Peer-to-Peer with VB .NET

MATTHEW MACDONALD

Apress™

Peer-to-Peer with VB .NET
Copyright ©2003 by Matthew MacDonald

ISBN (pbk): 1-59059-105-4

Printed and bound in the United States of America 12345678910

Technical Reviewer: Ron Miller

Editorial Board: Dan Appleman, Craig Berry, Gary Cornell, Tony Davis, Steven Rycroft, Julian Skinner, Martin Streicher, Jim Sumser, Karen Watterson, Gavin Wright, John Zukowski

Assistant Publisher: Grace Wong

Project Manager: Beth Christmas

Copy Editor: Mark Nigara

Production Manager: Kari Brooks

Production Editor: Lori Bring

Compositor and Proofreader: Kinetic Publishing Services, LLC

Indexer: Kevin Broccoli

Cover Designer: Kurt Krames

Manufacturing Manager: Tom Debolski

Distributed to the book trade in the United States by Springer-Verlag New York, Inc., 175 Fifth Avenue, New York, NY, 10010 and outside the United States by Springer-Verlag GmbH & Co. KG, Tiergartenstr. 17, 69112 Heidelberg, Germany.

In the United States: phone 1-800-SPRINGER, email `orders@springer-ny.com`, or visit `http://www.springer-ny.com`. Outside the United States: fax +49 6221 345229, email `orders@springer.de`, or visit `http://www.springer.de`.

For information on translations, please contact Apress directly at 2560 Ninth Street, Suite 219, Berkeley, CA 94710. Phone 510-549-5930, fax 510-549-5939, email `info@apress.com`, or visit `http://www.apress.com`.

The source code for this book is available to readers at `http://www.apress.com` in the Downloads section.

CHAPTER 4

Building a Simple Messenger

THE LAST CHAPTER CONDUCTED a whirlwind tour of Remoting, .NET's object-based model for communication between applications and across a network. Remoting is a surprisingly flexible technology. By default, it's tailored for traditional enterprise computing, in which all the work is performed by a central group of powerful server computers. But with a little more effort, you can use Remoting as the basis for a peer-to-peer system that uses brokered communication. In this chapter, we'll explore one such example with an instant-messaging application that relies on a central coordinator. Along the way, you'll learn the advantages and drawbacks involved with using Remoting in a peer-to-peer project.

Though Remoting is fairly easy to use, there can be a fair bit of subtlety involved in using it *correctly*. In the example presented in this chapter, it's easy to ignore threading and concurrency problems, scalability considerations, and security. These details are explored in more detail in the next chapter. In this chapter, however, we'll concentrate on creating a basic, reliable framework for a messaging application based on Remoting.

Because the code is quite lengthy, it won't be presented in this chapter all at once. Instead, it's broken down and dissected in detail throughout the chapter. But before we consider a single line of code, we need to plan the overall architecture of the system, which we'll call Talk .NET.

Envisioning Talk .NET

Every Internet user is familiar with the basic model for an instant-messaging application. Users log on to some sort of central authority, retrieve a list that indicates who else is currently online, and exchange simple text messages. Some messaging platforms include additional enhancements, such as file-transfer features and group conversations that can include more than two parties.

All current-day instant-messaging applications rely on some sort of central component that stores a list of who is currently online as well as the information needed to contact them. Depending on the way the system is set up, peers may retrieve this information and contact a chosen user directly, or they may route all activity through the central coordinator. This chapter will consider both alternatives. We'll use the central coordinator approach first.

Conceptually, there are two types of applications in Talk .NET: the single server and the clients (or peers). Both applications must be divided into two parts: a remotable MarshalByRefObject that's exposed to the rest of the world and used for communication over the network, and a private portion, which manages the user interface and local user interaction. The server runs continuously at a fixed, well-known location, while the clients are free to appear and disappear on the network. Figure 4-1 diagrams these components.

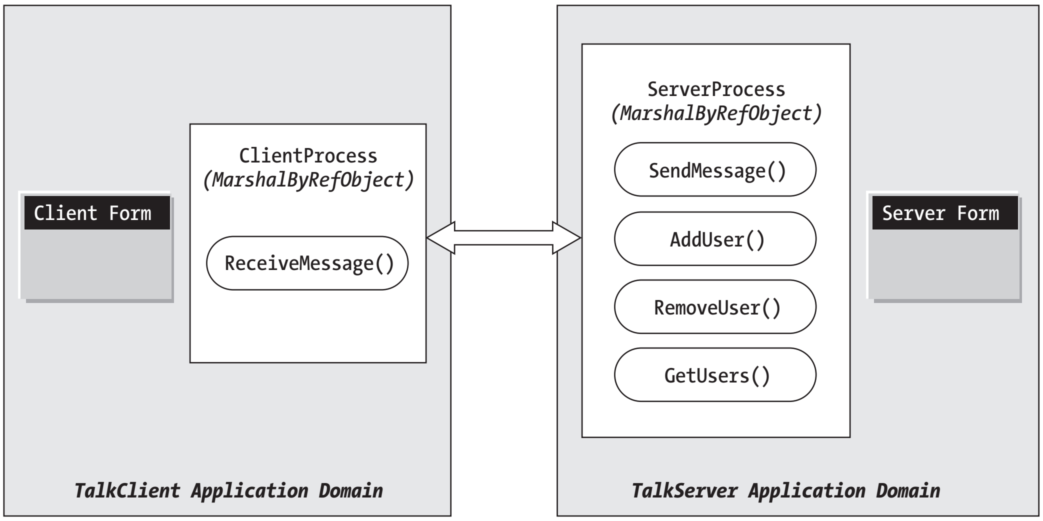


Figure 4-1. Components of the Talk .NET system

In order for the server to contact the client, the client must maintain an open bidirectional channel. When a message arrives, the server notifies the client. This notification can take place in several ways—it might use a callback or event, or the server could just call a method on the client object or interface, which is the approach taken in Talk .NET. Communication between these components uses TCP channels and binary formatting in our example, although these details are easy enough to change through the configuration files.

One of the most important aspects of the Talk .NET design is the fact that it uses interfaces to manage the communication process. Interfaces help to standardize how any two objects interact in a distributed system. Talk .NET includes two interfaces: ITalkServer, which defines the methods that a client can call on the server, and ITalkClient, which defines the methods that the server (or another client) can call on a client. Before actually writing the code for the Talk .NET components, we'll define the functionality by creating these interfaces.

NOTE *You can examine the full code for Talk .NET with the online samples for this chapter. There are a total of four projects that make up this solution; each is contained in a separate directory under the* Talk .NET *directory.*

Defining the Interfaces

The first step in creating the system is to lock down the methods that will be used for communication between the server and client components. These interfaces must be created in a separate DLL assembly so that they can be used by both the TalkClient and TalkServer applications. In the sample code, this class library project is called TalkComponent. It contains the following code:

```
Public Interface ITalkServer

    ' These methods allow users to be registered and unregistered
    ' with the server.
    Sub AddUser(ByVal [alias] As String, ByVal callback As ITalkClient)
    Sub RemoveUser(ByVal [alias] As String)

    ' This returns a collection of user names that are currently logged in.
    Function GetUsers() As ICollection

    ' The client calls this to send a message to the server.
    Sub SendMessage(ByVal senderAlias As String, _
      ByVal recipientAlias As String, ByVal message As String)

End Interface

Public Interface ITalkClient

    ' The server calls this to forward a message to the appropriate client.
    Sub ReceiveMessage(ByVal message As String, ByVal senderAlias As String)

End Interface

' This delegate is primarily for convenience on some server-side code.
Public Delegate Sub ReceiveMessageCallback(ByVal message As String, _
  ByVal senderAlias As String)
```

> **TIP** *Remember to consider security when designing the interfaces. The interfaces define the methods that will be exposed publicly to other application domains. Don't include any methods that you don't want a user at another computer to be able to trigger.*

ITalkServer defines the basic AddUser() and RemoveUser() methods for registering and unregistering users. It also provides a GetUsers() method that allows peers to retrieve a complete list of online users, and a SendMessage() method that actually routes a message from one peer to another. When SendMessage() is invoked, the server calls the ReceiveMessage() method of the ITalkClient interface to deliver the information to the appropriate peer.

Finally, the ReceiveMessageCallback delegate represents the method signature for the ITalkClient.ReceiveMessage() method. Strictly speaking, this detail isn't required. However, it makes it easier for the server to call the client asynchronously, as you'll see later.

One design decision has already been made in creating the interfaces. The information that's being transferred—the sender's user name and the message text—is represented by separate method parameters. Another approach would be to create a custom serializable Message object, which would be added to the TalkComponent project. Both approaches are perfectly reasonable.

Creating the TraceComponent

In Figure 4-1, both the client and the server are depicted as Windows applications. For the client, this design decision makes sense. For the server, however, it's less appropriate because it makes the design less flexible. For example, it might make more sense to implement the server component as a Windows service instead of a stand-alone application (as demonstrated in the next chapter).

A more loosely coupled option is possible. The server doesn't need to include any user-interface code. Instead, it can output messages to another source, such as the Windows event log. The Talk .NET server will actually output diagnostic messages using tracing code. These messages can then be dealt with in a variety of ways. They can be captured and recorded in a file, sent to an event log, shown in a console window, and so on. In the Talk .NET system, these messages will be caught by a custom trace listener, which will then display the trace messages in a Windows form. This approach is useful, flexible, and simple to code.

In .NET, any class can intercept, trace, and debug messages, provided it inherits from TraceListener in the System.Diagnostics namespace. This abstract class is the basis for DefaultTraceListener (which echoes messages to the Visual Studio .NET debugger), TextWriterTraceListener (which sends messages to a TextWriter or Stream, including a FileStream) and EventLogTraceListener (which records messages in the Windows event log).

All custom trace listeners work by overriding the Write() and WriteLine() methods. The entire process works like this:

1. The program calls a method such as Debug.Write() or Trace.Write().
2. The common language runtime (CLR) iterates through the current collection of debug listeners (Debug.Listeners) or trace listeners (Trace.Listeners).
3. Each time it finds a listener object, it calls its Write() or WriteLine() method with the message.

The solution used in this example creates a generic listener that forwards trace messages to a form, which then handles them appropriately. This arrangement is diagrammed in Figure 4-2.

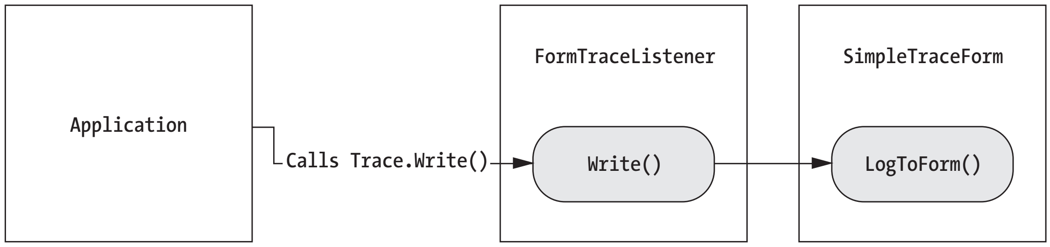


Figure 4-2. Forwarding trace messages to a form

The following is the outline for a FormTraceListener. This class is implemented in a separate class library project named TraceComponent.

```
' The form listener is a TraceListener object that
' maps trace messages to an ITraceForm instance, which
' will then display them in a window.
Public Class FormTraceListener
    Inherits TraceListener

    Public TraceForm As ITraceForm

    ' Use the default trace form.
    Public Sub New()
        MyBase.New()
        Me.TraceForm = New SimpleTraceForm()
    End Sub
    ' Use a custom trace form.
    Public Sub New(ByVal traceForm As ITraceForm)
        MyBase.New()

        If Not TypeOf traceForm Is Form Then
            Throw New InvalidCastException( _
              "ITraceForm must be used on a Form instance.")
        End If

        Me.TraceForm = traceForm
    End Sub

    Public Overloads Overrides Sub Write(ByVal value As String)
        TraceForm.LogToForm(value)
    End Sub

    Public Overloads Overrides Sub WriteLine(ByVal message As String)
        ' WriteLine() and Write() are equivalent in this simple example.
        Me.Write(message)
    End Sub

End Class
```

The FormTraceListener can send messages to any form that implements an ITraceForm interface, as shown here:

```
' Any custom form can be a "trace form" as long as it
' implements this interface.
Public Interface ITraceForm

    ' Determines how trace messages will be displayed.
    Sub LogToForm(ByVal message As String)

End Interface
```

Finally, the TraceComponent assembly also includes a sample form that can be used for debugging. It simply displays received messages in a list box and automatically scrolls to the end of the list each time a message is received.

```
Public Class SimpleTraceForm
    Inherits System.Windows.Forms.Form
    Implements ITraceForm

    ' (Designer code omitted.)
    Public Sub LogToForm(ByVal message As String) Implements ITraceForm.LogToForm
        ' Add the log message.
        lstMessages.Items.Add(message)

        ' Scroll to the bottom of the list.
        lstMessages.SelectedIndex = lstMessages.Items.Count - 1
    End Sub

End Class
```

This approach is useful for the Talk .NET server, but because it's implemented as a separate component, it can easily be reused in other projects.

The Coordination Server

Now that we've defined the basic building blocks for the Talk .NET system, it's time to move ahead and build the server. The TalkServer application has the task of tracking clients and routing messages from one user to another. The core of the application is implemented in the remotable ServerProcess class, which is provided to clients as a Singleton object. A separate module, called Startup, is used to start the TalkServer application. It initializes the Remoting configuration settings, creates and initializes an instance of the FormTraceListener, and displays the trace form modally. When the trace form is closed, the application ends, and the ServerProcess object is destroyed.

The startup code is shown here:

```
Imports System.Runtime.Remoting

Public Module Startup

    Public Sub Main()
        ' Create the server-side form (which displays diagnostic information).
        ' This form is implemented as a diagnostic logger.
        Dim frmLog As New TraceComponent.FormTraceListener()
        Trace.Listeners.Add(frmLog)

        ' Configure the connection and register the well-known object
        ' (ServerProcess), which will accept client requests.
        RemotingConfiguration.Configure("TalkServer.exe.config")
        ' From this point on, messages can be received by the ServerProcess
        ' object. The object will be created for the first request,
        ' although you could create it explicitly if desired.

        ' Show the trace listener form. By using ShowDialog(), we set up a
        ' message loop on this thread. The application will automatically end
        ' when the form is closed.
        Dim frm As Form = frmLog.TraceForm
        frm.Text = "Talk .NET Server (Trace Display)"
        frm.ShowDialog()
    End Sub

End Module
```

When you start the server, the ServerProcess Singleton object isn't created. Instead, it's created the first time a client invokes one of its methods. This will typically mean that the first application request will experience a slight delay, while the Singleton object is created.

The server configuration file is shown here. It includes three lines that are required if you want to run the Talk .NET applications under .NET 1.1 (the version of .NET included with Visual Studio .NET 2003). These lines enable full serialization, which allows the TalkServer to use the ITalkClient reference. If you are using .NET 1.0, these lines must remain commented out, because they will not be recognized. .NET 1.0 uses a slightly looser security model and allows full serialization support by default.

```
<configuration>
   <system.runtime.remoting>
      <application name="TalkNET">
         <service>
            <wellknown
               mode="Singleton"
               type="TalkServer.ServerProcess, TalkServer"
               objectUri="TalkServer" />
         </service>
         <channels>
            <channel port="8000" ref="tcp" >
               <!-- If you are using .NET 1.1, uncomment the lines below. -->
               <!--
               <serverProviders>
                   <formatter ref="binary" typeFilterLevel="Full" />
               </serverProviders>
               -->
            </channel>
         </channels>
      </application>
   </system.runtime.remoting>
</configuration>
```

Most of the code for the ServerProcess class is contained in the methods implemented from the ITalkServer interface. The basic outline is shown here:

```
Public Class ServerProcess
    Inherits MarshalByRefObject
    Implements ITalkServer

    ' Tracks all the user aliases, and the "network pointer" needed
    ' to communicate with them.
    Private ActiveUsers As New Hashtable()

    Public Sub AddUser(ByVal [alias] As String, ByVal client As ITalkClient) _
      Implements TalkComponent.ITalkServer.AddUser
        ' (Code omitted.)
    End Sub

    Public Sub RemoveUser(ByVal [alias] As String) _
      Implements TalkComponent.ITalkServer.RemoveUser
        ' (Code omitted.)
    End Sub

    Public Function GetUsers() As System.Collections.ICollection _
      Implements TalkComponent.ITalkServer.GetUsers
        ' (Code omitted.)
    End Function

    <System.Runtime.Remoting.Messaging.OneWay()> _
    Public Sub SendMessage(ByVal senderAlias As String, _
      ByVal recipientAlias As String, ByVal message As String) _
      Implements TalkComponent.ITalkServer.SendMessage
        ' (Code omitted.)
    End Sub

End Class
```

You'll see each method in more detail in the next few sections.

Tracking Clients

The Talk .NET server tracks clients using a Hashtable collection. The Hashtable provides several benefits compared to arrays or other types of collections:

- The Hashtable is a key/value collection (unlike some collections, which do not require keys). This allows you to associate two pieces of information: the user name and a network reference to the client.
- The Hashtable is optimized for quick key-based lookup. This is ideal, because users send messages based on the user's name. The server can speedily retrieve the client's location information.
- The Hashtable allows easy synchronization for thread-safe programming. We'll look at these features in the next chapter.

The collection stores ITalkClient references, indexed by user name. Technically, the ITalkClient reference really represents an instance of the System.Runtime.Remoting.ObjRef class. This class is a kind of network pointer—it contains all the information needed to generate a proxy object to communicate with the client, including the client channel, the object type, and the computer name. This ObjRef can be passed around the network, thus allowing any other user to locate and communicate with the client.

Following are the three collection-related methods that manage user registration. They're provided by the server.

```
Public Sub AddUser(ByVal [alias] As String, ByVal client As ITalkClient) _
  Implements TalkComponent.ITalkServer.AddUser
    Trace.Write("Added user '" & [alias] & "'")
    ActiveUsers([alias]) = client
End Sub

Public Sub RemoveUser(ByVal [alias] As String) _
  Implements TalkComponent.ITalkServer.RemoveUser
    Trace.Write("Removed user '" & [alias] & "'")
    ActiveUsers.Remove([alias])
End Sub

Public Function GetUsers() As System.Collections.ICollection _
  Implements TalkComponent.ITalkServer.GetUsers
    Return ActiveUsers.Keys
End Function
```

The AddUser() method doesn't check for duplicates. If the specified user name doesn't exist, a new entry is created. Otherwise, any entry with the same key is overwritten. The next chapter introduces some other ways to handle this behavior, but in a production application, you would probably want to authenticate users against a database with password information. This allows you to ensure that each user has a unique user name. If a user were to log in twice in a row, only the most recent connection information would be retained.

Note that only one part of the collection is returned to the client through the GetUsers() method: the user names. This prevents a malicious client from using the connection information to launch attacks against the peers on the system. Of course, this approach isn't possible in a decentralized peer-to-peer situation (wherein peers need to interact directly), but in this case, it's a realistic level of protection to add.

Sending Messages

The process of sending a message requires slightly more work. The server performs most of the heavy lifting in the SendMessage() method, which looks up the appropriate client and invokes its ReceiveMessage() method to deliver the message. If the recipient cannot be found (probably because the client has recently disconnected from the network), an error message is sent to the message sender by invoking *its* ReceiveMessage() method. If neither client can be found, the problem is harmlessly ignored.

```
Public Sub SendMessage(ByVal senderAlias As String, _
  ByVal recipientAlias As String, ByVal message As String) _
  Implements TalkComponent.ITalkServer.SendMessage

    ' Deliver the message.
    Dim Recipient As ITalkClient
    If ActiveUsers.ContainsKey(recipientAlias) Then
        Trace.Write("Recipient '" & recipientAlias & "' found")
        Recipient = CType(ActiveUsers(recipientAlias), ITalkClient)
    Else
        ' User wasn't found. Try to find the sender.
        If ActiveUsers.ContainsKey(senderAlias) Then
            Trace.Write("Recipient '" & recipientAlias & "' not found")
            Recipient = CType(ActiveUsers(senderAlias), ITalkClient)
            message = "'" & message & "' could not be delivered."
            senderAlias = "Talk .NET"
        Else
            Trace.Write("Recipient '" & recipientAlias & "' and sender '" & _
                        senderAlias & "' not found")
            ' Both sender and recipient weren't found.
            ' Ignore this message.
        End If
    End If

    Trace.Write("Delivering message to '" & recipientAlias & "' from '" & _
                senderAlias & "'")
    If Not Recipient Is Nothing Then
        Dim callback As New ReceiveMessageCallback( _
          AddressOf Recipient.ReceiveMessage)
        callback.BeginInvoke(message, senderAlias, Nothing, Nothing)
    End If

End Sub
```

You'll see that the server doesn't directly call the ClientProcess.ReceiveMessage() method because this would stall the thread and prevent it from continuing other tasks. Instead, it makes the call on a new thread by using the BeginInvoke() method provided by all delegates. It's possible to use a server-side callback to determine when this call completes, but in this case, it's not necessary.

This completes the basic framework for the TalkServer application. The next step is to build a client that can work with the server to send instant messages around the network.

The TalkClient

The client portion of Talk .NET is called TalkClient. It's designed as a Windows application (much like Microsoft's Windows Messenger). It has exactly two responsibilities: to allow the user to send a message to any other online user and to display a log of sent and received messages.

When the TalkClient application first loads, it executes a startup procedure, which presents a login form and requests the name of the user that it should register. If one isn't provided, the application terminates. Otherwise, it continues by taking two steps:

- It creates an instance of the ClientProcess class and supplies the user name. The ClientProcess class mediates all communication between the remote server and the client user interface.

- It creates and shows the main chat form, named Talk, around which most of the application revolves.

The startup code is shown here:

```
Public Class Startup

    Public Shared Sub Main()
        ' Create the login window (which retrieves the user identifier).
        Dim frmLogin As New Login()

        ' Only continue if the user successfully exits by clicking OK
        ' (not the Cancel or Exit button).
        If frmLogin.ShowDialog() = DialogResult.OK Then
            ' Create the new remotable client object.
            Dim Client As New ClientProcess(frmLogin.UserName)

            ' Create the client form.
            Dim frm As New Talk()
            frm.TalkClient = Client

            ' Show the form.
            frm.ShowDialog()
        End If
    End Sub

End Class
```

On startup, the ClientProcess object registers the user with the coordination server. Because ClientProcess is a remotable type, it will remain accessible to the server for callbacks throughout the lifetime of the application. These callbacks will, in turn, be raised to the user interface through local events. We'll dive into this code shortly.

The login form (shown in Figure 4-3) is quite straightforward. It exposes a public UserName property, which allows the Startup routine to retrieve the user name without violating encapsulation. This property could also be used to pre-fill the txtUser textbox by retrieving the previously used name, which could be stored in a configuration file or the Windows registry on the current computer.

```
Public Class Login
    Inherits System.Windows.Forms.Form

    ' (Designer code omitted.)

    Private Sub cmdExit_Click(ByVal sender As System.Object, _
      ByVal e As System.EventArgs) Handles cmdExit.Click
        Me.Close()
    End Sub

    Public Property UserName()
        Get
            Return txtUser.Text
        End Get
        Set(ByVal Value)
            txtUser.Text = UserName
        End Set
    End Property

End Class
```

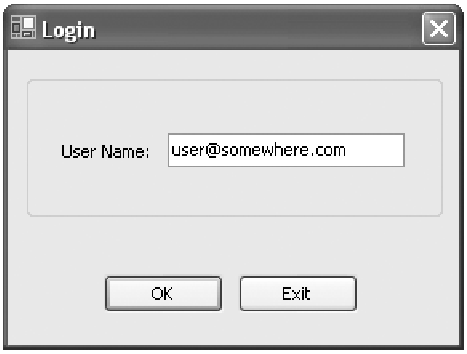


Figure 4-3. The login form

The Remotable ClientProcess Class

The ClientProcess class does double duty. It allows the TalkClient to interact with the TalkServer to register and unregister the user or send a message destined for another user. The ClientProcess also receives callbacks from the TalkServer and forwards these to the TalkClient through an event. In the Talk .NET system, the only time the TalkServer will call the ClientProcess is to deliver a message sent from another user. At this point, the ClientProcess will forward the message along to the user interface by raising an event. Because the server needs to be able to call ClientProcess.ReceiveMessage() across the network, the ClientProcess class must inherit from MarshalByRefObject. ClientProcess also implements ITalkClient.

Here's the basic outline for the ClientProcess class. Note that the user name is stored as a member variable named _Alias, and exposed through the public property Alias. Because alias is a reserved keyword in VB .NET, you will have to put this word in square brackets in the code.

```
Imports System.Runtime.Remoting
Imports TalkComponent

Public Class ClientProcess
    Inherits MarshalByRefObject
    Implements ITalkClient

    ' This event occurs when a message is received.
    ' It's used to transfer the message from the remotable
    ' ClientProcess object to the Talk form.
    Event MessageReceived(ByVal sender As Object, _
      ByVal e As MessageReceivedEventArgs)

    ' The reference to the server object.
    ' (Technically, this really holds a proxy class.)
    Private Server As ITalkServer

    ' The user ID for this instance.
    Private _Alias As String
    Public Property [Alias]() As String
        Get
            Return _Alias
        End Get
        Set(ByVal Value As String)
            _Alias = Value
        End Set
    End Property

    Public Sub New(ByVal [alias] As String)
        _Alias = [alias]
    End Sub
    ' This override ensures that if the object is idle for an extended
    ' period, waiting for messages, it won't lose its lease and
    ' be garbage collected.
    Public Overrides Function InitializeLifetimeService() As Object
        Return Nothing
    End Function

    Public Sub Login()
        ' (Code omitted.)
    End Sub

    Public Sub LogOut()
        ' (Code omitted.)
    End Sub

    Public Sub SendMessage(ByVal recipientAlias As String, _
      ByVal message As String)
        ' (Code omitted.)
    End Sub

    Private Sub ReceiveMessage(ByVal message As String, _
      ByVal senderAlias As String) Implements ITalkClient.ReceiveMessage
        ' (Code omitted.)
    End Sub

    Public Function GetUsers() As ICollection
        ' (Code omitted.)
    End Function

End Class
```

The InitializeLifetimeService() method must be overridden to preserve the life of all ClientProcess objects. Even though the startup routine holds a reference to a ClientProcess object, the ClientProcess object will still disappear from the network after its lifetime lease expires, unless you explicitly configure an infinite lifetime. Alternatively, you can use configuration file settings instead of overriding the InitializeLifetimeService() method, as described in the previous chapter.

One other interesting detail is found in the ReceiveMessage() method. This method is accessible remotely to the server because it implements ITalkClient.ReceiveMessage. However, this method is also marked with the Private keyword, which means that other classes in the TalkClient application won't accidentally attempt to use it.

The Login() method configures the client channel, creates a proxy to the server object, and then calls the ServerProcess.AddUser() method to register the client. The Logout() method simply unregisters the user, but it doesn't tear down the Remoting channels—that will be performed automatically when the application exits. Finally, the GetUsers() method retrieves the user names of all the users currently registered with the coordination server.

```
Public Sub Login()

    ' Configure the client channel for sending messages and receiving
    ' the server callback.
    RemotingConfiguration.Configure("TalkClient.exe.config")

    ' You could accomplish the same thing in code by uncommenting
    ' the following two lines:
    ' Dim Channel As New System.Runtime.Remoting.Channels.Tcp.TcpChannel(0) and
    ' ChannelServices.RegisterChannel(Channel).

    ' Create the proxy that references the server object.
    Server = CType(Activator.GetObject(GetType(ITalkServer), _
                   "tcp://localhost:8000/TalkNET/TalkServer"), ITalkServer)

    ' Register the current user with the server.
    ' If the server isn't  running, or the URL or class information is
    ' incorrect, an error will most likely occur here.
    Server.AddUser(_Alias, Me)

End Sub

Public Sub LogOut()
    Server.RemoveUser(_Alias)
End Sub

Public Function GetUsers() As ICollection
    Return Server.GetUsers()
End Function
```

Following is the client configuration, which only specified channel information. The client port isn't specified and will be chosen dynamically from the available ports at runtime. As with the server configuration file, you must enable full serialization if you are running the Talk .NET system with .NET 1.1. Otherwise, the TalkClient will not be allowed to transmit the ITalkClient reference over the network to the server.

```
<configuration>
   <system.runtime.remoting>
      <application>
         <channels>
            <channel port="0" ref="tcp" >
               <!-- If you are using .NET 1.1, uncomment the lines below. -->
               <!--
               <serverProviders>
                   <formatter ref="binary" typeFilterLevel="Full" />
               </serverProviders>
               -->
            </channel>
         </channels>
      </application>
   </system.runtime.remoting>
</configuration>
```

You'll notice that the Login() method mingles some dynamic Remoting code (used to create the TalkServer instance) along with a configuration file (used to create the client channel). Unfortunately, it isn't possible to rely exclusively on a configuration file when you use interface-based programming with Remoting. The problem is that the client doesn't have any information about the server, only an interface it supports. The client thus cannot register the appropriate object type and create it directly because there's no way to instantiate an interface. The previous solution, which uses the Activator.GetObject() method, forces you to include several distribution details in your code. This means that if the object is moved to another computer or exposed through another port, you'll need to recompile the code.

You can resolve this problem in several ways. One option is simply to add a custom configuration setting with the full object URI. This will be an application setting, not a Remoting setting, so it will need to be entered in the <appSettings> section of the client configuration file, as shown here:

```
<configuration>

  <appSettings>
    <add key="TalkServerURL"
         value="tcp://localhost:8000/TalkNET/TalkServer" />
    </appSettings>
    <system.runtime.remoting>
      <application>
         <channels>
            <channel port="0" ref="tcp" >
               <!-- If you are using .NET 1.1, uncomment the lines below. -->
               <!--
               <serverProviders>
                   <formatter ref="binary" typeFilterLevel="Full" />
               </serverProviders>
               -->
            </channel>
         </channels>
      </application>
    </system.runtime.remoting>

</configuration>
```

You can then retrieve this setting using the ConfigurationSettings.AppSettings collection:

```
  Server = CType(Activator.GetObject(GetType(ITalkServer), _
                 ConfigurationSettings.AppSettings("TalkServer")), ITalkServer)
```

Note that in this example, we use the loopback alias localhost, indicating that the server is running on the same computer. You should replace this value with the name of the computer (if it's on your local network), the domain name, or the IP address where the server component is running.

The last ingredient is the ClientProcess methods for sending and receiving messages. The following code shows the SendMessage() and ReceiveMessage() methods. The SendMessage() simply executes the call on the server and the ReceiveMessage() raises a local event for the client, which will be handled by the Talk form.

```
Public Sub SendMessage(ByVal recipientAlias As String, ByVal message As String)
    Server.SendMessage(_Alias, recipientAlias, message)
End Sub

Private Sub ReceiveMessage(ByVal message As String, _
  ByVal senderAlias As String) Implements ITalkClient.ReceiveMessage
    RaiseEvent MessageReceived(Me, New MessageReceivedEventArgs(message, _
                               senderAlias))
End Sub
```

The MessageReceived event makes use of the following custom EventArgs class, which adds the message-specific information:

```
Public Class MessageReceivedEventArgs
    Inherits EventArgs

    Public Message As String
    Public SenderAlias As String

    Public Sub New(ByVal message As String, ByVal senderAlias As String)
        Me.Message = message
        Me.SenderAlias = senderAlias
    End Sub

End Class
```

The Talk Form

The Talk form is the front-end that the user interacts with. It has four key tasks:

- Log the user in when the form loads and log the user out when the form closes.
- Periodically refresh the list of active users by calling ClientProcess.GetUsers(). This is performed using a timer.
- Invoke ClientProcess.SendMessage() when the user sends a message.
- Handle the MessageReceived event and display the corresponding information on the form.

The form is shown in Figure 4-4. Messages are recorded in a RichTextBox, which allows the application of formatting, if desired. The list of clients is maintained in a ListBox.

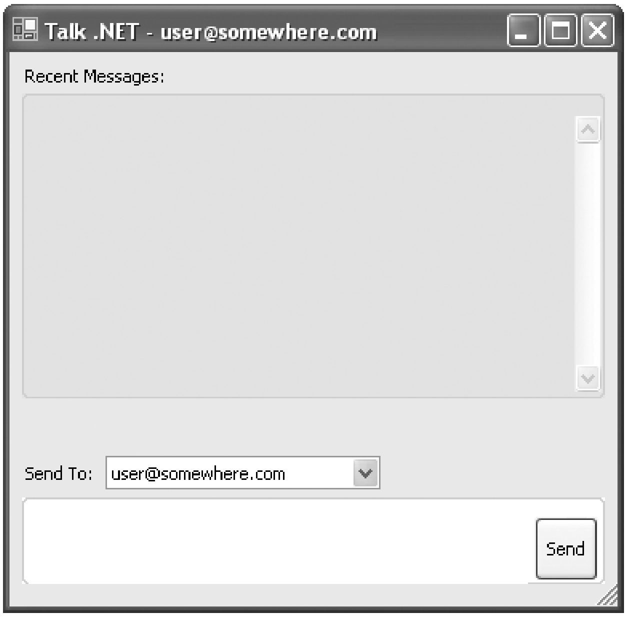


Figure 4-4. The Talk form

The full form code is shown here:

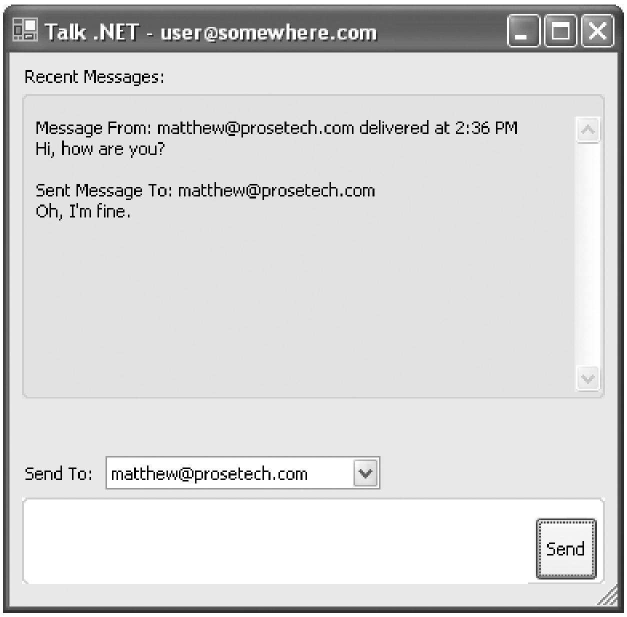
```
Public Class Talk
    Inherits System.Windows.Forms.Form

    ' (Designer code omitted.)

    ' The remotable intermediary for all client-to-server communication.
    Public WithEvents TalkClient As ClientProcess

    Private Sub Talk_Load(ByVal sender As System.Object, _
      ByVal e As System.EventArgs) Handles MyBase.Load

        Me.Text &= " - " & TalkClient.Alias

        ' Attempt to register with the server.
        TalkClient.Login()

        ' Ordinarily, a user list is periodically fetched from the
        ' server. In this case, the code enables the timer and calls it
        ' once (immediately) to initially populate the list box.
        tmrRefreshUsers_Tick(Me, EventArgs.Empty)
        tmrRefreshUsers.Enabled = True
        lstUsers.SelectedIndex = 0

    End Sub

    Private Sub TalkClient_MessageReceived(ByVal sender As Object, _
      ByVal e As MessageReceivedEventArgs) Handles TalkClient.MessageReceived

        txtReceived.Text &= "Message From: " & e.SenderAlias
        txtReceived.Text &= " delivered at " & DateTime.Now.ToShortTimeString()
        txtReceived.Text &= Environment.NewLine & e.Message
        txtReceived.Text &= Environment.NewLine & Environment.NewLine

    End Sub

    Private Sub cmdSend_Click(ByVal sender As System.Object, _
      ByVal e As System.EventArgs) Handles cmdSend.Click

        ' Display a record of the message you're sending.
        txtReceived.Text &= "Sent Message To: " & lstUsers.Text
        txtReceived.Text &= Environment.NewLine & txtMessage.Text
        txtReceived.Text &= Environment.NewLine & Environment.NewLine

        ' Send the message through the ClientProcess object.
        Try
            TalkClient.SendMessage(lstUsers.Text, txtMessage.Text)
            txtMessage.Text = ""
        Catch Err As Exception
            MessageBox.Show(Err.Message, "Send Failed", _
                            MessageBoxButtons.OK, MessageBoxIcon.Exclamation)
        End Try

    End Sub

    ' Checks every 30 seconds.
    Private Sub tmrRefreshUsers_Tick(ByVal sender As System.Object, _
      ByVal e As System.EventArgs) Handles tmrRefreshUsers.Tick

        ' Prepare list of logged-in users.
        ' The code must copy the ICollection entries into
        ' an ordinary array before they can be added.
        Dim UserArray() As String
        Dim UserCollection As ICollection = TalkClient.GetUsers
        ReDim UserArray(UserCollection.Count - 1)
        UserCollection.CopyTo(UserArray, 0)
        ' Replace the list entries. At the same time,
        ' the code will track the previous selection and try
        ' to restore it, so the update won't be noticeable.
        Dim CurrentSelection As String = lstUsers.Text
        lstUsers.Items.Clear()
        lstUsers.Items.AddRange(UserArray)
        lstUsers.Text = CurrentSelection

    End Sub

    Private Sub Talk_Closed(ByVal sender As Object, _
      ByVal e As System.EventArgs) Handles MyBase.Closed
        TalkClient.LogOut()
    End Sub

End Class
```

The timer fires and refreshes the list of user names seamlessly every 30 seconds. In a large system, you would lower this value to ease the burden on the coordinator. For a very large system with low user turnover, it might be more efficient to have the server broadcast user-added and user-removed messages. To support this infrastructure, you would add methods such as ITalkClient.NotifyUserAdded() and ITalkClient.NotifyUserRemoved(). Or you might just use a method such as ITalkClient.NotifyListChanged(), which tells the client that it must contact the server at some point to update its information.

The ideal approach isn't always easy to identify. The goal is to minimize the network chatter as much as possible. In a system with 100 users who query the server every 60 seconds, approximately 100 request messages and 100 response messages will be sent every minute. If the same system adopts user-added and user-removed broadcasting instead, and approximately 5 users join or leave the system in a minute, the server will likely need to send 5 messages to each of 100 users, for a much larger total of 500 messages per minute. The messages themselves would be smaller (because they would not contain the full user list), but the network overhead would probably be great enough that this option would work less efficiently.

In a large system, you might use "buddy lists" so that clients only receive a user list with a subset of the total number of users. In this case, the server broadcast approach would be more efficient because a network exchange would only be required for those users who are on the same list as the entering or departing peer. This reduces the total number of calls dramatically. Overall, this is probably the most sustainable option if you want to continue to develop the Talk .NET application to serve a larger audience.

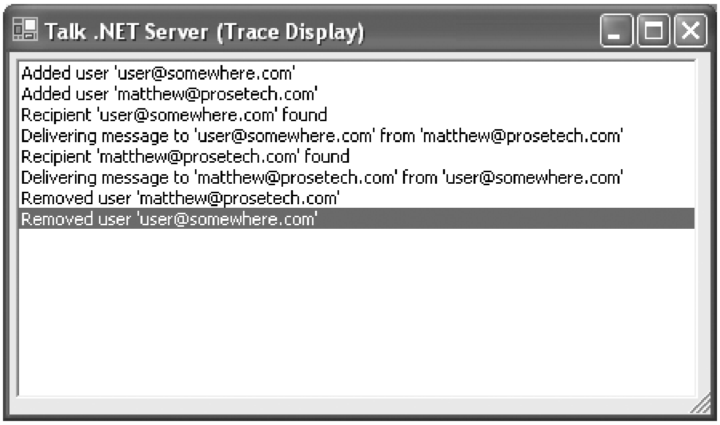
Because the client chooses a channel dynamically, it's possible to run several instances of the TalkClient on the same computer. After starting the new instances, the user list of the original clients will quickly be refreshed to represent the full user list. You can then send messages back and forth, as shown in Figure 4-5. Clients can also send messages to themselves.



```
Message From: matthew@prosetech.com delivered at 2:36 PM
Hi, how are you?

Sent Message To: matthew@prosetech.com
Oh, I'm fine.
```

Figure 4-5. Multiple client interaction

In each case, the coordination server brokers the communication. The trace output for a sample interaction on the server computer is shown in Figure 4-6.



```
Added user 'user@somewhere.com'
Added user 'matthew@prosetech.com'
Recipient 'user@somewhere.com' found
Delivering message to 'user@somewhere.com' from 'matthew@prosetech.com'
Recipient 'matthew@prosetech.com' found
Delivering message to 'matthew@prosetech.com' from 'user@somewhere.com'
Removed user 'matthew@prosetech.com'
Removed user 'user@somewhere.com'
```

Figure 4-6. The server trace display

Enhancing Talk .NET

Talk .NET presents a straightforward way to reinvent the popular instant-messaging application in .NET code. However, as it currently stands, it's best suited for small groups of users and heavily reliant on a central coordination server. In fact, in many respects it's hard to call this a true peer-to-peer application at all.

Fortunately, Talk .NET is just a foundation that you can build on. This section considers possible enhancements, stumbling blocks, and a minor redesign that allows true peer-to-peer communication.

Cleaning Up After Clients

Currently, the system assumes that all clients will log out politely when they've finished using the system. Due to network problems, program error, or some other uncontrollable factor, this may not be the case. Remember, one of the defining characteristics of any peer-to-peer system is that it must take into account the varying, fragile connectivity of users on the Internet. For this reason, Talk .NET needs to adopt a more defensive approach.

Currently, the SendMessage() method raises an unhandled exception if it can't contact the specified user. This exception will propagate back to the user-interface code, where it will be handled and will result in a user error message. The problem with this approach is that the user remains in the server's collection and continues to "appear" online. If another user attempts to send a message to this user, valuable server seconds will be wasted attempting to contact the offline user, thereby raising the exception. This problem will persist until the missing user logs back in to the system.

To account for this problem, users should be removed from the collection if they cannot be contacted. Here's the important portion of the SendMessage() code, revised accordingly:

```
If Not Recipient Is Nothing Then

    Dim callback As New ReceiveMessageCallback( _
      AddressOf Recipient.ReceiveMessage)

    Try
        callback.BeginInvoke(message, senderAlias, Nothing, Nothing)
    Catch Err As Exception
        ' Client could not be contacted.
        Trace.Write("Message delivery failed")
        ActiveUsers.Remove(recipientAlias)
    End Try

End If
```

You may also want to send a message explaining the problem to the user. However, you also need to protect yourself in case the user who sent the message can't be contacted or found. To prevent the code from becoming too fragmented, you can rewrite it using recursion, as shown here:

```
Public Sub SendMessage(ByVal senderAlias As String, _
  ByVal recipientAlias As String, ByVal message As String) _
  Implements TalkComponent.ITalkServer.SendMessage

    Dim Recipient As ITalkClient
    If ActiveUsers.ContainsKey(recipientAlias) Then
        Trace.Write("Recipient '" & recipientAlias & "' found")
        Recipient = CType(ActiveUsers(recipientAlias), ITalkClient)

        If Not Recipient Is Nothing Then

            Trace.Write("Delivering message to '" & recipientAlias & "' from _
                         '" & senderAlias & "'")
            Dim callback As New ReceiveMessageCallback( _
              AddressOf Recipient.ReceiveMessage)

            ' Deliver the message.
            Try
                callback.BeginInvoke(message, senderAlias, Nothing, Nothing)

            Catch Err As Exception
                ' Client could not be contacted.
                ActiveUsers.Remove(recipientAlias)

                If senderAlias <> "Talk .NET"
                   ' Try to send a warning message.
                   message = "'" & message & "' could not be delivered."
                   SendMessage("Talk .NET", senderAlias, message)

            End Try
        End If

    Else
        ' User was not found. Try to find the sender.
        Trace.Write("Recipient '" & recipientAlias & "' not found")
        If senderAlias <> "Talk .NET"
            ' Try to send a warning message.
            message = "'" & message & "' could not be delivered."
            SendMessage("Talk .NET", senderAlias, message)
        End If

    End If

End Sub
```

Of course, in order for this approach to work, you'll need to ensure that no other user can take the user name "Talk .NET." You could add this restriction in your logon or authentication code.

Toward Decentralization

Talk .NET will always requires some sort of centralized server component in order to store information about logged-on users and their locations. However, it's not necessary to route all communication through the server. In fact, Remoting allows clients to communicate directly—with a few quirks.

Remoting is designed as an object-based networking technology. In order for clients to communicate directly, they need to have a reference to each other's remotable ClientProcess object. As you've already learned, you can create this reference through a configuration file or .NET Remoting code, if you know the appropriate URL. This is how the client contacts the coordination server in the Talk .NET system—by knowing the computer and port where it's located. But there's also another approach: by passing an object reference. The server calls the client back by using one of its stored ITalkClient references.

The ITalkClient reference isn't limited to exchanges between the server and client. In fact, this reference can be passed to any computer on the network. Because ITalkClient references a remotable object (in this case, ClientProcess), whenever the reference travels to another application domain, it actually takes the form of an ObjRef: a network pointer that encapsulates all the information needed to describe the object and its location on the network. With this information, any .NET application can dynamically construct a proxy and communicate with the client it references. You can use the ObjRef as the basis for decentralized communication.

To see this in action, modify the ITalkServer interface to expose an additional method that returns an ITalkClient reference for a specific user:

```
Public Interface ITalkServer

    ' (Other code omitted.)
    Function GetUser(ByVal [alias] As String) As ITalkClient

End Interface
```

Now, implement the GetUser() method in the ServerProcess class:

```
Public Function GetUser(ByVal [alias] As String) As TalkComponent.ITalkClient _
  Implements TalkComponent.ITalkServer.GetUser

    Return ActiveUsers([alias])

End Function
```

Now the ClientProcess class can call GetUser() to retrieve the ITalkUser reference of the peer it wants to communicate with; it can then call the ITalkClient.ReceiveMessage() method directly:

```
Public Sub SendMessage(ByVal recipientAlias As String, ByVal message As String)

    Dim Peer As ITalkClient = Server.GetUser(recipientAlias)
    Peer.ReceiveMessage(message, Me.Alias)

End Sub
```

With this change in place, the system will work exactly the same. However, the coordination server is now simply being used as a repository of connection information. Once the lookup is performed, it's no longer required.

NOTE *You can find this version of the application in the* Talk .NET Decentralized *directory with the online samples for this chapter.*

Which approach is best? There's little doubt that the second choice is more authentically peer-to-peer. But the best choice for your system depends on your needs. Some of the benefits of the server-focused approach include the following:

- The server can track system activity, which could be useful, depending on your reporting needs. If you run the second version of this application, you'll see that the server trace log reflects when users are added or removed, but it doesn't contain any information when messages are sent.

- The connectivity is likely to be better. Typically, if a client can contact the server, the server will be able to call the client. However, two arbitrary clients may not be able to interact, depending on firewalls and other aspects of network topology.

- The server can offer some special features that wouldn't be possible in a decentralized system, such as multiuser broadcasts that involve thousands of users.

On the other hand, the benefits of the decentralized approach include the following:

- The server has no ability to monitor conversations. This translates into better security (assuming peers don't fully trust the behavior of the server).

- The possibility for a server bottleneck decreases. This is because the server isn't called on to deal with messages, but rather, only to provide client lookup, thereby reducing its burden and moving network traffic out to the edges of the network.

Most peer-to-peer supporters would prefer the decentralized approach. However, the current generation of instant-messaging applications avoid it for connectivity reasons. Instead, they use systems that more closely resemble the client-server model.

In some cases you might want to adopt a blended approach that makes use of both of these techniques. One option is to allow the client to specify the behavior through a configuration setting. Another option would be to use peer-to-peer communication only when large amounts of data need to be transmitted. This is the approach used in the next section to provide a file transfer service for Talk .NET.

In any case, if you adopt the decentralized approach, you can further reduce the burden on the central coordinator by performing the client lookup once, and then reusing the connection information for all subsequent messages. For example, you could cache the retrieved client reference in a local ActiveUsers collection, and update it from the server if an error is encountered while sending a message. Or, you might modify the system so that the GetUsers() method returns the entire collection, complete with user names and ITalkClient network pointers. The central coordinator would then simply need to support continuous requests to three methods: AddUser(), RemoveUser(), and GetUsers(). This type of design works well if you use "buddy lists" to determine who a user can communicate with. That way, users will only retrieve information about a small subset of the total number of users when they call GetUsers().

Adding a File Transfer Feature

Using the decentralized approach, it's easy to implement a file transfer feature that's similar to the one provided by Microsoft's Windows Messenger. This feature wouldn't be practical with the centralized approach because it encourages the server to become a bottleneck. Although transferring files isn't a complex task, it can take time, and the CLR only provides a limited number of threads to handle server requests. If all the threads are tied up with sending data across the network (or waiting as data is transferred over a low-bandwidth connection), subsequent requests will have to wait—and could even time out.

The file transfer operation can be broken down into four steps:

1. Peer A offers a file to Peer B.

2. Peer B accepts the file offer and initiates the transfer.

3. Peer A sends the file to Peer B.

4. Peer B saves the file locally in a predetermined directory.

These steps require several separate method calls. Typically, in step 2, the user will be presented with some sort of dialog box asking whether the file should be transferred. It's impractical to leave the connection open while this message is being displayed because there's no guarantee the user will reply promptly, and the connection could time out while waiting. Instead, the peer-to-peer model requires a looser, disconnected architecture that completely separates the file offer and file transfer.

The first step needed to implement the file transfer is to redefine the ITalkClient interface. It's at this point that most of the coding and design decisions are made.

```
Public Interface ITalkClient

    ' (Other code omitted.)
    Sub ReceiveFileOffer(ByVal filename As String, _
      ByVal fileIdentifier As Guid, ByVal senderAlias As String)
    Function TransferFile(ByVal fileIdentifier As Guid, _
      ByVal senderAlias As String) As Byte()

End Interface
```

You'll notice that both methods use a globally unique identifier (GUID) to identify the file. There are several reasons for this approach, all of which revolve around security. If the TransferFile() method accepted a full file name, it would be possible for the client to initiate a transfer even if the file had not been offered, thereby compromising data security. To circumvent this problem, all files are identified uniquely. The identifier used is a GUID, which guarantees that a client won't be able to guess the identifier for a file offered to another user. Also, because GUIDs are guaranteed to be unique, a peer can offer multiple files to different users without confusion. More elaborate security approaches are possible, but this approach is a quick and easy way to prevent users from getting ahold of the wrong files.

The file itself is transferred as a large byte array. While this will be sufficient in most cases, if you want to control how the data is streamed over the network, you'll need to use a lower-level networking class, such as the ones described in the second part of this book.

Once the ITalkClient interface is updated, you can begin to revise the ClientProcess class. The first step is to define a Hashtable collection that can track all the outstanding file offers since the application was started:

```
Private OfferedFiles As New Hashtable()
```

To offer a file, the TalkClient calls the public SendFileOffer() method. This method looks up the client reference, generates a new GUID to identify the file, stores the information, and sends the offer.

```
Public Function SendFileOffer(ByVal recipientAlias As String, _
  ByVal sourcePath As String)

    ' Retrieve the reference to the other user.
    Dim peer As ITalkClient = Server.GetUser(recipientAlias)

    ' Create a GUID to identify the file, and add it to the collection.
    Dim fileIdentifier As Guid = Guid.NewGuid()
    OfferedFiles(fileIdentifier) = sourcePath
    ' Offer the file.
    peer.ReceiveFileOffer(Path.GetFileName(sourcePath), fileIdentifier, Me.Alias)

End Function
```

Notice that only the file name is transmitted, not the full file path. The full file path is stored for future reference in the Hashtable collection, but it's snipped out of the offer using the Path class from the System.IO namespace. This extra step is designed to prevent the recipient from knowing where the offered file is stored on the offering peer.

TIP *Currently, the TalkClient doesn't go to any extra work to "expire" an offered file and remove its information from the collection if it isn't transferred within a set period of time. This task could be accomplished using a separate thread that would periodically examine the collection. However, because the in-memory size of the OfferedFiles collection will always remain relatively small, this isn't a concern, even after making a few hundred unclaimed file offers.*

The file offer is received by the destination peer with the ReceiveFileOffer() method. When this method is triggered, the ClientProcess class raises a local event to alert the user:

```
Event FileOfferReceived(ByVal sender As Object, _
  ByVal e As FileOfferReceivedEventArgs)

Private Sub ReceiveFileOffer(ByVal filename As String, _
  ByVal fileIdentifier As System.Guid, ByVal senderAlias As String) _
  Implements TalkComponent.ITalkClient.ReceiveFileOffer

    RaiseEvent FileOfferReceived(Me, _
      New FileOfferReceivedEventArgs(filename, fileIdentifier, senderAlias))

End Sub
```

The FileOfferReceivedEventArgs class simply provides the file name, file identifier, and sender's name:

```
Public Class FileOfferReceivedEventArgs
    Inherits EventArgs
    Public Filename As String
    Public FileIdentifier As Guid
    Public SenderAlias As String

    Public Sub New(ByVal filename As String, ByVal fileIdentifier As Guid, _
      ByVal senderAlias As String)
        Me.Filename = filename
        Me.FileIdentifier = fileIdentifier
        Me.SenderAlias = senderAlias
    End Sub

End Class
```

The event is handled in the form code, which will then ask the user whether the transfer should be accepted. If it is, the next step is to call the ClientProcess.AcceptFile() method, which initiates the transfer.

```
Public Sub AcceptFile(ByVal recipientAlias As String, _
  ByVal fileIdentifier As Guid, ByVal destinationPath As String)

    ' Retrieve the reference to the other user.
    Dim peer As ITalkClient = Server.GetUser(recipientAlias)

   ' Create an array to store the data.
    Dim FileData As Byte()

    ' Request the file.
    FileData = peer.TransferFile(fileIdentifier, Me.Alias)
    Dim fs As FileStream

    ' Create the local copy of the file in the desired location.
    ' Warning: This method doesn't bother to check if it's overwriting
    ' a file with the same name.
    fs = File.Create(destinationPath)
    fs.Write(FileData, 0, FileData.Length)

    ' Clean up.
    fs.Close()

End Sub
```

There are several interesting details in this code:

- It doesn't specify the destination file path and file name. This information is supplied to the AcceptFile() method through the destinationPath parameter. This allows the form code to stay in control, perhaps using a default directory or prompting the user for a destination path.
- It includes no exception-handling code. The assumption is that the form code will handle any errors that occur and inform the user accordingly.
- It doesn't worry about overwriting any file that may already exist at the specified directory with the same name. Once again, this is for the form code to check. It will prompt the user before starting the file transfer.

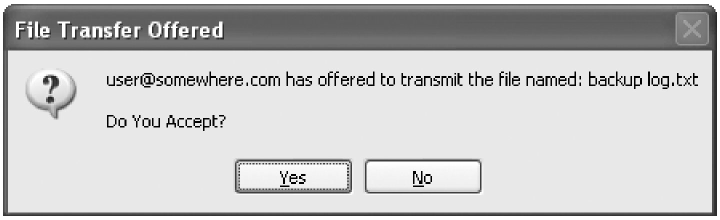
The peer offering the file sends it over the network in its TransferFile() method, which is in many ways a mirror image of AcceptFile().

```
Private Function TransferFile(ByVal fileIdentifier As System.Guid, _
  ByVal senderAlias As String) As Byte() _
  Implements TalkComponent.ITalkClient.TransferFile

    ' Ensure that the GUID corresponds to a valid file offer.
    If Not OfferedFiles.Contains(fileIdentifier) Then
        Throw New ApplicationException( _
          "This file is no longer available from the client.")
    End If

    ' Look up the file path from the OfferedFiles collection and open it.
    Dim fs As FileStream
    fs = File.Open(OfferedFiles(fileIdentifier), FileMode.Open)

    ' Fill the FileData byte array with the data from the file.
    Dim FileData As Byte()
    ReDim FileData(fs.Length)
    fs.Read(FileData, 0, FileData.Length)

    ' Remove the offered file from the collection.
    OfferedFiles.Remove(fileIdentifier)

    ' Clean up.
    fs.Close()
    ' Transmit the file data.
    Return FileData

End Function
```

The only detail we haven't explored is the layer of user-interface code in the Talk form. The first step is to add an "Offer File" button that allows the user to choose a file to send. The file is chosen using the OpenFileDialog class.

```
Private Sub cmdOffer_Click(ByVal sender As System.Object, _
  ByVal e As System.EventArgs) Handles cmdOffer.Click

    ' Prompt the user for a file to offer.
    Dim dlgOpen As New OpenFileDialog()
    dlgOpen.Title = "Choose a File to Transmit"

    If dlgOpen.ShowDialog() = DialogResult.OK Then
        Try
            ' Send the offer.
            TalkClient.SendFileOffer(lstUsers.Text, dlgOpen.FileName)
        Catch Err As Exception
            MessageBox.Show(Err.Message, "Send Failed", _
                            MessageBoxButtons.OK, MessageBoxIcon.Exclamation)
        End Try
    End If

End Sub
```

The Talk form code also handles the FileOfferReceived event, prompts the user, and initiates the transfer if accepted (see Figure 4-7).

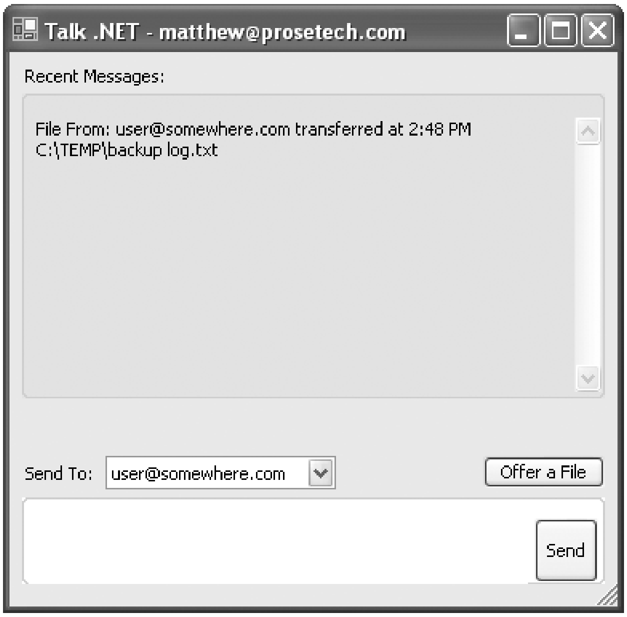

File Transfer Offered

user@somewhere.com has offered to transmit the file named: backup log.txt

Do You Accept?

Figure 4-7. Offering a file transfer

```
Private Sub TalkClient_FileOfferReceived(ByVal sender As Object, _
  ByVal e As TalkClient.FileOfferReceivedEventArgs) _
  Handles TalkClient.FileOfferReceived

    ' Create the user message describing the file offer.
    Dim Message As String
    Message = e.SenderAlias & " has offered to transmit the file named: "
    Message &= e.Filename & Environment.NewLine
    Message &= Environment.NewLine & "Do You Accept?"

    ' Prompt the user.
    Dim Result As DialogResult = MessageBox.Show(Message, _
      "File Transfer Offered", MessageBoxButtons.YesNo, MessageBoxIcon.Question)

    If Result = DialogResult.Yes Then

        Try
            ' The code defaults to the TEMP directory, although a more
            ' likely option would be to read information from a registry or
            ' configuration file setting.
            Dim DestinationPath As String = "C:\TEMP\" & e.Filename

            ' Receive the file.
            TalkClient.AcceptFile(e.SenderAlias, e.FileIdentifier, _
                                  DestinationPath)

            ' Assuming no error occurred, display information about it
            ' in the chat window.
            txtReceived.Text &= "File From: " & e.SenderAlias
            txtReceived.Text &= " transferred at "
            txtReceived.Text &= DateTime.Now.ToShortTimeString()
            txtReceived.Text &= Environment.NewLine & DestinationPath
            txtReceived.Text &= Environment.NewLine & Environment.NewLine

        Catch Err As Exception
            MessageBox.Show(Err.Message, "Transfer Failed", _
                            MessageBoxButtons.OK, MessageBoxIcon.Exclamation)

        End Try

    End If

End Sub
```



File From: user@somewhere.com transferred at 2:48 PM
C:\TEMP\backup log.txt

Figure 4-8. A completed file transfer

NOTE *Adding a file transfer feature such as this one is a notorious security risk. Because the communication is direct, there's no way to authenticate the recipient. (A central server, on the other hand, could verify that users are who they claim to be.) That means that a file could be offered to the wrong user or a malicious user who is impersonating another user. To reduce the risk, the server component could require user ID and password information before returning any information from the GetUsers() collection. We'll deal with security more closely in Chapter 11.*

Scalability Challenges with the Simple Implementation

In its current form, the Talk .NET application is hard pressed to scale in order to serve a large audience. The key problem is the server component, which could become a critical bottleneck as the traffic increases. To reduce this problem, you can switch to the decentralized approach described earlier, although this is only a partial solution. It won't deal with the possible problems that can occur if the number of users grows so large that storing them in an in-memory hashtable is no longer effective.

Databases and a Stateless Server

To combat this problem, you would need to store the list of logged-on users and their connection information in an external data store such as a database. This would reduce the performance for individual calls (because they would require database lookups), but it would increase the overall scalability of the system (because the memory overhead would be lessened).

This approach also allows you to create a completely stateless coordination server. In this case, you could replace your coordination server by a web farm of computers, each of which would access the same database. Each client request could be routed to the computer with the least traffic, guaranteeing performance. Much of the threading code presented in the next chapter would not be needed anymore, because all of the information would be shared in a common database that would provide its own concurrency control. In order to create the cluster farm and expose it under a single IP, you would need to use hardware clustering or a software load-balancing solution such as Microsoft's Application Center. All in all, this is a fairly good idea of how a system such as Microsoft's Windows Messenger works. It's also similar to the approach followed in the third part of this book, where you'll learn how to create a discovery server using a web service.

OneWay Methods

There is also a minor messaging enhancement you can implement using the OneWay attribute from the System.Runtime.Remoting.Messaging namespace. When you apply this attribute to a method, you indicate that, when this method is called remotely, the caller will disconnect immediately without waiting for the call to complete. This means that the method cannot return a result or modify a ByVal parameter. It also means that any exception thrown in the method will not be detected by the caller. The advantage of this approach is that it eliminates waiting. In the Talk .NET system, the coordination server automatically calls a client if a message cannot be delivered. Thus, there's no reason for the client to wait while the message is actually being delivered.

There are currently two methods that could benefit from the OneWay attribute: ClientProcess.ReceiveMessage() and ServerProcess.SendMessage(). Here's an example:

```
<System.Runtime.Remoting.Messaging.OneWay()> _
Private Sub ReceiveMessage(ByVal message As String, _
  ByVal senderAlias As String) Implements ITalkClient.ReceiveMessage
    ' (Code omitted.)
End Sub
```

Note that there's one reason you might *not* want to apply the OneWay attribute to ServerProcess.SendMessage(). If you do, you won't be able to detect an error that might result if the user has disconnected without logging off correctly. Without catching this error, it's impossible to detect the problem, notify the sender, and remove the user from the client collection. This error-handling approach is implemented in the next chapter.

Optional Features

Finally, there are a number of optional features that you can add to Talk .NET. These include variable user status, user authentication with a password, and buddy lists. The last of these is probably the most useful, because it allows you to limit the user list information. With buddy lists, users only see the names of the users that they want to contact. However, buddy lists must be stored on the server permanently, and so can't be held in memory. Instead, this information would probably need to be stored in a server-side database.

Another option would be to store a list on the local computer, which would then be submitted with the login request. This would help keep the system decentralized, but it would also allow the information to be easily lost, and make it difficult for users to obtain location transparency and use the same buddy list from multiple computers. As you'll see, users aren't always prepared to accept the limitations of decentralized peer-to-peer applications.

Firewalls, Ports, and Other Issues

Remoting does not provide any way to overcome some of the difficulties that are inherent with networking on the Internet. For example, firewalls, depending on their settings, can prevent communication between the clients and the coordination server. On a local network, this won't pose a problem. On the Internet, you can lessen the possibility of problems by following several steps:

- Use the centralized design in which all communication is routed through the coordination server.

- Make sure the coordination server is not behind a firewall (in a company network, you would place the coordination server in the demilitarized zone, or DMZ). This helps connectivity because often communication will succeed when the client is behind a firewall, but not when both the client and server are behind firewalls.

- Change the configuration files so that HTTP channels are used instead. They're typically more reliable over the Internet and low-bandwidth connections. You should still use binary formatting, however, unless you're trying to interoperate with non-.NET clients.

It often seems that developers and network administrators are locked in an endless battle, with developers trying to extend the scope of their applications while network administrators try to protect the integrity of their network. This battle has escalated to such a high point that developers tout new features such as .NET web services because they use HTTP and can communicate through a firewall. All this ignores the fact that, typically, the firewall is there to *prevent* exactly this type of communication. Thwarting this protection just means that firewall vendors will need to go to greater lengths building intelligence into their firewall products. They'll need to perform more intensive network analysis that might reject SOAP messages or deny web-service communication based on other recognizable factors. These changes, in turn, raise the cost of the required servers and impose additional overhead.

In short, it's best to deal with firewall problems by configuring the firewall. If your application needs to use a special port, convince the network administrators to open it. Similarly, using port 80 for a peer-to-peer application is sure to win the contempt of system administrators everywhere. If you can't ensure that your clients can use another port, you may need to resort to this sleight-of-hand, but it's best to avoid the escalating war of Internet connectivity altogether.

NOTE *Ports are generally divided into three groups: well-known ports (0–1023), registered ports (1024–49151), and dynamic ports (49152–65535). Historically, well-known ports have been used for server-based applications such as web servers (80), FTP (20), and POP3 mail transfer (110). In your application, you would probably do best to use a registered or dynamic port that isn't frequently used. These are less likely to cause a conflict (although more likely to be blocked by a firewall). For example, 6346 is most commonly used by Gnutella. For a list of frequently registered ports, refer to the* C:\{WinDir]\System32\Drivers\Etc\Services *file or the* `http://www.iana.org/assignments/port-numbers` *site.*

Remoting and Network Address Translation

.NET Remoting, like many types of distributed communication, is challenged by firewalls, proxy servers, and network address translation (NAT). Many programmers (and programming authors) assume that using an HTTP channel will solve these problems. It may—if the intervening firewall restricts packets solely based on whether they contain binary information. However, this won't solve a much more significant problem: Most firewalls allow outgoing connections but prevent all incoming ones. Proxy servers and NAT devices work in the same way. This is a significant limitation. It means that a Talk .NET peer can contact the server (and the server can respond), but the server cannot call back to the client to deliver a message.

There's more than one way to solve this problem, but none is easy (or ideal). You could implement a polling mechanism, whereby every client periodically connects to the server and asks for any unsent messages. The drawback of this approach is that the message latency will be increased, and the load on the server will rise dramatically with the number of clients.

Another approach is to use some sort of bidirectional communication method. For example, you might want to maintain a connection and allow the server to fire its event or callback at any time using the existing connection. This also reduces the number of simultaneous clients the server can handle, and it requires a specially modified type of Remoting channel. Ingo Rammer has developed one such channel, and it's available at `http://www.dotnetremoting.cc/projects/modules/BidirectionalTcpChannel.asp`. However, this bidirectional channel isn't yet optimized for a production environment, so enterprise developers will need to wait.

Unfortunately, neither of these two proposed solutions will work if you want to use decentralized communication in which peers contact each other directly. In this case, you'll either need to write a significant amount of painful low-level networking code (which is beyond the scope of this book), or use a third-party platform such as those discussed in Part Three.

The Last Word

In this chapter, we developed an instant-messaging application using Remoting and showed how it could be modified into a peer-to-peer system with a central lookup service. However, the current version of the Talk .NET system still suffers from some notable shortcomings, which will become particularly apparent under high user loads. If different users attempt to register, unregister, or send messages at the same time, the user collection may be updated incorrectly, and information could be lost. To guard against these problems, which are almost impossible to replicate under modest loads, you'll need to add multithreading, as described in the next chapter.