Programowanie gniazd w C# - część I

Jeśli masz zamiar pracować z gniazdami w przyszłości ,możesz być zainteresowany jak to zrobić korzystając z **technologii C#.** Celem tego artykułu jest pokazanie jak można oprogramować gniazda w C#. Artykuł ten zakłada twoją pewną znajomość programowania gniazd, chociaż nie musi to być na poziomie eksperta. Jest kilka stron programowania gniazd – jak strona klienta, strona serwera, blokowanie lub synchronizacja, nie – blokowanie lub asynchronizacja itd.

Pamiętając o tym wszystkim, postanowiłem podzielić to na dwie części. W części pierwszej zacznę od zablokowania gniazda strony klienta. Później w drugiej części pokażę jak stworzyć stronę serwera nie zablokowaną.

Programowanie sieciowe w Windows jest możliwe dzięki gniazdom. Gniazdo jest jak uchwyt dla pliku Oprogramowanie gniazda jest podobne do pliku IO w komunikacji szeregowej. Możesz użyć oprogramowania gniazd aby dwie aplikacje mogły komunikować się wzajemnie. Aplikacje są zazwyczaj na różnych komputerach ale mogą też być na tym samym komputerze. Przy rozmowie dwóch aplikacji na tym samym lub różnych komputerach używając gniazd, jedna aplikacja jest generalnie serwerem, który nasłuchuje nadchodzących żądań a druga aplikacja działa jako klient i tworzy połączenei do aplikacji serwera.

Aplikacja serwera może albo przyjąć albo odrzucić połączenie. Jeśli serwer zaakceptuje połączenie, może zacząć się dialog pomiędzy klientem a serwerem. W momencie kiedy klient wykona wszystko co miał do zrobienia, może zamknąć połączenie z serwerem. Połączenia są kosztowne w tym sensie, że serwery pozwalają na wystąpienie zakończenia połączenia. W czasie kiedy klient ma *aktywne połączenie* może wysyłać dane do serwera i/lub odbierać dane.

Cała złożoność zaczyna się tu. Kiedy jedna ze stron (klient lub serwer) wysyła dane do drugiej strony przypuszczalnie aby te dane odczytać. Ale skąd ta druga strona wie kiedy nadejdą dane. Są dwie opcje – albo aplikacja musi *zapytywać* o dane w regularnych odstępach czasu albo potrzebuje pewnego rodzaju mechanizmu, który umożliwiłby aplikacji uzyskanie zawiadomienia i aplikacja mogłaby odczytać dane w tym czasie. Cóż, jak by nie było Windows jest systemem opartym o zdarzenia i system powiadamiania wydaje się być oczywisty i najlepszym wyborem.

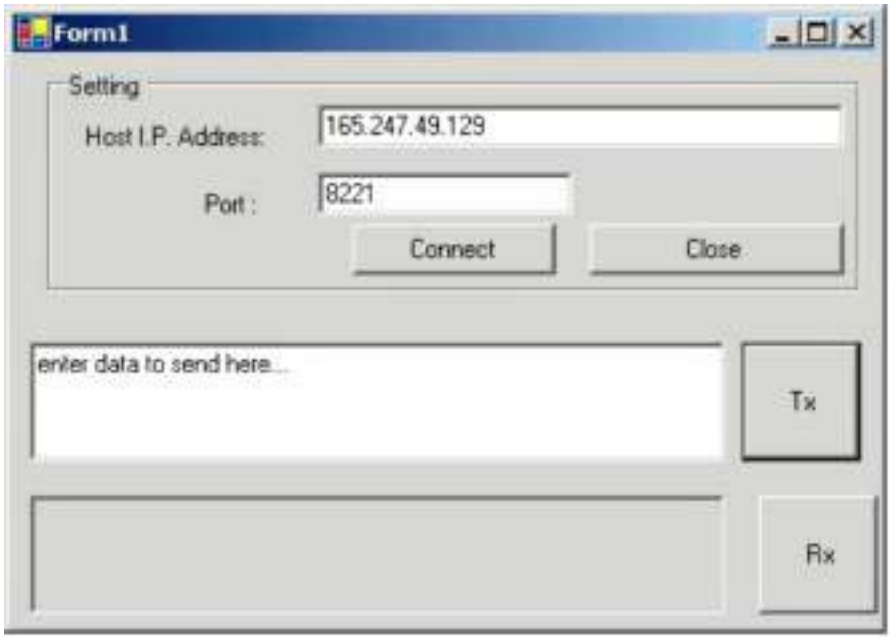
Jak powiedziałem, dwie aplikacje chcące komunikować się między sobą, muszą najpierw utworzyć połączenie. Aby dwie aplikacje mogły utworzyć połączenie, muszą najpierw się wzajemnie zidentyfikować (lub komputer). Komputery w sieci mają unikalne identyfikatory nazywane adresem IP, który jest przedstawiany w *notacji kropki* jak 10.20.120.127 itp. Zobaczmy jak wszystko to działa w .NET.

Przestrzeń nazw System.Net. Sockets

Zanim pójdziemy dalej, ściągnij kod źródłowy dołączony do tego artykułu. Rozpakuj plik zip do folderu, powiedzmy c:\Temp gdzie zobaczysz dwa foldery:

- Server
- Client

W folderze Server będzie jeden plik EXE . A w folderze Client będzie kod źródłowy w C# , będącyt naszym klientem. Będzie tam jeden plik nazwany SocketClient.sln. Jeśli klikniesz go dwukrotnie , przy uruchomionym VS.NET zobaczysz projekt SocketClientProj jako rozwiązanie. Pod tym projektem kryje się plik SocketClientForm.cs. Teraz budujemy kod (przez naciśnięcie Ctrl-Shift-B) a po uruchomieniu kodu zobaczysz poniższe okienko dialogowe:



Jak widać , okienko dialogowe ma pole dla adresu IP Hosta (który jest adresem IP maszyny na której uruchomisz Server Application (umieszczoną w folderze Server). Jest również pole gdzie możesz określić numer portu na którym Server jest nasłuchiwany. Aplikacja serwera ,jaką podałem nasłuchiwała na porcie 8221. Więc określiłem port na 8221.

Po określeniu tych parametrów łaczymy się z serwerem. Więc naciskamy Connect dla połączenia się z serwerem a dla zamknięcia połączenia naciskamy Close. Aby wysłać dane do serwera wpiszz jakąś daną w polu blisko przycisku nazwanego Tx a jeśli naciśniesz Rx apliakcaj będzie zablokwoana ,chyba ,że jest jakaś dana do odczytania.

Oprogramowanie gniazd w .NET jest tworzone przez klasę Socket obecną wewnątrz przestrzeni nazw System.Net.Sockets .
Klasa Socket ma kilka metod, właściwości i konstruktor.
Pierwszym krokiem jest stworzenie obiektu tej klasy.
Ponieważ jest tylko jedne konstruktor nie mamy wyboru i musimy go użyć.
Tu mamy jak stworzyć gniazdo:
m_socListener = new Socket(AddressFamily.InterNetwork,SocketType.Stream,ProtocolType.IP);
Pierwszym parametrem jest address family , którego będziemy używać z interNetwork – inne opcje obejmują NetBios itd.
AddressFamily jest typem wyliczeniowym zdefiniowanym w przestrzeni nazw Sockets.
Następnie musimy okreslić typ gniazda: i użyjemy pewnych dwóch sposobów połączeń opartych o gniazda (potok) zamiast zawodnych gniazd bezpołączeniowych (datagramów) .Więc oczywiście określimy potok jako typ gniazda i w końcu użyjemy TCP/IP aby określić typ protokołu jako Tcp. Od chwili stworzenia Socket musimy nawiązać połączenie z serwerem ponieważ używamy komunikacji zorientowanej połączeniowo. Aby połączyć się ze zdalnym komputerem musimy znać Adres IP i port na którym się łączymy. W .NET jest klasa w przestrzeni nazw System.Net nazywana IPEndPoint która reprezentuje komputer sieciowy jako adres IP i numer portu.

IPEndPoint ma dwa konstruktory – jeden który pobiera adres IP i numer Portu i jeden , który pobiera długość i numer portu. Ponieważ mamy adres IP komputera możemy użyć

public IPEndPoint(System.Net.IPAddress address, int port);

Jak widać pierwszy parametr pobiera obiekt IPAddress. Jeśli zbadamy klasę IPAddress zobaczymy ,że ma ona statyczną metodę nazywaną Parse ,która zwraca dany IPAddress jako ciąg (z notacją kropki) a drugi parametr będzie numerem portu. Kiedy mamy gotowy punkt końcowy, możemy użyć metody Connect z klasy Socket do połączenia z punktem końcowym (komputer zdalnego serwera).Tu mamy kod:
System.Net.IPAddress ipAdd = System.Net.IPAddress.Parse("10.10.101.200");
System.Net.IPEndPoint remoteEP = new IPEndPoint(iAdd,8221);
M._socClient.Connect(remoteEP)

Te trzy linijki kodu stworzą połączenie do zdalnego hosta uruchomionego na komputerzez z IP 10.10.101.200 i nasłuchującego na porcie 8221. Jeśli Serwer jest uruchomiony i zastartowany(

nasłuchuje),połączeniei zakończy się sukcesem. Jeśli jednak serwer nie jest uruchomiony wyjątek SocketException zostanie wywołany. Jeśli przechwycisz ten wyjątek i sprawdzisz właściwość Message tego wyjątku w tym przypadku zobaczysz tekst:

"No connection could be made because the target machine actively refused it."

Podobnie jeśli stworzyłeś połaczenie a serwer jakoś zamilkł, otrzymasz wyjątek jeśli sprobujesz wysłać dane.

"An existing connection was forcibly closed by the remote host"

Zakładając ,że połaczenie istnieje , możesz wysłąć dane używając metody Send klasy Socket. Metoda Send ma kilka przeciążeń. Każde z nich pobiera tablicę bajtów. Na przykład jeśli chcesz wysłać "Hello There" do hosta możesz użyć następującego wywyołania:

```
try
{
String szData = "Hello There";
byte[] byData = System.Text.Encoding.ASCII.GetBytes(szData);
m_socClient. Send(byData);
}
catch (SocketException se)
{
MessageBox.Show ( se.Message );
}
```

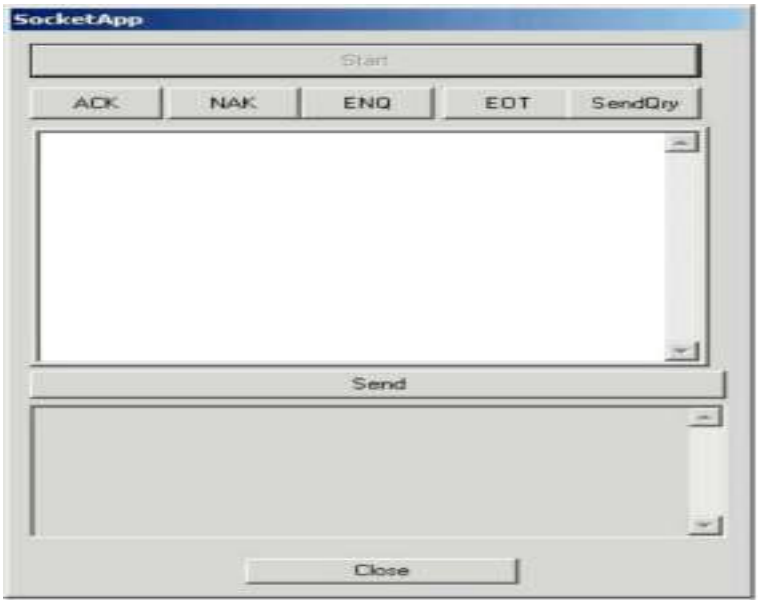
Zauważ,że moetoda Send jest zablokowana. Co oznacza ,że funkcja będzie zablokowana aż do wysłania danych lub dopóki nie pojawi się wyjątek. Jest też wersja nie zblokowana wysyłania, którą omówimy w następnej części tego artykułu. Podobnie jak Send jest metoda Receive w klasie Socket. Możesz odbierać dane używając nastepującego wywołania:

```
byte [] buffer = new byte[1024];
int iRx = m_socClient.Receive (buffer);
```

Tu również metoda Receive jest zablokowana . To ozancza ,że jeśli żadna dana nie jest dostępna, wywołanie będzie zablokowane dopóki jakaś dana nie nadejdzie, lub nie wystąpi wyjątek

Wersja Recevive nie zablkowana jest bardziej użyteczna, niż nie zabloowana wersja Send z powodu ,że jeśli zdecydujemy się zablokować Receive , skutecznie wykonamy odpytanie. Nie ma żadnegozdarzenia przy przyjściu danych. Ten model nie działa dobrzez w poważnych aplikacjach. Ale jest to tema kolejnej części artykułu. Teraz rozprawimy się z wersją zablokowaną. Aby użyć kodu źródłowego i aplikacji tutaj musmy najpierw uruchomić :

Tak jak tu:



Kiedy uruchomimy Server, kliknij Start aby rozpocząć nasłuchiwanie. Server nasłuchuje na porcie 8221. Więc upewnij się ,że określiłeś numer portu na 8221 w polu portu aplikacji klienta. A w polu IPAddress Client App wpisz adres IP Address maszyny na której jest uruchomiony Server. Jeśli wysyłasz jakieś dane do serwera od klienta przez naciśnięcie przycisku Tx zobaczymy, że dana jest w szarym polu edycyjnym

Programowanie gniazd w C# - część II

To jest druga część artykułu poświęconego obsłudze gniazd w języku C#. Przeczytaj ten artykuł aby nauczyć się jak uzywać gniazd w .Net Framework.To jest druga część poprzedniego artykułu o programowaniu gniazd. We wcześniejszym artykule stworzyliśmy klienta, ale ten klient używał zablokowanych funkcji IO (Receive) do odczytu danych przy regularnych odstępach czasu (poprzez kliknięcie przyciksu Rx). Ale jak wspomniałem we wcześniejszym artykule, taki model nie działa dobrze w świecie rzeczywsitych aplikacji. Ponieważ Windows jest systemem opartym o zdarzenia ,aplikacja (klient) powinien uzyskać notifyfikację pewnego rodzaju gdy tylko dane są odebrane aby klient mógł je odczytać zamiast klient miałby stale odpytywać o dane. Cóż jest to możliwe przy odrobinie wysiłku. Jeśli przeczytałeś pierwszą część, wiesz już, że klasa Socket w przestrzeni nazw Systems.Net.Sockets ma kilka metod takich jak Receive i Send które są funkcjami blokującymi. Poza tym są również funkcje takie jak BeginReceive , BeginSend itd. Są one przeznaczone dla asynchronicznych IO . Na przykład są przynajmniej dwa problemy z blokowaniem Receive:

1 Kiedy wywołujesz funkcję Receive funkcja jest blokowan jeśli żadna dana nie jest obecna, tak długo dopóki nie nadejdzie jakaś dana

2. Nawet jeśli jest dana kiedy wywołujesz receive ,nie wiesz kiedy wywołasz ją następnym razem. Musisz dokonac odpytania , co nie jest wydajnym sposobem.

Chociaż możesz dowodzić, że można przełamać tą *wadę*sprzez wielowątkowść, co oznaczaloby ,że można stworzyć nowy wątek i pozwolić aby wątek wykonał odpytanie i powiadomił główny wątek o danej. Cóż ta koncepcja działa dobrze. Ale nawet jeśli stworzysz nowy wątek, będzie wymagany głóny wątek do współdzielenia czasu CPU z tym nowym wątkiem. System Windows (Windows NT /2000 /XP) dostarcza modelu nazywanego Completion Port IO dla wykonywania zakładkowania (asynchronicznego) IO. Szczegóły portu IO Completion są poza zakresem obecnego omówienia, ale upraszcza to myślenie o IO Completion Port jako najwydajniejszego mechanizmu dla wykonania asynchronicznego IO w Windows dostarczanego przez system operacyjny.Model Completion Port może być stosowany do każdego rodzaju IO wliczajac w to plik odczyt /zapis i komunikację szeregową. Programowanie gniazd asynchronicznych .NET klasy Socket dostarca pdobnego modelu.

BeginReceive

Klasa Socket .NET Framework dostarcza metody BeginReceive do odbioru danych asynchronicznie tj. w sposób nie blokujący . Metoda BeginReceive ma następującą sygnaturę:

public **IAsyncResult BeginReceive(** byte[] buffer, int offset, int size, SocketFlags socketFlags, AsyncCallback callback, object state);

Sposób w jaki działa funkcja BeginReceive jest taki ,że przekazujesz funkcji bufor, funkcja callback (przekazanie) będzie wywołana kiedy nadejdą dane.

Ostatni parameter, object, BeginReceive, może być dowolna klasą pochodzącą z object (nawet null) Kiedy została wywołana funkcja callback, oznacz to ,że funkcja BeginReceive jest zakończona ,co oznacza ,że dane nadeszły.

Funkcja callback musi mmieć następującą sygnaturę:

void AsyncCallback(IAsyncResult ar);

Jak widzisz, callback zwraca void i przekazywany jest jeden parametr ,interfejs **IAsyncResult** , który zawiera stan operacji asynchronicznego odbierania.

Interfejs IAsyncResult ma kilka właściwości. Pierwszy parametr - AsyncState – jest obiektem,który jest taki sam jak ostatni parametr przekazany do BeginReceive(). Druga właściwość to AsyncWaitHandle , którą omówimy za chwilę. Trzecia właściwość wskazuje czy odbiór rzeczywiście był asynchroniczny lub zakończył się synchronicznie. Ważną rzeczą jest tu to ,że niekonieczne dla funkcji asynchronicznej jest zawsze kończenie asynchronicznie – może być całkowicie bezpośrednie jeśli dana jest już obecna. Kolejny parametr to IsComplete , który wskazuje czy operacja się zaończyła czy nie . Jeśli popatrzysz na sygnaturę BeginReceive zauważysz ,że funkcja zwraca również ***IAsyncResult.*** To jest ciekwae. Jak powiedziałem, będziemy mówili o drugiej właściwości IAsyncResult za moment. Teraz jest etn moment. Drugi parametr nazywa się **AsyncWaitHandle.** AsyncWaitHandle jest typu **WaitHandle**, klasy zdefiniowanej w przestrzeni nazw System.Threading. Klasa WaitHandle hermetyzujeHandle (który jest wskaxnikiem na int lub handle) i dostarcza sposobu na oczekwianie , kiedy zostanie zasygnalizowane. Ta klasa ma kilka metod statycznych jak WaitOne (która jest podobna do WaitForSingleObject), WaitAll (podobna do WaitForMultipleObjects z waitAll true) , WaitAny itd . Również te funckje są przeładowane.

Wracając do naszego omawiania interfejsu IAsyncResult, uchwyt w in AsyncWaitHandle (WaitHandle) jest sygnalizowany kiedy zakończy się operacja odbioru. Więc jeśli oczekujemy na to ,że uchwyt będzie nieskończony, będziemy wiedzieli kiedy odbiór się zakończył. To o znacza ,że jeśli przkekażemy WaitHandle do różnych wątków, różne wątki mogą oczekiwać na ten uchwyt i mogą zapewnić nas o tym fakcie, że dane nadeszły i ,że możemy je odczytać. Musimy być więc ostrozni korzystając z tego mechanizmu korzystając z funkcji callback. To prawda. Jeśli wybierzemy użycie tego mechanizmu WaitHandle wtedy parameterm funkcji callback do BeginReceive może być null jak pokazano tu:

```
//m_asynResult is declared of type IAsyncResult and assumming that m_socClient has made a
connection. m_asynResult =
m_socClient.BeginReceive(m_DataBuffer,0,m_DataBuffer.Length,SocketFlags.None,null,null); if (
m_asynResult.AsyncWaitHandle.WaitOne () )
{
int iRx = 0 ;
iRx = m_socClient.EndReceive (m_asynResult);
char[] chars = new char[iRx + 1];
System.Text.Decoder d = System.Text.Encoding.UTF8.GetDecoder();
int charLen = d.GetChars(m_DataBuffer, 0, iRx, chars, 0);
System.String szData = new System.String(chars);
txtDataRx.Text = txtDataRx.Text + szData;
}
```

Chociaż ten mechanizm będzie działał dobrze pryz zastosowaniu wielu wątków, skupimy się na mechaniźmie callback gdzie system powiadamai nas o całkowitym zakończeniu operacji asynchronicznej, w tym przypadku.

Powiedzmy ,że wywołaliśmy BeginReceive a po jakimś czasie nadeszły danei nasza funkcja callback została wywołana. Pozstaje pytanie gdzie są dane?Dane sa dostępne w buforzez, który przekazałeś jako pierwszy parametr podczas wywoływania metody BeginReceive(). W poniższym przykładzie dana będzie dostępna w m_DataBuffer :

```
BeginReceive(m_DataBuffer,0,m_DataBuffer.Length,SocketFlags.None, pfnCallBack,null);
```

Ale przed dostępem do bufora musimy wywołać funkcję EndReceive() w gieździe. EndReceive zwraca liczbę odebranych bajtów . To nie jest poprawny dostęp do bufora przed wywołaniem EndReceive.

Złożywszy to wszystko razem spójrzmy na poniższy prosty kod:

```
byte[] m_DataBuffer = new byte [10];
IAsyncResult m_asynResult; public
AsyncCallback pfnCallBack ; public
Socket m_socClient; // create the
socket... public void OnConnect()
{
m_socClient = new Socket (AddressFamily.InterNetwork,SocketType. Stream ,ProtocolType.Tcp );
// get the remote IP address...
IPAddress ip = IPAddress.Parse ("10.10.120.122");
int iPortNo = 8221;
//create the end point
IPEndPoint ipEnd = new IPEndPoint (ip.Address,iPortNo);
//connect to the remote host...
m_socClient.Connect ( ipEnd );
//watch for data ( asynchronously )...
WaitForData();
}
public void WaitForData()
{
if ( pfnCallBack == null )
pfnCallBack = new AsyncCallback (OnDataReceived);
// now start to listen for any data...
m_asynResult =
m_socClient.BeginReceive (m_DataBuffer,0,m_DataBuffer.Length,SocketFlags.None,
pfnCallBack,null);
}
public void OnDataReceived(IAsyncResult asyn)
{
//end receive...
```

```
int iRx = 0 ;
iRx = m_socClient.EndReceive (asyn);
char[] chars = new char[iRx + 1];
System.Text.Decoder d = System.Text.Encoding.UTF8.GetDecoder();
int charLen = d.GetChars(m_DataBuffer, 0, iRx, chars, 0);
System.String szData = new System.String(chars);
WaitForData();
}
```

Funkcja OnConnect tworzy połączenie do serwera a potem wywołuje WaitForData. WaitForData tworzy funkcję callback i wywołuje BeginReceive przekazując globalny bufor i funkcję callback. Kiedy nadejdzie dana, wywoływana jest OnDataReceive i m_socClient's EndReceive , która zwraca liczbę odebranych bajtów, potem dana jest kopiowana do ciągu i wywoływnana jest nowa WaitForData , która wywoła ponowni BeginReceive i tak w kółko. Działa to poprawnie jeśli masz jedno gniazdo w aplikacji.

WIELE GNIAZD

Teraz pomówimy o połączeniu dwóch gniazd z dwoma różnymi serwerami, lub tym samym serwerem (który jest poprawny) . Sposobem na to jest stworzenie dwóch różnych delegatur i dołączenie różnych delaegatur do różnych funkcji BeginReceive.A co jeśli mamy 3 gniazda lub powiedzmy n gnaizas , to podejście tworzenia wielu delegatur nie spełni się w tym przypadku. Rozwiązaniem powinno być użycie tylko jednej delagatury callback. Ale wtedy problemem jest to jak poznać które gniazdo zakończyło działanie. Na szczęście jest lepsze rozwiązanie. Jeśli popatrzymy na funkcję BeginReceive, ostatni parametr jest stanem obiektu. Możesz tu przekazać cokolwiek . A to co przekazesz tam, zostanie ci przekazane z powrotem jako część parametru funkcji callback. W rzeczywistości ten obiekt będzie przekazany później jako IAsyncResult.AsyncState. Tak więc kiedy wywołujesz callback, możesz użyć tej informacji do idnetyfikacji gniazda ,które zakończyło działanie. Ponieważ przekazujesz cokolwiek jako ostatni parametr, możesz przekazać klasę obiektu, która zwiera tak dużo informacji jak chcesz. Na przykład możemy zadeklarować klasę jak następuje:

```
public class CSocketPacket
{
public System.Net.Sockets.Socket thisSocket;
public byte[] dataBuffer = new byte[1024];
}
```

and call BeginReceive as follows:

```
CSocketPacket theSocPkt = new CSocketPacket ();
theSocPkt. thisSocket = m_socClient;
// now start to listen for any data...
m_asynResult = m_socClient.BeginReceive (theSocPkt.dataBuffer ,0,theSocPkt.dataBuffer.Length ,
SocketFlags.None,pfnCallBack,theSocPkt);
```

A w funkcji callback możemy uzyskać dane tak:

```
public void OnDataReceived(IAsyncResult asyn)
  { try
  {
CSocketPacket theSockId =
(CSocketPacket)asyn.AsyncState ; //end receive... int iRx =
0 ;
iRx = theSockId.thisSocket.EndReceive
(asyn); char[] chars = new char[iRx + 1];
System.Text.Decoder d =
System.Text.Encoding.UTF8.GetDecoder(); int charLen =
d.GetChars(theSockId.dataBuffer, 0, iRx, chars, 0);
System.String szData = new System.String(chars);
txtDataRx.Text = txtDataRx.Text + szData; WaitForData();
  }
  catch (ObjectDisposedException )
  { System.Diagnostics.Debugger.Log(0," 1 ","\nOnDataReceived: Socket has been
closed\n");
  }
  catch(SocketException se)
  { MessageBox.Show (se.Message );
  }
  }
```

Kiedy wywołasz BeginReceive , musisz przekazać bufor i liczbę bajtów do odbioru. Pytaniem pozostaje jak duży powinien być. Cóz odpowiedź jest uzalezniona. Możesz mieć bardzo mały rozmiar bufora, powiedzmy, 10 bajtów,i jeśli jest 20 bajtów do odczytania, wtedy wumagane są 2 wywołania do odbioru danych. Z drugiej strony, jeśli określisz długość na 1024 i wiesz,że zawsze będziesz odbierał dane w 10 bajtowych kawałkach,niepotrzebnie będziemy marnować pamięć. Tak więc długość zależy od apliakcji.

Strona Serwera

Jeśli zrozumiałeś to co napisałem wcześniej, łatwo zrozumiesz część Serwera apliakcji gniazda. Jak dotąd mówilismy o połaczeniach tworzonych przez klienta do serwera i wysyłanie oraz odbiór danych. Na końcu Serwera, aplikacja musi wysłąć i odebrać dane. Ale dodatkowo do dodawania i odbioru danych, serwer musi pozwolić klientowi utworzyć połączenie przez nasłuch na jakimś porcie. Serwer nie musi znać adresów IP klienta. Nie ma znaczenia gdzie jest klient ponieważ to nie serwer ale klientem który jest odpwoiedzialny za utworzenie połaczenia. Serwer odpowiada za zarządzanie połączeniami klienta. Po stronie serwera musi być gniazdo nazwane gniazdem Listener , które nasłuchuje na określonym numerze portu dla połączeń klienta. Kiedy klient wtorzy połączenie, serwer musi zaakceptować połączenie aby potem serwer wysyłał i odbierał dame z połączonego klienta. Poniższy kod ilustruje jak serwer nasłuchuje połaczenia i akceptuje połączenie:

```
public Socket m_socListener;
public void StartListening()
 try
//tworzy gniazdo nasłuchiwania...
m_socListener = new Socket(AddressFamily.InterNetwork,SocketType.Stream,ProtocolType.Tcp);
IPEndPoint ipLocal = new IPEndPoint ( IPAddress.Any ,8221);
//bind to local IP Address...
m_socListener.Bind( ipLocal );
//start listening...
m_socListener.Listen (4);
// create the call back for any client connections...
m_socListener.BeginAccept(new AsyncCallback ( OnClientConnect ),null);
cmdListen.Enabled = false;
catch(SocketException se)
MessageBox.Show ( se.Message
);
```

Jeśli popatrzysz na powyższy kod zonbaczysz , że jest podobny do asynchronicznego klienta. Przede wszystkim musimy stworzyć gnaizdo nasłuchujace i podpiąć do lokalnego adresu IP. Zauważ ,że mamy dane Any jako IPAddress .Wyjaśnię co mam na myśli później. Przekazujemy numer portu jako 8221. Następnie stworzymy wywołanie funkcji Listen. 4 jest parametrem wskazującym *backlog* wskazujący maksymalną długość kolejki połączeń będących w toku. Następnie tworzymy wywołanie BeginAccept przekazując mu wydelegowany callback. BeginAccept jest metodą nie zablokowaną, która zwracana jest bezpośrednio, a kiedy klient żąda połączenia, podprogram callback jest wywoływany i możemy zaakceptować połączenie przez wywołanie EndAccept. EndAccept zwraca obiekt gniazda, która reprezentuje nadchodzące połaczenie. Tu mamy kod dla delegatury callback:

```
public void OnClientConnect(IAsyncResult asyn)
{
try
{
  m_socWorker = m_socListener.EndAccept
(asyn); WaitForData(m_socWorker);
}
catch(ObjectDisposedException)
{
        System.Diagnostics.Debugger.Log(0,"1","\n OnClientConnection: Socket has been closed\n");
}
catch(SocketException se)
{
```

MessageBox.Show (se.Message);
Tu akceptuejmy połączenie i wywołujemyl WaitForData która z kolei wywołuje BeginReceive dla m_socWorker. Jeśli chcesz wysłać dane do klienta użyjemy use m_socWorker socket do tego celu:
Object objData = txtDataTx.Text;
byte[] byData = System.Text.Encoding.ASCII.GetBytes(objData.ToString ());
m_socWorker. Send (byData);

Tak wygląda nasz klient:

Tak wygląda nasz serwer