaktować. Dozwolone jest mieszanie języków (np. serwer w Javie, klient w Python'ie). Do programu należy załączyć raport ze sposobem użycia programu i opisem zaimplementowanych funkcjonalności.

2. Krótki opis realizacji projektu

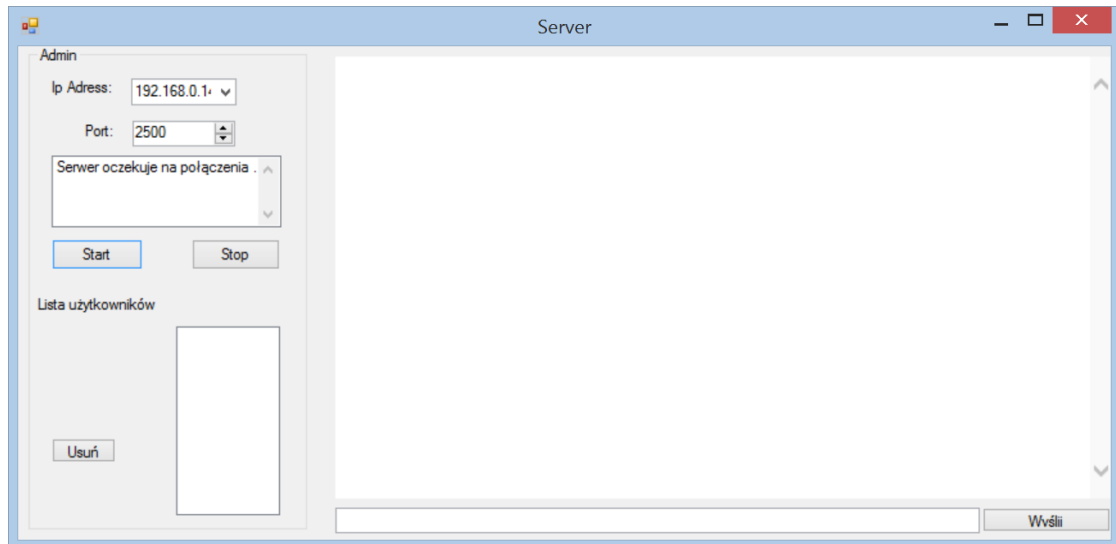
Swój projekt komunikatora sieciowego zaimplementowałem w języku C#. Za cel obrałem napisanie komunikatora z GUI Windows Form Application dostępnym dzięki .NET framework, obsługiwanego w programie Visual Studio. Ponadto na podstawie protokołu UDP napisałem serwer współbieżny, który dzięki podejściu wielowątkowemu jest w stanie obsłużyć wielu klientów na jednym porcie. Każde połączenie jest reprezentowane przez osobny wątek, tworzony podczas logowania klienta do serwera i utrzymywany, do momentu rozłączenia użytkownika z serwerem. Za podstawowe założenie przyjąłem napisanie serwera, który będzie działał w trybie rozgłoszeniowym. Każda wiadomość (o odpowiednim formacie) wysłana od klienta trafia do serwera, który rozsyła ją do innych użytkowników. Dzięki przyjętemu formacie wiadomości byłem w stanie rozróżniać logowanie do serwera, próbę wysłania wiadomości oraz ewentualne błędy, które mogły wystąpić przy próbie autoryzacji, nawiązania połączenia klient- serwer. W następnej kolejności, zacząłem rozwijać projekt o nowe funkcjonalności.

3. Sposób użycia programu

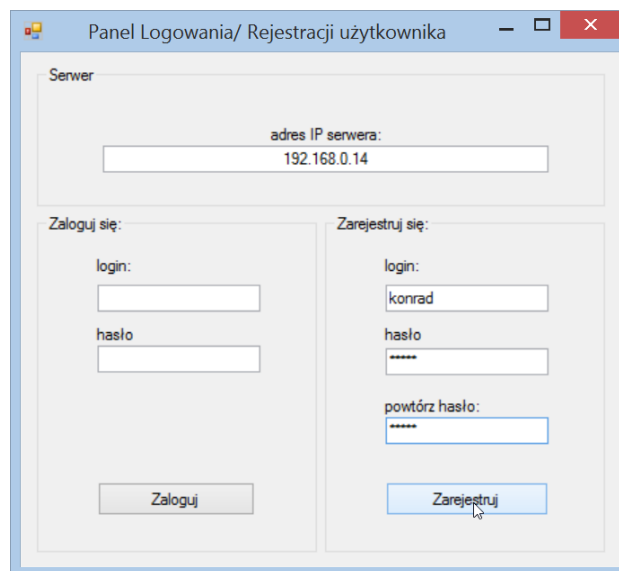
Swój program testowałem w sieci LAN. Trzy komputery podłączyłem pod jedną sieć lokalną. Jeden z nich działał jako serwer i klient, a dwa pozostałe tylko jako klient. W celu ustalenia adresu IP w sieci lokalnej skorzystałem z programu ipconfig. Krótko opiszę sposób połączenia klienta z serwerem.



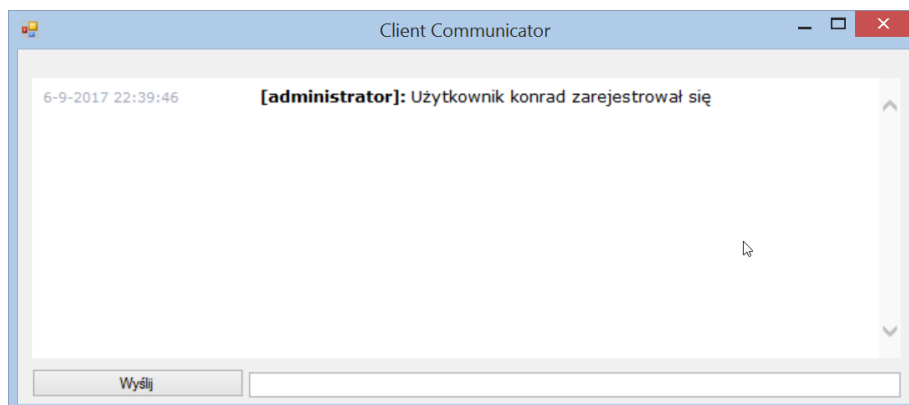
Adres IP sieci lokalnej (IPv4 Address) wpisujemy do aplikacji serwera. Po poprawnym uzupełnieniu adresu, serwer informuje nas o nasłuchiowaniu łącza, wyświetlając wiadomość "Serwer oczekuje na połączenie" w dzienniku aktywności.



Teraz podłączymy pierwszego klienta, w naszym przypadku z tego samego komputera. Adres IP użytkownika, powinien być tożsamy z adresem serwera. W tym celu uruchamiamy aplikację Client.exe. Uzupełniając pola rejestracji oraz pole adresu IP, uruchamiamy czat.

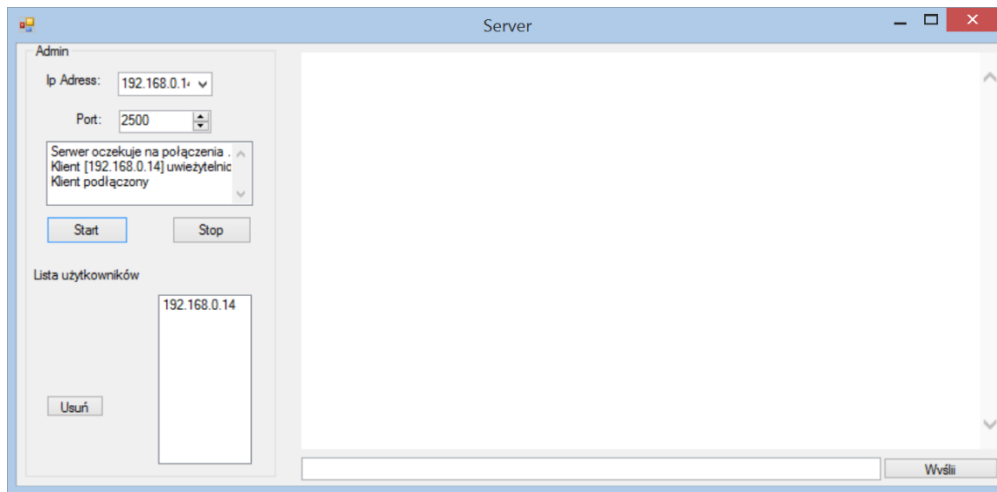


Po wciśnięciu "Zarejestruj" uzyskujemy dostęp do panelu czata, wraz z wiadomością powitalną:

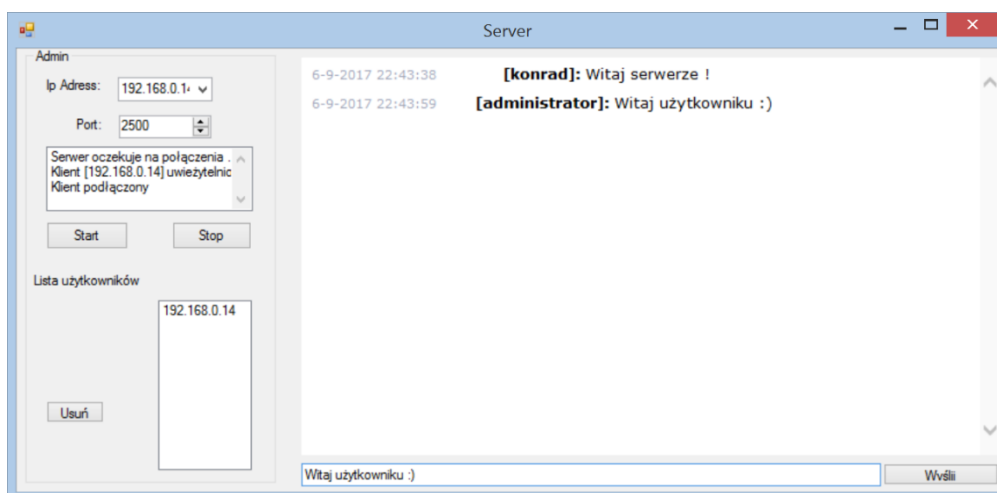
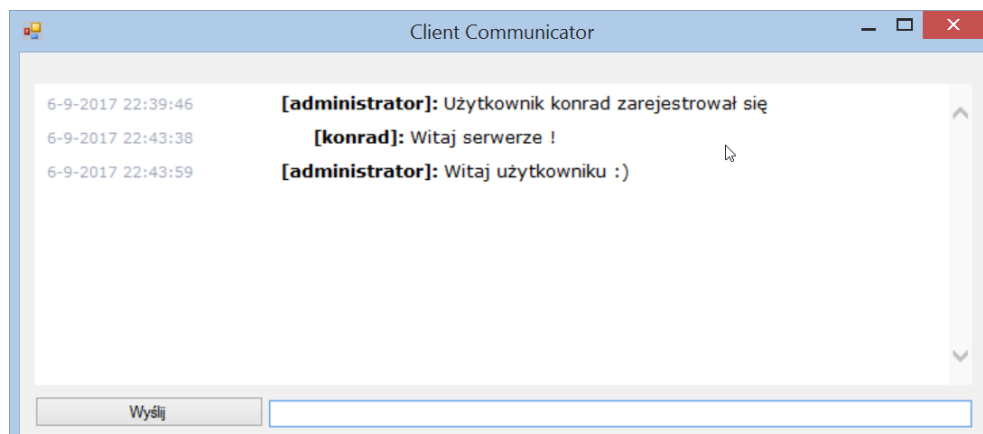


Programowanie Aplikacji Sieciowych- Komunikator- raport

Serwer dodaje połączenie do rejestru użytkowników oraz w polu aktywności serwera informuje o podłączeniu nowego użytkownika:

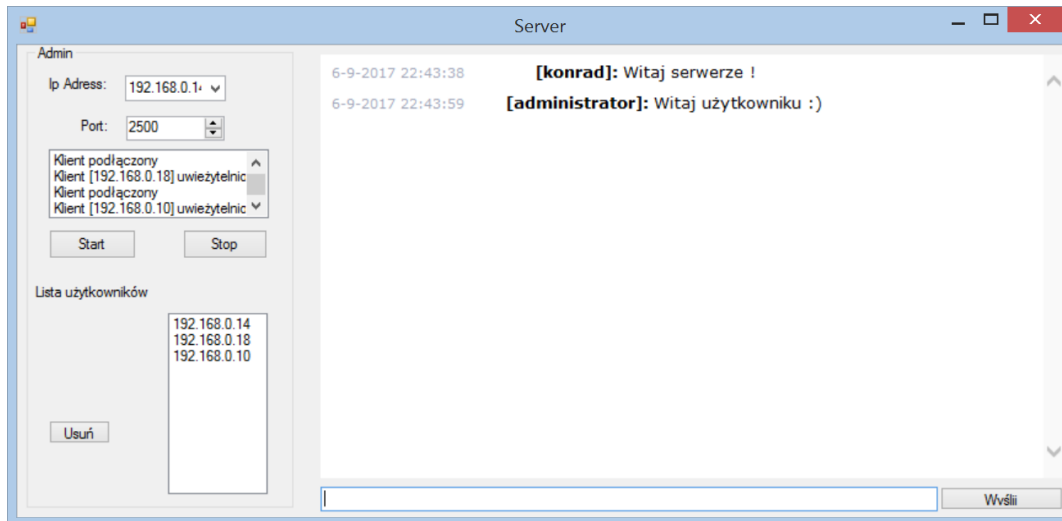


Teraz wyślemy pierwszą wiadomość do serwera, a z serwera wyślemy przykładową wiadomość do wszystkich zalogowanych klientów (w naszym przypadku do jednego)

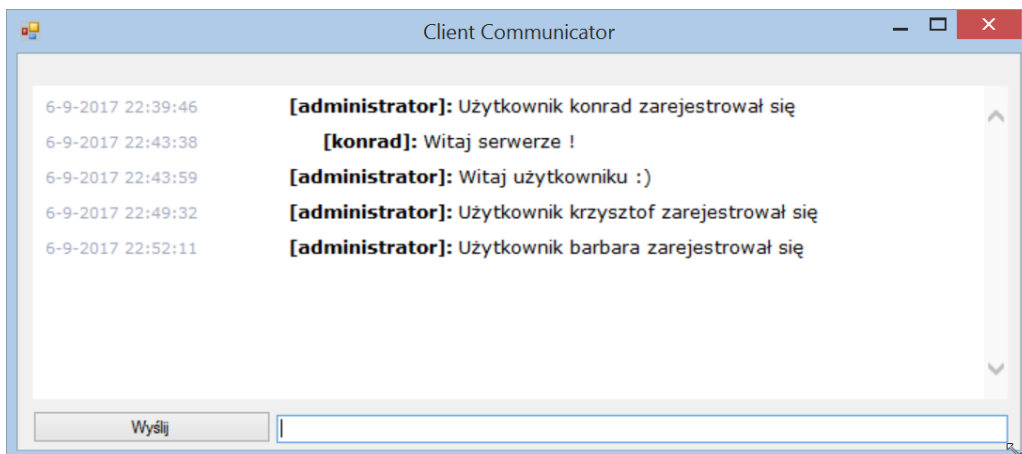


W celu uwiarygodnienia poprawnego działania serwera, podłączymy drugiego i trzeciego klienta w analogiczny sposób. Oto efekt widoczny z poziomu aplikacji serwera:

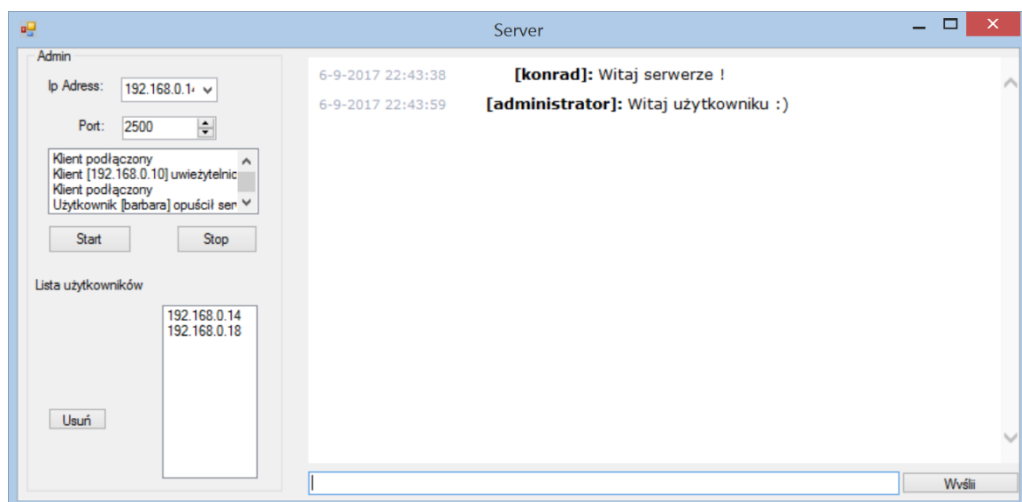
Programowanie Aplikacji Sieciowych- Komunikator- raport



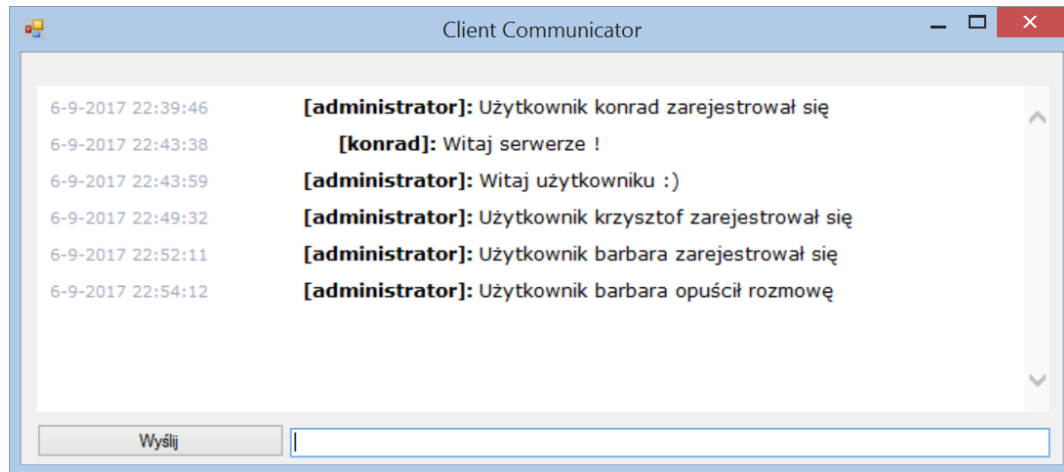
Wszyscy klienci podłączeni do serwera, również dostają powiadomienie o zalogowaniu się użytkownika:



Teraz dla przykładu rozłączymy użytkownika barbara:



Serwer widzi już tylko dwóch użytkowników. Dziennik aktywności informuje o wylogowaniu użytkownika barbara. Klienci również otrzymują taką informację:



W następnym punkcie będę opisywał każdą funkcjonalność oraz do każdej funkcjonalności pokażę przykładowe użycie programu.

4. Format/Protokół wiadomości

W ogólności format wiadomości składa się z trzech pól: *NADAWCA*, *KOMENDA* oraz *TREŚĆ*. Wprowadziłem następujące komendy: *HI* — przy uwierzytelnianiu, *SAY* — jeżeli użytkownik chce wysłać wiadomość, *BYE* — jeżeli klient się rozłączył, *ERROR* — jeżeli wystąpią błędy podczas próby połączenia, które nie są związane z działaniem samych aplikacji. Format wiadomości będzie następujący:

NADAWCA:KOMENDA:TREŚĆ.

Przykład wiadomości wymienianych między klientem a serwerem:

Klient: *UżytkownikX:HI:pusty*

Serwer: *administrator:HI:pusty*

Serwer: *administrator:SAY:Użytkownik UżytkownikX dołączył do rozmowy*

Klient: *UżytkownikX:SAY:Cześć wszystkim! Jak się macie?*

...

Klient: *UżytkownikX:SAY:Do widzenia!*

Klient: *UżytkownikX:BYE:pusty*

Serwer: *administrator:SAY:Użytkownik UżytkownikX opuścił rozmowę*

Pisząc w ogólności, miałem na myśli format bazowy. Podczas tworzenia niektórych funkcjonalności, jak przesyłanie do konkretnego użytkownika, czy też logowania lub rejestracji z użyciem hasła, rozszerzyłem powyższy format. Przykładowo, dla autoryzacji z użyciem hasła:

Klient: *UżytkownikX:HI:login:hasło*

Serwer: *administrator:HI:pusty*

Serwer: *administrator:SAY:Użytkownik UżytkownikX dołączył do rozmowy*

Klient: *UżytkownikX:SAY:Cześć wszystkim! Jak się macie?*

gdzie hasło przesyłane jest w postaci Hash'a.

Dla logowania, czy też rejestracji oraz wysłania wiadomości do konkretnego UżytkownikaY, wymiana informacji mogłaby wyglądać następująco:

Klient: *UżytkownikX:HI:login:hasło*

Serwer: *administrator:HI:pusty*

Serwer: *administrator:SAY:Użytkownik UżytkownikX dołączył do rozmowy*

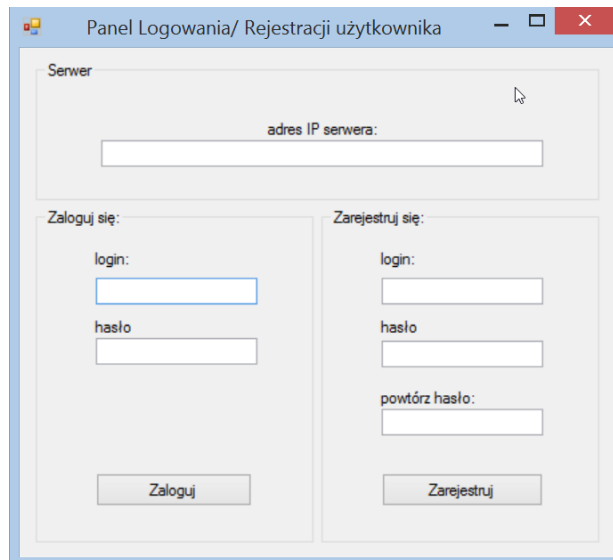
Klient: *UżytkownikX:SAY:#sendto UżytkownikY Cześć użytkownikuY*

5. Funkcjonalność/ sposób użycia programu

Rejestracja użytkowników z podaniem loginu- funkcjonalność:

Otwierając aplikację, mamy do wyboru opcje logowania oraz rejestracji. Serwer na czas swojego działania zapamiętuje w listach login, hasło w postaci hash'u oraz numer IP hosta. W celach rejestracji klient musi wpisać swój login, hasło oraz powtórzone hasło. Program Client składa się z dwóch form. Zawiera odpowiednio panel logowania/ rejestracji, nazwany LoginPanel oraz interfejs komunikatora zwany Form, który służy do wymiany danych z serwerem.

Proces rejestracji:



Podczas procesu rejestracji, na kliknięcie przycisku zarejestruj, uruchamiana jest walidacja formularza rejestracji:


```
private void buttonRegister_Click(object sender, EventArgs e)
{
    if (textBoxRegisterPassword.Text != textBoxRegisterPasswordRepeat.Text)
    {
        MessageBox.Show("hasło nie zgadza się z powtórzonym hasłem");
        return;
    }
    if (form1.validUser(textBoxRegisterLogin.Text, textBoxRegisterPassword.Text, textBoxIp.Text, "register")
    {
        this.Hide();
        form1.Show();
    }
}
```

Walidacja polega na sprawdzeniu, czy powtórzone hasło jest takie same jak hasło oraz czy na serwerze nie ma użytkownika o takim samym loginie. Jak widać w kodzie powyżej, w przypadku niezgodności haseł pojawi nam się komunikat "hasło nie zgadza się z powtórzonym hasłem". Jeżeli hasła są identyczne, przekazujemy login, hasło oraz adres IP hosta do panelu klienta (Form), który dzięki metodzie validUser wysyła do serwera wiadomość autoryzacyjną, w celu próby nawiązania połączenia.

```
public bool validUser(string userLogin, string userPassword, string serverIp, string operation)
{
    try
    {
        addressIPServer = serverIp;
        string hashPassword = getHash(userPassword);
        client = new TcpClient(addressIPServer, 2500);
        NetworkStream ns = client.GetStream();
        write = new BinaryWriter(ns);
        write.Write(userLogin + ":HI:" + operation + ":" + hashPassword);
        BinaryReader read = new BinaryReader(ns);
        if (read.ReadString() == "HI")
        {
            backgroundWorkerMainThread.RunWorkerAsync();
            isActive = true;
            login = userLogin;
            password = hashPassword;
            return true;
        }
        LoginPanel loginPanel = new LoginPanel();
        loginPanel.validMessage("Błędne hasło lub login. Spróbuj ponownie");
        return false;
    }
    catch (Exception ex)
    {
        MessageBox.Show("Nie można nawiązać połączenia " + ex.Message);
        return false;
    }
}
```

Programowanie Aplikacji Sieciowych- Komunikator- raport

Zgodnie z wcześniej opisanym protokołem, wysyłamy wiadomość:

login:HI:registration:password.

dla przykładu:

login: Konrad

hasło: admin

Wiadomość autoryzacyjna wygląda następująco:

Konrad:HI:registration:??*?H??f\u000f?\u0014\n?5?\fM??

Wiadomość za pomocą protokołu TCP, wysyłana jest na serwer, którego adres IP odczytywany jest z pola textBoxIP, a port, na którym przesyłana jest wiadomość to 2500. W przypadku, gdy serwer odpowie "HI", wiadomo, że użytkownik dokonał rejestracji i został zapisany do listy użytkowników na serwerze. Proces autoryzacji po stronie serwera, realizowany jest za pomocą metody `backgroundWorkerMainLoop_DoWork`. Serwer nasłuchuje port 2500 i w przypadku otrzymania ciągu bajtów, wykonuje określone czynności. Proces rejestracji obsługiwany jest przez następującą instrukcję warunkową:

```
if (cmd[2] == "register" && cmd[1] == "HI")
{
    if (namesClients.BinarySearch(cmd[0]) > -1)
    {
        write.Write("ERROR:Użytkownik o podanej nazwie już istnieje");
    }
    else
    {
        write.Write("HI");
        BackgroundWorker clientThread = new BackgroundWorker();
        clientThread.WorkerSupportsCancellation = true;
        clientThread.DoWork += new DoWorkEventHandler(clientThread_DoWork);
        namesClients.Add(cmd[0]);
        clientsList.Add(clientThread);
        namesLoggedClients.Add(cmd[0]);
        passwordClients.Add(cmd[3]);
        clientThread.RunWorkerAsync();
        SendUdpMessage("administrator:SAY:Użytkownik " + cmd[0] + " zarejestrował się");
        SetText(listBoxServer, "Klient podłączony");
    }
}
```

`cmd`, jest to tablica, Split odebranej wiadomości, przekonwertowanej na Stringa, zawierająca odpowiednio: Login, komendę, operację, hasło (Konrad, HI, register, `??*?H??f\u000f?\u0014\n?5?\fM??`). W przypadku, metoda wyszukiwania binarnego znajdzie indeks w tablicy `namesClients`, wysyłana jest informacja ERROR, którą w moim protokole oznacza błąd. W przypadku wiadomości ERROR, aplikacja po stronie klienta wypisze błąd "Użytkownik o podanej nazwie już istnieje". Jeżeli `BinarySearch` nie znalazł użytkownika o walidowanym loginie, serwer otwiera połączenie z klientem, a właściwie, przesyła zgodnie z protokołem wiadomość HI, która autoryzuje użytkownika, wpisuje go na listę użytkowników, zapisuje jego hasło oraz adres IP. Proces autoryzacji klient- serwer odbywa się za pomocą protokołu TCP. Sama komunikacja, pod względem przesyłania wiadomości SAY, odbywa się za pomocą protokołu UDP. W związku z

powyższym, po autoryzacji, serwer wysyła wiadomość rozgłoszeniową, do wszystkich zalogowanych klientów z informacją "użytkownik Konrad zarejestrował się". Dzięki temu użytkownicy, również są informowani o bieżących logowaniach oraz rejestracjach. Warto nadmienić, że podczas autoryzacji, każdemu połączeniu przypisywany jest osobny wątek, w celu identyfikacji użytkowników i pominięcia problemu jedno połączenie- jeden port.

Rejestracja użytkowników z podaniem loginu- sposób użycia:

Sposób użycia został przedstawiony w punkcie 3.

Logowanie z użyciem hasła- funkcjonalność:

Uruchamiając program, wpisujemy w odpowiednie pola login oraz hasło.

```
private void buttonLogin_Click_1(object sender, EventArgs e)
{
    try
    {
        if (textBoxLogin.Text != String.Empty && textBoxIp.Text != String.Empty && textBoxPassword.Text != String.Empty)
        {
            if (form1.validUser(textBoxLogin.Text, textBoxPassword.Text, textBoxIp.Text, "login"))
            {
                this.Hide();
                form1.Show();
            }
        }
        else
        {
            MessageBox.Show("Uzupełnij pola login oraz hasło");
        }
    }
    catch (Exception ex)
    {
        MessageBox.Show("nie można się zalogować " + ex.Message);
    }
}
```

Podobnie jak w procesie rejestracji, użytkownik podaje login oraz hasło. Formularz waliduje wyżej wymienione pola. W przypadku braku, któregośkolwiek z nich wyświetla komunikat: "Uzupełnij pola login oraz hasło". W przypadku, gdy formularz logowania jest uzupełniony, po kliknięciu przycisku zaloguj uruchamia się wcześniej przedstawiona metoda walidacyjna, zwracająca wartość logiczną, zależną od odpowiedzi serwera. Hasło, zarówno podczas logowania jak i rejestracji jest mieszane funkcją getHash(). Zapewnia to bezpieczeństwo, przechowywanych haseł po stronie serwera, a także chroni przed podsłuchem. Dodatkowym zabezpieczeniem jest szyfrowanie wiadomości, o którym napiszę później.

```
private string getHash(string password)
{
    var sha1 = new SHA1CryptoServiceProvider();
    var sha1data = sha1.ComputeHash(Encoding.ASCII.GetBytes(password));
    return new ASCIIEncoding().GetString(sha1data);
}
```

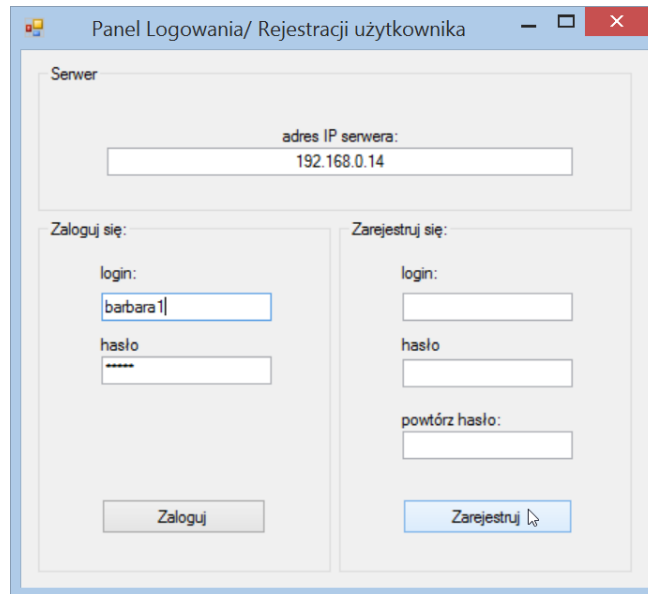
Powyższa metoda działa w oparciu o funkcję mieszającą sha-1, którą możemy odnaleźć w bibliotekach .NET. Argumentem metody jest hasło, które zamieniane jest na tablicę bajtów, która następnie jest poddawana funkcji mieszającej, zgodnie ze standardem sha-1. Wartością zwracaną jest przekonwertowana wartość funkcji mieszającej na ciąg znaków. Wracając do procesu logowania,

metoda `validUser`, przesyła zapytanie do serwera. Serwer tak samo jak przy rejestracji, w zależności od przebiegu autoryzacji, zwraca wiadomość "HI:pusty" lub "ERROR:użytkownik o podanej nazwie już istnieje". Mówiąc bardziej szczegółowo, serwer sprawdza listę zarejestrowanych użytkowników oraz im przypisane hasła. Jeżeli login i hasło są zapisane na serwerze, tworzy on wątek dla użytkownika, dodaje tekst do pola informującego o aktywności serwera oraz przesyła wiadomość do wszystkich użytkowników, informując ich o zalogowaniu danego klienta.

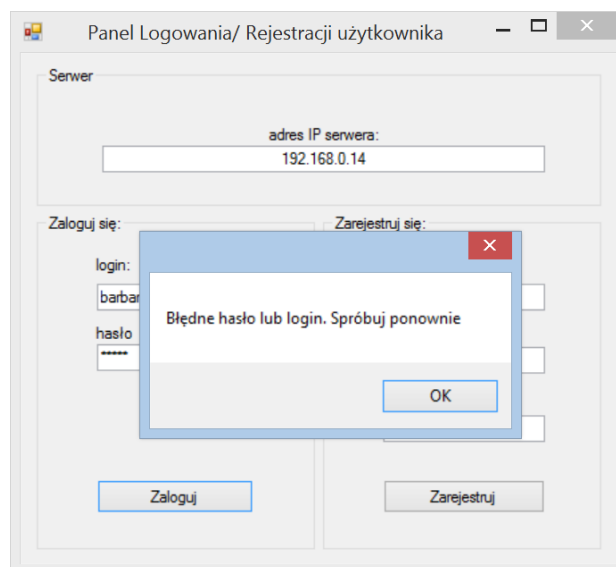
```
if (cmd[2] == "login" && cmd[1] == "HI")
{
    if(namesClients.Count <= 0)
    {
        write.Write("ERROR:Użytkownik o podanej nazwie już istnieje");
        return;
    }
    for (int i = 0; i < namesClients.Count; ++i)
    {
        if (cmd[0] == namesClients[i].ToString() && cmd[3] == passwordClients[i].ToString())
        {
            write.Write("HI");
            BackgroundWorker clientThread = new BackgroundWorker();
            clientThread.WorkerSupportsCancellation = true;
            clientThread.DoWork += new DoWorkEventHandler(clientThread_DoWork);
            clientsList.Add(clientThread);
            namesLoggedClients.Add(cmd[0]);
            clientThread.RunWorkerAsync();
            SendUdpMessage("administrator:SAY:Użytkownik " + cmd[0] + " zalogował się");
            SetText(listBoxServer, "Klient podłączony");
        }
    }
    write.Write("ERROR:Użytkownik o podanej nazwie już istnieje");
}
```

Logowanie z użyciem hasła- użycie programu:

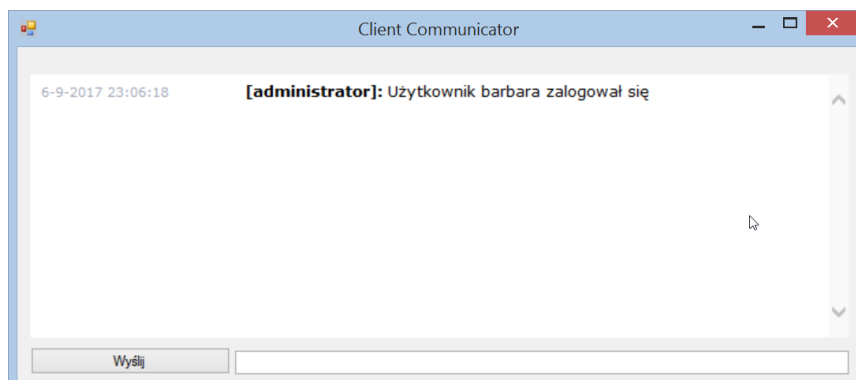
Kontynuując punkt 3, po wylogowaniu użytkownika barbara, spróbujemy ponownie zalogować się na to konto. W tym celu na komputerze użytkownika barbara uzupełniamy formularz logowania



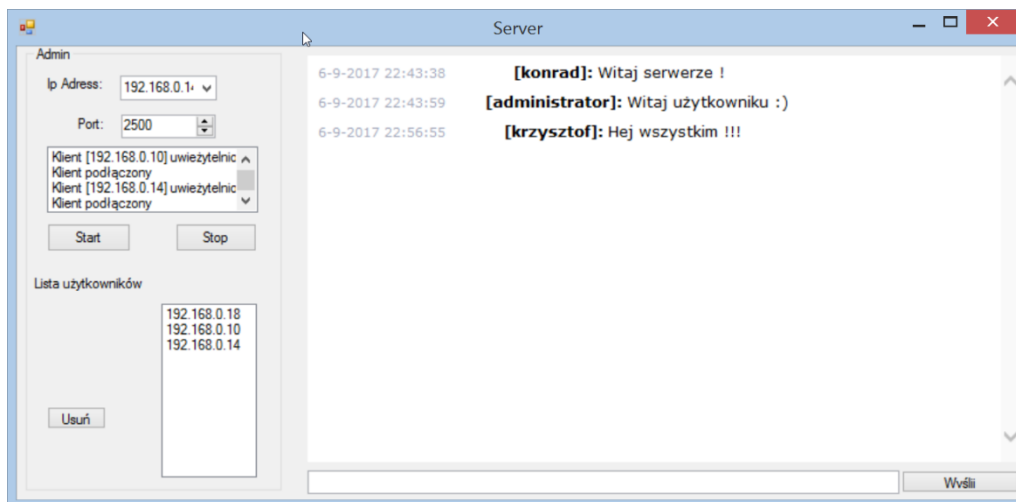
Dla testów poprawnej funkcjonalności programu, celowo wpisałem zły login, dopisując "jedynek".
Efekt pomyłki:



Jak widać serwer nie zarejestrował użytkownika "barbara1". Teraz wpisujemy poprawny login, którym wcześniej rejestrowaliśmy się do serwera("barbara")



Stan serwera:



Jak widać proces logowania działa poprawnie. Serwer jak i aplikacja klienta odpowiednio reaguje na zmiany na łączu klient- serwer.

Komunikacja rozgłoszeniowa do wszystkich klientów zarejestrowanych w serwerze- funkcjonalność:

Swój projekt zacząłem właśnie od tej funkcjonalności. Założeniem początkowym, było napisanie serwera, który obsługuje komunikację rozgłoszeniową. Funkcjonalność tą realizuje metoda SendUdpMessage.

```
private void SendUdpMessage(string message)
{
    string encryptedMessage = Cipher.EncryptStringAES(message, "message");
    foreach (string user in listBoxUsers.Items)
        using (UdpClient klientUDP = new UdpClient(user, 2500))
        {
            byte[] bufor = Encoding.UTF8.GetBytes(encryptedMessage);
            klientUDP.Send(bufor, bufor.Length);
        }
}
```

Metoda, jako argument przyjmuje typ string, w którym zapisana jest wiadomość, w formie wcześniej ustalonego protokołu. W pierwszej kolejności wiadomość jest szyfrowana, następnie pętla foreach przegląda listę zalogowanych użytkowników. Tworząc instancję UdpClient, zamieniamy zaszyfrowaną wiadomość na tablicę bajtów, a następnie przesyłamy ją do wszystkich użytkowników z listy, za pomocą protokołu bezpołączeniowego UDP.

Po stronie klienta wiadomość odbierana wygląda następująco:


```
private void backgroundWorkerMainThread_DoWork(object sender, DoWorkEventArgs e)
{
    UdpClient client = new UdpClient(2500);
    IPEndPoint addressIP = new IPEndPoint(IPAddress.Parse(addressIPServer), 0);
    string message = "";

    while (!backgroundWorkerMainThread.CancellationPending){
        Byte[] bufor = client.Receive(ref addressIP);
        string data = Encoding.UTF8.GetString(bufor);
        data = Cipher.DecryptStringAES(data, "message");
        string[] cmd = data.Split(new char[] { ':' });

        if (cmd[1] == "BYE")
        {
            AddText("system", "klient odłączony");
            client.Close();
            return;
        }
        if (cmd.Length > 2)
        {
            message = cmd[2];
            for (int i = 3; i < cmd.Length; i++)
                message += ":" + cmd[i];
            AddText(cmd[0], message);
        }
    }
}
```

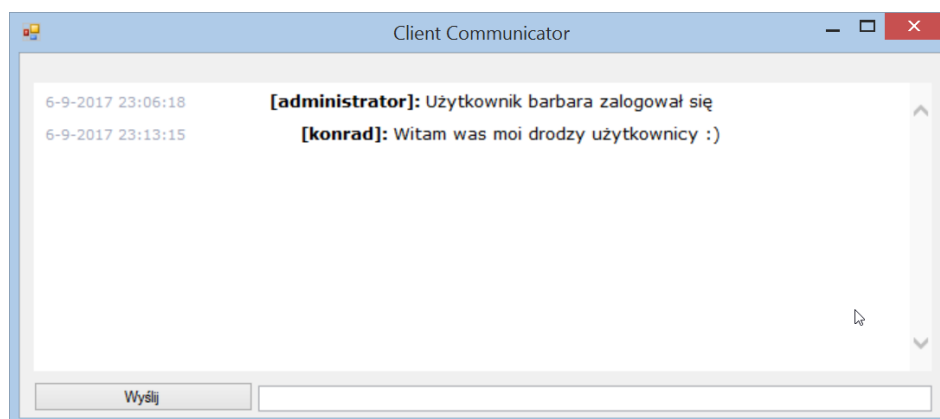
Główny wątek backgroundWorkerMainThread do momentu rozłączenia nasłuchuje stan łącza. Dopóki flaga CancellationPending jest ustawiona na false, dopóty istnieje połączenie i klient jest w stanie odbierać informacje. Odbiór wiadomości rozpoczyna się od odebrania tablicy bajtów, pobrania ich w postaci String, a następnie odszyfrowania informacji. Odszyfrowana informacja jest dzielona względem znaku ":" i zapisywana do tablicy typu string. Takie przygotowanie danych usprawnia wszystkie działania, na odebranej i odszyfrowanej wiadomości. Metoda backgroundWorkerMainThread_DoWork, sprawdza komendę, czy też rodzaj wiadomości (SAY, BYE), a następnie dokonuje odpowiednie operacje. W przypadku wiadomości BYE, aplikacja klienta dodaje do pola tekstowego czatu, wiadomość "Klient odłączony", a następnie zamyka połączenie. Otrzymując wiadomość, której po rozdzieleniu względem znaku ":", rozmiar tablicy jest większy od 2, wiemy, że mamy do czynienia z wiadomością "SAY". Po odpowiednich działaniach wyświetlamy ją na ekran aplikacji klienta.

Komunikacja ogłoszeniowa do wszystkich klientów zarejestrowanych w serwerze- użycie programu:

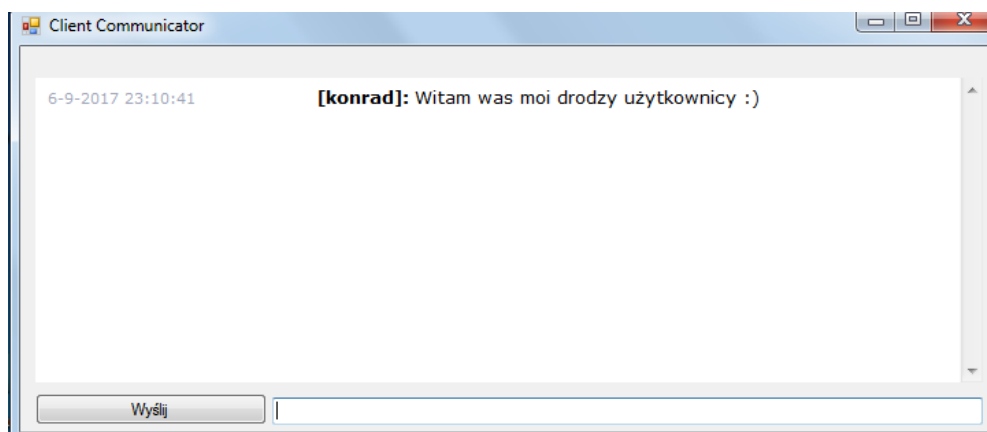
W tym podpunkcie, bazując na wcześniejszych analizach wyślemy wiadomość od użytkownika konrad i sprawdzimy, czy wszyscy użytkownicy odebrali tę samą wiadomość:

Programowanie Aplikacji Sieciowych- Komunikator- raport

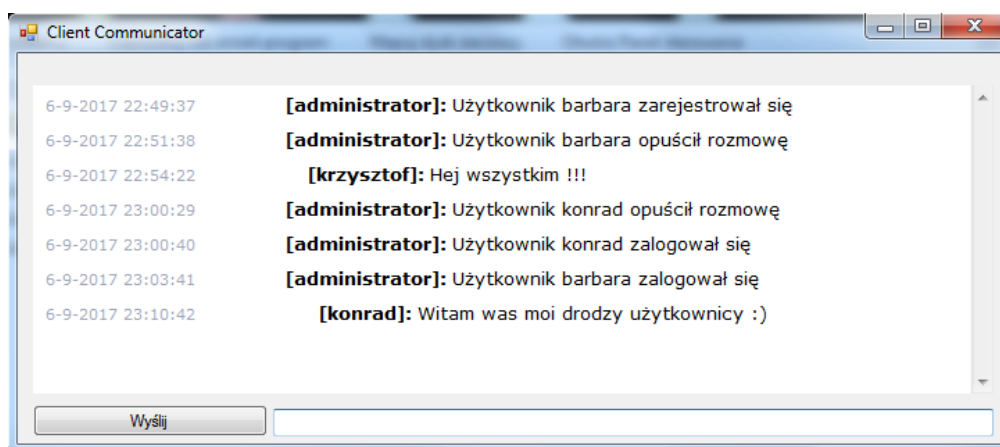
użytkownik barbara:



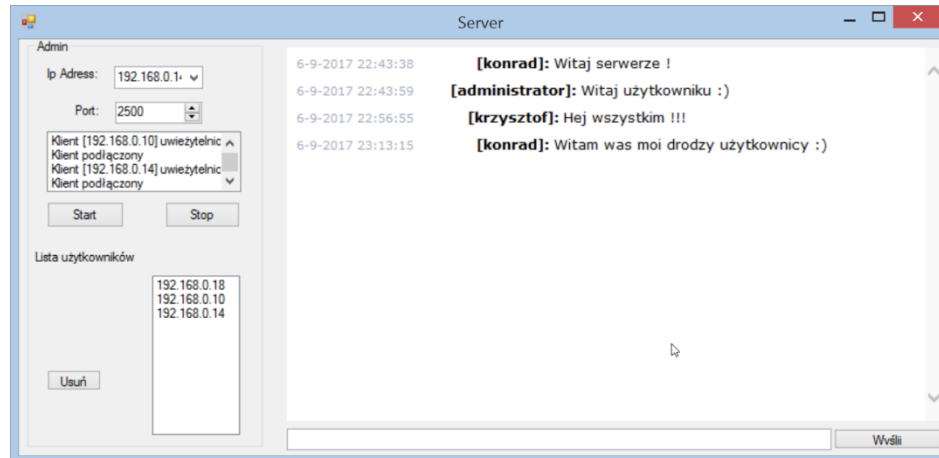
użytkownik konrad:



użytkownik krzysztof:



serwer:



Komunikacja tylko do wybranego użytkownika- funkcjonalność:

Główny wątek serwera `clientThread_DoWork`, po odczytaniu wiadomości o formacie "użytkownik1:SAY:#sendto użytkownik2 cześć", wie że dostał wiadomość typu SAY od użytkownika 1. Po odczytaniu komendy `#sendto`, uruchamia procedurę związaną z wysłaniem wiadomości do jednego użytkownika. W naszym przypadku jest to użytkownik 2. Treść wiadomości, jak można się domyślić to "Cześć". Procedurę tę realizuje poniższa instrukcja warunkowa:

```
string message = null;
if (cmd.Length > 2)
{
    message = cmd[2];
    for (int i = 3; i < cmd.Length; i++)
        message += ":" + cmd[i];
}
switch (cmd[1])
{
    case "SAY":
        if (getCommand(message) == "#showall")
        {
            AddText(cmd[0], "Lista użytkowników: " + parseListToString(namesLoggedClients));
            SendUdpMessage(cmd[0] + ":" + cmd[1] + ":" + "Lista użytkowników: " +
                parseListToString(namesLoggedClients), getUserByName(cmd[0]));
            break;
        }
        if (getCommand(message) == "#sendto")
        {
            sendTo(getNameFromMessage(message), "{Priv}" + getTextFromMessage(message), cmd[0]);
            AddText(cmd[0], "{Priv}" + (message));
            break;
        }
        AddText(cmd[0], message);
        SendUdpMessage(cmd[0] + ":" + cmd[1] + ":" + message);
        break;
}
```

Programowanie Aplikacji Sieciowych- Komunikator- raport

Inicjujemy zmienną message, która jest treścią wiadomości przesłanej od użytkownika 1 do użytkownika 2. W naszym przykładzie będzie to "#sendto użytkownik2 cześć". Za pomocą metody getCommand, która w argumencie przyjmuje treść wiadomości, jesteśmy w stanie wyodrębnić samą komendę, jaką jest "#sendto".

```
private String getCommand(string word)
{
    return word.IndexOf(" ") > -1
        ? word.Substring(0, word.IndexOf(" "))
        : word;
}
```

Jeżeli serwer otrzyma tego typu wiadomość, uruchamia część kodu, w którym obsługiwane jest przesyłanie wiadomości do konkretnego użytkownika. W celu rozróżnienia wiadomości rozgłoszeniowej, od wyżej opisanej, stworzona została metoda sendTo, która w odróżnieniu od wiadomości do wszystkich użytkowników, w swoich argumentach przyjmuje adres odbiorcy (name), treść wiadomości(message) oraz nadawcę wiadomości(who). Wszystkie argumenty są typu string.

```
private void sendTo(string name, string message, string who)
{
    for (int i = 0; i < listBoxUsers.Items.Count; ++i)
    {
        if (namesClients[i].ToString() == name)
        {
            AddText(name, message);
            SendUdpMessage(who + ":SAY:" + message, listBoxUsers.Items[i].ToString());
            break;
        }
    }
}
```

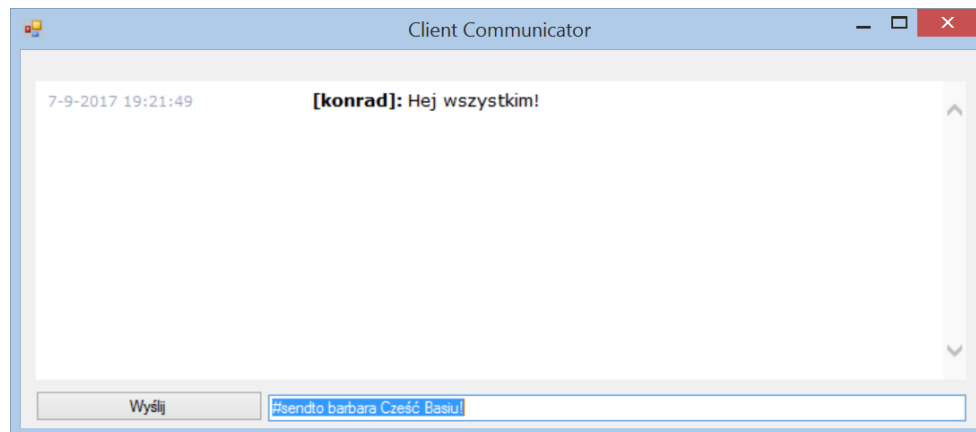
Metoda sendTo, wyszukuje spośród wszystkich zalogowanych użytkowników, użytkownika, do którego ma trafić wiadomość. W przypadku znalezienia takiego klienta (w naszym przypadku użytkownik 2), uruchamiana jest metoda sendUdpMessage, która podobnie jak w przypadku wiadomości rozgłoszeniowej, wysyła wiadomość, przy użyciu protokołu UDP, z tym wyjątkiem, że do jednego, konkretnego użytkownika, sprecyzowanego w drugim argumencie metody.

```
private void SendUdpMessage(string message, string user)
{
    string encryptedMessage = Cipher.EncryptStringAES(message, "message");
    using (UdpClient klientUDP = new UdpClient(user, 2500))
    {
        byte[] bufor = Encoding.UTF8.GetBytes(encryptedMessage);
        klientUDP.Send(bufor, bufor.Length);
    }
}
```

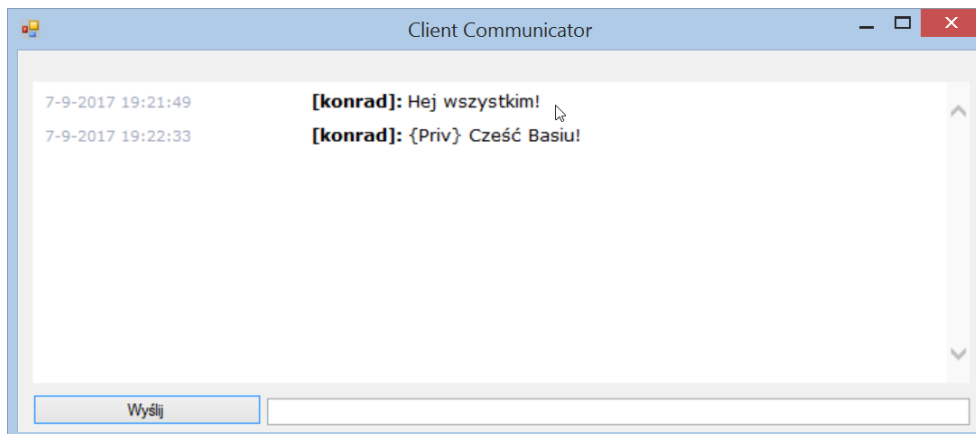
Komunikacja tylko do wybranego użytkownika- użycie programu:

Dla przykładu użytkownik konrad napisze wiadomość tylko do użytkownika barbara:

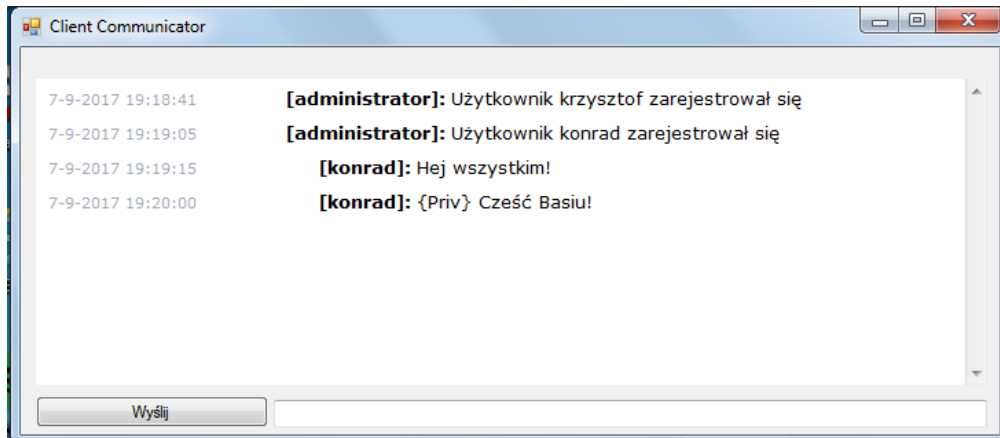
użytkownik konrad:



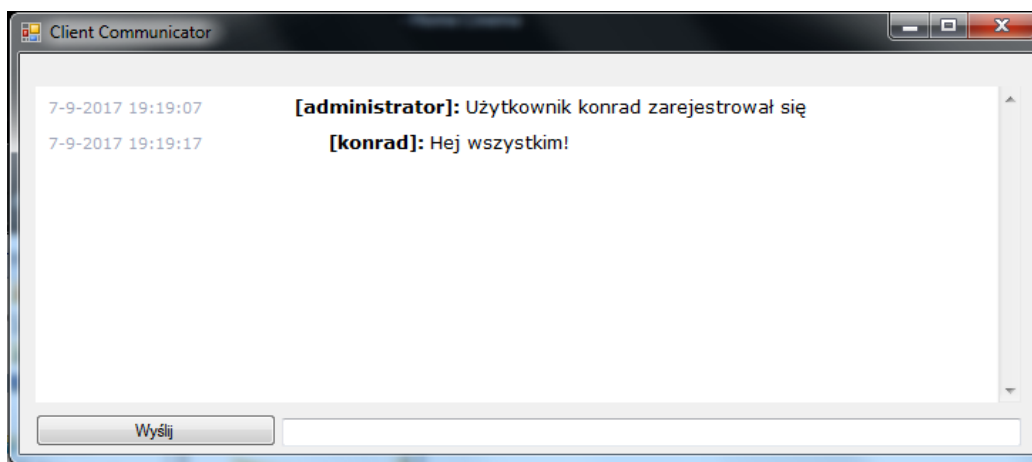
użytkownik konrad po wysłaniu wiadomości:



użytkownik barbara po odebraniu wiadomości prywatnej od użytkownika konrad:



użytkownik krzysztof (nie dostał wiadomości, wymienionej między użytkownikiem konrad a barbara):



Wyświetlanie w kliencie zalogowanych użytkowników- funkcjonalność:

```
switch (cmd[1])
{
    case "SAY":
        if (getCommand(message)=="#showall")
        {
            AddText(cmd[0], "Lista użytkowników: " + parseListToString(namesLoggedClients));
            SendUdpMessage(cmd[0] + ":" + cmd[1] + ":" + "Lista użytkowników: " +
                parseListToString(namesLoggedClients), getUserByName(cmd[0]));
            break;
        }
}
```

Tak jak w przypadku komunikacji z wybranym użytkownikiem, serwer, po odczytaniu z wiadomości słowa "SAY", wcześniej zapisanego do tablicy cmd, sprawdza co następuje po tym słowie. Ponownie używamy metody getCommand, która zwraca nam gotowy łańcuch tekstowy, który jest pierwszym słowem po "SAY".

```
private String getCommand(string word)
{
    return word.IndexOf(" ") > -1
        ? word.Substring(0, word.IndexOf(" "))
        : word;
}
```

W tym przypadku będzie to komenda #showall, która ma na celu poinformować serwer, że chcemy wysłać wiadomość do konkretnego użytkownika. W tym wypadku serwer dodaje do swojego webBrowser tekst w postaci:

27.04.2017 12:20 [PanX]: Lista użytkowników: użytkownik 1,użytkownik 2, użytkownik 3.

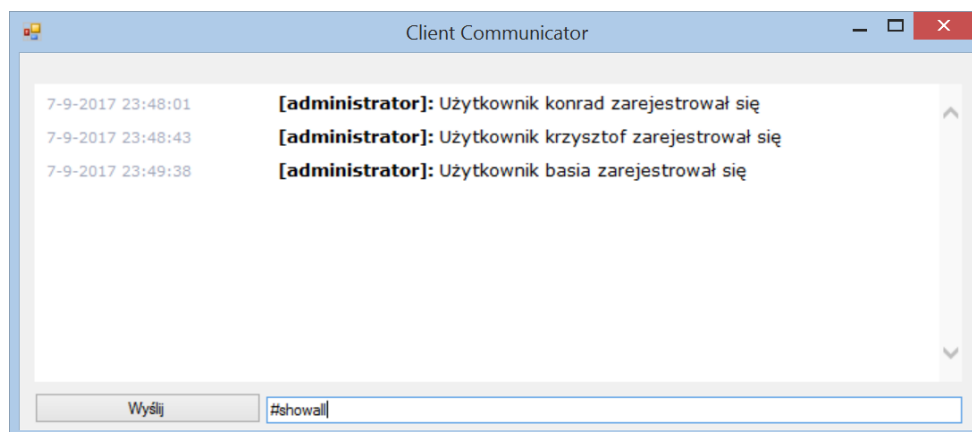
W liście namesLoggedClients, są przechowywane loginy wszystkich aktualnie zalogowanych użytkowników. Metoda parseListToString, za pomocą metody Join, zamienia listę na ciąg znaków, występujących po przecinku.

```
private String parseListToString(ArrayList listToParse)
{
    return String.Join(", ", listToParse.ToArray());
}
```

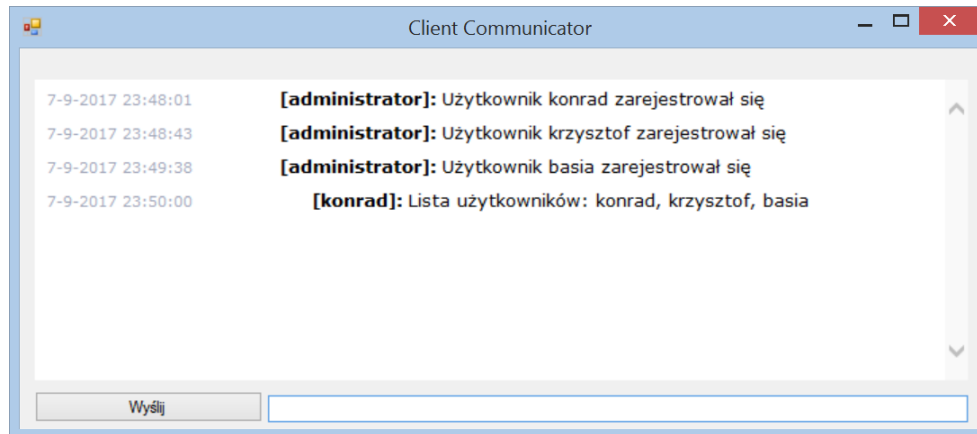
Dwuargumentowa metoda sendUdpMessage wysyła wiadomość do konkretnego użytkownika, w naszym wypadku, do osoby, która chce uzyskać listę wszystkich zalogowanych użytkowników.

```
private void SendUdpMessage(string message, string user)
{
    string encryptedMessage = Cipher.EncryptStringAES(message, "message");
    using (UdpClient klientUDP = new UdpClient(user, 2500))
    {
        byte[] bufor = Encoding.UTF8.GetBytes(encryptedMessage);
        klientUDP.Send(bufor, bufor.Length);
    }
}
```

Wyświetlanie w kliencie zalogowanych użytkowników- sposób użycia programu:



w rezultacie komendy #showall otrzymujemy, jak poniżej:



Wyświetlanie znaczników czasowych przy wiadomościach- funkcjonalność:

Tę funkcjonalność zrealizowałem, dzięki bibliotece System, używając metody DateTime(). Znaczniki czasowe wykorzystuję, tylko podczas wyświetlania wiadomości, dlatego też dodałem je do metody AddText, która do pola webBrowserChat dodaje tekst, sformatowany przez HTML. Czas wyświetlany przy wiadomościach, to czas, który odczytywany jest z systemu operacyjnego. Format czasu przyjąłem następujący: "d/M/yyyy HH:mm:ss"

```
private void AddText(string who, string message)
{
    SetTextHTML("<table><tr><td style='font-size:11px;' width=\"30%\"><span style='color:#a1a7ba'>" +
        String.Format("{0:d/M/yyyy HH:mm:ss}", DateTime.Now) +
        "</span></td><td width=\"10%\"><b style='text-align:left'>" +
        "[" + who + "]:</b></td>");

    SetTextHTML("<td colspan=2>" + message + "</td></tr></table>");
    SetScroll();
}
```

Wyświetlanie znaczników czasowych przy wiadomościach- użycie w programie:

Zarówno po stronie klienta jak i serwera widać przy wiadomościach znaczniki czasowe (patrz powyżej)

Szyfrowanie wiadomości- funkcjonalność:

W celach zapewnienia bezpiecznej transmisji danych między serwerem, a klientem i na odwrót, został użyty szyfr AES. AES, to symetryczny szyfr blokowy przyjęty przez NIST jako standard FIPS-197 w wyniku konkursu ogłoszonego w roku 1997. W 2001 roku został przyjęty jako standard. AES jest oparty na algorytmie Rijndael'a, którego autorami są Belgijscy kryptografowie, Joan Daemen i Vincent Rijmen. Zaprezentowali oni swoją propozycję szyfru Instytucji NIST w ramach ogłoszonego

konkursu. Rijndael jest rodziną szyfrów o różnych długościach klucza oraz różnych wielkościach bloków.

Za implementację algorytmu szyfrującego, odpowiedzialna jest klasa Cipher, znajdująca się po stronie klienta oraz serwera. Klasa przechowuje ziarno (salt), które w celach bezpieczeństwa powinno być generowane i zmieniane. W mojej aplikacji jest stosowane stałe ziarno - "E1F53135E559C253". Klasa zawiera metodę szyfrującą- EncryptStringAES oraz metodę odszyfrowującą DescryptStringAES. Metoda deszyfrująca, działa tak samo, jak metoda szyfrująca, z tym wyjątkiem, że kolejność jest odwrócona. W związku z tym opiszę metodę szyfrującą. Metoda przyjmuje dwa argumenty: plainText- treść wiadomości, do zaszyfrowania oraz sharedSecret- hasło używane do generacji klucza (w programie "message"). Generowanie klucza jest oparte na generatorze liczb pseudolosowych Rfc2898, który możemy odnaleźć w dokumentacji grupy .NET. W celu zakodowania wiadomości, w pierwszej kolejności tworzony jest obiekt RijndaelManaged, następnie Encryptor, po którym otwierany jest strumień, do którego wprowadzamy dane. Po zaszyfrowaniu danych, tworzony jest string o podstawie 64, na podstawie tablicy bajtów.

```
class Cipher
{
    private static byte[] _salt = Encoding.ASCII.GetBytes("E1F53135E559C253");
    public static string EncryptStringAES(string plainText, string sharedSecret)
    {
        if (string.IsNullOrEmpty(plainText))
            throw new ArgumentNullException("plainText");
        if (string.IsNullOrEmpty(sharedSecret))
            throw new ArgumentNullException("sharedSecret");

        string outStr = null;
        RijndaelManaged aesAlg = null;
```

```
try
{
    Rfc2898DeriveBytes key = new Rfc2898DeriveBytes(sharedSecret, _salt);
    aesAlg = new RijndaelManaged();
    aesAlg.Key = key.GetBytes(aesAlg.KeySize / 8);
    ICryptoTransform encryptor = aesAlg.CreateEncryptor(aesAlg.Key, aesAlg.IV);
    using (MemoryStream msEncrypt = new MemoryStream())
    {
        msEncrypt.Write(BitConverter.GetBytes(aesAlg.IV.Length), 0, sizeof(int));
        msEncrypt.Write(aesAlg.IV, 0, aesAlg.IV.Length);
        using (CryptoStream csEncrypt = new CryptoStream(msEncrypt, encryptor, CryptoStreamMode.Write))
        {
            using (StreamWriter swEncrypt = new StreamWriter(csEncrypt))
            {
                swEncrypt.Write(plainText);
            }
        }
        outStr = Convert.ToBase64String(msEncrypt.ToArray());
    }
}
finally
{
    if (aesAlg != null)
        aesAlg.Clear();
}

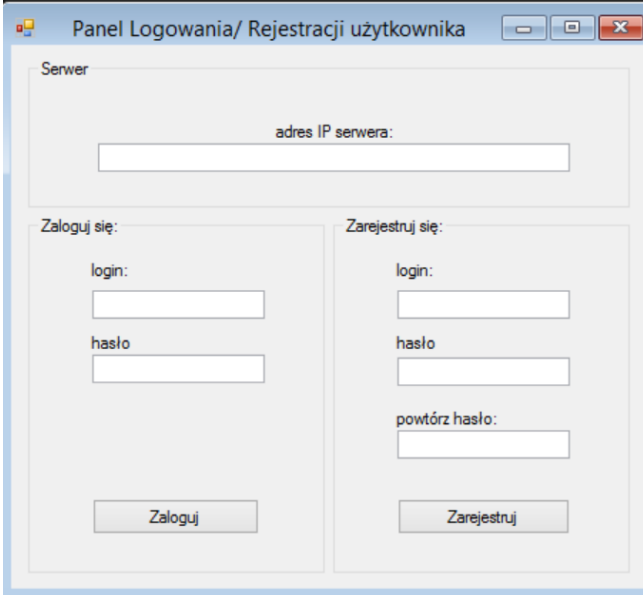
return outStr;
```

Graficzny interfejs użytkownika (GUI)

Projekt został stworzony, w postaci okienkowej. Korzystając z WindowsFormApplication, zostały stworzone trzy formy.

I. Po stronie klienta:

a) LoginPanel, który pełnił rolę panelu rejestracji i logowania:

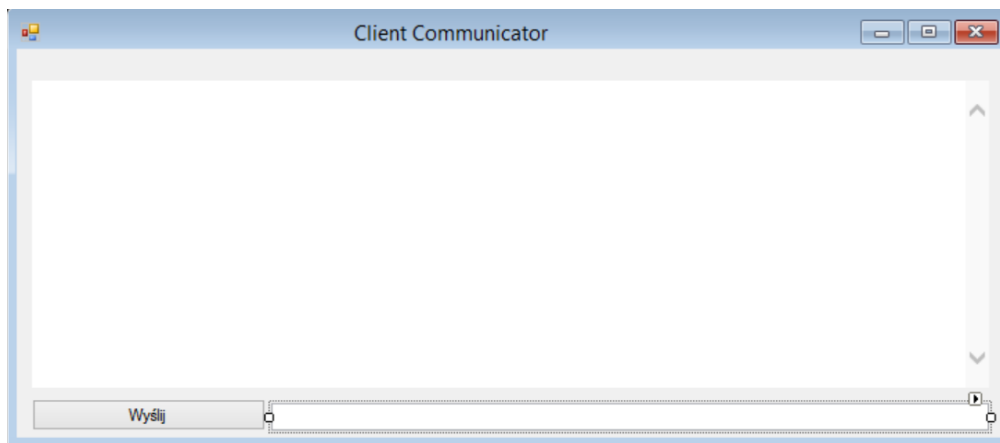


LoginPanel zawiera następujące trzy grupy:

- Serwer, w który wpisujemy adres IP serwera;
- Zaloguj się, w którym realizowany jest proces logowania. Użytkownik podaje login i hasło, klikając "Zaloguj" uzyskuje dostęp do formularza czatu.
- Zarejestruj się, w którym realizowany jest proces rejestracji. Użytkownik podając login, hasło oraz powtórzone hasło, klikając "Zarejestruj" jest w stanie zarejestrować się do serwera, uzyskując dostęp do formularza czatu.

Wszystkie pola są poddawane walidacji.

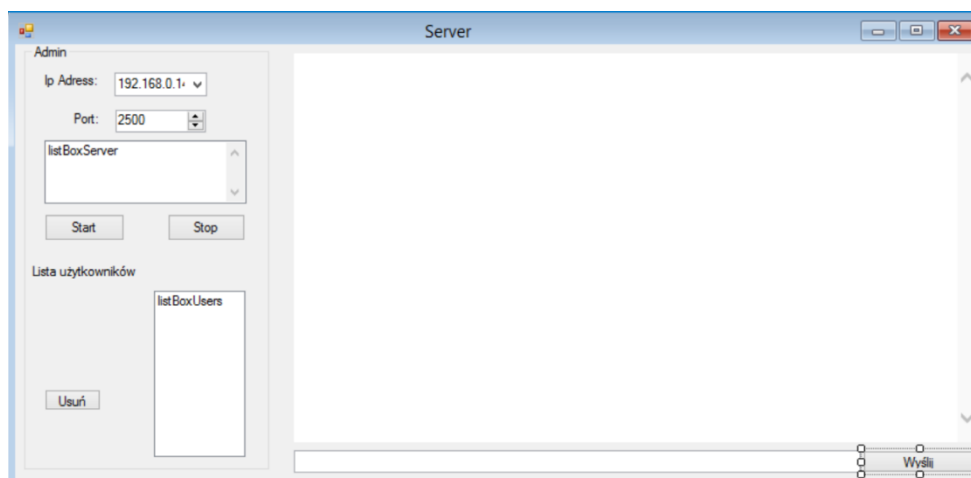
b) Form, który pełni rolę interfejsu czatu



Form zawiera:

- Web Browser, działający podobnie jak przeglądarka internetowa. Element ten, umożliwia wyświetlanie wiadomości w formacie HTML.
- Textbox, w którym użytkownik wpisuje treść wiadomości
- Button, umożliwiający wysłanie wpisanej wiadomości w pole Textbox.

II. Po stronie serwera:



a) grupa Admin:

- adres IP serwera;
- port, na którym serwer będzie nasłuchiwał i przez który będzie wysyłał;
- element listBoxServer, w który wyświetlane są logi informujące o aktywności serwera;
- przycisk start oraz stop, inicjując i kończący pracę serwera;

b) grupa Lista użytkowników:

- element listBoxUsers, w którym wyświetlani są adresy IP, aktualnie podłączonych użytkowników;
- przycisk usuń, który usuwa wybranego użytkownika z listy listBoxUsers.

c) Tak samo jak w przypadku klienta, webBrowser, który wyświetla wszystkie wiadomości, przechodzące przez serwer oraz TextBox, pozwalający administratorowi serwera wysłać wiadomość do wszystkich zalogowanych użytkowników.

6. Wnioski

Z wymienionych w zadaniach funkcjonalności zabrakło u mnie logowania do pliku, wiadomości offline oraz ciekawej funkcjonalności. Resztę zadań wykonałem i przetestowałem. Zdaję sobie sprawę, że mój kod nie jest zbyt czysty, nie spełnia SOLID. Swoją uwagę w głównej mierze przyłożyłem do funkcjonalności. Nad niektórymi trzeba było dłużej się zastanowić. W ogólności projekt nie sprawił mi większych problemów. Pierwszy raz pisałem większą aplikację w języku C# i uważam to za ciekawe doświadczenie. C# jest prosty, przystępny, ma bogatą dokumentację i dosyć sporą społeczność w sieci.