Simplifying Sockets in Visual Basic .NET

by Carl Franklin

Microsoft has done a great job of wrapping the Winsock API into some very cool managed classes in the .NET Framework, namely the classes in the System.Net and System.Net.Sockets namespaces. However, VB developers who are used to dealing with higher-level socket controls are now faced with new issues such as thread-affinity and asynchronous calls if they really want to do sockets right.

If you are new to sockets and TCP/IP, read my article entitled "TCP/IP, Sockets, and You." In that article, I show some simple examples of how to connect to a host, send a request, and process received data all using the application thread.

The issue is this. You can receive data three ways. First you can block the application thread until a packet has been received by calling the Receive method of a Socket object that's connected. Secondly, you can go into a loop, polling the Socket object to determine when data has been received. When polling, you could also use a Timer control to poll for received data. Lastly, you can receive data asynchronously by telling the socket to **begin** receiving data into a byte array, and then once data has been received, to call a special sub that you have set up to receive the data.

In this article I'll take you through all of these scenarios, and wrap it up with a component that greatly simplifies client-side socket programming. So, what's preventing you from just using my component without reading this article? After all, it works, right? Well, like any good implementation of technology, you are way ahead of the game if you understand why the code was written as it was. Not to mention the fact that you've got more time than you know what to do with. Right?

First off, let me revisit the simple client-side socket code that I wrote in my aforementioned article. Figure A shows code for the Sockets1 project. This is a simple Windows Forms project, with a Button and a Textbox. When you press the button, a socket object is created, connected to www.franklins.net (my website) on port 80 (HTTP), and a request is sent for a text file called sockets1.txt with an HTTP GET command.

Then the fun begins, and this is what I was just talking about. How do you deal with receiving the data? This code goes into a polling loop, calling DoEvents (bad idea.) The Socket object has a Poll method that returns True if data is available. In this case, we are telling the socket to poll for 100 microseconds, or 10 milliseconds.

Figure A. The Sockets1 project

```
Imports System.Net
Imports System.Net.Sockets
Public Class Form1
   Inherits System.Windows.Forms.Form
   '-- This is a System.Net.Sockets.Socket object
   Private ClientSocket As Socket
   Private Sub Button1_Click(ByVal sender As System.Object, _
     ByVal e As System.EventArgs) Handles Button1.Click
        '-- Create a new Socket for TCP/IP use
        ClientSocket = New Socket(AddressFamily.InterNetwork, _
          SocketType.Stream, ProtocolType.Tcp)
        '-- Create an endpoint to www.franklins.net on port 80
        Dim EP As New _
        IPEndPoint(Dns.Resolve("www.franklins.net").AddressList(0), 80)
        '-- Connect to the Endpoint
        ClientSocket.Connect(EP)
        '-- Send an HTTP string to retrieve a document
        Dim SendThis() As Byte = _
          System.Text.ASCIIEncoding.ASCII.GetBytes("GET " & _
             "http://www.franklins.net/sockets1.txt" & vbCrLf)
        ClientSocket.Send(SendThis)
        '-- Wait for received data
        Do
            '-- Is data available?
            If ClientSocket.Poll(100, SelectMode.SelectRead) = True Then
                '-- Available property returns number of bytes ready
                '   to be read. You must read into a byte array.
                '   This is because a character can be more than
                '   one byte if Unicode.
                Dim ReceiveThis(ClientSocket.Available) As Byte
                '-- Read the data
                ClientSocket.Receive(ReceiveThis)
                '-- Convert the byte array to an ASCII string
                TextBox1.Text = _
                  System.Text.ASCIIEncoding.ASCII.GetString(ReceiveThis)
                '-- All done!
                Exit Do
            Else
                '-- Allow other processes to occur
                Application.DoEvents()
            End If
        Loop
        '-- Shutdown and Close the socket. You must always do this
        ClientSocket.Shutdown(SocketShutdown.Both)
        ClientSocket.Close()
    End Sub

End Class
```

This method is ok, except that you are slowing down your main application thread every time you tell the socket to poll for 10 milliseconds. That could make your application feel choppy, especially if you are downloading a large amount of data. Not to mention the fact that using DoEvents at all is a bad idea. If you were paying attention in Visual Basic class in college (or wherever… please don't email me about this) you'll remember that using DoEvents can lead to unexpected results in your applications, and is generally not advised. If you can get around using it, do so.

This polling method is actually the second method of receiving data that I want to show you. The first is even worse. That is, to call the Receive method on the Socket and block the main application thread until all data has been received. Figure B shows the Sockets2 project, which does exactly that.

Figure B. The Sockets2 project blocks while receiving data

```
Imports System.Net
Imports System.Net.Sockets
Public Class Form1
   Inherits System.Windows.Forms.Form
   '-- This is a System.Net.Sockets.Socket object
   Private ClientSocket As Socket
    Private Sub Button1_Click(ByVal sender As System.Object, _
      ByVal e As System.EventArgs) Handles Button1.Click
        '-- Create a new Socket for TCP/IP use
        ClientSocket = New Socket(AddressFamily.InterNetwork, _
          SocketType.Stream, ProtocolType.Tcp)
        '-- Create an endpoint to www.franklins.net on port 80
        Dim EP As New _
        IPEndPoint(Dns.Resolve("www.franklins.net").AddressList(0), 80)
        '-- Connect to the Endpoint
        ClientSocket.Connect(EP)
        '-- Send an HTTP string to retrieve a document
        Dim SendThis() As Byte = _
          System.Text.ASCIIEncoding.ASCII.GetBytes("GET " & _
             "http://www.franklins.net/sockets1.txt" & vbCrLf)
        ClientSocket.Send(SendThis)
        '-- Wait for received data
        Dim ReceiveThis(4095) As Byte
        Dim Numbytes As Int32
        Numbytes = ClientSocket.Receive(ReceiveThis, _
           ReceiveThis.Length, SocketFlags.None)
        If Numbytes > 0 Then
            ReDim Preserve ReceiveThis(Numbytes - 1)
            '-- Convert the byte array to an ASCII string
            TextBox1.Text = _
              System.Text.ASCIIEncoding.ASCII.GetString(ReceiveThis)
            '-- All done!
        End If
        '-- Shutdown and Close the socket. You must always do this
        ClientSocket.Shutdown(SocketShutdown.Both)
        ClientSocket.Close()
    End Sub

End Class
```

I don't even want you to consider using this method. Once you see how easy it is to receive data asynchronously every other method is meaningless.

The Asynchronous Programming model makes it easy to use sockets asynchronously for both sending and receiving data. Instead of calling Receive, which blocks your main thread until a packet has been received, you can call the Socket object's BeginReceive method. Let me show you how it works.

Take a look at the AsyncClient project. The code – minus the designer-generated code - is shown in Figure C.

Figure C. The AsyncClient project shows how to receive data asynchronously.

```
Imports System.Net
Imports System.Net.Sockets
Imports System.IO

Public Class Form1
    Inherits System.Windows.Forms.Form

    Private myClient As Socket
    Private bytesDownloaded As Int32
    Private Const packetSize As Int32 = 4096    '-- Maximum receive buffer size.
    Private Bytes(packetSize - 1) As Byte
    Private MyStream As Stream
    Private DownloadFileName As String = Application.StartupPath & "\contaigen.mp3"

    Private Sub Button1_Click(ByVal sender As System.Object, _
      ByVal e As System.EventArgs) Handles Button1.Click
        '-- Set the Progressbar maximum property to the size of the file (cheating)
        Const FileSize As Int32 = 4220134
        Me.ProgressBar1.Maximum = FileSize

        '-- kill the file if it's there
        If File.Exists(DownloadFileName) Then
            If MsgBox(DownloadFileName & " exists. Overwrite?", _
               MsgBoxStyle.YesNo) = MsgBoxResult.Yes Then
                File.Delete(DownloadFileName)
            Else
                Exit Sub
            End If
        End If

        '-- Create a new socket
        myClient = New Socket(AddressFamily.InterNetwork, _
           SocketType.Stream, ProtocolType.Tcp)

        '-- Create an endpoint to the host application
        Dim EP As New IPEndPoint(Dns.Resolve("www.pwop.com").AddressList(0), 80)

        '-- Connect to the endpoint
        Try
            myClient.Connect(EP)
        Catch ex As Exception
            MsgBox("Could not connect")
            Exit Sub
        End Try

        '-- Send an HTTP GET command to retrieve an MP3 of a song
        '   that Carl wrote and recorded :-)
        Try
            Dim SendThis As String = "GET http://www.pwop.com/contaigen.mp3" & vbCrLf
            Dim SendBytes() As Byte = New System.Text.ASCIIEncoding().GetBytes(SendThis)
            myClient.Send(SendBytes)
        Catch ex As Exception
            myClient.Shutdown(SocketShutdown.Both)
            myClient.Close()
            MsgBox(ex.Message)
            Exit Sub
        End Try

        '-- Begin receiving data asynchronously
        Try
            myClient.BeginReceive(Bytes, 0, packetSize, SocketFlags.None, _
               AddressOf ReceiveCallback, Nothing)
        Catch ex As Exception
            myClient.Shutdown(SocketShutdown.Both)
            myClient.Close()
            MsgBox(ex.Message)
        End Try
```

```
    End Sub

    Public Sub ReceiveCallback(ByVal ar As IAsyncResult)
        Dim ByteCount As Integer
        Try
            ByteCount = myClient.EndReceive(ar)
            If ByteCount < 1 Then
                '-- Done. Exit
                MsgBox("Done")
                Exit Sub
            End If

            bytesDownloaded += ByteCount

            '-- If you're going to update the user interface, you
            '   have to do that from the main app thread. This code
            '   is running on the socket receive thread, and will
            '   cause problems if your UI elements are being modified
            '   by the user or your main app code at the same time
            '   that this code is executing. In this case, it does not
            '   matter, because no other code touches the progress bar.
            '   I will show you how to deal with this issue in another
            '   example.
            '-- fudge the progressbar (maximum property set with knowledge
            '   of the file size)
            ProgressBar1.Value = bytesDownloaded

            '-- Save the file data here
            MyStream = File.Open(DownloadFileName, FileMode.Append, _
               FileAccess.Write, FileShare.Read)
            With MyStream
                .Write(Bytes, 0, ByteCount)
                .Flush()
                .Close()
            End With

            '-- Receive the next packet if this was not the last packet
            myClient.BeginReceive(Bytes, 0, packetSize, SocketFlags.None, _
               AddressOf ReceiveCallback, Nothing)
        Catch ex As Exception
            MsgBox(ex.Message)
        End Try
    End Sub

End Class
```

Now, this code is far from perfect, but it's a step in the right direction. After a connection is established, the Socket object's BeginReceive is called like so:

```
        myClient.BeginReceive(Bytes, 0, packetSize, SocketFlags.None, _
           AddressOf ReceiveCallback, Nothing)
```

The first argument (Bytes) is a byte array that will receive the data. This array has to be dimensioned, but the data in it will be overwritten. Next is the first element within the array where the data will be written. This allows you to use one big array to receive all of a large file or some other large construct where the length is known, and you can manage where in the buffer the data is actually written to for each packet received.

The next argument is the maximum number of bytes to receive. Since we are defining the receive buffer as being 4K (4096 bytes) in length, we'll just use the packetSize constant for this value. The next argument lets you specify Winsock flags (if you don't know that you need them, select None).

The next argument is the address of a method that will be called by the Socket object when either the array is full or the packet received is smaller than the array. Use the AddressOf keyword followed by the name of the method. The last argument is a state object, which we will leave alone for now. But, we'll use it later, I promise.

Now, let's look at the ReceiveCallback method. You can call it anything. The only requirement is that it be a subroutine that accepts a single argument, an IAsyncResult object. More correctly, any object that implements the IAsyncResult interface.

To finalize the call, and return the actual number of bytes that were received, call EndReceive on the socket like so:

```
ByteCount = myClient.EndReceive(ar)
```

The ar variable is the IAsyncResult I just mentioned. Now, the data is in the bytes() array! Next, check to see if ByteCount is less than 1, in which case, there is no more data to be received.

The next line of code sets the ProgressBar value, and here is where we get into multithreading issues.

```
ProgressBar1.Value = bytesDownloaded
```

We can't help but bump up against thread synchronization issues. We didn't specifically create a thread object here, but when we called BeginReceive, we told the socket to receive data on a new thread (which it pulled from the system thread pool).

Here's the critical idea. The ReceiveCallback function is being called from a thread *other* than your main application thread. If, in this sub, you access a module-level variable in that function (such as a custom control reference) and you *also* access that same variable in the main form thread (say, from a Button_Click event or in any UI generated event), it is possible for both of those threads to operate on those variables at *virtually the same time (even in between lines of code!)*

You may have heard about thread synchronization, which is required to keep multiple threads that access the same variables from accessing the same module-level variables at the same time. In this case, locking will not help.

It's not a problem with this particular example because there is no other code that accesses the ProgressBar control except for this ReceiveCallback sub, and that is called one shot at a time, and not by multiple client sockets. If we had multiple client sockets, then using SyncLock would be appropriate.

Even though this code is safe by design, we will revisit this issue in the FranklinSocket.SocketClient object, which I will discuss next.

The next task is to open the output file for append (meaning that new data will be appended to the end of the existing file), and write the received data to it. Keeping the file open between packets is an option, but I chose to do it this way so that if the connection was broken during the download, at least I'd have some of the data.

```
'-- Save the file data here
MyStream = File.Open(DownloadFileName, FileMode.Append, _
   FileAccess.Write, FileShare.Read)
With MyStream
    .Write(Bytes, 0, ByteCount)
    .Flush()
    .Close()
End With
```

Here's what makes this thing beautiful. After we're done processing this packet, we can just call BeginReceive again with the exact same parameters. More data will be received, and our callback sub will be called again at least one more time. If no more data is available, EndReceive will return 0 and we know we're done downloading.

Using a Delegate for Thread Safety

Like I said before, this code is safe by design, but let's assume that it isn't for a minute, and implement the proper way to handle calling UI code from another thread. Figure D shows the critical code in the next sample project, ThreadSafeAsyncClient, which defines a delegate as a new data type.

Figure D. Create a Delegate data type and a private sub to execute code that accesses the user interface.

```
Private Delegate Sub dlgUpdateProgressBar(ByVal Value As Int32)

Private Sub UpdateProgressBar(ByVal Value As Int32)
    '-- fudge the progressbar (maximum property set with knowledge of the file size)
    ProgressBar1.Value = Value
End Sub


Public Sub ReceiveCallback(ByVal ar As IAsyncResult)
    ...
    Dim dlg As New dlgUpdateProgressBar(AddressOf UpdateProgressBar)
    Dim args() As Object = {bytesDownloaded}
    Me.Invoke(dlg, args)
End Sub
```

I suppose we should start with Sub UpdateProgressBar. Any time you have code that needs to be called on the main application thread, yet you want to call it from another thread, create a private sub and put the code in there.

Then, create a delegate data type to match that sub. In this case it's called dlgUpdateProgressBar. The syntax is self-explanatory. Just make sure you define the same signature as your sub.

To invoke the code on the main app thread, you simply create a variable of the delegate type, create an object array to hold the arguments (in this case, it's a single int32 value), and call the Invoke method of the form, passing the delegate variable and the array.

This is only possible because System.Windows.Forms.Form eventually derives from Control, which implements an interface called IsynchronizeInvoke. This interface is where the Invoke method comes from. Invoke insures that the delegate you pass it is called on the control's thread, which is also the main application thread.

You Call This Productivity?

No. If I had to write all that code every time I wanted to make a socket connection and receive some data, I'd be wasting time and distracting myself from the task at hand. That's why I have written a socket client component that completely encapsulates asynchronous receiving of data, and automatically raises its Receive event using the thread of the underlying form.

Let's take a look at it now. Load up the FranklinSocket project. This is a DLL project that contains a single component, SocketClient. Figure E shows the code in its entirety, except for the auto-generated code.

Figure E. The FranklinSocket.SocketClient component simplifies development of asynchronous socket calls.

```
'-------------------------------------------------------------
' FranklinSocket.SocketClient Class                          '
' Developed by Carl Franklin for Franklins.Net               '
' Copyright 2003 by Franklins.Net                           '
' All Rights Reserved                                       '
'                                                           '
' Use of this code or any derivative is only allowed if this   '
' Documentation header is not removed. The original code is    '
' available at http://www.franklins.net/dotnet              '
'                                                           '
' If you really want to learn how to use Sockets, Remoting,    '
' Multithreading, Asynchronous Calls, and understand the       '
' .NET Framework, attend Carl Franklin's Advanced VB.NET       '
' Class. Details online at http://www.franklins.net          '
'-------------------------------------------------------------
Imports System.Net
Imports System.Net.Sockets
Imports System.ComponentModel.Design

Public Class SocketClient
    Inherits System.ComponentModel.Component

    '-- Used to convert byte arrays to strings and vice-versa
    Private ASCII As New System.Text.ASCIIEncoding

    Private sock As Socket                      '-- Does the actual send/receive
    Private Shared dummy As New ArrayList       '-- Used to sync value type variables
    Private Shared socksyncobj As New ArrayList '-- Used to sync the socket

    Private _packetSize As Int32 = 4096         '-- Default packet size.
    Private _linemode As Boolean = True         '-- LineMode property is True by default.
    Private _eolchar As Char = vbLf          '-- Linefeed character denotes end of line.

    '-- This is required to switch the context from the socket thread to the
    '   underlying calling container's thread when raising events from the socket's
    '   receive data thread.
    Private _syncrhonizingObject As System.ComponentModel.ISynchronizeInvoke

    '-- Delegates are required to raise events on the container's thread
    Private Delegate Sub RaiseReceiveEvent(ByVal ReceiveData As String)
    Private Delegate Sub RaiseConnectedEvent(ByVal Connected As Boolean)
    Private Delegate Sub RaiseExceptionEvent(ByVal Ex As Exception)
```

```
'-- Event definitions
Public Event Receive(ByVal ReceiveData As String)
Public Event Connected(ByVal Connected As Boolean)
Public Event Exception(ByVal Ex As Exception)

'-- This must be set to the container control (for example, Form1 in a
'   Windows Forms app)
<System.ComponentModel.Browsable(False)> _
Public Property SynchronizingObject() As System.ComponentModel.ISynchronizeInvoke
    Get
        If _syncrhonizingObject Is Nothing And Me.DesignMode Then
            Dim designer As IDesignerHost = Me.GetService(GetType(IDesignerHost))
            If Not (designer Is Nothing) Then
                _syncrhonizingObject = designer.RootComponent
            End If
        End If
        Return _syncrhonizingObject
    End Get
    Set(ByVal Value As System.ComponentModel.ISynchronizeInvoke)
        If Not Me.DesignMode Then
            If Not (_syncrhonizingObject Is Nothing) And _
               Not (_syncrhonizingObject Is Value) Then
                Throw New Exception("Property can not be set at run-time.")
            Else
                _syncrhonizingObject = Value
            End If
        End If
    End Set
End Property

'-- When LineMode is True, this character denotes the end of a line of text
Public Property EolChar() As Char
    Get
        Dim c As Char
        SyncLock dummy
            c = _eolchar
        End SyncLock
        Return c
    End Get
    Set(ByVal Value As Char)
        SyncLock dummy
            _eolchar = Value
        End SyncLock
    End Set
End Property

'-- When Set to True, received data is parsed into lines of text
'   and the Receive event is fired once for each line of text.
'   When false, the Receive event is fired for each packet received.
Public Property LineMode() As Boolean
    Get
        Dim l As Boolean
        SyncLock dummy
            l = _linemode
        End SyncLock
        Return l
    End Get
    Set(ByVal Value As Boolean)
        SyncLock dummy
            _linemode = Value
        End SyncLock
    End Set
End Property

'-- Closes the socket
Public Sub Close()
    If Me.IsConnected = True Then
        SyncLock socksyncobj
            sock.Shutdown(SocketShutdown.Both)
            sock.Close()
```

```
            sock = Nothing
            If IsConnected() = False Then
                RaiseEvent Connected(False)
            End If
        End SyncLock
    End If
End Sub

'-- The size of the receive buffer
Public Property PacketSize() As Int32
    Get
        Dim ps As Int32
        SyncLock dummy
            ps = _packetSize
        End SyncLock
        Return ps
    End Get
    Set(ByVal Value As Int32)
        SyncLock dummy
            _packetSize = Value
        End SyncLock
    End Set
End Property

'-- Sends the string data and blocks until all data is sent
Public Sub Send(ByVal Data As String)
    If IsConnected() = False Then
        RaiseEvent Connected(False)
    End If
    Dim bytes() As Byte = ASCII.GetBytes(Data)
    If Me.IsConnected = True Then
        SyncLock socksyncobj
            sock.Send(bytes, bytes.Length, SocketFlags.None)
        End SyncLock
    Else
        Throw New Exception("Socket is not connected")
    End If
End Sub

'-- Overridable routine called on the container form's thread
Protected Overridable Sub OnReceive(ByVal ReceiveData As String)
    RaiseEvent Receive(ReceiveData)
End Sub

'-- Overridable routine called on the container form's thread
Protected Overridable Sub OnConnected(ByVal Connected As Boolean)
    RaiseEvent Connected(Connected)
End Sub

'-- Overridable routine called on the container form's thread
Protected Overridable Sub OnException(ByVal Ex As Exception)
    RaiseEvent Exception(Ex)
End Sub

'-- Returns True of the socket is connected
Public Function IsConnected() As Boolean
    '-- Make sure it isn't Nothing... that would be bad
    Dim f As Boolean
    If sock Is Nothing Then
        Return False
    Else
        SyncLock socksyncobj
            f = sock.Poll(1, SelectMode.SelectRead)
        End SyncLock
        '-- Still could be Nothing here.
        If sock Is Nothing Then
            Return False
        Else
            '-- The true test for a disconnected socket
            Dim a As Int32
            SyncLock socksyncobj
```

```
                a = sock.Available
            End SyncLock
            If f = True And a = 0 Then
                Return False
            Else
                Return True
            End If
        End If
    End If
End Function

'-- Callback for data received asynchronously by the socket
Private Sub ReceiveCallback(ByVal ar As IAsyncResult)
    Dim lm As Boolean = Me.LineMode '-- Line Mode?
    '-- Retrieve the array of bytes
    Dim bytes() As Byte = ar.AsyncState
    '-- Get the number of bytes received
    Dim numbytes As Int32 = sock.EndReceive(ar)
    '-- Did we get anything?
    If numbytes > 0 Then
        '-- Yes. Resize the array to match the number of bytes received
        ReDim Preserve bytes(numbytes - 1)
        '-- Convert to an ASCII string
        Dim recv As String = ASCII.GetString(bytes)
        '-- Now we need to raise the Receive event.
        '   args() is used to pass an argument from this thread
        '   to the synchronized container's UI thread
        Dim args(0) As Object
        '-- Create a new delegate for the OnReceive event
        Dim d As New RaiseReceiveEvent(AddressOf OnReceive)
        '-- Line Mode?
        If lm = True Then
            '-- Yes. Split the string into an array based on the EOL character
            Dim sep() As Char = {Me.EolChar}
            Dim lines() As String = recv.Split(sep)
            Dim i As Int32
            '-- Raise the Receive event once for every line of text.
            For i = 0 To lines.Length - 1
                If i = lines.Length - 1 Then
                    args(0) = lines(i)
                Else
                    args(0) = lines(i) & EolChar
                End If
                '-- This is the magic, right here!
                _syncrhonizingObject.Invoke(d, args)
            Next
        Else
            '-- Not line mode. Easy. Pass the entire string once.
            args(0) = recv
            _syncrhonizingObject.Invoke(d, args)
        End If
    End If

    '-- Are we still connected?
    If IsConnected() = False Then
        '-- Nope. Raise the Connected event passing False
        Dim args() As Object = {False}
        Dim d As New RaiseConnectedEvent(AddressOf OnConnected)
        _syncrhonizingObject.Invoke(d, args)
    Else
        '-- Yes. We're still connected.
        '-- Resize bytes to the packetsize
        ReDim bytes(Me.PacketSize - 1)
        '-- Call BeginReceive again, catching any errors
        Try
            sock.BeginReceive(bytes, 0, bytes.Length, SocketFlags.None, _
               AddressOf ReceiveCallback, bytes)
        Catch ex As Exception
            '-- If there was an exception, raise the Exception event
            '   This is better than throwing an exception, because there
            '   is most likely no code to catch the exception
```

```
            Dim args() As Object = {ex}
            Dim d As New RaiseExceptionEvent(AddressOf OnException)
            _syncrhonizingObject.Invoke(d, args)

            '-- Are we still connected? If not, raise the Connected event
            If IsConnected() = False Then
                args(0) = False
                Dim dl As New RaiseConnectedEvent(AddressOf OnConnected)
                _syncrhonizingObject.Invoke(dl, args)
            End If
        End Try
    End If
End Sub

'-- Overload for Connect(EndPoint)
Public Sub Connect(ByVal HostNameOrAddress As String, ByVal Port As Int32)
    Dim ServerAddress As IPAddress = Dns.Resolve(HostNameOrAddress).AddressList(0)
    Try
        Connect(ServerAddress, Port)
    Catch ex As Exception
        Throw ex
    End Try
End Sub

'-- Overload for Connect(EndPoint)
Public Sub Connect(ByVal ServerAddress As IPAddress, ByVal Port As Int32)
    Dim EP As New IPEndPoint(ServerAddress, Port)
    Try
        Connect(EP)
    Catch ex As Exception
        Throw ex
    End Try
End Sub

'-- This Overload does the actual connecting.
Public Sub Connect(ByVal EndPoint As IPEndPoint)
    '-- Create a new socket
    sock = New Socket(AddressFamily.InterNetwork, _
       SocketType.Stream, ProtocolType.Tcp)
    '-- Try connecting
    Try
        sock.Connect(EndPoint)
        If Me.IsConnected = True Then
            RaiseEvent Connected(True)
        End If
    Catch ex As Exception
        '-- There was a problem
        Throw New Exception("Could not connect", ex)
    End Try

    '-- Create a local byte array to hold received data
    Dim bytes(Me.PacketSize - 1) As Byte

    Try
        '-- This call tells the socket to wait for received data on a new
        '   thread from the system thread pool. It passes the byte array as
        '   the buffer to receive the data, passes the address of ReceiveCallback
        '   as the delegate to be called when data is received, and passes
        '   the array again as a reference that will be passed back to the
        '   ReceiveCallback event (so we don't have to define the array as
        '   a module-level variable thereby making it more difficult to be
        '   thread-safe).
        sock.BeginReceive(bytes, 0, bytes.Length, SocketFlags.None, _
            AddressOf ReceiveCallback, bytes)
    Catch ex As Exception
        Throw New Exception("Error receiving Data", ex)
        If IsConnected() = False Then
            RaiseEvent Connected(False)
        End If
    End Try
End Sub
```

```
End Class
```

Let's start here with the big picture by describing how to use this component. You go over to your toolbar and drag a SocketClient component on any form that you want to use to make a socket connection. The default name is SocketClient1.

You add a Button and a TextBox to the form. Behind the button you enter the following code to connect to www.franklins.net and send a request for the file sockets1.txt. The code shown here is from the sample project named SocketClientTest.

```
Private Sub Button1_Click(ByVal sender As System.Object, _
  ByVal e As System.EventArgs) Handles Button1.Click
    Try
        SocketClient1.LineMode = False
        SocketClient1.Connect("www.franklins.net", 80)
        SocketClient1.Send("GET http://www.franklins.net/sockets1.txt" & vbCrLf)
    Catch ex As Exception
        MsgBox(ex.Message)
        SocketClient1.Close()
    End Try
End Sub
```

The reason I have the connect and request each in their own Try/Catch is because I want Next, handle the Receive event to get the data. You don't need to worry about thread safety. It's all taken care of automatically. I'll show you how in a minute, but first, just check out how easy it is to get this data.

```
Private Sub SocketClient1_Receive(ByVal ReceiveData As String) Handles _
  SocketClient1.Receive
    Dim DownloadFileName As String = Application.StartupPath & "/sockets1.txt"
    Dim MyStream As New System.io.StreamWriter(DownloadFileName)
    With MyStream
        .Write(ReceiveData)
        .Flush()
        .Close()
    End With
End Sub
```

A Connected event fires when you are either connected or disconnected. In this case, the server will close the socket connection after sending the file. Just respond to the Connected event with some signal to the user that the file has been downloaded:

```
Private Sub SocketClient1_Connected(ByVal Connected As Boolean) Handles _
   SocketClient1.Connected
    If Connected = False Then
        MsgBox("Done")
    End If
End Sub
```

Of course, if there is a problem during operation, you don't want this component to throw an exception, because there will be no code there to catch it. Instead, there is an Exception event that fires if an exception occurs. We can just respond to that by displaying any error message. Again, there is no need for special thread-handling code:

```
Private Sub SocketClient1_Exception(ByVal Ex As System.Exception) Handles _
   SocketClient1.Exception
    MsgBox(Ex.Message)
```

```
End Sub
```

How easy is that??

Not only does this component shield you from the thread-safety issues associated with the user interface, but it is also thread safe in and of itself. In other words, if you wanted to make multiple client connections using multiple instances of the SocketClient component, and have them operate on multiple threads, you do not have to worry about thread-safety.

Now, let's take a look at this component and see exactly what it's doing.

The callback for processing received data is called ReceiveCallback. In order to raise the Receive event safely, this code must invoke a delegate of a private sub that raises the event. And, here's the tricky part, it must do it using the form or control that the component is placed on.

Just for fun, drop a Timer component on a form, and look at the Property window. There is a property called SynchronizingObject. And look, it's set to Form1 automatically! How did it do that?

The answer came to me from my friend Chris Sells, and it makes perfect sense. Figure F shows the SynchronizingObject property handler for the SocketClient component.

Figure F. The SynchronizingObject property works at design time to get the root component of the designer that the component is sitting on. In other words, the parent Form.

```
'-- This must be set to the container control (for example, Form1 in a
'   Windows Forms app)
<System.ComponentModel.Browsable(False)> _
Public Property SynchronizingObject() As System.ComponentModel.ISynchronizeInvoke
    Get
        If _syncrhonizingObject Is Nothing And Me.DesignMode Then
            Dim designer As IDesignerHost = Me.GetService(GetType(IDesignerHost))
            If Not (designer Is Nothing) Then
                _syncrhonizingObject = designer.RootComponent
            End If
        End If
        Return _syncrhonizingObject
    End Get
    Set(ByVal Value As System.ComponentModel.ISynchronizeInvoke)
        If Not Me.DesignMode Then
            If Not (_syncrhonizingObject Is Nothing) And _
               Not (_syncrhonizingObject Is Value) Then
                Throw New Exception("Property can not be set at run-time.")
            Else
                _syncrhonizingObject = Value
            End If
        End If
    End Set
End Property
```

This property works at design time to get the root component of the designer that the component is sitting on. In other words, the parent Form. It does this by calling GetService, which is a method of the Component class to return a service that is provided by the component. It passes the IDesignerHost type, returning the designer that the

component is sitting on. Then it gets the RootComponent of that designer, which translates to the Form itself.

The Form reference is only returned once, at design time. And, if you try to set the SynchronizingObject property at runtime, the Set handler throws an exception.

The Attribute <System.ComponentModel.Browsable(False)> tells .NET not to display the property in the property window.

The end result is a seamless automatic way to pass the component a reference to the form that it sits on without requiring *any* intervention from the programmer.

IsConnected

Figure G shows the IsConnected function, which returns true only if the socket is connected. It will check to see if the socket is nothing as well. It should be noted that in order to successfully test to see if the socket is connected you have to AND the results of the Poll method and the Available property, which returns zero if there is no data available to be read. The Socket object has a Connected property, but it does not accurately reflect the state of the socket connection.

```
'-- Returns True of the socket is connected
Public Function IsConnected() As Boolean
    '-- Make sure it isn't Nothing... that would be bad
    Dim f As Boolean
    If sock Is Nothing Then
        Return False
    Else
        SyncLock socksyncobj
            f = sock.Poll(1, SelectMode.SelectRead)
        End SyncLock
        '-- Still could be Nothing here.
        If sock Is Nothing Then
            Return False
        Else
            '-- The true test for a disconnected socket
            Dim a As Int32
            SyncLock socksyncobj
                a = sock.Available
            End SyncLock
            If f = True And a = 0 Then
                Return False
            Else
                Return True
            End If
        End If
    End If
End Function
```

LineMode

Another great feature of this control is Line mode, or Text mode. When downloading ASCII text, it is more useful for the Receive event to fire once for every line of text. When not in Line mode, the Receive event either when an entire packet is received, or the

buffer (the byte array that receives the data) fills up. The size of the buffer is determined by the PacketSize property. The default value for the LineMode property is True.

When LineMode is True, the code has to know what character denotes the end of a line. It is usually the LineFeed character (ASCII 10) but its good not to hard-code magic numbers like that, so an EOLChar property is exposed.

Connecting

There are three Connect overloads. Here is the first one:

```
'-- Overload for Connect(EndPoint)
Public Sub Connect(ByVal HostNameOrAddress As String, ByVal Port As Int32)
    Dim ServerAddress As IPAddress = Dns.Resolve(HostNameOrAddress).AddressList(0)
    Try
        Connect(ServerAddress, Port)
    Catch ex As Exception
        Throw ex
    End Try
End Sub
```

This allows you to pass any text representation of a name like www.franklins.net or an IP address like 127.0.0.1, and a port.

This overload gets an IPAddress object from the name or address string, and calls the next overload, which accepts an IPAddress object and a port:

```
'-- Overload for Connect(EndPoint)
Public Sub Connect(ByVal ServerAddress As IPAddress, ByVal Port As Int32)
    Dim EP As New IPEndPoint(ServerAddress, Port)
    Try
        Connect(EP)
    Catch ex As Exception
        Throw ex
    End Try
End Sub
```

The next step is to make an IPEndPoint object from the address and port. This overload does that, and then calls the next overload, which does the real work:

```
 '-- This Overload does the actual connecting.
Public Sub Connect(ByVal EndPoint As IPEndPoint)
    '-- Create a new socket
    sock = New Socket(AddressFamily.InterNetwork, _
       SocketType.Stream, ProtocolType.Tcp)
    '-- Try connecting
    Try
        sock.Connect(EndPoint)
        If Me.IsConnected = True Then
            RaiseEvent Connected(True)
        End If
    Catch ex As Exception
        '-- There was a problem
        Throw New Exception("Could not connect", ex)
    End Try

    '-- Create a local byte array to hold received data
    Dim bytes(Me.PacketSize - 1) As Byte
```

```
        Try
            '-- This call tells the socket to wait for received data on a new
            '   thread from the system thread pool. It passes the byte array as
            '   the buffer to receive the data, passes the address of ReceiveCallback
            '   as the delegate to be called when data is received, and passes
            '   the array again as a reference that will be passed back to the
            '   ReceiveCallback event (so we don't have to define the array as
            '   a module-level variable thereby making it more difficult to be
            '   thread-safe).
            sock.BeginReceive(bytes, 0, bytes.Length, SocketFlags.None, _
                AddressOf ReceiveCallback, bytes)
        Catch ex As Exception
            Throw New Exception("Error receiving Data", ex)
            If IsConnected() = False Then
                RaiseEvent Connected(False)
            End If
        End Try
    End Sub
```

Having three overloads calling each other like this makes it very convenient for the programmer, allowing them to connect with whatever representation of an IP address they have at the time. If the programmer already has an IPEndPoint, great! They can call the third overload directly. However, if all they have is the host name and port, then they can pass that information.

Once connected, the code immediately creates a byte array named bytes() – original, huh? - and calls BeginReceive, passing the bytes() array as the State object. Remember I told you we'd talk about that last parameter to BeginReceive? You can pass any object (or, more accurately, a reference to any object) and in the receive callback method that reference is made available through the IAsyncResult variable (ar).

To recap, at this point the CLR takes a thread from the system threadpool and waits for received data on that thread. When either an entire packet is received, or the buffer fills up (meaning that the packet is bigger than the byte array) the ReceiveCallback method is called.

Let's take a look at the ReceiveCallback method now.

You should see familiar code up to about this point:

```
            '-- Now we need to raise the Receive event.
            '   args() is used to pass an argument from this thread
            '   to the synchronized container's UI thread
            Dim args(0) As Object
            '-- Create a new delegate for the OnReceive event
            Dim d As New RaiseReceiveEvent(AddressOf OnReceive)
```

This is the code that creates the delegate object. But, instead of modifying user interface inside this component, we're going to raise an event (Receive) via the OnReceive method, which is private to our component. Since the Receive event is raised on the main form thread, it's completely thread-safe.

The next bit of code prepares to call the Receive event. If LineMode is True, we call the Split method to split the data string into an array using the EOLChar character as the delimiter. Then, the Receive event is called for each line. This can be slow.

```
            '-- Line Mode?
            If lm = True Then
                '-- Yes. Split the string into an array based on the EOL character
                Dim sep() As Char = {Me.EolChar}
                Dim lines() As String = recv.Split(sep)
                Dim i As Int32
                '-- Raise the Receive event once for every line of text.
                For i = 0 To lines.Length - 1
                    If i = lines.Length - 1 Then
                        args(0) = lines(i)
                    Else
                        args(0) = lines(i) & EolChar
                    End If
                    '-- This is the magic, right here!
                    _syncrhonizingObject.Invoke(d, args)
                Next
```

If LineMode is False, then the entire string is passed all at once.

```
            Else
                '-- Not line mode. Easy. Pass the entire string once.
                args(0) = recv
                _syncrhonizingObject.Invoke(d, args)
            End If
        End If
```

Now we check to see if we're still connected. If we are, then BeginReceive is called again in a Try/Catch. I prefer to err on the side of caution here, because at any time the Socket could be disconnected. If you don't handle that possible error situation, a riot could ensue.

```
        '-- Are we still connected?
        If IsConnected() = False Then
            '-- Nope. Raise the Connected event passing False
            Dim args() As Object = {False}
            Dim d As New RaiseConnectedEvent(AddressOf OnConnected)
            _syncrhonizingObject.Invoke(d, args)
        Else
            '-- Yes. We're still connected.
            '-- Resize bytes to the packetsize
            ReDim bytes(Me.PacketSize - 1)
            '-- Call BeginReceive again, catching any errors
            Try
                sock.BeginReceive(bytes, 0, bytes.Length, SocketFlags.None, _
                   AddressOf ReceiveCallback, bytes)
            Catch ex As Exception
                '-- If there was an exception, raise the Exception event
                '   This is better than throwing an exception, because there
                '   is most likely no code to catch the exception
                Dim args() As Object = {ex}
                Dim d As New RaiseExceptionEvent(AddressOf OnException)
                _syncrhonizingObject.Invoke(d, args)

                '-- Are we still connected? If not, raise the Connected event
                If IsConnected() = False Then
                    args(0) = False
                    Dim dl As New RaiseConnectedEvent(AddressOf OnConnected)
                    _syncrhonizingObject.Invoke(dl, args)
                End If
            End Try
        End If
    End Sub
```

Summary

Well, that's about all there is to it. I hope you stuck with me and went through that code. It's really not that difficult to understand. Using sockets can mean a significant increase in performance and efficiency in your connected .NET applications. Last time I checked, .NET was all about being connected. What could be more connected than a socket?

In my next article, I'll show you how to write multithreaded server applications!

Now, go write some code!