helion.pl
KSIĘGARNIA INTERNETOWA · WYDAWNICTWO

IDŹ DO
PRZYKŁADOWY ROZDZIAŁ
SPIS TREŚCI

KATALOG KSIĄŻEK
KATALOG ONLINE
ZAMÓW DRUKOWANY KATALOG

TWÓJ KOSZYK
DODAJ DO KOSZYKA

CENNIK I INFORMACJE
ZAMÓW INFORMACJE O NOWOŚCIACH
ZAMÓW CENNIK

CZYTELNIA
FRAGMENTY KSIĄŻEK ONLINE

C#. Tworzenie aplikacji sieciowych. 101 gotowych projektów



Autor: Sławomir Orłowski
ISBN: 83-246-0572-X
Format: B5, stron: 264

Stwórz własne aplikacje sieciowe

Komunikacja sieciowa jest jednym z najważniejszych elementów współczesnych aplikacji. Coraz więcej narzędzi wykorzystuje zasoby internetu, umożliwia pracę grupową lub łączy się z siecią w innym celu, np. aktywacji lub rejestracji. Stosowane obecnie platformy programistyczne umożliwiają stosunkowo łatwą implementację mechanizmów sieciowych, a zyskujący coraz większą popularność język C# wydaje się stworzony do tego typu zastosowań. Język ten, oparty na obiektach i komponentach, jest prosty do opanowania. Za jego pomocą można tworzyć nie tylko aplikacje dla „dużych" komputerów, ale także dla platformy PocketPC.

Książka „C#. Tworzenie aplikacji sieciowych. 101 gotowych rozwiązań" to zbiór przykładów ilustrujących sposoby implementacji mechanizmów komunikacji sieciowej w aplikacjach tworzonych w języku C#. Czytając ją, poznasz zasady korzystania z protokołów sieciowych ICMP, HTTP i FTP, tworzenia programów służących do analizy ruchu w sieci, przeglądania stron WWW i pobierania plików z serwerów. Nauczysz się korzystać z technologii ADO.NET i ASP.NET do pisania aplikacji internetowych i komunikacji z bazami danych. Przeczytasz także o przetwarzaniu plików XML oraz tworzeniu i wykorzystywaniu usług sieciowych.

- środowiska programistyczne Visual C# 2005 Express Edition oraz Visual Web Developer 2005 Express Edition
- Protokoły sieciowe
- Analiza połączeń sieciowych
- Edytor stron WWW
- Przesyłanie plików za pomocą FTP
- Komunikatory i czaty
- Tworzenie aplikacji internetowych
- Usługi sieciowe

Wydawnictwo Helion
ul. Kościuszki 1c
44-100 Gliwice
tel. 032 230 98 63
e-mail: helion@helion.pl

Spis treści

Wstęp **7**

Rozdział 1. Język C# i platforma .NET **9**
Technologia .NET. Krótki wstęp 9
Elementy języka C# i programowanie zorientowane obiektowo 10
Przestrzenie nazw 13
Kolekcje 14
Zdarzenia i metody zdarzeniowe 15
Delegacje 15
Wyjątki 15

Rozdział 2. Visual C# 2005 Express Edition. Opis środowiska **17**
Projekt 1. Budujemy interfejs pierwszej aplikacji 18
Projekt 2. Poznajemy pliki projektu pierwszej aplikacji 21
Projekt 3. Interakcja aplikacji z użytkownikiem. Metody zdarzeniowe 25

Rozdział 3. Visual Web Developer 2005 Express Edition. Opis środowiska **29**
Projekt 4. Pierwsza strona ASP.NET. Tworzymy interfejs 29
Projekt 5. Pierwsza strona ASP .NET. Poznajemy pliki projektu 33
Projekt 6. Pierwsza strona ASP.NET. Metody zdarzeniowe 37

Rozdział 4. Programowanie sieciowe **39**
Sieci komputerowe 39
Protokoły TCP i UDP 42
Protokół IP i adresy MAC 44
Programowanie klient-serwer i peer-to-peer 45
Popularne protokoły sieciowe 46
Protokół ICMP 46
Protokół HTTP 47
Protokół FTP 47

Rozdział 5. Aplikacje TCP i UDP **49**
Projekt 7. Połączenie TCP. Klient 49
Projekt 8. Połączenie TCP. Serwer 52
Projekt 9. Odczytanie adresu IP przyłączonego hosta 55
Projekt 10. Połączenie UDP. Klient 56
Projekt 11. Połączenie UDP. Serwer 58
Projekt 12. Prosty skaner otwartych portów 59
Projekt 13. Sprawdzenie adresu IP naszego komputera 61

Projekt 14. Komplet informacji na temat połączeń sieciowych ... 63
Projekt 15. Ping ... 65
Projekt 16. Ping. Przeciwdziałanie zablokowaniu interfejsu ... 69
Projekt 17. NetDetect. Sprawdzanie dostępnych komputerów w sieci ... 70
Projekt 18. Traceroute. Śledzenie drogi pakietu ICMP ... 72
Projekt 19. Traceroute. Zatrzymanie wywołania asynchronicznego ... 76
Projekt 20. Protokół HTTP. Sprawdzanie dostępnych uaktualnień ... 76
Projekt 21. Pobieranie pliku z użyciem protokołu HTTP ... 77
Projekt 22. Pobranie źródła strony z serwera WWW ... 79
Projekt 23. Przeglądarka WWW ... 80
Projekt 24. Edytor HTML. Budowanie interfejsu ... 83
Projekt 25. Edytor HTML. Obsługa plików tekstowych ... 84
Projekt 26. Edytor HTML. Współpraca ze schowkiem ... 86
Projekt 27. Edytor HTML. Wprowadzanie tagów ... 87
Projekt 28. Edytor HTML. Podgląd bieżącej strony ... 90
Projekt 29. Wysyłanie wiadomości e-mail bez uwierzytelnienia ... 91
Projekt 30. Wysyłanie sformatowanej wiadomości e-mail z załącznikami ... 94
Projekt 31. Wysyłanie poczty za pomocą serwera wymagającego uwierzytelnienia ... 97
Projekt 32. Masowe wysyłanie wiadomości e-mail ... 98
Projekt 33. Klient FTP. Interfejs aplikacji ... 102
Projekt 34. Klient FTP. Definiowanie pól i własności klasy FTPClient ... 103
Projekt 35. Klient FTP. Listowanie katalogów serwera FTP ... 107
Projekt 36. Klient FTP. Zmiana katalogu ... 110
Projekt 37. Klient FTP. Metoda pobierająca plik asynchronicznie ... 112
Projekt 38. Klient FTP. Wywołanie metody pobierającej plik asynchronicznie ... 115
Projekt 39. Klient FTP. Metoda wysyłająca plik asynchronicznie ... 117
Projekt 40. Klient FTP. Wywołanie metody wysyłającej plik asynchronicznie ... 119
Projekt 41. Klient FTP. Kasowanie pliku ... 121
Projekt 42. Menedżer pobierania plików w tle. Budowa interfejsu ... 122
Projekt 43. Menedżer pobierania plików w tle. Pobieranie pliku ... 124
Projekt 44. Menedżer pobierania plików w tle. Przerwanie pobierania pliku ... 126
Projekt 45. Serwer Uśmiechu. Budowa interfejsu ... 127
Projekt 46. Serwer Uśmiechu. Lista kontaktów ... 129
Projekt 47. Serwer Uśmiechu. Wysyłanie danych do wielu odbiorców ... 132
Projekt 48. Klient Uśmiechu. Umieszczenie ikony w zasobniku systemowym ... 133
Projekt 49. Klient Uśmiechu. Oczekiwanie na połączenie w osobnym wątku ... 136
Projekt 50. Klient Uśmiechu. Bezpieczne odwoływanie się do własności kontrolek formy z poziomu innego wątku ... 138
Projekt 51. Komunikator. Serwer. Budowa interfejsu ... 139
Projekt 52. Komunikator. Serwer. Bezpieczne odwoływanie się do własności kontrolek formy z poziomu innego wątku ... 142
Projekt 53. Komunikator. Serwer. Obsługa rozmowy ... 143
Projekt 54. Komunikator. Klient ... 148
Projekt 55. Zdalny screenshot. Klient. Zrzut ekranu ... 150
Projekt 56. Zdalny screenshot. Klient ... 151
Projekt 57. Klient. Wysyłanie informacji o dostępności klienta ... 153
Projekt 58. Serwer screenshot. Budowa interfejsu ... 154
Projekt 59. Serwer screenshot. Bezpieczne odwoływanie się do własności kontrolek formy z poziomu innego wątku ... 155
Projekt 60. Serwer screenshot. Lista aktywnych klientów ... 156
Projekt 61. Serwer screenshot. Pobranie zrzutu ekranu ... 157

Projekt 62. Serwer. Czat. Budowanie interfejsu ... 159
Projekt 63. Serwer Czat. Bezpieczne odwoływanie się do własności kontrolek formy z poziomu innego wątku ... 160
Projekt 64. Serwer Czat. Klasa formy oraz pętla główna programu ... 161
Projekt 65. Serwer Czat. Obsługa wątków związanych z klientami ... 165
Projekt 66. Serwer Czat. Rozłączenie klienta ... 167
Projekt 67. Czat. Klient ... 167

Rozdział 6. Remoting ... 173
Projekt 68. Serwer HTTP ... 174
Projekt 69. Klient HTTP ... 177
Projekt 70. Serwer TCP ... 178
Projekt 71. Klient TCP ... 180
Projekt 72. Serwer TCP. Plik konfiguracyjny ... 181
Projekt 73. Klient TCP. Plik konfiguracyjny ... 183
Projekt 74. Czat. Klasa serwera ... 184
Projekt 75. Czat. Serwer ... 186
Projekt 76. Czat. Klient ... 187

Rozdział 7. ASP.NET i ADO.NET ... 191
Projekt 77. Pozycjonowanie kontrolek na stronie. Pozycja względna ... 192
Projekt 78. Pozycjonowanie kontrolek na stronie. Pozycja bezwzględna ... 196
Projekt 79. Ping ... 197
Projekt 80. Wysyłanie wiadomości e-mail ... 198
Projekt 81. Pobieranie plików na serwer ... 200
Projekt 82. Wysyłanie wiadomości e-mail z załącznikami ... 201
Projekt 83. Księga gości. Współpraca z plikiem XML ... 202
Projekt 84. Księga gości. Wyświetlanie zawartości pliku XML ... 206
Projekt 85. Księga gości. Sprawdzanie poprawności wpisywanych danych ... 208
Projekt 86. Księga gości. Liczba gości online ... 210
Projekt 87. Wielojęzyczny serwis internetowy. Zasoby lokalne ... 212
Projekt 88. Wielojęzyczny serwis internetowy. Zasoby globalne ... 217
Projekt 89. Wielojęzyczny serwis internetowy. Wybór języka przez użytkownika ... 219
Projekt 90. Identyfikacja użytkowników ... 221
Projekt 91. Rejestrowanie nowych użytkowników ... 225
Projekt 92. Identyfikacja użytkowników II ... 226
Projekt 93. Baza książek. Stworzenie bazy danych ... 227
Projekt 94. Baza książek. Przyłączenie się do bazy danych ... 230
Projekt 95. Baza książek. Prezentacja danych ... 231

Rozdział 8. Web Services ... 235
Projekt 96. Pierwsza usługa sieciowa ... 236
Projekt 97. Korzystanie z usługi sieciowej ... 239
Projekt 98. Usługa Google Web API. Rejestracja usługi ... 240
Projekt 99. Usługa Google Web API. Klient ... 242
Projekt 100. Usługa Google Web API. Klient. Podpowiedź ... 245
Projekt 101. Usługa Google Web API. Klient. Zapisywanie kopii stron ... 247

Skorowidz ... 249

Rozdział 5.

Aplikacje TCP i UDP

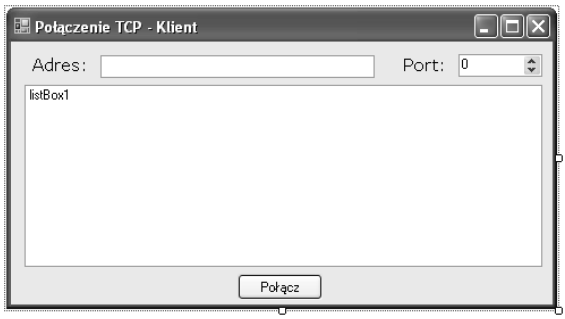
Rozdział ten stanowi główną część całej książki. Opisane zostały w nim podstawy programowania aplikacji sieciowych używających protokołów TCP i UDP dla platformy .NET. Bez zbytnich uogólnień oraz wywodów teoretycznych zamieszczone są tu praktyczne rozwiązania, jakie stosuje się w programowaniu. Problemy, z jakimi przyjdzie nam się zmierzyć, mają różne stopnie trudności. Począwszy od ustanowienia połączenia TCP po programowanie serwera współbieżnego. Przedstawiony jest tutaj również język C#: jego składnia i sposób użycia. Wybór projektów jest oczywiście dosyć subiektywny i opiera się na problemach, z jakim spotkał się autor podczas pracy nad konkretnymi projektami. Niemniej jednak stanowić powinien dobry przegląd metod oraz tworzyć solidne podstawy do dalszych, samodzielnych studiów.

Projekt 7. Połączenie TCP. Klient

Naszym pierwszym projektem będzie prosta aplikacja nawiązująca połączenie TCP ze wskazanym komputerem w sieci. Po ustanowieniu połączenia będzie ono zamykane. Na początek nie będziemy przesyłać żadnych dodatkowych informacji pomiędzy klientem i serwerem. W tym celu użyjemy klasy `TcpClient` z przestrzeni nazw `System.Net.Socket`. Sam protokół TCP opisany został w rozdziale 4.

1. Z menu *File* wybieramy pozycję *New Project*....
2. W oknie wyboru projektu zaznaczamy ikonę *Windows Application*.
3. W polu *Name* wpisujemy nazwę `PoloczenieTCPKlient`.
4. W widoku projektu do formy dodajemy kontrolkę `textBox1` z okna *Toolbox*. W to pole użytkownik będzie mógł wpisywać adres serwera, do którego będzie chciał się przyłączyć.
5. Na formę wrzucamy kontrolkę `numericUpDown1`. Będzie ona przechowywać numer portu, na którym ma nastąpić połączenie.
6. Własność `Maximum` kontrolki `numericUpDown1` ustawiamy na `655353`. Jest to numer najwyższego portu.

7. Dodajemy również kontrolkę `listBox1`, która zawierać będzie komunikaty dotyczące nawiązania połączenia.
8. Do formy dodajemy przycisk `button1`. Kliknięcie go spowoduje próbę nawiązania połączenia.
9. Za pomocą okna właściwości własność `Text` kontrolki `button1` ustawiamy na *Połącz*.
10. Dodajemy również opisy wszystkich pól naszej aplikacji. W tym celu na formę wrzucamy dwie kontrolki `label1` i `label2`. Opisujemy je jako `Adres:` i `Port:`. Proponowane umieszczenie kontrolek przedstawia rysunek 5.1.



Rysunek 5.1. *Wygląd projektu aplikacji TCP Klient*

11. Oprogramujemy teraz metodę zdarzeniową `Click` kontrolki `button1`. Ponieważ jest to domyślna metoda zdarzeniowa, możemy to zrobić na dwa sposoby:
    a) Klikamy dwukrotnie kontrolkę `button1`.
    b) W oknie własności klikamy ikonę `Event`, odnajdujemy zdarzenie `Click` i dwukrotnie je klikamy.
12. Bez względu na sposób wywołania zdarzenia środowisko przenosi nas do widoku kodu. Nagłówek naszej metody został utworzony automatycznie. Piszemy teraz kod wyróżniony w listingu 5.1.

Listing 5.1. *Kod klienta TCP*

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Text;
using System.Windows.Forms;
using System.Net.Sockets;

namespace PoloczenieTCPKlient
{
    public partial class Form1 : Form
    {
        public Form1()
        {
            InitializeComponent();
        }
```

```
        private void button1_Click(object sender, EventArgs e)
        {
            string host = textBox1.Text;
            int port = System.Convert.ToInt16(numericUpDown1.Value);
            try
            {
                TcpClient klient = new TcpClient(host, port);
                listBox1.Items.Add("Nawiązano połączenie z " + host + " na porcie:
                ➥" + port);
                klient.Close();
            }
            catch(Exception ex)
            {
                listBox1.Items.Add("Błąd: Nie udało się nawiązać połączenia!");
                MessageBox.Show(ex.ToString());
            }
        }
    }
}
```

Na początku w sekcji `using` dodajemy standardowo przestrzeń nazw `System.Net.Socket`. Zawiera ona klasę `TCPClient`. Wewnątrz metody zdarzeniowej `Click` przycisku `button1` definiujemy zmienną `host` typu `string` i podstawiamy pod nią zawartość pola *Adres* (kontrolki `textBox1`). Druga zmienna `port` zawierać będzie numer portu wprowadzonego przez użytkownika do kontrolki `numericUpDown1`. Jak widać, dokonaliśmy konwersji wartości z pola `Value` typu `decimal` na `int`. Do konwersji służą metody klasy `System.Convert`. Dalej inicjujemy obiekt klasy `TcpClient` i do jego konstruktora przekazujemy wartości zmiennych `host` i `port`. Po nawiązaniu połączenia, wypisujemy odpowiedni komunikat w kontrolce `listBox1`, po czym zamykamy połączenie metodą `Close`. Próbę połączenia musimy umieścić w bloku `try/catch`, aby wyeliminować komunikaty ewentualnych wyjątków zgłaszanych przez system w momencie braku możliwości nawiązania połączenia ze wskazanym hostem. Powstałe wyjątki obsługujemy sami w bloku `catch`. Do kontrolki `listBox1` dodajemy informację o tym, że próba nawiązania połączenia się nie powiodła. Dodatkowo za pomocą metody `MessageBox.Show` wyświetlamy okno informacji z dokładnymi danymi dotyczącymi wyjątku.

13. Zapisujemy nasz projekt (kombinacja klawiszy *Ctrl+S*).
14. Klawiszem *F5* kompilujemy i uruchamiamy aplikację.
15. W pole adres wprowadźmy `www.helion.pl`.
16. Jako numer portu wybierzmy *80*. Oznacza to, że będziemy chcieli połączyć się serwerem WWW o nazwie domenowej *helion.pl*.

W przypadku braku połączenia z Internetem, zamiast łączyć się z hostem zdalnym (punkty 15. i 16.), możemy poprzez adres pętli zwrotnej 127.0.0.1 ustanowić połączenie sami ze sobą. W pole adresu, jak widać, oprócz nazw DNS można również wprowadzać numer IP. Najprawdopodobniej nie będziemy w stanie nawiązać połączenia na porcie 80 (chyba że lokalnie mamy uruchomiony serwer WWW). W kolejnym projekcie wykonamy więc serwer otwierający port wskazany przez nas i oczekujący na połączenie.



Jeżeli w naszym systemie działa zapora sieciowa (ang. *firewall*), wówczas musimy ją ustawić tak, aby umożliwiała naszej aplikacji połączenie się z internetem. W przeciwnym wypadku nie będziemy w stanie nawiązać połączenia.

Projekt 8. Połączenie TCP. Serwer

Pora teraz zaimplementować serwer oczekujący na połączenie. Nie będziemy obsługiwać tego połączenia, a jedynie wyświetlimy odpowiedni komunikat po stronie serwera.

1. Tworzymy nowy projekt o nazwie `PolocznieTCPSerwer`.
2. W widoku projektu do formy dodajemy kontrolkę `textBox1`.
3. Na formę wrzucamy kontrolkę `numericUpDown1`.
4. Dodajemy także komponent `listBox1`. Będzie on wyświetlał wszystkie komunikaty odnoszące się do pracy naszego serwera.
5. Jak w poprzednim projekcje, również tutaj do formy dodajemy kontrolki `label1` i `label2`, które posłużą nam do opisu pól edycyjnych formy (`textBox1` i `numericUpDown1`).
6. Do deklaracji przestrzeni nazw dodajemy wpis `using System.Net.Socket`.
7. Dodajemy również przestrzeń nazw `System.Net`, która dostarcza nam klasę `IPAdress`. Klasa ta będzie potrzebna do prawidłowego przekształcenia obiektu typu `string` w adres IP akceptowany przez metodę inicjującą serwer.
8. Do klasy formy dodajemy prywatne pole `serwer` typu `TCPListener` oraz pole klient typu `TcpClient` (listing 5.2).

Listing 5.2. *Deklaracja potrzebnych przestrzeni nazw oraz prywatnego pola serwer klasy Form1*

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Text;
using System.Windows.Forms;
using System.Net.Sockets;
using System.Net;

namespace PoloczenieTCPSerwer
{
    public partial class Form1 : Form
    {
        public Form1()
        {
            InitializeComponent();
        }
```

```
    private TcpListener serwer = null;
      private TcpClient klient = null;
    }
}
```

9. Na formę przeciągamy przycisk `button1`. Własność `Text` zmieniamy na *Start*. Za jego pomocą będziemy uruchamiać serwer.
10. Do formy dodajemy kontrolkę `button2`. Własność `Text` ustalamy na *Stop*. Posłuży nam ona do zatrzymania serwera. Własność `Enabled` tego przycisku ustawiamy na `false`.
11. Dla domyślnej metody zdarzeniowej kontrolki `button1` piszemy kod z listingu 5.3.

Listing 5.3. *Metoda inicjująca serwer TCP*

```
private void button1_Click(object sender, EventArgs e)
        {
            IPAddress adresIP = null;

            try
            {
                adresIP = IPAddress.Parse(textBox1.Text);
            }
            catch
            {
                MessageBox.Show("Błędny format adresu IP!","Błąd");
                textBox1.Text = String.Empty;
                return;
            }

            int port = System.Convert.ToInt16(numericUpDown1.Value);

            try
            {
                serwer = new TcpListener(adresIP, port);
                serwer.Start();
                klient = serwer.AcceptTcpClient();
                listBox1.Items.Add("Nawiązano połączenie");
                button1.Enabled = false;
                button2.Enabled = true;
               klient.Close();
                serwer.Stop();
            }
            catch (Exception ex)
            {
                listBox1.Items.Add("Błąd inicjacji serwera!");
                MessageBox.Show(ex.ToString(),"Błąd");
            }
        }
```

Kod ten wymaga kilku komentarzy. Na początku metody tworzymy obiekt `IPAddress`. Jest nam on dalej potrzebny do inicjacji serwera. Inicjacja serwera polega na wywołaniu konstruktora klasy `TcpListener` oraz użyciu metody `Start()`.

Konstruktor klasy `TcpListener` jest w wersji .NET 2.0 dwuargumentowy. Pierwszym argumentem jest adres IP naszego serwera, co na początku wydaje się nieco dziwne. Ale może się zdarzyć, że nasz serwer chcemy uruchomić lokalnie (adres 127.0.0.1) do testów. Możemy również posiadać więcej niż jeden adres IP przypisany naszej maszynie. Drugim argumentem konstruktora jest port, na jakim nasza aplikacja ma oczekiwać na połączenia. Istnieje również konstruktor bezparametrowy, po którym należy użyć metody `Connect` z argumentami identycznymi jak w przypadku konstruktora dwuparametrowego. Port jest otwierany w momencie użycia metody `Start()`. Po przyłączeniu się klienta za pomocą metody `AcceptTcpClient()` tworzymy obiekt `klient` klasy `TCPClient`, który w przyszłości będzie służył do wymiany danych pomiędzy serwerem a klientem. Metoda ta działa na zasadzie blokowania gniazda (ang. *blocking socket*). Blokowany jest interfejs całej aplikacji, aż do momentu uzyskania połączenia. W kolejnych projektach będziemy temu przeciwdziałać poprzez stworzenie osobnego wątku. Odpowiednie fragmenty mogące potencjalnie generować wyjątki otoczone są blokami ochronnymi `try/catch`. Dodatkowo jeżeli nastąpi połączenie, wówczas przycisk *Start* zmieniamy na nieaktywny. Zapobiegnie to powtórnemu utworzeniu serwera i próbie otwarcia portu. Jest to zabieg bardziej estetyczny, ponieważ interfejs aplikacji w momencie uruchomienia i tak jest zablokowany, przez co forma i wszystkie jej kontrolki są dla nas niedostępne.

12. Zajmijmy się teraz funkcją wyłączającą nasz serwer. Dla metody zdarzeniowej `Click` kontrolki `button2` piszemy kod z listingu 5.4.

Listing 5.4. *Metoda zatrzymująca pracę serwera TCP*

```
private void button2_Click(object sender, EventArgs e)
        {
            serwer.Stop();
            klient.Close();
            listBox1.Items.Add("Zakończono pracę serwera ...");
            button1.Enabled = true;
            button2.Enabled = false;
        }
```

W metodzie tej wyłączamy serwer oraz zamykamy klienta uruchomionego po stronie serwera. Zmieniamy również przycisk *Start* na aktywny.

Pora teraz na przetestowanie naszej aplikacji:

1. Uruchamiamy serwer, naciskając klawisz *F5*.
2. W polu adresu wpiszemy `127.0.0.1`. Jest to adres pętli zwrotnej interfejsu sieciowego.
3. Jako numer portu wybieramy *20000*.
4. Klikamy przycisk *Start*. W tym momencie interfejs naszej aplikacji zostaje „zamrożony" i serwer oczekuje na połączenia od klientów.

5. Za pomocą np. eksploratora plików przechodzimy teraz do katalogu, gdzie zapisaliśmy naszą aplikację klienta TCP. Zwykle znajduje się ona w katalogu *Moje Dokumenty*. Wchodzimy do katalogu *(ścieżka do katalogu projektu)**bin**Debug*. Znajduje się tam plik wykonywalny naszej aplikacji *PoloczenieTCPKlient.exe*. Uruchamiamy go.
6. W pole *adres aplikacji klienta* wpisujemy adres IP `127.0.0.1` (lub `localhost`).
7. Port ustawiamy ten sam, na którym działa nasz serwer (*20000*).
8. Klikamy przycisk *Połącz*.

Napisany przez nas klient powinien łączyć się z aplikacją serwerem. Jeżeli coś nie działa, przypominam o odpowiednim ustawieniu zapory sieciowej.

Spróbujmy jeszcze nieco inaczej. Nasz serwer oczekuje na połączenie od dowolnej aplikacji TCP, nie tylko od napisanego przez nas klienta. Użyjmy więc programu telnet:

1. Uruchamiamy ponownie nasz serwer.
2. Wybieramy adres IP oraz port. Niech będzie to ten sam adres i port, który wybraliśmy poprzednio (*127.0.0.1:20000*). Przyciskamy *Start*. Serwer oczekuje na połączenie.
3. Z menu *Start* systemu Windows wybieramy opcję *Uruchom…*.
4. W polu *Otwórz* wpisujemy `cmd`. W ten sposób otworzymy terminal systemowy.
5. W terminalu wpisujemy polecenie: `telnet 127.0.0.1 20000`. W ten sposób nawiązane zostanie połączenie z naszym serwerem.

Jak widać, dowolna aplikacja używająca połączenia TCP może połączyć się z naszym serwerem. W celu ochrony przed niechcianymi połączeniami możemy zastosować filtrowanie adresów IP (patrz projekt 9.) bądź wprowadzić uwierzytelnianie. Uwierzytelnianie może polegać na przesyłaniu hasła lub używaniu specjalnego protokołu. Więcej na ten temat znaleźć można w projektach 51 – 54 i 62 – 67.

Projekt 9. Odczytanie adresu IP przyłączonego hosta

Często zdarza się, że chcemy odczytać adres IP komputera, który łączy się z naszym serwerem TCP. Możemy to zrobić np. w celu weryfikacji klientów.

1. Otwieramy powtórnie projekt *TCPSerwer*.
2. Odnajdujemy metodę `Click` przycisku `button1` i uzupełniamy ją o kod z listingu 5.5.

Listing 5.5. *Odczytywanie adresu IP i portu klienta*

```
private void button1_Click(object sender, EventArgs e)
        {
            IPAddress adresIP = null;
            try
            {
                adresIP = IPAddress.Parse(textBox1.Text);
            }
            catch
            {
                MessageBox.Show("Błędny format adresu IP!", "Błąd");
                textBox1.Text = String.Empty;
                return;
            }
            int port = System.Convert.ToInt16(numericUpDown1.Value);
            try
            {
                serwer = new TcpListener(adresIP, port);
                serwer.Start();
                klient = serwer.AcceptTcpClient();
                IPEndPoint IP = (IPEndPoint)klient.Client.RemoteEndPoint;
                listBox1.Items.Add("["+IP.ToString()+"] :Nawiązano połączenie");
                klient.Close();
                serwer.Stop();
            }
            catch (Exception ex)
            {
                listBox1.Items.Add("Inicjacja serwera nie powiodła się!");
                MessageBox.Show(ex.ToString(), "Błąd");
            }
        }
```

Klasa `Socket` reprezentuje punkt końcowy połączenia (ang. *end point*). W tym przypadku jest to klient przyłączony do serwera. Dzięki niej możemy odczytać adres IP i numer portu, na jakim działa klient. Dane te odczytujemy za pomocą metody `RemoteEndPoint`, która zwraca obiekt klasy `IPEndPoint`. Klasa `TCPClient` używa klasy `Socket` do przesyłania danych przez sieć. Aby otrzymać obiekt `Socket` z klasy `TCPClient`, musimy posłużyć się klasą `Client`. Przesyłanie danych przez sieć opisane będzie w kolejnych przykładach. W tym i dwóch poprzednich projektach skupiliśmy się na samym połączeniu.

Projekt 10. Połączenie UDP. Klient

Ten projekt jest odpowiednikiem projektu numer 7, jednak zamiast protokołu TCP użyjemy protokołu UDP. Nie będziemy mieli więc gwarancji tego, że dane przesyłane przez nas będą odbierane przez serwer i odwrotnie. Aplikacja ta będzie miała za zadanie wysłanie krótkiej wiadomości do serwera. Serwer, który napiszemy w następnym projekcie, będzie tę wiadomość odbierał i wyświetlał.

1. Tworzymy nowy projekt o nazwie `UDPKlient`.
2. Do formy dodajemy kontrolkę `textBox1`. W to pole będziemy wprowadzać adres DNS lub IP hosta, z którym chcemy nawiązać połączenie.
3. Na formę wrzucamy kontrolkę `numericUpDown1`. Za jej pomocą będziemy ustalać numer portu, poprzez który będziemy się łączyć.
4. Wartość `Maximum` kontrolki `numericUpDown1` ustalamy na *65535*.
5. Kolejną kontrolką dodaną do projektu będzie `listBox1`.
6. Do projektu dodajemy jeszcze przycisk `button1`. Wartość `Text` ustalamy na *Wyślij*.
7. Do formy dodajemy kontrolkę `textBox2`, w którą będziemy mogli wprowadzać tekst komunikatu wysyłanego do serwera.
8. Dodajemy dwie kontroli: `label1` i `label2`. Ich własność `Text` ustalamy odpowiednio na *Adres:* i *Port:*. Będą one opisywały nam pole `textBox1` oraz `numericUpDown1`.
9. Dodajemy przestrzeń nazw `System.Net.Socket`.
10. Dla kontrolki `button1` tworzymy domyślną metodę zdarzeniową. Odpowiedni kod zawiera listing 5.6.

Listing 5.6. *Metoda zdarzeniowa Click kontrolki textBox1*

```
private void button1_Click(object sender, EventArgs e)
        {
            string host = textBox1.Text;
            int port = (int)numericUpDown1.Value;
            try
            {
                UdpClient klient = new UdpClient(host, port);
                Byte[] dane = Encoding.ASCII.GetBytes(textBox2.Text);
                klient.Send(dane, dane.Length);
                listBox1.Items.Add("Wysłanie wiadomośći do hosta " + host
                ➥+ ":" + port);
                klient.Close();

            }
            catch(Exception ex)
            {
                listBox1.Items.Add("Błąd: Nie udało się wysłać wiadomości!");
                MessageBox.Show(ex.ToString(), "Błąd");
            }
        }
```

Jak widać, kod tej aplikacji jest zbliżony do projektu *TCPKlient* (listing 5.1). Wyjątkiem jest korzystanie z obiektu klasy UDPClient zamiast TCPClient. Za jego pomocą wysyłana jest wiadomość. Konstruktor przyjmuje te same parametry co w przypadku połączenia TCP. Kod próbujący nawiązać połączenie umieszczamy w bloku ochronnym `try/catch`, aby wyłączyć generowanie ewentualnych wyjątków. Korzystając z klasy

`Encoding`, zamieniamy ciąg znaków z kontrolki `textBox2` na ciąg bajtów, który następnie będziemy mogli wysłać pod wskazany adres IP. Posłuży nam do tego metoda `Send`. Przyjmuje ona dwa argumenty: dane jako tablicę typu `Byte` oraz długość tej tablicy. Tak jak poprzednio, tutaj również dodano przestrzeń nazw `System.Net.Socket`.



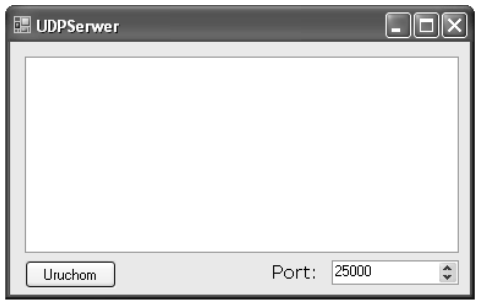
Proszę zauważyć, że próba wysłania wiadomości na dowolny port pod adres np. *www.fizyka.umk.pl* nie spowoduje zgłoszenia wyjątku. Dzieje się tak dlatego, że protokół UDP nie troszczy się o to, czy wysłany pakiet dotrze do celu i zostanie odczytany.

Projekt 11. Połączenie UDP. Serwer

Serwer UDP zasadniczo różni się implementacyjnie od serwera TCP. Po pierwsze, nie mamy odpowiednika klasy `TcpListener`[1]. Pozostaje nam użycie klasy `UDPClient` i związanie obiektu tej klasy z lokalnym portem. Jak wiadomo, połączenie UDP nie gwarantuje dostarczenia danych. Brak kontroli pakietów prowadzi z jednej strony do możliwości ich utraty w trakcie przesyłania, jednak z drugiej strony połączenie UDP generuje mniej ruchu w sieci. Jest więc szybsze. Nasz serwer będzie oczekiwał na dane wysłane od klienta. Po ich otrzymaniu zostaną one wyświetlone w kontrolce `listBox1`.

1. Tworzymy nową aplikację *Windows Forms* o nazwie `UDPSerwer`.
2. Interfejs tej aplikacji będzie zbliżony do projektu 7. (rysunek 5.2). Do projektu dodajemy kontrolkę `numericUpDown1`. Jej własność `Maximum` ustawiamy na *65535*.

Rysunek 5.2. *Wygląd interfejsu aplikacji UDPSerwer*



3. Do formy dodajemy kontrolkę `listBox1`. Będzie ona odpowiedzialna za wypisywanie odbieranych komunikatów.
4. Na formę wrzucamy jeszcze kontrolkę `label1`. Posłuży nam ona do opisu pola `numericUpDown1`. Własność `Text` ustawiamy na *Port:*.
5. Dodajemy jeszcze przycisk `button1`. Będzie on odpowiedzialny za start naszego serwera.

[1] Jest to oczywiste następstwo stosowania protokołu UDP, gdzie nie ma mowy o nawiązywaniu połączenia (patrz rozdział 4.).

6. Dodajemy przestrzeń nazw `System.Net` oraz `System.Socket`.

7. Dla domyślnej metody zdarzeniowej kontrolki `button1` piszemy kod z listingu 5.7.

Listing 5.7. *Serwer oparty na protokole UDP*

```
private void button1_Click(object sender, EventArgs e)
        {
            int port = (int)numericUpDown1.Value;
            IPEndPoint zdalnyIP = new IPEndPoint(IPAddress.Any, 0);
            try
            {
                UdpClient serwer = new UdpClient(port);
                Byte[] odczyt = serwer.Receive(ref zdalnyIP);
                string dane = Encoding.ASCII.GetString(odczyt);
                listBox1.Items.Add(dane);
                serwer.Close();
            }
            catch (Exception ex)
            {
                MessageBox.Show(ex.Message, "Błąd");
            }
        }
```

Przykład ten pokazuje, w jaki sposób możemy odebrać wysłaną przez klienta wiadomość. Obiekt klasy `IPEndPoint` reprezentuje nam hosta wysyłającego do nas wiadomość. W naszym przypadku dopuszczamy połączenia od dowolnego komputera w sieci. W bloku ochronnym tworzona jest referencja do klasy `UdpClient` użytej tutaj jako serwer. Przygotowujemy następnie zmienną `odczyt`, będącą tablicą typu `Byte`, i do niej zapisujemy wysłaną przez klienta zdalnego wiadomość. Używamy w tym celu metody `Receive`, która jako argument przyjmuje referencję klasy `IPEndPoint` reprezentującą zdalnego hosta. Wiadomość ta jest dalej zamieniana na obiekt typu `string` i wyświetlana. Do zamiany ciągu bajtów na znaki Unicode używamy wygodnej klasy `Encoding`. Działamy przy tym odwrotnie niż w przypadku poprzedniego projektu. Po odebraniu wiadomości zamykamy nasz serwer metodą `Close`.



Z racji swoich ograniczeń protokół UDP zwykle nie jest używany dla architektury klient-serwer (żądanie-odpowiedź). Powyższe przykłady mają jedynie pokazać różnice w implementacji protokołu TCP i UDP.

Projekt 12. Prosty skaner otwartych portów

Zmodyfikujemy teraz nieco naszą aplikację klienta TCP, tak aby działała jako prosty skaner otwartych portów. Będziemy próbować nawiązać połączenie ze wskazanym komputerem na wszystkich portach z pewnej listy. Podstawową niedogodnością jest czas oczekiwania na zakończenie operacji skanowania wszystkich portów. Czas ten wydłuża się, jeżeli skanowany komputer ma większość portów zamkniętych.

1. Tworzymy nową aplikację *Windows Forms* o nazwie `SkanerPortow`.
2. Na formę wrzucamy kontrolkę `textBox1`. Użytkownik będzie mógł do niej wprowadzić nazwę komputera, który będzie chciał skanować.
3. Do projektu dodajemy kontrolkę `listBiox1`. Będzie ona przechowywać komunikaty dotyczące działania naszej aplikacji.
4. Dodajemy również przycisk `button1`, który będzie uruchamiał skanowanie portów dla wskazanego komputera.
5. Dla metody zdarzeniowej `Click` kontrolki `button1` piszemy kod z listingu 5.8.

Listing 5.8. *Skanowanie portów za pomocą klasy TcpClient*

```
private void button1_Click(object sender, EventArgs e)
        {
            short[] ListaPortow = { 20, 21, 22, 23, 25, 53, 70, 80, 109, 110, 119,
            ➥143, 161, 162, 443, 3389 };
            string host = textBox1.Text;
            listBox1.Items.Add("Skanowanie portów dla "+host);
            listBox1.Items.Add("To może potrwać chwilę ...");
            foreach (short port in ListaPortow)
            {
                this.Refresh();
                try
                {
                    TcpClient klient = new TcpClient(host, port);
                    listBox1.Items.Add("Port:"+port.ToString()+" jest otwarty");
                }
                catch
                {
                    listBox1.Items.Add("Port:"+port.ToString()+" jest zamknięty");
                }

            }
        }
```

Na początku naszej metody tworzymy tablicę zawierającą listę kilkunastu portów. Jak widać, są to ogólnie znane porty, na których działają podstawowe usługi (por. rozdział 4.). W pętli `foreach` staramy się nawiązać połączenie na kolejnym porcie z listy. Jeżeli operacja się powiedzie, wyświetlony zostanie odpowiedni komunikat z numerem portu. W przeciwnym wypadku podniesiony zostanie wyjątek. Ponieważ próbę połączenia umieściliśmy w bloku ochronnym `try/catch`, to mamy możliwość wygenerowania odpowiedniego komunikatu. Jak widać, sam kod jest stosunkowo prosty. Jego wspomnianą wcześniej wadą jest szybkość działania. Jako proste zadanie Czytelnik może zmodyfikować kod tak, aby oprócz numeru portu, wyświetlana była również usługa standardowo do niego przypisana. Pętla `foreach` użyta w tym projekcie szczególnie nadaje się do iteracji po obiektach z pewnej kolekcji. W tym przypadku iteracja wykonywana jest na elementach z tablicy.