Sławomir Orłowski
Maciej Grabek

C#

Tworzenie aplikacji sieciowych

GOTOWE PROJEKTY

Wykorzystaj rewolucję sieciową i twórz nowatorskie aplikacje!

Helion

Autorstwo: Sławomir Orłowski (wstęp, rozdziały 1-8), Maciej Grabek (rozdział 9).

Redaktor prowadzący: Ewelina Burska

Projekt okładki: Jan Paluch

Materiały graficzne na okładce zostały wykorzystane za zgodą Shutterstock.

Wydawnictwo HELION
ul. Kościuszki 1c, 44-100 GLIWICE
tel. 32 231 22 19, 32 230 98 63
e-mail: helion@helion.pl
WWW: http://helion.pl (księgarnia internetowa, katalog książek)

Drogi Czytelniku!
Jeżeli chcesz ocenić tę książkę, zajrzyj pod adres
http://helion.pl/user/opinie?cshta2
Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

ISBN: 978-83-246-2910-7

Printed in Poland.

Spis treści

Rozdział 6.
Remoting

Technologia Remoting pozwala na komunikację pomiędzy aplikacjami środowiska .NET niezależnie od tego, czy znajdują się one na tym samym komputerze, czy w sieci. Technologia ta jest składową platformy .NET. Jej odpowiednikiem w języku Java jest technologia RMI (ang. *Remote Method Invocation*). Innym przykładem może być rozwiązanie CORBA (ang. *Common Object Request Broker Architecture*), które zapewnia komunikację między obiektami znajdującymi się w różnych systemach komputerowych. Przodkiem Remotingu była technologia DCOM (ang. *Distributed Component Object Model*), która obecnie nie jest już rozwijana. Dobra definicja .NET Remoting znajduje się na stronach MSDN:

> „Microsoft® .NET Remoting zapewnia bogatą i rozszerzalną infrastrukturę komunikacyjną dla obiektów znajdujących się w różnych domenach aplikacji AppDomains, w różnych procesach i na różnych komputerach. .NET Remoting oferuje silny, a jednocześnie prosty model programistyczny i wsparcie w czasie wykonania, co sprawia, że infrastruktura komunikacyjna jest niewidoczna".

Myślę, że bardzo dobrze oddaje ona ideę tej składowej platformy .NET. Zwykle budowa rozproszonej aplikacji, opartej na technologii Remoting, składa się z trzech faz:

- Stworzenie obiektu odpowiedzialnego za funkcjonalność rozproszonej aplikacji.
- Utworzenie serwera, który będzie odpowiadał za rejestrację, udostępnienie oraz zarządzanie zdalnym obiektem.
- Napisanie aplikacji klienckich korzystających z udostępnionej przez serwer usługi.

Wszystkie projekty zebrane w tym rozdziale będą budowane w trzech opisanych wyżej krokach. Ogromną zaletą technologii Remoting jest to, że nie musimy dbać o połączenia i wymianę danych, tak jak miało to miejsce w projektach z rozdziału 5. Tutaj dzielony obiekt jest automatycznie udostępniany przez serwer. Do dyspozycji mamy trzy rodzaje połączeń: TCP, IPC i HTTP. Pierwsze dwa połączenia przesyłają dane w formacie binarnym. Połączenie HTTP przesyła informacje w formacie tekstowym. Udostępniany obiekt może być przekazywany przez referencję bądź wartość. W przypadku referencji obiekt zdalny istnieje wyłącznie na serwerze. Podczas komunikowania się z klientem otrzymuje on jedynie referencje do obiektu. Aby klasa zdalna przekazywała dane przez

referencję, musi dziedziczyć po klasie `MarshalByRefObject`. Klasa ta zapewnia niezbędne mechanizmy udostępniania. Jeżeli zdecydujemy się na przekazywanie przez wartość, wówczas klient będzie miał pełną wiedzę na temat obiektu zdalnego oraz będzie posiadał kopię przeznaczoną tylko dla niego. W definicji klasy obiektu zdalnego musimy wstawić wtedy atrybut `[Serializable]`. W naszych projektach będziemy używać przekazywania przez referencję. Z racji tego, że technologia Remoting jest częścią składową platformy .NET, może być używana jedynie przez aplikacje stworzone dla tej platformy. Jeżeli chcielibyśmy używać pewnych usług z poziomu różnych systemów komputerowych, wówczas lepszym rozwiązaniem wydają się być usługi sieciowe (ang. *web services*) opisane w rozdziale 8. Technologia .NET Remoting jest obecnie zastąpiona przez *Windows Communication Foundation* (rozdział 9.). Stanowi więc technologię, która powoli schodzi ze sceny. Nie zaleca się używania jej w nowych projektach. Projekty, które już są w użyciu, powinny migrować w stronę WCF. Jeśli to niemożliwe, powinny przynajmniej integrować się z WCF.

W .NET Remoting istnieją trzy podstawowe typy obiektów zdalnych, które pośredniczą w transakcjach pomiędzy klientem a serwerem:

- **SingleCall.** Obiekt, który obsługuje jedno żądanie. Nie przechowuje on informacji pomiędzy wywołaniami. Nie zapamiętuje więc stanu.
- **Singleton.** Zgodnie z wzorcem projektowym o nazwie Singleton jest to obiekt wspólny dla wielu klientów. Jedna instancja przechowująca stan dla wszystkich wywołań. Umożliwia jawne współdzielenie danych.
- **CAO (Client Activated Object).** Obiekt aktywowany na żądanie klienta, działający po stronie serwera. Do komunikacji wykorzystuje klasę pośredniczącą. Zapamiętuje stan dla danego klienta, nie dla wszystkich klientów.

Rozdział ten stanowi jedynie bardzo krótki wstęp, który ma za zadanie zaznajomić Czytelnika z podstawami technologii Remoting. Brak tu takich zagadnień, jak szyfrowanie danych, ukrywanie implementacji obiektu zdalnego przed klientami oraz zarządzanie czasem życia. Myślę jednak, że w opisie podstaw programowania sieciowego dla platformy .NET nie mogło zabraknąć choćby podstawowego opisu technologii Remoting. Istnieje jeszcze wiele projektów, które z niej korzystają. Warto ją więc poznać w celu zakonserwowania dotychczasowych rozwiązań i ich przeniesienia do WCF. Jeśli stoisz przed wyborem: .NET Remoting czy WCF, to należy wybrać WCF.

Projekt 69. Serwer HTTP

Napiszemy teraz serwer oparty na połączeniu HTTP. Jako pierwszą utworzymy klasę odpowiedzialną za usługę zdalną. Nasza usługa będzie polegała na wyświetlaniu odpowiedniego komunikatu w przypadku połączenia się z klientem. Będzie to bardzo prosty projekt, odpowiednik projektu „witaj świecie”. Jego zadaniem jest pokazanie, w jaki sposób należy budować projekty .NET Remoting. Nie będziemy skupiać się w tym miejscu na jakimś skomplikowanym projekcie.

1. Z menu *File* wybieramy opcję *New project*....
2. Zaznaczamy ikonę *Class Library*. W polu *Name* wpisujemy `RemotingExample`. Przypominam, że jest to nazwa przestrzeni nazw, w której znajdą się nasze przykładowe klasy. Do nazywania zmiennych będziemy używać języka angielskiego. Myślę, że jest to dobry nawyk. Warto już teraz przyzwyczajać się do anglojęzycznych nazw zmiennych w programie.
3. Piszemy prostą klasę `RemotingObject` z jedną publiczną metodą, która zwraca pewien tekst. Konstruktor bezparametrowy zostanie pusty. Kod klasy jest pokazany na listingu 6.1. Należy zwrócić uwagę na fakt, że nasza klasa ma dziedziczyć z klasy `MarshalByRefObject`. Dla aplikacji, które obsługują Remoting, oznacza to udostępnienie obiektu tej klasy nie tylko w ramach jednej domeny aplikacji.

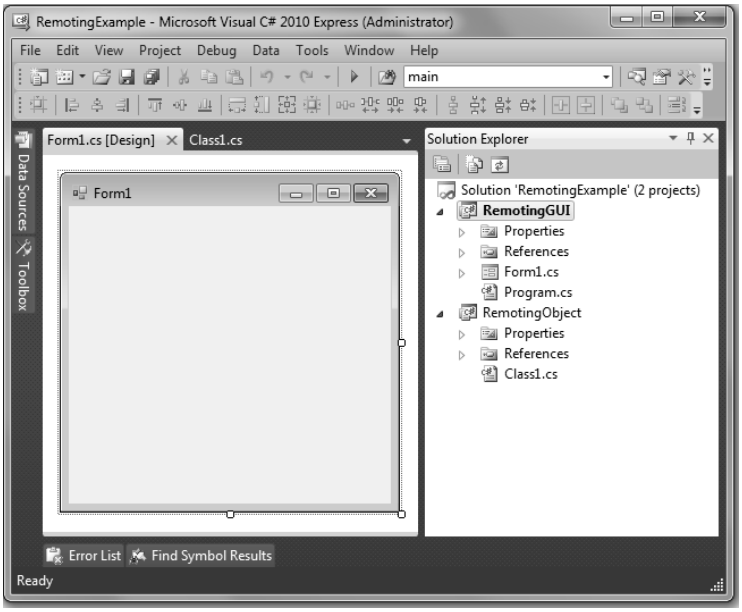
Listing 6.1. *Klasa odpowiedzialna za usługę*

```
using System;
using System.Text;

namespace RemotingExample
{
    public class RemotingObject: MarshalByRefObject
    {
        public RemotingObject() { }

        public string Test()
        {
            return ".NET Remoting test";
        }
    }
}
```

4. Zapisujemy teraz nasz projekt pod nazwą *RemotingObject* oraz kompilujemy go (klawisz *F6*). W wyniku kompilacji otrzymujemy bibliotekę *RemotingObject.dll*.

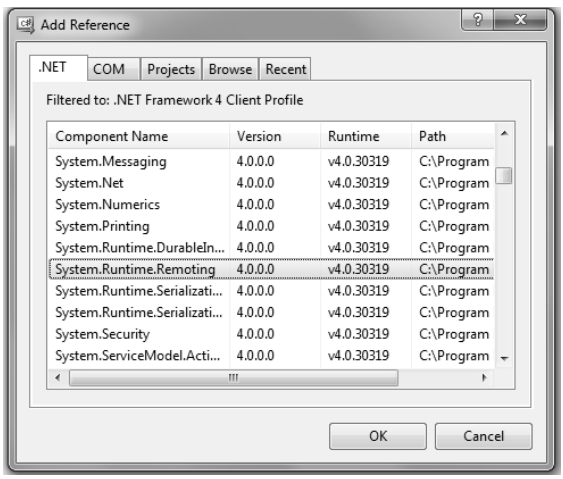
   Naszą klasę nazwaliśmy `RemotingObject`. Dziedziczy ona po klasie `MarshalByRefObject`, co zapewnia przesyłanie do przyłączonych klientów jedynie referencji obiektu `RemotingObject`. Stworzyliśmy pusty konstruktor oraz jedną publiczną, bezparametrową metodę `Test`, zwracającą tekst `.NET Remoting test`. Jak widać, przykład ten nie jest skomplikowany. Przejdźmy teraz do napisania aplikacji serwera, która będzie korzystała ze stworzonej przed chwilą biblioteki.
5. Do naszego rozwiązania dołączymy nowy projekt. Przypominam, że najszybciej można to zrobić, klikając prawym klawiszem w oknie *Solution Explorer* nazwę naszego rozwiązania (ang. *solution*). Z menu podręcznego wybieramy opcję *Add* oraz *New Project*.... Rozpoczynamy nowy projekt *Windows Forms Application*. W polu *Name* wpisujemy `RemotingGUI`. Dzięki temu poprzednia biblioteka i nasz projekt będą znajdowały się w tej samej przestrzeni nazw. Oczywiście nie jest to konieczne, ale wygodne.

6. Ustawiamy projekt `RemotingGUI` jako startowy. Aby to zrobić, klikamy prawym klawiszem nazwę projektu, a następnie wybieramy opcję *Set as StarUp Project*. Po tych czynnościach okno Visual Studio powinno przypominać to z rysunku 6.1.



Rysunek 6.1. *Dwa projekty w ramach jednego rozwiązania*

7. Do projektu dodajemy referencję do biblioteki *System.Runtime.Remoting.dll*, która nie jest standardowo zamieszczana w projektach typu *Windows Forms Application*. W tym celu z menu głównego wybieramy opcję *Project*, a następnie klikamy pozycję *Add Reference*.... Otworzy się okno z wyborem referencji (rysunek 6.2). Należy pamiętać, aby wybrać zakładkę *.NET*.



Rysunek 6.2. *Okno Add Reference*

8. Na zakładce *.NET* wyszukujemy i zaznaczamy komponent o nazwie *System.Runtime.Remoting*. Klikamy przycisk *OK*. W ten sposób referencja została dodana. Aby to sprawdzić, możemy w oknie *Solution Explorer* rozwinąć gałąź *References*. Powinien się tam znajdować wpis *System.Runtime.Remoting*.
9. Przełączmy się do widoku kodu projektu `RemotingGUI` (klawisz *F7*). W sekcji odpowiedzialnej za użyte w projekcie przestrzenie nazw dodajemy nowe wpisy:

```
using System.Runtime.Remoting;
using System.Runtime.Remoting.Channels;
using System.Runtime.Remoting.Channels.Http;
```

10. W oknie widoku projektu na formę wrzucamy kontrolkę `button1`. Jej własność `Text` ustawiamy na *Start*.
11. Do projektu dodajemy kontrolkę `textBox1`. Własność `ReadOnly` zmieniamy na `true`.
12. Musimy dodać jeszcze referencje do stworzonej wcześniej biblioteki *RemotingObject.dll*. Robimy to podobnie jak w przypadku dodawania referencji *System.Runtime.Remoting*. Po otwarciu okna *Add Reference* zmieniamy zakładkę na *Projects* i wybieramy projekt *RemotingObject*. Klikamy przycisk *OK*. Biblioteka została dodana do referencji naszego projektu.
13. Serwer powinien wystartować w momencie kliknięcia przycisku *Start*. Wobec tego piszemy metodę zdarzeniową `Click` (listing 6.2).

Listing 6.2. *Metoda uruchamiająca serwer*

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Linq;
using System.Text;
using System.Windows.Forms;
using System.Runtime.Remoting;
using System.Runtime.Remoting.Channels;
using System.Runtime.Remoting.Channels.Http;


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

        private void button1_Click(object sender, EventArgs e)
        {
            try
            {
                HttpChannel channel = new HttpChannel(25000);
                ChannelServices.RegisterChannel(channel, false);
```

```
RemotingConfiguration.RegisterWellKnownServiceType(typeof(RemotingObject),
"Remoting", WellKnownObjectMode.SingleCall);
                textBox1.Text = "Serwer oczekuje .. ";
            }
            catch (Exception ex)
            {
                MessageBox.Show(ex.Message, "Błąd");
            }

        }
    }
}
```

Dla naszej aplikacji rezerwujemy port **25000**, na którym będzie ona nasłuchiwać wywołań opartych na protokole HTTP. Używając metody `RegisterChannel` klasy `ChannelServices`, rejestrujemy nasz kanał HTTP. Pierwszym argumentem jest kanał. Drugi argument odpowiada za zabezpieczenie (`true` — zabezpieczony, `false` — niezabezpieczony). Jeżeli spróbowalibyśmy jeszcze raz zarejestrować ten kanał, zostałby zgłoszony wyjątek informujący nas, że kanał już jest zarejestrowany. Dalej korzystamy z metody `RegisterWellKnownServiceType` klasy `RemotingConfiguration`. Metoda ta rejestruje nasz serwis w usłudze *Remoting*. Jej pierwszym argumentem jest typ klasy, która będzie dostępna jako usługa. Drugi argument to nazwa `Uri`, czyli unikalna nazwa łącząca usługę z obiektem. Celowo użyliśmy nazwy *Remoting*, aby pokazać, iż klient wcale nie musi znać nazwy klasy odpowiedzialnej za usługę. Wystarczy, że zna adres i port serwera oraz nazwę samej usługi. W ten sposób możemy również udostępniać wiele usług na jednym porcie. Usługi te muszą mieć inne `Uri`. Ostatni parametr mówi, w jaki sposób obiekt ma być tworzony po stronie serwera. Mamy do dyspozycji dwie opcje: `SingleCall` i `Singleton`. Jeżeli wybieramy pierwszą, wówczas każda przychodząca wiadomość jest obsługiwana przez nową instancję obiektu. W drugim przypadku wszystkie przychodzące wiadomości są obsługiwane przez tę samą instancję obiektu. Całość jest zamknięta w bloku ochronnym `try/catch`, ponieważ metody tu użyte mogą generować wyjątki. Serwer będzie działał, dopóki nie wyłączymy aplikacji.

14. Uruchamiamy teraz nasz projekt. Włączamy serwer. Musimy pamiętać o tym, aby ustawić odpowiednio firewall.

15. Sprawdzimy teraz, czy nasz serwer rzeczywiście działa. Otwieramy przeglądarkę internetową i jeżeli nie jesteśmy podłączeni do sieci, wpisujemy w niej adres `http://localhost:25000/Remoting?wsdl`. W przypadku gdy nasz komputer posiada adres IP, zamiast `localhost` wpisujemy ten adres. Powinna ukazać się strona XML z informacjami o naszym obiekcie *RemotingObject*. Język WSDL (ang. *Web Services Description Language*) opisuje usługi sieciowe. Odpowiedź w przypadku lokalnego hosta zawiera listing 6.3.

Listing 6.3. *Odpowiedź serwera*

```
<definitions name="RemotingObject"
targetNamespace="http://schemas.microsoft.com/clr/nsassem/RemotingExample/RemotingO
bject%2C%20Version%3D1.0.0.0%2C%20Culture%3Dneutral%2C%20PublicKeyToken%3Dnull">
```

```
<message name="RemotingObject.TestInput"></message>

<message name="RemotingObject.TestOutput"><part name="return"
type="xsd:string"/></message>

<portType name="RemotingObjectPortType"><operation name="Test"><input
name="TestRequest" message="tns:RemotingObject.TestInput"/><output
name="TestResponse"
message="tns:RemotingObject.TestOutput"/></operation></portType>

<binding name="RemotingObjectBinding"
type="tns:RemotingObjectPortType"><soap:binding style="rpc"
transport="http://schemas.xmlsoap.org/soap/http"/><suds:class
type="ns0:RemotingObject" rootType="MarshalByRefObject">
        </suds:class><operation name="Test"><soap:operation
soapAction="http://schemas.microsoft.com/clr/nsassem/RemotingExample.RemotingObject
/RemotingObject#Test"/><suds:method attributes="public"/><input
name="TestRequest"><soap:body use="encoded"
encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
namespace="http://schemas.microsoft.com/clr/nsassem/RemotingExample.RemotingObject/
RemotingObject"/></input><output name="TestResponse">
<soap:body use="encoded" encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
namespace="http://schemas.microsoft.com/clr/nsassem/RemotingExample.RemotingObject/
RemotingObject"/></output></operation>
</binding>

<service name="RemotingObjectService"><port name="RemotingObjectPort"
binding="tns:RemotingObjectBinding"><soap:address
location="http://localhost:25000/Remoting"/></port></service></definitions>
```

W odpowiedzi opisane są wszystkie szczegóły naszego serwisu .NET Remoting. Na listingu 6.3 wyróżniono kilka najważniejszych.

Projekt 70. Klient HTTP

Nasz klient będzie miał za zadanie połączyć się z usługą (z serwerem), uruchomić metodę i wyświetlić jej wyniki.

1. Tworzymy nowy projekt *Windows Forms*. W polu *Name* wpisujemy `RemotingClient`.
2. Podobnie jak w poprzednim projekcie tutaj także dodajemy referencje do bibliotek *RemotingObject.dll* oraz *System.Runtime.Remoting.dll*. W tym celu można użyć zakładki *Recent* z okna *Add Reference*, gdzie powinny się już znajdować wcześniej dodane referencje.
3. Zapisujemy projekt pod nazwą `RemotingClient`. Zmieniamy przestrzeń nazw projektu na `RemotingExample`. Można to zrobić np. za pomocą narzędzia Refactor.
4. W podglądzie kodu dodajemy odpowiednie przestrzenie nazw:

```
using System.Runtime.Remoting;
using System.Runtime.Remoting.Channels;
using System.Runtime.Remoting.Channels.Http;
```

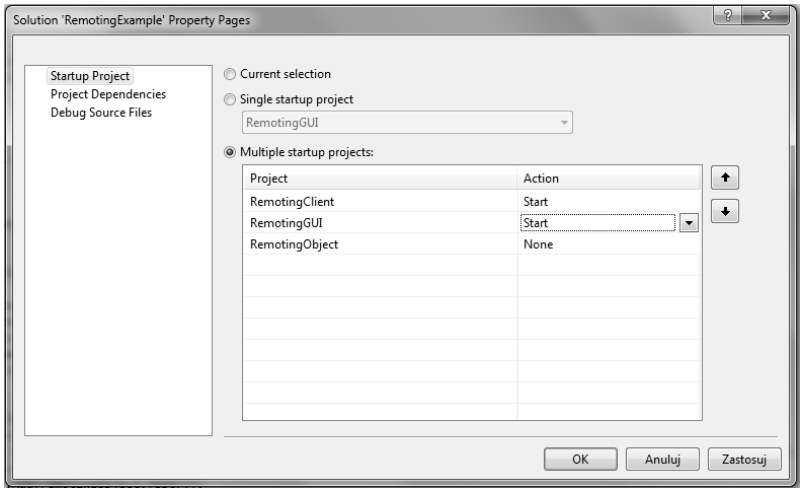
5. Na formę wrzucamy pole edycyjne `textBox1`. Tutaj będziemy wprowadzać adres serwera.
6. Do projektu dodajemy kontrolkę `numericUpDown1`— będzie ona przechowywała numer portu, na którym nasłuchuje serwer.
7. Formę uzupełniamy o kontrolkę `listBox1`. Będziemy tu wypisywać wszystkie komunikaty.
8. Na formę wrzucamy przycisk `button1`. Własność `Text` zmieniamy na *Połącz*.
9. Dla domyślnej metody zdarzeniowej kontrolki `button1` piszemy kod z listingu 6.4.

Listing 6.4. *Uzyskanie połączenia z serwerem*

```
private void button1_Click(object sender, EventArgs e)
        {
            string address = textBox1.Text;
            if (!address.StartsWith("http://"))
                address = "http://" + textBox1.Text;
            int port = (int)numericUpDown1.Value;
            HttpClientChannel channel = null;
            try
            {
                channel = new HttpClientChannel();
                ChannelServices.RegisterChannel(channel, false);
                RemotingObject remotingObject = (RemotingObject)Activator.GetObject
                ➥(typeof(RemotingObject), address + ":" + port.ToString() +
                ➥"/Remoting");
                listBox1.Items.Add("Połączenie: " + address + ":" + port.ToString()
                ➥+ "/Remoting");
                listBox1.Items.Add(remotingObject.Test());
                ChannelServices.UnregisterChannel(channel);
                listBox1.Items.Add("Połączenie zakończone");
            }
            catch (Exception ex)
            {
                MessageBox.Show(ex.Message, "Błąd");
                listBox1.Items.Add("Połączenie przerwane");
                ChannelServices.UnregisterChannel(channel);
            }
        }
```

W tej metodzie tworzymy kanał typu `HttpClientChannel`. Następnie za pomocą metody `GetObject` klasy `Activator` tworzony jest obiekt typu `RemotingObject`. Pierwszym argumentem tej metody jest typ obiektu. Drugi argument to opisany wcześniej obiekt `Uri`. Jeżeli aktywacja się powiedzie, wówczas mamy dostęp do metody `Test()`. Jej wywołanie zostało wyróżnione na listingu 6.4. Należy pamiętać, że w tym przypadku mamy do czynienia z obiektem zdalnym, kontrolowanym przez serwer. Po wywołaniu metody `Test()` zamykamy kanał metodą `UnregisterChannel`. Całość musimy umieścić w bloku ochronnym, ponieważ w trakcie działania aplikacji mogą być zgłaszane wyjątki.

Pora teraz przetestować nasze rozwiązanie. W jego skład wchodzą dwa projekty, które powinny być uruchomione jednocześnie. W Visual Studio poradzimy sobie z tym bez problemu. W oknie *Solution Explorer* klikamy prawym klawiszem nazwę rozwiązania. Otworzy się podręczne menu, z którego wybieramy opcję *Properties*. Możemy również użyć skrótu klawiszowego *Alt+Enter*. Wybieramy teraz opcję *Startup Project*. W kolejnym kroku zaznaczamy opcję *Multiple startup projects*. Możemy wskazać, które projekty mają wystartować w momencie kompilacji i uruchomienia rozwiązania. Wybieramy projekty *RemotingClient* oraz *RemotingGUI* (rysunek 6.3). Po naciśnięciu klawisza *F5* uruchomią się wskazane projekty. W projekcie serwera klikamy przycisk *Start*. W polu adresu na formie klienta wpisujemy `localhost` oraz port `25000`. Teraz klikamy *Połącz*. Klient powinien nawiązać połączenie i wyświetlić wynik działania metody `Test()`.



Rysunek 6.3. *Okno właściwości rozwiązania*

Projekt 71. Serwer TCP

Protokół TCP również może służyć do udostępniania usług za pomocą technologii .NET Remoting. W ramach obiektu zdalnego napiszemy prostą metodę dodającą dwie liczby i zwracającą wynik. Tak jak w poprzednich projektach z tego rozdziału tutaj również nie zależy nam na rozbudowanych przykładach. W tym i kolejnym projekcie skupimy się na zasadzie tworzenia rozwiązania opartego na protokole TCP.

1. Rozpoczynamy nowy projekt *Class Library*. W polu *Name* wpisujemy `RemotingTCP`.
2. Naszą klasę nazwiemy, trochę na wyrost, `RemotingMath`. Dodamy bezparametrowy konstruktor oraz publiczną metodę `Add`, która zwróci wynik dodawania dwóch liczb przekazanych jako argumenty. Metodę przeciążymy dla typów `double` i `int`. Oczywiście to nie koniec. Powinno się jeszcze bardziej przeciążyć metodę `Add`. Nie będziemy się tym zajmować teraz, ponieważ co innego jest celem tego ćwiczenia. Cały kod tej klasy zawiera listing 6.5. Klasa ta nie różni się znacząco od klasy napisanej w projekcie 69. Tutaj także dziedziczymy po klasie `MarshalByRefObject`.

Listing 6.5. *Klasa Matematyka*

```
using System;
using System.Collections.Generic;
using System.Text;

namespace RemotingTCP
{
    public class RemotingMath: MarshalByRefObject
    {
        public RemotingMath() { }

        public double Add(double a, double b)
        {
            return a + b;
        }
        public int Add(int a, int b)
        {
            return a + b;
        }
    }
}
```

3. Zapisujemy projekt pod nazwą `RemotingMath`, a rozwiązanie pod nazwą `RemotingTCP`, następnie całość kompilujemy. W ten sposób otrzymamy bibliotekę *RemotingMath.dll*.
4. Tworzymy nowy projekt *Windows Forms* o nazwie *RemotingServer* i umieszczamy go w tym samym rozwiązaniu. Sposób został opisany w poprzednich projektach z tego rozdziału (projekty 69. i 70.).
5. Do projektu dodajemy referencje do *System.Runtime.Remoting.dll* oraz do projektu *RemotingMath* (patrz projekt 70.).
6. Przenosimy się do widoku kodu (klawisz *F7*). W bloku definiującym przestrzenie nazw dodajemy wpis:

```
using System.Runtime.Remoting;
using System.Runtime.Remoting.Channels;
using System.Runtime.Remoting.Channels.Tcp;
```

7. Na formę wrzucamy kontrolkę `textBox1`. Jej własność `ReadOnly` zmieniamy na `true`.
8. Do projektu dodajemy przycisk `button1`. Tworzymy dla niego metodę zdarzeniową `Click` i wpisujemy kod z listingu 6.6. Komentarz z projektu 69. odnosi się również do tego listingu. Jedyną różnicą jest użycie obiektu klasy `TcpServerChannel`.

Listing 6.6. *Inicjacja serwera TCP*

```
private void button1_Click(object sender, EventArgs e)
        {
            try
            {
                TcpServerChannel channel = new TcpServerChannel(20000);
```

```
                ChannelServices.RegisterChannel(channel, false);

RemotingConfiguration.RegisterWellKnownServiceType(typeof(RemotingMath), "Math",
WellKnownObjectMode.SingleCall);
                textBox1.Text = "Serwer oczekuje na połączenia ...";
            }
            catch (Exception ex)
            {
                MessageBox.Show(ex.Message, "Błąd");
            }
        }
```

Projekt 72. Klient TCP

Idea klienta .NET Remoting została już opisana w projekcie 70. Ten projekt będzie nieco bardziej rozbudowany. Rozważymy różne przypadki użycia tej małej aplikacji. Zabieramy się więc od razu do kodowania:

1. Tworzymy nowy projekt *Windows Forms*. W polu *Name* wpisujemy `RemotingClient`. Jak poprzednio projekt ten powinien być dodany do tego samego rozwiązania. Ułatwi nam to zarządzanie projektami i testowanie ich.
2. Dodajemy referencję do *System.Runtime.Remoting.dll* oraz *RemotingMath* (patrz projekt 69.).
3. Projekt uzupełniamy o następujące definicje przestrzeni nazw:

```
using System.Runtime.Remoting;
using System.Runtime.Remoting.Channels;
using System.Runtime.Remoting.Channels.Tcp;
```

4. Na formę wrzucamy kontrolki `textBox1`, `textBox2` i `textBox3` (rysunek 6.4).

Rysunek 6.4. *Widok formy klienta .Net Remoting TCP*

5. Własność `ReadOnly` kontrolki `textBox3` ustawiamy na `true`.
6. Do projektu dodajemy przycisk `button1`. Własność `Text` zmieniamy na =.
7. Dodajemy dwa prywatne pola klasy formy (listing 6.7).

Listing 6.7. *Klient TCP. Pola prywatne*

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Text;
```

```
using System.Windows.Forms;
using System.Runtime.Remoting;
using System.Runtime.Remoting.Channels;
using System.Runtime.Remoting.Channels.Tcp;


namespace RemotingTCP
{
    public partial class Form1 : Form
    {
        private TcpClientChannel channel;
        private RemotingMath remotingMath;

        public Form1()
        {
            InitializeComponent();
        }
}
```

8. Tworzymy metodę zdarzeniową `Load` formy. Zainicjalizujemy w niej obiekty, których dalej użyjemy w połączeniu .NET Remoting (listing 6.8). Należy zwrócić uwagę, że inicjalizacja obiektów nie oznacza nawiązania połączenia. Innymi słowy, nawet jeśli serwer nie jest włączony, metoda `Load` nie spowoduje błędów.

Listing 6.8. *Klient TCP. Metoda inicjalizująca połączenie .NET Remoting*

```
private void Form1_Load(object sender, EventArgs e)
        {
            channel = new TcpClientChannel();
            ChannelServices.RegisterChannel(channel, false);
            WellKnownClientTypeEntry config = new WellKnownClientTypeEntry(typeof
            ↪(RemotingMath), "tcp://localhost:20000/Math");
            RemotingConfiguration.RegisterWellKnownClientType(config);
            remotingMath = new RemotingMath();
        }
```

9. Tworzymy nową metodę zdarzeniową `Click` kontrolki `button1` i wpisujemy do niej kod z listingu 6.9. Metoda ta użyje zdalnego obiektu i wykona na nim metodę `Add`.

Listing 6.9. *Klient TCP*

```
private void button1_Click(object sender, EventArgs e)
        {
           if (textBox1.Text == null && textBox2.Text == null && textBox3.Text == null)
                return;

            double a = 0.0;
            double b = 0.0;

            if (!Double.TryParse(textBox1.Text, out a))
            {
                textBox3.Text = "Błąd!";
                return;
```

```
        }
        if (!Double.TryParse(textBox2.Text, out b))
        {
            textBox3.Text = "Błąd!";
            return;
        }


        try
        {
            textBox3.Text = (remotingMath.Add(a, b)).ToString();
        }
        catch (Exception ex)
        {
            MessageBox.Show(ex.Message, "Błąd");
            ChannelServices.UnregisterChannel(channel);
        }
    }
```

Jak to często bywa, kod osłaniający ewentualne złe użycie metody jest bardziej rozbudowany niż samo użycie metody. Ta aplikacja jest odpowiednikiem programu z listingu 6.4. Istnieją jednak pewne różnice. Aplikacja klienta używa metody `RegisterWellKnown ↪ClientType` klasy `RemotingConfiguration` do połączenia się z serwerem. Dalej tworzona jest instancja klasy `RemotingMath`. Pomimo że dzieje się to w klasie klienta, instancja ta nie jest w pełni lokalnym obiektem. Zarządzana jest przez aplikację serwera. Będzie to dobrze widoczne w projektach 75 – 77. W ten sposób pokazaliśmy dwa rozwiązania, które możemy zastosować do aktywacji dostępu do zdalnego obiektu po stronie klienta.

Projekt 73. Serwer TCP. Plik konfiguracyjny

W poprzednich projektach aplikacje serwerowe były konfigurowane w kodzie programu. Każdorazowa zmiana nazwy udziału, portu lub innego parametru wiąże się z ponownym skompilowaniem projektu. Zamiast konfigurować serwer w kodzie, możemy skonfigurować go za pomocą pliku konfigurującego. Jest on standardu XML. Takie rozwiązanie jest o wiele bardziej elastyczne.

1. Tworzymy nowy projekt *Windows Forms*.
2. Do projektu dodajemy referencje *System.Runtime.Remoting.dll* i *RemotingMath.dll*.
3. Dodamy teraz do projektu plik XML. Z menu *Project* wybieramy opcję *Add New Item…*.
4. Zaznaczamy ikonę *XML File* i w polu *Name* wpisujemy `config.xml`. Klikamy *Add*.
5. W otwartym pliku XML wpisujemy kod z listingu 6.10.

Listing 6.10. *Plik konfiguracyjny serwera*

```
<configuration>
   <system.runtime.remoting>
      <application>
         <service>
            <wellknown mode="SingleCall" objectUri="Math"
      type="RemotingTCP.RemotingMath, RemotingMath"/>
            <activated type=" RemotingTCP.RemotingMath, RemotingMath " />
         </service>
         <channels>
            <channel ref="tcp" port="20000"/>
         </channels>
      </application>
   </system.runtime.remoting>
</configuration>
```

Rozpoczynamy kluczem `<configuration>`. Kolejny klucz to definicja przestrzeni nazw. W kluczu `<service>` definiujemy parametry serwisu. W znaczniku `<wellknown>` podajemy tryb inicjalizacji obiektu (`SingleCall`), `Uri` oraz przestrzeń nazw i nazwę klasy obiektu. Znacznik `<activated>` zapewnia możliwość tworzenia przez klienta lokalnej kopii obiektu. Na końcu zdefiniowaliśmy typ transmisji oraz numer portu, na którym serwer będzie prowadził nasłuch.

6. Teraz przechodzimy do kodu *Form1.cs*. Jak zwykle dodajemy trzy przestrzenie nazw:

```
using System.Runtime.Remoting;
using System.Runtime.Remoting.Channels;
using System.Runtime.Remoting.Channels.Tcp;
```

7. Do projektu dodajemy pole edycyjne `textBox1`.

8. Na formę wrzucamy przycisk `button1`. Dla jego domyślnej metody zdarzeniowej piszemy kod z listingu 6.11.

Listing 6.11. *Inicjacja serwera na podstawie pliku konfiguracyjnego*

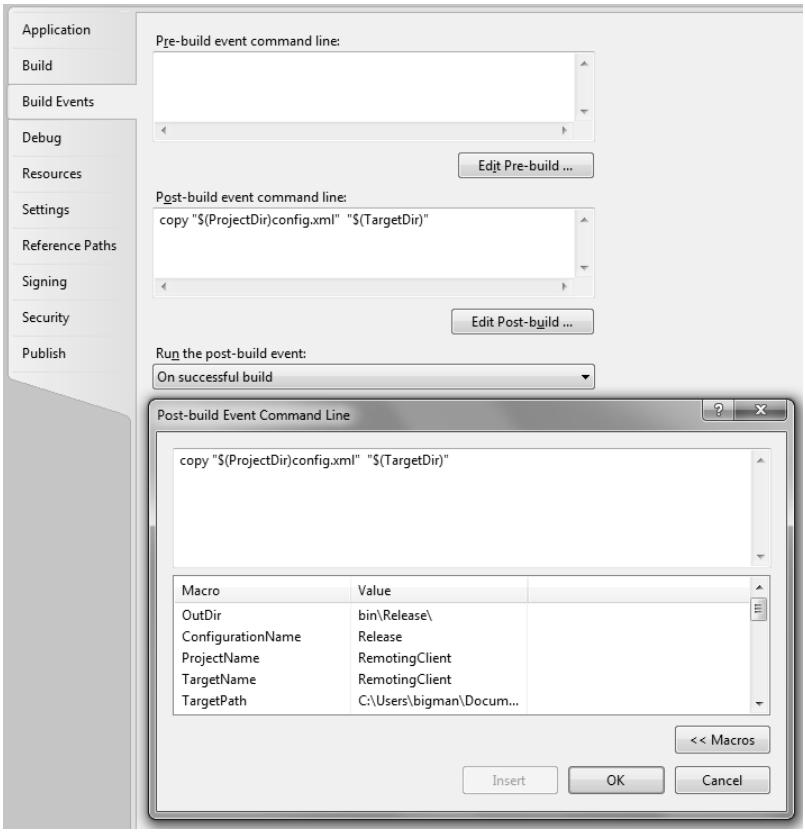
```
private void button1_Click(object sender, EventArgs e)
        {
            try
            {
                RemotingConfiguration.Configure("config.xml", false);
                textBox1.Text = "Serwer nasłuchuje ...";
            }
            catch (Exception ex)
            {
                MessageBox.Show(ex.Message);
            }
        }
```

Inicjacja serwera polega jedynie na użyciu metody `Configure` klasy `RemotingConfiguration`, za pomocą której wskazujemy plik konfiguracyjny. Jeżeli do naszego serwisu dodaliśmy plik *config.xml*, wówczas standardowo będzie się on znajdował w katalogu projektu. W momencie uruchomienia

serwera jesteśmy przenoszeni do katalogu *katalog_projektu/bin/Debug*. Aby plik *config.xml* był kopiowany do katalogu wynikowego aplikacji, musimy zmienić konfigurację budowania aplikacji. Otwieramy okno własności (*Properties*) projektu. Odnajdujemy zakładkę *Build Events*. W miejscu *Post-build event command line* wpisujemy następujące polecenie:

```
copy "$(ProjectDir)config.xml"  "$(TargetDir)"
```

Spowoduje to skopiowanie pliku *config.xml* do katalogu wynikowego aplikacji. Posłużyliśmy się w tym miejscu makrodefinicjami. Warto sprawdzić opcję *Edit Post-build*.... Jest to edytor, który zawiera wszystkie makrodefinicje dostępne w Visual Studio (rysunek 6.5). Napisaną aplikację możemy teraz przetestować.



Rysunek 6.5. *Okno własności projektu z otwartym edytorem Post-build event*

9. Zapisujemy i uruchamiamy serwer.
10. W katalogu projektu 72. odnajdujemy plik wykonywalny klienta TCP. Uruchamiamy tę aplikację. W polach edycyjnych wpisujemy dowolne liczby i klikamy przycisk =. Połączenie z nowo utworzonym serwerem powinno zostać ustanowione i odpowiednie działanie powinno być wykonane.

Projekt 74. Klient TCP. Plik konfiguracyjny

Klient również może zostać skonfigurowany za pomocą pliku XML zawierającego odpowiednie wpisy.

1. Tworzymy nowy projekt *Windows Forms*. W polu *Name* wpisujemy, tak jak poprzednio, `RemotingTCP`. Dzięki temu będziemy się znajdować w tej samej przestrzeni nazw co serwer. Nazwę projektu możemy zmienić potem w oknie *Solution Explorer*.
2. Interfejs naszej aplikacji będzie identyczny z interfejsem klienta TCP z projektu 71.
3. Dodajemy również te same referencje (*RemotingMath.dll* i *System.Runtime.Remoting.dll*).
4. Projekt uzupełniamy o definicje następujących przestrzeni nazw:

```
using System.Runtime.Remoting;
using System.Runtime.Remoting.Channels;
using System.Runtime.Remoting.Channels.Tcp;
```

5. Do projektu dodajemy plik XML (patrz punkt 3. w projekcie 73.). Nazwiemy go *config.xml*.
6. W utworzonym pliku wpisujemy kod z listingu 6.12.

Listing 6.12. *Plik konfiguracyjny klienta TCP*

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
<system.runtime.remoting>
   <application>
      <channels>
         <channel ref="tcp">
            <clientProviders>
               <formatter ref="binary"/>
            </clientProviders>
         </channel>
      </channels>
      <client>
         <wellknown  url="tcp://localhost:20000/Math"
type="RemotingTCP.RemotingMath,  RemotingMath "/>
      </client>
   </application>
</system.runtime.remoting>
</configuration>
```

W pliku tym użyliśmy tych samych tagów co w projekcie 73. Znacznik `<formater>` definiuje format przesyłania danych. W tym przypadku dane zostają przesyłane binarnie.

Projekt 75. Czat. Klasa serwera

Program, który teraz napiszemy, będzie odpowiadał projektowi *czat* z rozdziału 5. Różnica będzie polegać na zastosowaniu technologii Remoting. W projekcie 72. przekonywałem, że obiekt stworzony na podstawie klasy `RemotingMath` nie jest obiektem wyłącznie lokalnym, choć samo stworzenie go za pomocą operatora `new` wskazywałoby na to, iż jest to obiekt lokalny, kontrolowany przez aplikację klienta. W tym projekcie i dwóch kolejnych będziemy mieli szansę sprawdzić, że tak utworzony obiekt jest dzielony przez wszystkich klientów, a jego stan jest kontrolowany przez serwer.

1. Tworzymy nowy projekt *Class Library* i nazywamy go `ChatRemoting`.
2. Stworzoną klasę nazywamy `Chat`. Jako klasę bazową dla nowej klasy wybierzemy `MarshalByRefObject` (listing 6.13). Do klasy dodamy jeszcze bezparametrowy konstruktor i przestrzeń nazw `System.Collections`.

Listing 6.13. *Szablon klasy Czat*

```
using System;
using System.Collections.Generic;
using System.Collections;
using System.Text;

namespace ChatRemoting
{
    public class Chat : MarshalByRefObject
    {
        private ArrayList usersList = new ArrayList();
        private String talk = String.Empty;

        public Chat() { }
    }
}
```

Jeżeli zlikwidowalibyśmy dziedziczenie po klasie `MarshalByRefObject`, to każdy klient na czas sesji posiadałby własną, niezależną kopię obiektu. Kopia ta nie byłaby uzupełniana o nowe wiadomości bądź nowych klientów. Dziedziczenie z klasy bazowej `MarshalByRefObject` umożliwia współdzielenie obiektu klasy pochodnej `Chat`.

3. W klasie `Chat` umieszczamy dwa prywatne pola: `usersList` i `talk` (listing 6.13). Pierwsze z nich będzie przechowywało listę aktualnie przyłączonych użytkowników. Zastosowaliśmy tutaj dynamiczną tablicę klasy `ArrayList`. Pole `talk` będzie przechowywało całą rozmowę, począwszy od startu serwera. Pola inicjalizujemy w miejscu definicji.
4. Do klasy `Chat` dodajemy metodę umożliwiającą dodawanie nowych użytkowników (listing 6.14). Zabezpieczymy się w niej przed dodawaniem użytkowników o tych samych pseudonimach. Ponieważ pole `usersList` jest de facto dzielone pomiędzy aplikacje klienckie wykorzystujące obiekt klasy `Chat`, dodawanie umieszczamy w sekcji krytycznej. W przypadku nieudanej próby dodania użytkownika metoda zwróci `false`.

Listing 6.14. *Dodawanie nowych użytkowników serwisu*

```
public bool AddUser(string userName)
        {
            if (usersList.Contains(userName))
                return false;

            if (userName != null && userName.Trim() != null)
            {
                lock (usersList)
                {
                    usersList.Add(userName);
                }
                return true;
            }
            return false;
        }
```

5. Jeżeli użytkownik się odłączy, wówczas powinien zostać usunięty z listy `usersList`. Odpowiedni kod realizujący to zadanie został umieszczony na listingu 6.15.

Listing 6.15. *Usunięcie użytkownika z listy użytkowników*

```
public void RemoveUser(string userName)
        {
            lock (usersList)
            {
                usersList.Remove(userName);
            }
        }
```

6. Do naszej klasy dodajemy metodę, która będzie umieszczała w polu `rozmowa` nową wiadomość wysłaną przez klienta (listing 6.16). Metoda ta sprawdza również, czy w podanym ciągu nie znajdują się znaki formatujące tabulacji i końca linii. Na potrzeby tego przykładu jest to wystarczające, ale powinno się zwiększyć kontrolę wprowadzanych znaków.

Listing 6.16. *Dodawanie nowej wiadomości*

```
public void AddMessage(string newMessage)
        {
            if (newMessage != null && newMessage.Trim() != null)
            {
                newMessage.Replace("\n", "");
                newMessage.Replace("\t", " ");
                lock (talk)
                {
                    talk += newMessage + "\n";
                }
            }
        }
```

7. Aby uzyskać dostęp do prywatnych pól klasy `Czat`, tworzymy własności tylko do odczytu (listing 6.17). Pola te nie mogą być modyfikowane bezpośrednio z zewnątrz klasy.

Listing 6.17. *Własności wyświetlające listę użytkowników i rozmowę*

```
public ArrayList UsersList
        {
            get { return usersList; }
        }

        public string Talk
        {
            get { return talk; }
        }
```

8. Projekt zapisujemy (np. pod nazwą *ChatRemoting*) i kompilujemy.

W ten sposób stworzyliśmy klasę, której następnie użyjemy do połączenia z klientami. Ponieważ jednocześnie kilku użytkowników usługi mogłoby chcieć uzyskać dostęp do pól klasy poprzez metody `AddUser`, `RemoveUser` i `AddMessage`, zastosowaliśmy metodę `lock`. Blokuje ona określone zasoby, tak aby metody z innych wątków programu nie miały do nich dostępu tak długo, jak długo aktualny wątek musi korzystać z tych zasobów. Jest to sposób często używany w synchronizowaniu wątków.

Projekt 76. Czat. Serwer

Aplikacja serwera będzie niezwykle prosta. Ograniczymy się w niej tylko do uruchomienia usługi związanej ze stworzoną wcześniej biblioteką *ChatRemoting.dll*. Ponieważ będziemy przesyłać wiadomości tekstowe, najwygodniejszym rozwiązaniem jest zastosowanie połączenia HTTP.

1. Rozpoczynamy nowy projekt *Windows Forms*. Dodajemy go do już istniejącego rozwiązania. W polu *Name* wpisujemy `ChatServer`. Niech przestrzeń nazw będzie ta sama co w projekcie 75., czyli `ChatRemoting`. Najszybciej można zamienić nazwę przestrzeni nazw, używając narzędzia *Refactor*.
2. Do projektu dodajemy referencję *System.Runtime.Remoting.dll* oraz referencję do projektu 75.
3. Tradycyjnie dodajemy następujące przestrzenie nazw:

```
using System.Runtime.Remoting;
using System.Runtime.Remoting.Channels;
using System.Runtime.Remoting.Channels.Http;
```

4. Na formę wrzucamy przycisk `button1`. Dla jego domyślnej metody zdarzeniowej piszemy kod z listingu 6.18.

Listing 6.18. *Główna metoda serwera*

```
private void button1_Click(object sender, EventArgs e)
        {
            try
            {
                HttpChannel channel = new HttpChannel(1978);
                ChannelServices.RegisterChannel(channel, false);
                RemotingConfiguration.RegisterWellKnownServiceType(typeof(Chat),
                ➥"Chat", WellKnownObjectMode.Singleton);
            }
            catch (Exception ex)
            {
                MessageBox.Show(ex.Message);
            }

        }
```

Połączenie będzie nawiązywane w trybie `Singleton`. Oznacza to, że każdy z klientów będzie miał dostęp do tego samego (jednego) obiektu. Zmiany wprowadzone przez jednego klienta będą widoczne z poziomu każdego innego (nawet nowo przyłączonego). W naszym przypadku jest to naturalny wybór, ponieważ chcemy, aby wszystkie aplikacje klienckie mogły ze sobą rozmawiać. Stąd również w metodach modyfikujących klasę `ChatRemoting` znajdują się sekcje krytyczne.

Projekt 77. Czat. Klient

Pora na aplikację klienta. Aplikacja ta będzie nieco bardziej skomplikowana od poprzedniej, choć jej napisanie nie przysporzy nam większych kłopotów.

1. Tworzymy nowy projekt *Windows Forms*. W polu *Name* wpisujemy `ChatClient`. Jak poprzednio musimy zadbać, aby klasa znajdowała się w tej samej przestrzeni nazw co pozostałe klasy projektu. Jest to oczekiwana własność, która uchroni nas przed dodawaniem w sekcji `using` referencji do nowych przestrzeni nazw.
2. Podobnie jak w projekcie 76. tutaj także dodajemy referencję do biblioteki *System.Runtime.Remoting.dll* oraz do projektu *CzatRemoting*.
3. Do projektu dodajemy nowe przestrzenie nazw:

```
using System.Collections;
using System.Runtime.Remoting;
using System.Runtime.Remoting.Channels;
using System.Runtime.Remoting.Channels.Http;
```

4. Klasę `Form1` uzupełniamy o nowe pola:

```
private Chat chat;
private HttpChannel channel;
private bool isConnected = false;
```

5. Zaprojektujemy teraz interfejs aplikacji. Na formę wrzucamy pole edycyjne `textBox1`. Tutaj użytkownik będzie mógł wprowadzać wiadomość do wysłania.
6. Do projektu dodajemy pole edycyjne `textBox2`, które będzie przechowywało nasze imię.
7. Formę uzupełniamy o kontrolkę `textBox3`. W niej będzie widoczna rozmowa. Własność `ReadOnly` ustawiamy na `true`. Własność `Multiline` ustawiamy na `true`.
8. Na formie umieszczamy listę `listBox1`, która będzie zawierała aktualnie podłączonych użytkowników.
9. Na formę wrzucamy przycisk `button1`. Jego własność `Text` zmieniamy na *Połącz*.
10. Dodajemy kontrolkę `timer1` klasy `Timer`. Jest to „stoper" odmierzający czas w milisekundach. Własność `Interval` ustawiamy na `1000`. Oznacza to, że w odstępach czasu równych 1000 ms, czyli 1 s, będzie następowało zdarzenie `Tick`. Jest to wygodne narzędzie do oprogramowania periodycznie powtarzających się funkcji aplikacji. Jego użycie zostanie przedyskutowane niżej.
11. Tworzymy metodę zdarzeniową `Click` kontrolki `button1` i umieszczamy w niej kod z listingu 6.19.

Listing 6.19. *Połączenie z serwerem*

```
private void button1_Click(object sender, EventArgs e)
        {
            if (isConnected)
            {
                MessageBox.Show("Już podłączony", "Błąd");
                return;
            }
            try
            {
                channel = new HttpChannel();
                ChannelServices.RegisterChannel(channel, false);
                RemotingConfiguration.RegisterWellKnownClientType(typeof(Chat),
                ➥"http://localhost/Chat");
                chat = new Chat();
                bool status = chat.AddUser(textBox2.Text.Trim());
                if (status == false)
                {
                    MessageBox.Show("Użytkownik już istnieje", "Błąd");
                    ChannelServices.UnregisterChannel(channel);
                    timer1.Enabled = false;
                    isConnected = false;
                    button1.Enabled = true;
                    textBox2.ReadOnly = false;
                    return;
                }
                chat.AddMessage("Użytkownik [" + textBox2.Text + "] dołączył
                ➥do rozmowy \n\n");
                timer1.Enabled = true;
                isConnected = true;
                button1.Enabled = false;
                textBox2.ReadOnly = true;
            }
```

```
            catch (Exception ex)
            {
                MessageBox.Show(ex.Message, "Błąd");
                textBox3.Text = "Nie udało się nawiązać połączenia ...";
                ChannelServices.UnregisterChannel(channel);
                timer1.Enabled = false;
                isConnected = false;
                button1.Enabled = true;
                textBox2.ReadOnly = false;
                channel = null;
            }
        }
```

Kod, za pomocą którego łączymy się z serwerem, jest prawie taki sam jak w projekcie 70. Jedyna różnica polega na tym, że po podłączeniu do serwera wysyłamy wiadomość, używając metody `AddUser`. W momencie przyłączenia się do serwera i utworzenia obiektu klasy `Chat` uzyskujemy dostęp do dzielonego obiektu. Jeżeli rozmowa będzie się już toczyć jakiś czas, a nowy klient dopiero się podłączy, wówczas dostanie on wszystkie wiadomości od momentu rozpoczęcia usługi (włączenia serwera).

12. Do projektu dodajemy przycisk `button2`. Jego kliknięcie spowoduje dodanie wiadomości do pola `talk`. Odpowiedni kod zawiera listing 6.20.

Listing 6.20. *Dodanie nowej wiadomości*

```
private void button2_Click(object sender, EventArgs e)
        {
            if (isConnected)
                if (textBox2.Text != null && textBox2.Text.Trim() != null)
                    chat.AddMessage("[" + textBox2.Text + "] " + textBox1.Text);
        }
```

13. Dla komponentu `timer1` tworzymy metodę zdarzeniową `Tick` (listing 6.21).

Listing 6.21. *Metoda pobierająca nowe wiadomości i nowych użytkowników*

```
private void timer1_Tick(object sender, EventArgs e)
        {
            ArrayList users = chat.UsersList;
            listBox1.Items.Clear();
            foreach (string user in users)
                listBox1.Items.Add(user);
            textBox3.Clear();
            textBox3.Text = chat.Talk;
        }
```

Jak sprawdzić, czy ktoś coś napisał lub czy ktoś nowy dołączył do rozmowy? Pomysł jest bardzo prosty. Co jedną sekundę odświeżamy zawartość pola edycyjnego `textBox3`, które przechowuje stan rozmowy oraz zawartość listy gości (kontrolka `listBox1`). Pomysł ten jest od razu najsłabszym punktem rozwiązania. Musimy mieć na uwadze, jakie wprowadza ograniczenia. Wielkość

pamięci rozmowy nie może być zbyt duża. Spowodowałoby to przeciążenie sieci. Dodatkowo po co od nowa pobierać rozmowę, którą już pobraliśmy? Proponuję, aby problemy te potraktować jako ciekawe ćwiczenie. Omówiony projekt jest jedynie początkiem. Należy go zmodyfikować i rozszerzyć.

14. Do projektu dodajemy jeszcze przycisk `button3`. Jego własność `Text` zmieniamy na *Rozłącz*.

15. Kod klasy uzupełniamy o metodę zdarzeniową `Click` kontrolki `button3` (listing 6.22).

Listing 6.22. *Rozłączenie użytkownika*

```
private void button3_Click(object sender, EventArgs e)
        {
            if (isConnected)
            {
                chat.AddMessage("Użytkownik [" + textBox2.Text + "] opuścił rozmowę");
                chat.RemoveUser(textBox2.Text);
                listBox1.Items.Clear();
                timer1.Enabled = false;
                ChannelServices.UnregisterChannel(channel);
                isConnected = false;
                button1.Enabled = true;
                textBox2.ReadOnly = false;
            }
        }
```

Aby rozłączyć użytkownika, należy usunąć go z listy rozmówców oraz użyć metody `UnregisterChannel` klasy `ChannelServices`, która zakończy połączenie z serwerem. Można również wysłać odpowiedni komunikat do wszystkich użytkowników usługi.

Rozdział ten, jak wspomniałem na początku, jest szybkim wstępem do technologii .NET Remoting, która… w zasadzie zanika. Jeśli nie musisz jej używać, po prostu jej nie używaj. Wsparcie dla niej może w jednej z kolejnych wersji platformy .NET zostać ograniczone. Niemniej jednak jest to ciekawa (moim zdaniem) technologia, która wprowadziła pewną warstwę abstrakcji pomiędzy gniazda sieciowe a wyższe warstwy aplikacji. Projekt już nie jest rozwijany, ale z pewnością przyczynił się do powstania nowych koncepcji w *Windows Communication Foundation*.

Skorowidz

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

R

S

Ś

T

U

V

W

X

Y

Z

Ź

Znaczna część współczesnego życia na stałe przeniosła się do Internetu. Nikt dziś nie wyobraża już sobie świata, w którym nie można sprawdzić pogody, zrobić zakupów czy szybko znaleźć potrzebnych informacji — właśnie w sieci. A skoro tak, wciąż wzrasta zapotrzebowanie na programistów mogących zaspokoić oczekiwania rzeszy klientów oraz dostarczyć im wygodnych, świetnie działających aplikacji sieciowych, skrojonych na potrzeby konkretnych użytkowników. To właśnie było źródłem niezwykłej popularności poprzedniej książki Sławomira Orłowskiego, w praktyczny sposób przekazującej wiedzę na temat sposobów tworzenia takich aplikacji i spełniania różnych warunków związanych ze specyfiką określonych projektów programistycznych.

C#. Tworzenie aplikacji sieciowych. Gotowe projekty to podręcznik przeznaczony dla osób z choćby pobieżną znajomością zasad programowania, które chcą rozwinąć i wykorzystać w praktyce swoje umiejętności. Autorzy tłumaczą, dlaczego wybrali język C#, a także wskazują możliwości i zalety platformy .NET oraz środowiska Visual Studio. Następnie skupiają się na projektach obejmujących wszystkie aspekty komunikacji internetowej, od wykorzystania różnych protokołów sieciowych, przez budowę interfejsu, wysyłanie e-maili, obsługę FTP, komunikatorów, czatów, aż po kontaktowanie się z bazami danych, tworzenie wielojęzycznych serwerów i udostępnianie usługi. Jeśli chcesz szybko i bez kłopotu podszkolić się w zakresie aplikacji sieciowych, by w krótkim czasie osiągnąć spektakularne efekty, nie znajdziesz nic lepszego!

- Język C# i platforma .NET
- Visual C# 2010 Express Edition. Opis środowiska
- Visual Web Developer 2010 Express Edition. Opis środowiska
- Programowanie sieciowe
- Aplikacje TCP i UDP
- Remoting
- ASP.NET i ADO.NET
- Web Services
- WCF — ponad transportem

Aplikacje sieciowe — wykorzystaj potencjał Internetu!

helion.pl
księgarnia internetowa

Nr katalogowy: 6822

Księgarnia internetowa
http://helion.pl

Zamówienia telefoniczne:
0 801 339900
0 601 339900

Helion

Sprawdź najnowsze promocje:
http://helion.pl/promocje
Książki najchętniej czytane:
http://helion.pl/bestsellery
Zamów informacje o nowościach:
http://helion.pl/nowosci

Helion SA
ul. Kościuszki 1c, 44-100 Gliwice
tel.: 32 230 98 63
e-mail: helion@helion.pl
http://helion.pl

Informatyka w najlepszym wydaniu

Cena: 49,00 zł

ISBN 978-83-246-2910-7

9 788324 629107