

Chapter 23. Networking: Streams-Based Sockets and Datagrams

If the presence of electricity can be made visible in any part of a circuit, I see no reason why intelligence may not be transmitted instantaneously by electricity.

Samuel F. B. Morse

Protocol is everything.

Francois Giuliani

What networks of railroads, highways and canals were in another age, the networks of telecommunications, information and computerization ... are today.

Bruno Kreisky

The port is near, the bells I hear, the people all exulting.

Walt Whitman

OBJECTIVES

In this chapter you will learn:

- To implement networking applications that use sockets and datagrams.
- To implement clients and servers that communicate with one another.
- To implement network-based collaborative applications.
- To construct a multithreaded server.
- To use the `WebBrowser` control to add Web browsing capabilities to any application.
- To use .NET remoting to enable an application executing on one computer to invoke methods from an application executing on a different computer.

Outline

[23.1 Introduction](#)

[23.2 Connection-Oriented vs. Connectionless Communication](#)

[23.3 Protocols for Transporting Data](#)

[23.4 Establishing a Simple TCP Server \(Using Stream Sockets\)](#)
[23.5 Establishing a Simple TCP Client \(Using Stream Sockets\)](#)
[23.6 Client/Server Interaction with Stream-Socket Connections](#)
[23.7 Connectionless Client/Server Interaction with Datagrams](#)
[23.8 Client/Server Tic-Tac-Toe Using a Multithreaded Server](#)
[23.9 WebBrowser Control](#)
[23.10 .NET Remoting](#)
[23.11 Wrap-Up](#)
[Summary](#)
[Terminology](#)
[Self-Review Exercises](#)
[Answers to Self-Review Exercises](#)
[Exercises](#)



[Page 1232 (continued)]

23.1. Introduction

There is much excitement about the Internet and the Web. The Internet ties the information world together. The Web makes the Internet easy to use and gives it the flair and sizzle of multimedia. Organizations see the Internet and the Web as crucial to their informationsystems strategies. The .NET FCL provides a number of built-in networking capabilities that make it easy to develop Internet- and Web-based applications. Programs can search the world for information and collaborate with programs running on other computers internationally, nationally or just within an organization.

In [Chapter 21](#), ASP.NET 2.0, Web Forms and Web Controls, and [Chapter 22](#), Web Services, we began our presentation of C#'s networking and distributed-computing capabilities. We discussed ASP.NET, Web Forms and Web serviceshigh-level networking technologies that enable programmers to develop distributed applications. In this chapter, we focus on the underlying networking technologies that support C#'s ASP.NET and Web services capabilities.

This chapter begins with an overview of the communication techniques and technologies used to transmit data over the Internet. Next, we present the basic concepts of establishing a connection between two applications using streams of data that are similar to File I/O. This connection-oriented approach enables programs to communicate with one another as easily as writing to and reading from files on disk. Then we present a simple chat application that uses these techniques to send messages between a client and a server. The chapter continues with a presentation and an example of connectionless techniques for transmitting data between applications that is less reliable than establishing a connection between applications, but much more efficient. Such techniques are

typically used in applications such as streaming audio and video over the Internet. Next, we present an example of a client-server Tic-Tac-Toe game that demonstrates how to create a simple multithreaded server. Then this chapter demonstrates the new *WebBrowser control* for adding Web browsing capabilities to any application. The chapter completes with a brief introduction to .NET remoting which, like Web services ([Chapter 22](#)) enable distributed computing over networks.



[Page 1233]

23.2. Connection-Oriented vs. Connectionless Communication

There are two primary approaches to communicating between applications: connection oriented and connectionless. Connection-oriented communications are similar to the telephone system, in which a connection is established and held for the length of the session. Connectionless services are similar to the postal service, in which two letters mailed from the same place and to the same destination may actually take two dramatically different paths through the system and even arrive at different times, or not at all.

In a connection-oriented approach, computers send each other control information through a technique called *handshaking* to initiate an end-to-end connection. The Internet is an unreliable network, which means that data sent across the Internet may be damaged or lost. Data is sent in packets, which contain pieces of the data along with information that helps the Internet route the packets to the proper destination. The Internet does not guarantee anything about the packets sent; they could arrive corrupted or out of order, as duplicates or not at all. The Internet makes only a "best effort" to deliver packets. A connection-oriented approach ensures reliable communications on unreliable networks, guaranteeing that sent packets will arrive at the intended receiver undamaged and be reassembled in the correct sequence.

In a connectionless approach, the two computers do not handshake before transmission, and reliability is not guaranteed: data sent may never reach the intended recipient. A connectionless approach, however, avoids the overhead associated with handshaking and enforcing reliability: less information often needs to be passed between the hosts.



[Page 1233 (continued)]

23.3. Protocols for Transporting Data

There are many protocols for communicating between applications. Protocols are sets of rules that govern how two entities should interact. In this chapter, we focus on Transmission Control Protocol (TCP) and User Datagram Protocol (UDP). .NET's TCP and UDP networking capabilities are defined in the `System.Net.Sockets` namespace.

Transmission Control Protocol (TCP) is a connection-oriented communication protocol which guarantees that sent packets will arrive at the intended receiver undamaged and in the correct sequence. TCP allows protocols like HTTP ([Chapter 21](#)) to send information across a network as simply and reliably as writing to a file on a local computer. If packets of information don't arrive at the recipient, TCP ensures that the packets are sent again. If the packets arrive out of order, TCP reassembles them in the correct order transparently to the receiving application. If duplicate packets arrive, TCP discards them.

Applications that do not require TCP's reliable end-to-end transmission guaranty typically use the connectionless User Datagram Protocol (UDP). UDP incurs the minimum overhead necessary to communicate between applications. UDP makes no guarantees that packets, called datagrams, will reach their destination or arrive in their original order.

There are benefits to using UDP over TCP. UDP has little overhead because UDP datagrams do not need to carry the information that TCP packets carry to ensure reliability. UDP also reduces network traffic relative to TCP due to the absence of handshaking, retransmissions, etc.

Unreliable communication is acceptable in many situations. First, reliability is not necessary for some applications, so the overhead imposed by a protocol that guarantees reliability can be avoided. Second, some applications, such as streaming audio and video, can tolerate occasional datagram loss. This usually results in a small pause (or "hiccup") in the audio or video being played. If the same application were run over TCP, a lost segment could cause a significant pause, since the protocol would wait until the lost segment was retransmitted and delivered correctly before continuing. Finally, applications that need to implement their own reliability mechanisms different from those provided by TCP can build such mechanisms over UDP.

[Page 1234]



[Page 1234 (continued)]

23.4. Establishing a Simple TCP Server (Using Stream Sockets)

Typically, with TCP, a server "waits" for a connection request from a client. Often, the server program contains a control statement or block of code that executes continuously until the server receives a request. On receiving a request, the server establishes a connection to the client. The server then uses this connection managed by an object of class `Socket` to handle future requests from that client and to send data to the client. Since programs that communicate via TCP process the data they send and receive as streams of bytes, programmers sometimes refer to Sockets as "stream Sockets."

Establishing a simple server with TCP and stream sockets requires five steps. First, create an object of class `TcpListener` of namespace `System.Net.Sockets`. This class represents a TCP stream socket through which a server can listen for requests. Creating a new `TcpListener`, as in

```
TcpListener server = new TcpListener( ipAddress, port );
```

binds (assigns) the server application to the specified port number. A port number is a numeric identifier that an application uses to identify itself at a given network address, also known as an Internet Protocol Address (IP Address). IP addresses identify computers on the Internet. In fact, Web-site names, such as www.deitel.com, are aliases for IP addresses. An IP address is represented by an object of class `IPAddress` of namespace `System.Net`. Any application that performs networking identifies itself via an IP address/port number pair. No two applications can have the same port number at a given IP address. Explicitly binding a socket to a connection port (using method `Bind` of class `Socket`) is usually unnecessary, because class `TcpListener` and other classes discussed in this chapter do it automatically, along with other socket-initialization operations.



Software Engineering Observation 23.1

Port numbers can have values between 0 and 65535. Many operating systems reserve port numbers below 1024 for system services (such as e-mail and Web servers). Applications must be granted special privileges to use these reserved port numbers.

To receive requests, `TcpListener` first must listen for them. The second step in the connection process is to call `TcpListener`'s *Start* method, which causes the `TcpListener` object to begin listening for connection requests. The server listens indefinitely for a request; i.e., the execution of the server-side application waits until some client attempts to connect with it. The server creates a connection to the client when it receives a connection request. An object of class `Socket` (namespace `System.Net.Sockets`) manages a connection to a client. Method *AcceptSocket* of class *TcpListener* accepts a connection request. This method returns a `Socket` object upon connection, as in the statement

[Page 1235]

```
Socket connection = server.AcceptSocket();
```

When the server receives a request, `AcceptSocket` calls method `Accept` of the `TcpListener`'s underlying `Socket` to make the connection. This is an example of how networking complexity is hidden from the programmer. You simply place the preceding statement in a server-side program; the classes of namespace `System.Net.Sockets` handle the details of accepting requests and establishing connections.

The third step establishes the streams used for communication with the client. In this step, we create a *NetworkStream* object that uses the `Socket` object representing the connection to perform the actual sending and receiving of data. In our forthcoming example, we use this `NetworkStream` object to create a `BinaryWriter` and a *BinaryReader* that will be used to send information to and receive information from the client, respectively.

Step four is the processing phase, in which the server and client communicate using the connection established in the third step. In this phase, the client uses *BinaryWriter* method *Write* and *BinaryReader* method *ReadString* to perform the appropriate communications.

The fifth step is the connection-termination phase. When the client and server have finished communicating, the server calls method `Close` of the `BinaryReader`, `BinaryWriter`, `NetworkStream` and `Socket` to terminate the connection. The server can then return to step two to wait for the next connection request. Note that the documentation for class `Socket` recommends that you call method `Shutdown` before method `Close` to ensure that all data is sent and received before the `Socket` closes.

One problem associated with the server scheme described in this section is that step four blocks other requests while processing the connected client's request, so no other client can connect with the server while the code that defines the processing phase is executing. The most common technique for addressing this problem is to use multithreaded servers, which place the processing-phase code in a separate thread. For each connection request the server receives, it creates a `Thread` to process the connection, leaving its `TcpListener` (or `Socket`) free to receive other connections. We demonstrate

a multithreaded server in [Section 23.8](#).



Software Engineering Observation 23.2

Multithreaded servers can efficiently manage simultaneous connections with multiple clients. This architecture is precisely what popular UNIX and Windows network servers use.



Software Engineering Observation 23.3

A multithreaded server can be implemented to create a thread that manages network I/O across a `Socket` object returned by method `AcceptSocket`. A multithreaded server also can be implemented to maintain a pool of threads that manage network I/O across newly created `Sockets`.



Performance Tip 23.1

In high-performance systems with abundant memory, a multithreaded server can be implemented to create a pool of threads. These threads can be assigned quickly to handle network I/O across multiple `Sockets`. Thus, when a connection is received, the server does not incur the overhead of thread creation.



[Page 1236]

23.5. Establishing a Simple TCP Client (Using Stream Sockets)

There are four steps to creating a simple TCP client. First, we create an object of class *TcpClient* (namespace `System.Net.Sockets`) to connect to the server. The connection is established by calling *TcpClient* method `Connect`. One overloaded version of this method takes two arguments: the server's IP address and its port number as in:

```
TcpClient client = new TcpClient();
client.Connect( serverAddress, serverPort );
```

The `serverPort` is an `int` that represents the port number to which the server application is bound to listen for connection requests. The `serverAddress` can be either an `IPAddress` instance (that encapsulates the server's IP address) or a `string` that specifies the server's hostname or IP address. Method `Connect` also has an overloaded version to which you can pass an *EndPoint* object that represents an IP address/port number pair. *TcpClient* method `Connect` calls `Socket` method `Connect` to establish the connection. If the connection is successful, *TcpClient* method `Connect`

returns a positive integer; otherwise, it returns 0.

In step two, the `TcpClient` uses its `GetStream` method to get a `NetworkStream` so that it can write to and read from the server. We then use the `NetworkStream` object to create a `BinaryWriter` and a `BinaryReader` that will be used to send information to and receive information from the server, respectively.

The third step is the processing phase, in which the client and the server communicate. In this phase of our example, the client uses `BinaryWriter` method `Write` and `BinaryReader` method `ReadString` to perform the appropriate communications. Using a process similar to that used by servers, a client can employ threads to prevent blocking of communication with other servers while processing data from one connection.

After the transmission is complete, step four requires the client to close the connection by calling method `Close` on each of `BinaryReader`, `BinaryWriter`, `NetworkStream` and `TcpClient`. This closes each of the streams and the `TcpClient`'s `Socket` to terminate the connection with the server. At this point, a new connection can be established through method `Connect`, as we have described.



[Page 1236 (continued)]

23.6. Client/Server Interaction with Stream-Socket Connections

The applications in [Fig. 23.1](#) and [Fig. 23.2](#) use the classes and techniques discussed in the previous two sections to construct a simple client/server chat application. The server waits for a client's request to make a connection. When a client application connects to the server, the server application sends an array of bytes to the client, indicating that the connection was successful. The client then displays a message notifying the user that a connection has been established.

Figure 23.1. Server portion of a client/server stream-socket connection.
(This item is displayed on pages 1237 - 1240 in the print version)

```

1  // Fig. 23.1: ChatServer.cs
2  // Set up a server that will receive a connection from a client, send a
3  // string to the client, chat with the client and close the connection.
4  using System;
5  using System.Windows.Forms;
6  using System.Threading;
7  using System.Net;
8  using System.Net.Sockets;
9  using System.IO;
10
11 public partial class ChatServerForm : Form
12 {
13     public ChatServerForm()
14     {
15         InitializeComponent();
16     } // end constructor
17
18     private Socket connection; // Socket for accepting a connection
19     private Thread readThread; // Thread for processing incoming messages
20     private NetworkStream socketStream; // network data stream
21     private BinaryWriter writer; // facilitates writing to the stream
22     private BinaryReader reader; // facilitates reading from the stream

```

```

23
24 // initialize thread for reading
25 private void ChatServerForm_Load( object sender, EventArgs e )
26 {
27     readThread = new Thread( new ThreadStart( RunServer ) );
28     readThread.Start();
29 } // end method ChatServerForm_Load
30
31 // close all threads associated with this application
32 private void ChatServerForm_FormClosing( object sender,
33     FormClosingEventArgs e )
34 {
35     System.Environment.Exit( System.Environment.ExitCode );
36 } // end method ChatServerForm_FormClosing
37
38 // delegate that allows method DisplayMessage to be called
39 // in the thread that creates and maintains the GUI
40 private delegate void DisplayDelegate( string message );
41

```

```

                                     [Page 1238]
42 // method DisplayMessage sets displayTextBox's Text property
43 // in a thread-safe manner
44 private void DisplayMessage( string message )
45 {
46     // if modifying displayTextBox is not thread safe
47     if ( displayTextBox.InvokeRequired )
48     {
49         // use inherited method Invoke to execute DisplayMessage
50         // via a delegate
51         Invoke( new DisplayDelegate( DisplayMessage ),
52             new object[] { message } );
53     } // end if
54     else // OK to modify displayTextBox in current thread
55         displayTextBox.Text += message;
56 } // end method DisplayMessage
57
58 // delegate that allows method DisableInput to be called
59 // in the thread that creates and maintains the GUI
60 private delegate void DisableInputDelegate( bool value );
61
62 // method DisableInput sets inputTextBox's ReadOnly property
63 // in a thread-safe manner
64 private void DisableInput( bool value )
65 {
66     // if modifying inputTextBox is not thread safe
67     if ( inputTextBox.InvokeRequired )
68     {
69         // use inherited method Invoke to execute DisableInput
70         // via a delegate
71         Invoke( new DisableInputDelegate( DisableInput ),
72             new object[] { value } );
73     } // end if
74     else // OK to modify inputTextBox in current thread
75         inputTextBox.ReadOnly = value;
76 } // end method DisableInput
77
78 // send the text typed at the server to the client
79 private void inputTextBox_KeyDown( object sender, KeyEventArgs e )
80 {
81     // send the text to the client
82     try
83     {

```

```

84         if ( e.KeyCode == Keys.Enter && inputTextBox.ReadOnly == false )
85         {
86             writer.Write( "SERVER>>> " + inputTextBox.Text );
87             displayTextBox.Text += "\r\nSERVER>>> " + inputTextBox.Text;
88
89             // if the user at the server signaled termination
90             // sever the connection to the client
91             if ( inputTextBox.Text == "TERMINATE" )
92                 connection.Close();
93


---


94             [Page 1239]
95             inputTextBox.Clear(); // clear the user's input
96         } // end if
97     } // end try
98     catch ( SocketException )
99     {
100         displayTextBox.Text += "\nError writing object";
101     } // end catch
102 } // end method inputTextBox_KeyDown
103
104 // allows a client to connect; displays text the client sends
105 public void RunServer()
106 {
107     TcpListener listener;
108     int counter = 1;
109
110     // wait for a client connection and display the text
111     // that the client sends
112     try
113     {
114         // Step 1: create TcpListener
115         IPAddress local = IPAddress.Parse( "127.0.0.1" );
116         listener = new TcpListener( local, 50000 );
117
118         // Step 2: TcpListener waits for connection request
119         listener.Start();
120
121         // Step 3: establish connection upon client request
122         while ( true )
123         {
124             DisplayMessage( "Waiting for connection\r\n" );
125
126             // accept an incoming connection
127             connection = listener.AcceptSocket();
128
129             // create NetworkStream object associated with socket
130             socketStream = new NetworkStream( connection );
131
132             // create objects for transferring data across stream
133             writer = new BinaryWriter( socketStream );
134             reader = new BinaryReader( socketStream );
135
136             DisplayMessage( "Connection " + counter + " received.\r\n" );
137
138             // inform client that connection was successful
139             writer.Write( "SERVER>>> Connection successful" );
140
141             DisableInput( false ); // enable inputTextBox
142
143             string theReply = "";
144
145             // Step 4: read string data sent from client

```

```

145         do
146         {


---


147             try
148             {
149                 // read the string sent to the server
150                 theReply = reader.ReadString();
151
152                 // display the message
153                 DisplayMessage( "\r\n" + theReply );
154             } // end try
155             catch ( Exception )
156             {
157                 // handle exception if error reading data
158                 break;
159             } // end catch
160         } while ( theReply != "CLIENT>>> TERMINATE" &&
161                 connection.Connected );
162
163         DisplayMessage( "\r\nUser terminated connection\r\n" );
164
165         // Step 5: close connection
166         writer.Close();
167         reader.Close();
168         socketStream.Close();
169         connection.Close();
170
171         DisableInput( true ); // disable InputTextBox
172         counter++;
173     } // end while
174 } // end try
175 catch ( Exception error )
176 {
177     MessageBox.Show( error.ToString() );
178 } // end catch
179 } // end method RunServer
180 } // end class ChatServerForm

```

Figure 23.2. Client portion of a client/server stream-socket connection.

(This item is displayed on pages 1242 - 1246 in the print version)

```

1 // Fig. 23.2: ChatClient.cs
2 // Set up a client that will send information to and
3 // read information from a server.
4 using System;
5 using System.Windows.Forms;
6 using System.Threading;
7 using System.Net.Sockets;
8 using System.IO;
9
10 public partial class ChatClientForm : Form
11 {


---


12     public ChatClientForm()
13     {
14         InitializeComponent();
15     } // end constructor

```

```
16
17 private NetworkStream output; // stream for receiving data
18 private BinaryWriter writer; // facilitates writing to the stream
19 private BinaryReader reader; // facilitates reading from the stream
20 private Thread readThread; // Thread for processing incoming messages
21 private string message = "";
22
23 // initialize thread for reading
24 private void ChatClientForm_Load( object sender, EventArgs e )
25 {
26     readThread = new Thread( new ThreadStart( RunClient ) );
27     readThread.Start();
28 } // end method ChatClientForm_Load
29
30 // close all threads associated with this application
31 private void ChatClientForm_FormClosing( object sender,
32     FormClosingEventArgs e )
33 {
34     System.Environment.Exit( System.Environment.ExitCode );
35 } // end method ChatClientForm_FormClosing
36
37 // delegate that allows method DisplayMessage to be called
38 // in the thread that creates and maintains the GUI
39 private delegate void DisplayDelegate( string message );
40
41 // method DisplayMessage sets displayTextBox's Text property
42 // in a thread-safe manner
43 private void DisplayMessage( string message )
44 {
45     // if modifying displayTextBox is not thread safe
46     if ( displayTextBox.InvokeRequired )
47     {
48         // use inherited method Invoke to execute DisplayMessage
49         // via a delegate
50         Invoke( new DisplayDelegate( DisplayMessage ),
51             new object[] { message } );
52     } // end if
53     else // OK to modify displayTextBox in current thread
54         displayTextBox.Text += message;
55 } // end method DisplayMessage
56
57 // delegate that allows method DisableInput to be called
58 // in the thread that creates and maintains the GUI
59 private delegate void DisableInputDelegate( bool value );
60
61 // method DisableInput sets inputTextBox's ReadOnly property
62 // in a thread-safe manner
63 private void DisableInput( bool value )
64 {
```

[Page 1244]

```
65     // if modifying inputTextBox is not thread safe
66     if ( inputTextBox.InvokeRequired )
67     {
68         // use inherited method Invoke to execute DisableInput
69         // via a delegate
70         Invoke( new DisableInputDelegate( DisableInput ),
71             new object[] { value } );
72     } // end if
73     else // OK to modify inputTextBox in current thread
74         inputTextBox.ReadOnly = value;
75 } // end method DisableInput
76
```

```

77 // sends text the user typed to server
78 private void inputTextBox_KeyDown( object sender, KeyEventArgs e )
79 {
80     try
81     {
82         if ( e.KeyCode == Keys.Enter && inputTextBox.ReadOnly == false )
83         {
84             writer.Write( "CLIENT>>> " + inputTextBox.Text );
85             displayTextBox.Text += "\r\nCLIENT>>> " + inputTextBox.Text;
86             inputTextBox.Clear();
87         } // end if
88     } // end try
89     catch ( SocketException )
90     {
91         displayTextBox.Text += "\nError writing object";
92     } // end catch
93 } // end method inputTextBox_KeyDown
94
95 // connect to server and display server-generated text
96 public void RunClient()
97 {
98     TcpClient client;
99
100    // instantiate TcpClient for sending data to server
101    try
102    {
103        DisplayMessage( "Attempting connection\r\n" );
104
105        // Step 1: create TcpClient and connect to server
106        client = new TcpClient();
107        client.Connect( "127.0.0.1", 50000 );
108
109        // Step 2: get NetworkStream associated with TcpClient
110        output = client.GetStream();
111
112        // create objects for writing and reading across stream
113        writer = new BinaryWriter( output );
114        reader = new BinaryReader( output );
115
116        DisplayMessage( "\r\nGot I/O streams\r\n" );
117        DisableInput( false ); // enable inputTextBox

```

[Page 1245]

```

118
119    // loop until server signals termination
120    do
121    {
122        // Step 3: processing phase
123        try
124        {
125            // read message from server
126            message = reader.ReadString();
127            DisplayMessage( "\r\n" + message );
128        } // end try
129        catch ( Exception )
130        {
131            // handle exception if error in reading server data
132            System.Environment.Exit( System.Environment.ExitCode );
133        } // end catch
134    } while ( message != "SERVER>>> TERMINATE" );
135
136    // Step 4: close connection
137    writer.Close();

```

```

138         reader.Close();
139         output.Close();
140         client.Close();
141
142         Application.Exit();
143     } // end try
144     catch ( Exception error )
145     {
146         // handle exception if error in establishing connection
147         MessageBox.Show( error.ToString(), "Connection Error",
148             MessageBoxButtons.OK, MessageBoxIcon.Error );
149         System.Environment.Exit( System.Environment.ExitCode );
150     } // end catch
151 } // end method RunClient
152 } // end class ChatClientForm

```

(a)



(b)



[Page 1246]

(c)



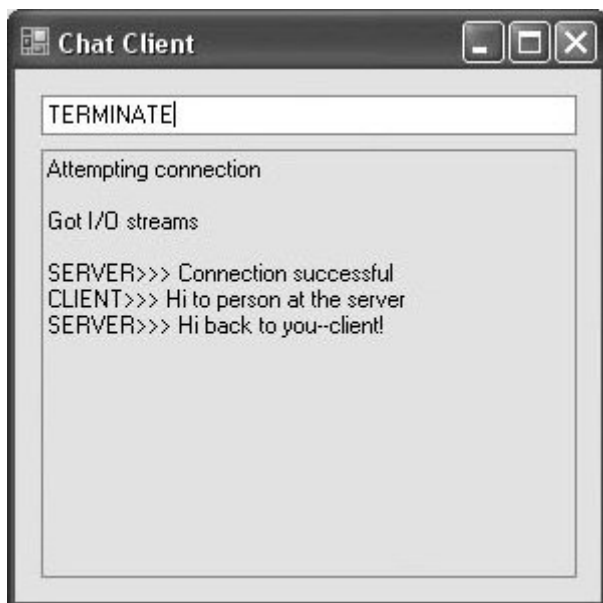
(d)



(e)



(f)



(g)



Both the client and the server applications contain TextBoxes that enable users to type messages and send them to the other application. When either the client or the server sends the message "TERMINATE," the connection between the client and the server terminates. The server then waits for another client to request a connection. [Figure 23.1](#) and [Fig. 23.2](#) provide the code for classes Server and Client, respectively. [Figure 23.2](#) also contains screen captures displaying the execution between the client and the server.

[Page 1237]

ChatServerForm Class

We begin by discussing class ChatServerForm ([Fig. 23.1](#)). In the constructor, line 27 creates a Thread that will accept connections from clients. The ThreadStart delegate object that is passed as the constructor's argument specifies which method the Thread should execute. Line 28 starts the Thread, which uses the ThreadStart delegate to invoke method RunServer (lines 104179). This method initializes the server to receive connection requests and process connections. Line 115 instantiates a TcpListener object to listen for a connection request from a client at port 50000 (Step 1). Line 118 then calls TcpListener method Start, which causes the TcpListener to begin waiting for requests (Step 2).

[Page 1240]

Accepting the Connection and Establishing the Streams

Lines 121173 declare an infinite loop that begins by establishing the connection requested by the client (Step 3). Line 126 calls method AcceptSocket of the TcpListener object, which returns a Socket upon successful connection. The thread in which method AcceptSocket is called blocks (i.e., stops executing) until a connection is established. The returned Socket object manages the connection. Line 129 passes this Socket object as an argument to the constructor of a NetworkStream object, which provides access to streams across a network. In this example, the NetworkStream object uses the streams of the specified Socket. Lines 132133 create instances of the BinaryWriter and BinaryReader classes for writing and reading data. We pass the NetworkStream object as an argument to each constructor. BinaryWriter can write bytes to the

NetworkStream, and BinaryReader can read bytes from NetworkStream. Line 135 calls DisplayMessage, indicating that a connection was received. Next, we send a message to the client indicating that the connection was received. BinaryWriter method Write has many overloaded versions that write data of various types to a stream. Line 138 uses method Write to send to the client a string notifying the user of a successful connection. This completes Step 3.

[Page 1241]

Receiving Messages from the Client

Next, we begin the processing phase (Step 4). Lines 145-161 declare a do...while statement that executes until the server receives a message indicating connection termination (i.e., CLIENT>>> TERMINATE). Line 150 uses BinaryReader method ReadString to read a string from the stream. Method ReadString blocks until a string is read. This is the reason that we execute method RunServer in a separate Thread (created at lines 27-28, when the Form loads). This Thread ensures that our Server application's user can continue to interact with the GUI to send messages to the client, even when this thread is blocked while awaiting a message from the client.

Modifying GUI Controls from Separate Threads

Windows Form controls are not thread safe; a control that is modified from multiple threads is not guaranteed to be modified correctly. The Visual Studio 2005 Documentation^[1] recommends that only the thread which created the GUI should modify the controls. Class Control provides method Invoke to help ensure this. Invoke takes two arguments: a delegate representing a method that will modify the GUI and an array of objects representing the parameters of the method. At some point after Invoke is called, the thread that originally created the GUI will (when it's not executing any other code) execute the method represented by the delegate, passing the contents of the object array as the method's arguments.

[1] The MSDN article "How to: Make Cross-Thread Calls to Windows Forms Controls" can be found at [msdn2.microsoft.com/library/ms171728\(en-us,vs.80\).aspx](http://msdn2.microsoft.com/library/ms171728(en-us,vs.80).aspx).

Line 40 declares a delegate type named DisplayDelegate, which represents methods that take a string argument and do not return a value. Method DisplayMessage (lines 44-56) meets those requirements: it receives a string parameter named message and does not return a value. The if statement in line 47 tests displayTextBox's InvokeRequired property (inherited from class Control), which returns True if the current thread is not allowed to modify this control directly and returns false otherwise. If the current thread executing method DisplayMessage is not the thread that created the GUI, then the if condition evaluates to true and lines 51-52 call method Invoke, passing to it a new DisplayDelegate representing the method DisplayMessage itself and a new object array consisting of the string argument message. This causes the thread that created the GUI to call method DisplayMessage again at a later time with the same string argument as the original call. When that call occurs from the thread that created the GUI, the method is allowed to modify displayTextBox directly, so the else body (line 55) executes and appends message to displayTextBox's Text property.

Lines 60-66 provide a delegate definition, DisableInputDelegate, and a method, DisableInput, to allow any thread to modify the ReadOnly property of inputTextBox using the same techniques. A thread calls DisableInput with a bool argument (True to disable; false to enable). If DisableInput is not allowed to modify the control from the current thread, DisableInput calls method Invoke. This causes the thread that created the GUI to call DisableInput at a later time and set inputTextBox.ReadOnly to the value of the bool argument.

[Page 1242]

Terminating the Connection with the Client

When the chat is complete, lines 166169 close the `BinaryWriter`, `BinaryReader`, `NetworkStream` and `Socket` (Step 5) by invoking their respective `Close` methods. The server then waits for another client connection request by returning to the beginning of the `while` loop (line 121).

Sending Messages to the Client

When the server application's user enters a string in the `TextBox` and presses the Enter key, event handler `inputTextBox_KeyDown` (lines 79101) reads the string and sends it via method `Write` of class `BinaryWriter`. If a user terminates the server application, line 92 calls method `Close` of the `Socket` object to close the connection.

Terminating the Server Application

Lines 3236 define event handler `ChatServerForm_FormClosing` for the `FormClosing` event. The event closes the application and calls method `Exit` of class `Environment` with parameter `ExitCode` to terminate all threads. Method `Exit` of class `Environment` closes all threads associated with the application.

ChatClientForm Class

[Figure 23.2](#) lists the code for the `ChatClientForm` class. Like the `ChatServerForm` object, the `ChatClientForm` object creates a `Thread` (lines 2627) in its constructor to handle all incoming messages. `ChatClientForm` method `RunClient` (lines 96151) connects to the `ChatServerForm`, receives data from the `ChatServerForm` and sends data to the `ChatServerForm`. Lines 106107 instantiate a `TcpClient` object, then call its `Connect` method to establish a connection (Step 1). The first argument to method `Connect` is the name of the server in our case, the server's name is "`localhost`", meaning that the server is located on the same machine as the client. The `localhost` is also known as the loopback IP address and is equivalent to the IP address `127.0.0.1`. This value sends the data transmission back to the sender's IP address. [Note: We chose to demonstrate the client/server relationship by connecting between programs that are executing on the same computer (`localhost`). Normally, this argument would contain the Internet address of another computer.] The second argument to method `Connect` is the server port number. This number must match the port number at which the server waits for connections.

[Page 1246]

The `ChatClientForm` uses a `NetworkStream` to send data to and receive data from the server. The client obtains the `NetworkStream` in line 110 through a call to `TcpClient` method `GetStream` (Step 2). The `do...while` statement in lines 120134 loops until the client receives the connection-termination message (`SERVER>>> TERMINATE`). Line 126 uses `BinaryReader` method `ReadString` to obtain the next message from the server (Step 3). Line 127 displays the message, and lines 137140 close the `BinaryWriter`, `BinaryReader`, `NetworkStream` and `TcpClient` objects (Step 4).

[Page 1247]

Lines 3975 declare `DisplayDelegate`, `DisplayMessage`, `DisableInputDelegate` and

DisableInput just as in lines 4076 of [Fig. 23.1](#). These once again are used to ensure that the GUI is modified by only the thread that created the GUI controls.

When the user of the client application enters a string in the TextBox and presses the Enter key, event handler `inputTextBox_KeyDown` (lines 7893) reads the string from the TextBox and sends it via `BinaryWriter` method `Write`. Notice here that the `ChatServerForm` receives a connection, processes it, closes it and waits for the next one. In a real-world application, a server would likely receive a connection, set up the connection to be processed as a separate thread of execution and wait for new connections. The separate threads that process existing connections could then continue to execute while the server concentrates on new connection requests.



[Page 1247 (continued)]

23.7. Connectionless Client/Server Interaction with Datagrams

Up to this point, we have discussed connection-oriented, streams-based transmissions using the TCP protocol to ensure that the packets of data are transmitted reliably. Now, we consider connectionless transmission using datagrams and UDP.

Connectionless transmission via datagrams resembles the method by which the postal service carries and delivers mail. Connectionless transmission bundles and sends information in packets called datagrams, which can be thought of as similar to letters you send through the mail. If a large message will not fit in one envelope, that message is broken into separate message pieces and placed in separate, sequentially numbered envelopes. All the letters are mailed at once. The letters might arrive in order, out of order or not at all. The person at the receiving end reassembles the message pieces in sequential order before attempting to interpret the message. If the message is small enough to fit in one envelope, the sequencing problem is eliminated, but it is still possible that the message will never arrive. (Unlike with postal mail, duplicate of datagrams could reach receiving computers.) C# provides the `UdpClient` class for connectionless transmission. Like `TcpListener` and `TcpClient`, `UdpClient` uses methods from class `Socket`. The `UdpClient` methods `Send` and `Receive` transmit data with `Socket`'s `SendTo` method and read data with `Socket`'s `ReceiveFrom` method, respectively.

The programs in [Fig. 23.3](#) and [Fig. 23.4](#) use datagrams to send packets of information between client and server applications. In the `PacketClient` application, the user types a message into a TextBox and presses Enter. The client converts the message to a byte array and sends it to the server. The server receives the packet and displays the packet's information, then echoes, or returns, the packet to the client. When the client receives the packet, the client displays the packet's information. In this example, the implementations of the `PacketClientForm` and `PacketServerForm` classes are similar.

Figure 23.3. Server-side portion of connectionless client/server computing.

(This item is displayed on pages 1248 - 1249 in the print version)

```
1 // Fig. 23.3: PacketServer.cs
2 // Set up a server that will receive packets from a
3 // client and send the packets back to the client.
4 using System;
5 using System.Windows.Forms;
6 using System.Net;
7 using System.Net.Sockets;
8 using System.Threading;
```

```

9
10 public partial class PacketServerForm : Form
11 {
12     public PacketServerForm()
13     {
14         InitializeComponent();
15     } // end constructor
16
17     private UdpClient client;
18     private IPEndPoint receivePoint;
19
20     // initialize variables and thread for receiving packets
21     private void PacketServerForm_Load( object sender, EventArgs e )
22     {
23         client = new UdpClient( 50000 );
24         receivePoint = new IPEndPoint( new IPAddress( 0 ), 0 );
25         Thread readThread =
26             new Thread( new ThreadStart( WaitForPackets ) );
27         readThread.Start();
28     } // end method PacketServerForm_Load
29
30     // shut down the server
31     private void PacketServerForm_FormClosing( object sender,
32         FormClosingEventArgs e )
33     {
34         System.Environment.Exit( System.Environment.ExitCode );
35     } // end method PacketServerForm_FormClosing
36
37     // delegate that allows method DisplayMessage to be called
38     // in the thread that creates and maintains the GUI
39     private delegate void DisplayDelegate( string message );
40
41     // method DisplayMessage sets displayTextBox's Text property
42     // in a thread-safe manner
43     private void DisplayMessage( string message )
44     {
45         [Page 1249]
46         // if modifying displayTextBox is not thread safe
47         if ( displayTextBox.InvokeRequired )
48         {
49             // use inherited method Invoke to execute DisplayMessage
50             // via a delegate
51             Invoke( new DisplayDelegate( DisplayMessage ),
52                 new object[] { message } );
53         } // end if
54         else // OK to modify displayTextBox in current thread
55             displayTextBox.Text += message;
56     } // end method DisplayMessage
57
58     // wait for a packet to arrive
59     public void WaitForPackets()
60     {
61         while ( true )
62         {
63             // set up packet
64             byte[] data = client.Receive( ref receivePoint );
65             DisplayMessage( "\r\nPacket received:" +
66                 "\r\nLength: " + data.Length +
67                 "\r\nContaining: " +
68                 System.Text.Encoding.ASCII.GetString( data ) );
69             // echo information from packet back to client

```

```

70         DisplayMessage( "\r\n\r\nEcho data back to client..." );
71         client.Send( data, data.Length, receivePoint );
72         DisplayMessage( "\r\nPacket sent\r\n" );
73     } // end while
74 } // end method WaitForPackets
75 } // end class PacketServerForm

```

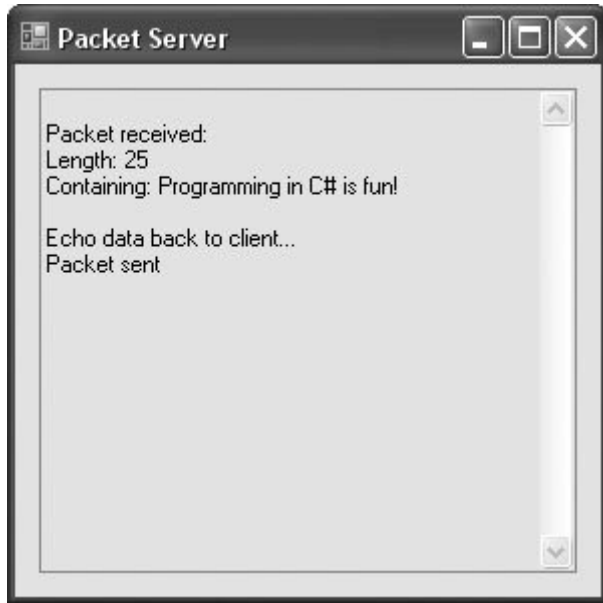


Figure 23.4. Client portion of connectionless client/server computing.

(This item is displayed on pages 1250 - 1252 in the print version)

```

1 // Fig. 23.4: PacketClient.cs
2 // Set up a client that sends packets to a server and receives
3 // packets from a server.
4 using System;
5 using System.Windows.Forms;
6 using System.Net;
7 using System.Net.Sockets;
8 using System.Threading;
9
10 public partial class PacketClientForm : Form
11 {
12     public PacketClientForm()
13     {
14         InitializeComponent();
15     } // end constructor
16
17     [Page 1251]
18     private UdpClient client;
19     private IPEndPoint receivePoint;
20
21     // initialize variables and thread for receiving packets
22     private void PacketClientForm_Load( object sender, EventArgs e )
23     {
24         receivePoint = new IPEndPoint( new IPAddress( 0 ), 0 );
25         client = new UdpClient( 50001 );
26         Thread thread =
27             new Thread( new ThreadStart( WaitForPackets ) );
28         thread.Start();
29     } // end method PacketClientForm_Load

```

```

29
30 // shut down the client
31 private void PacketClientForm_FormClosing( object sender,
32     FormClosingEventArgs e )
33 {
34     System.Environment.Exit( System.Environment.ExitCode );
35 } // end method PacketClientForm_FormClosing
36
37 // delegate that allows method DisplayMessage to be called
38 // in the thread that creates and maintains the GUI
39 private delegate void DisplayDelegate( string message );
40
41 // method DisplayMessage sets displayTextBox's Text property
42 // in a thread-safe manner
43 private void DisplayMessage( string message )
44 {
45     // if modifying displayTextBox is not thread safe
46     if ( displayTextBox.InvokeRequired )
47     {
48         // use inherited method Invoke to execute DisplayMessage
49         // via a delegate
50         Invoke( new DisplayDelegate( DisplayMessage ),
51             new object[] { message } );
52     } // end if
53     else // OK to modify displayTextBox in current thread
54         displayTextBox.Text += message;
55 } // end method DisplayMessage
56
57 // send a packet
58 private void inputTextBox_KeyDown( object sender, KeyEventArgs e )
59 {
60     if ( e.KeyCode == Keys.Enter )
61     {
62         // create packet (datagram) as string
63         string packet = inputTextBox.Text;
64         displayTextBox.Text +=
65             "\r\nSending packet containing: " + packet;
66
67         // convert packet to byte array
68         byte[] data = System.Text.Encoding.ASCII.GetBytes( packet );
69


---


70         // send packet to server on port 50000
71         client.Send( data, data.Length, "127.0.0.1", 50000 );
72         displayTextBox.Text += "\r\nPacket sent\r\n";
73         inputTextBox.Clear();
74     } // end if
75 } // end method inputTextBox_KeyDown
76
77 // wait for packets to arrive
78 public void WaitForPackets()
79 {
80     while ( true )
81     {
82         // receive byte array from server
83         byte[] data = client.Receive( ref receivePoint );
84
85         // output packet data to TextBox
86         DisplayMessage( "\r\nPacket received:" +
87             "\r\nLength: " + data.Length + "\r\nContaining: " +
88             System.Text.Encoding.ASCII.GetString( data ) + "\r\n" );
89     } // end while

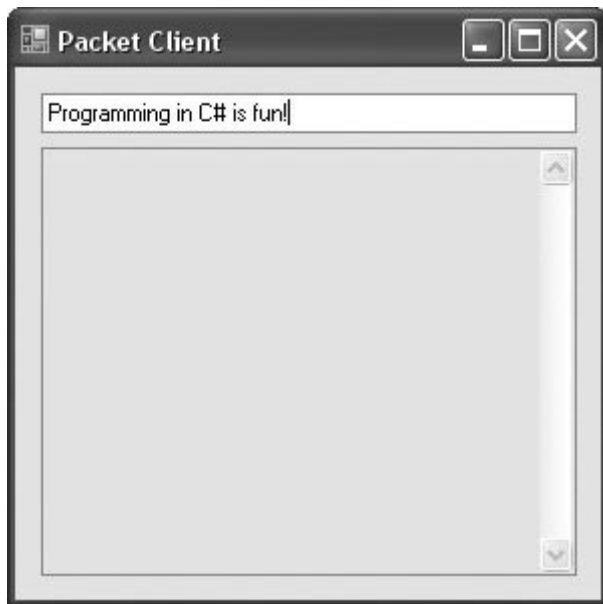
```

```

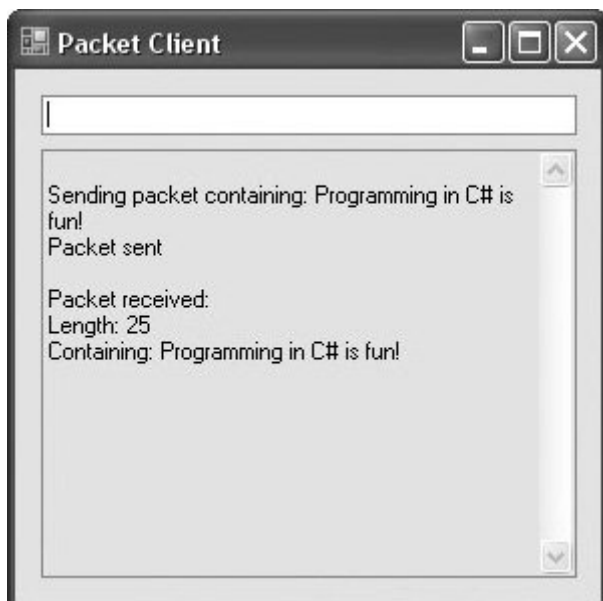
90     } // end method WaitForPackets
91 } // end class PacketClientForm

```

(a) Packet Client window before sending a packet to the server



(b) Packet Client window after sending a packet to the server and receiving it back



PacketServerForm Class

The code in [Fig. 23.3](#) defines the PacketServerForm for this application. Line 23 in the Load event handler for class PacketServerForm creates an instance of the UdpClient class that receives data at port 50000. This initializes the underlying Socket for communications. Line 24 creates an instance of class IPEndPoint to hold the IP address and port number of the client(s) that transmit to PacketServerForm. The first argument to the IPEndPoint constructor is an IPAddress object; the second argument is the port number of the endpoint. These values are both 0, because we need only instantiate an empty IPEndPoint object. The IP addresses and port numbers of clients are copied into the IPEndPoint when datagrams are received from clients.

[Page 1248]

[Page 1249]

Lines 3955 define `DisplayDelegate` and `DisplayMessage`, allowing any thread to modify `displayTextBox`'s `Text` property.

`PacketServerForm` method `waitForPackets` (lines 5874) executes an infinite loop while waiting for data to arrive at the `PacketServerForm`. When information arrives, `UdpClient` method `Receive` (line 63) receives a byte array from the client. We pass to `Receive` the `IPEndPoint` object created in the constructor; this provides the method with an `IPEndPoint` to which the program copies the client's IP address and port number. This program will compile and run without an exception even if the reference to the `IPEndPoint` object is `null`, because method `Receive` initializes the `IPEndPoint` if it is `null`.

[Page 1250]

Lines 6467 update the `PacketServerForm`'s display to include the packet's information and content. Line 71 echoes the data back to the client, using `UdpClient` method `Send`. This version of `Send` takes three arguments: the byte array to send, an `int` representing the array's length and the `IPEndPoint` to which to send the data. We use array data returned by method `Receive` as the data, the length of array data as the length and the `IPEndPoint` passed to method `Receive` as the data's destination. The IP address and port number of the client that sent the data are stored in `receivePoint`, so merely passing `receivePoint` to `Send` allows `PacketServerForm` to respond to the client.

PacketClientForm Class

Class `PacketClientForm` ([Fig. 23.4](#)) works similarly to class `PacketServerForm`, except that the `Client` object sends packets only when the user types a message in a `TextBox` and presses the `Enter` key. When this occurs, the program calls event handler `inputTextBox_KeyDown` (lines 5875). Line 68 converts the string that the user entered in the `TextBox` to a byte array. Line 71 calls `UdpClient` method `Send` to send the byte array to the `PacketServerForm` that is located on `localhost` (i.e., the same machine). We specify the port as 50000, which we know to be `PacketServerForm`'s port.

Lines 3955 define `DisplayDelegate` and `DisplayMessage`, allowing any thread to modify `displayTextBox`'s `Text` property.

Line 24 instantiates a `UdpClient` object to receive packets at port 50001; we choose port 50001 because the `PacketServerForm` already occupies port 50000. Method `waitForPackets` of class `PacketClientForm` (lines 7890) uses an infinite loop to wait for these packets. `UdpClient` method `Receive` blocks until a packet of data is received (line 83). The blocking performed by method `Receive` does not prevent class `PacketClientForm` from performing other services (e.g., handling user input), because a separate thread runs method `waitForPackets`.

[Page 1252]

When a packet arrives, lines 8688 display its contents in the `TextBox`. The user can type information in the `PacketClientForm` window's `TextBox` and press the `Enter` key at any time, even while a packet is being received. The event handler for the `TextBox` processes the event and sends the data to

the server.



[Page 1252 (continued)]

23.8. Client/Server Tic-Tac-Toe Using a Multithreaded Server

In this section, we present a networked version of the popular game Tic-Tac-Toe, implemented with stream sockets and client/server techniques. The program consists of a `TicTacToeServer` application ([Fig. 23.5](#)) and a `TicTacToeClient` application ([Fig. 23.6](#)). The `TicTacToeServer` allows two `TicTacToeClient` instances to connect to the server and play Tic-Tac-Toe against each other. We depict the output in [Fig. 23.6](#). When the server receives a client connection, lines 7887 of [Fig. 23.5](#) create instances of class `Player` to process each client in a separate thread of execution. This enables the server to handle requests from both clients. The server assigns value "X" to the first client that connects (player X makes the first move), then assigns value "O" to the second client. Throughout the game, the server maintains information regarding the status of the board so that the server can validate players' requested moves. However, neither the server nor the client can establish whether a player has won the game in this application, method `GameOver` (lines 139143) always returns false. Each `Client` maintains its own GUI version of the Tic-Tac-Toe board to display the game. The clients can place marks only in empty squares on the board. Class `Square` ([Fig. 23.7](#)) is used to define squares on the Tic-Tac-Toe board.

[Page 1253]

Figure 23.5. Server side of client/server Tic-Tac-Toe program.

```

1 // Fig. 23.5: TicTacToeServer.cs
2 // This class maintains a game of Tic-Tac-Toe for two
3 // client applications.
4 using System;
5 using System.Windows.Forms;
6 using System.Net;
7 using System.Net.Sockets;
8 using System.Threading;
9 using System.IO;
10
11 public partial class TicTacToeServerForm : Form
12 {
13     public TicTacToeServerForm()
14     {
15         InitializeComponent();
16     } // end constructor
17
18     private byte[] board; // the local representation of the game board
19     private Player[] players; // two Player objects
20     private Thread[] playerThreads; // Threads for client interaction
21     private TcpListener listener; // listen for client connection
22     private int currentPlayer; // keep track of whose turn it is
23
24     private Thread getPlayers; // Thread for acquiring client connections
25     internal bool disconnected = false; // true if the server closes

```

[Page 1254]

```

25
26 // initialize variables and thread for receiving clients
27 private void TicTacToeServerForm_Load( object sender, EventArgs e )
28 {
29     board = new byte[ 9 ];
30     players = new Player[ 2 ];
31     playerThreads = new Thread[ 2 ];
32     currentPlayer = 0;
33
34     // accept connections on a different thread
35     getPlayers = new Thread( new ThreadStart( SetUp ) );
36     getPlayers.Start();
37 } // end method TicTacToeServerForm_Load
38
39 // notify Players to stop Running
40 private void TicTacToeServerForm_FormClosing( object sender,
41     FormClosingEventArgs e )
42 {
43     disconnected = true;
44     System.Environment.Exit( System.Environment.ExitCode );
45 } // end method TicTacToeServerForm_FormClosing
46
47 // delegate that allows method DisplayMessage to be called
48 // in the thread that creates and maintains the GUI
49 private delegate void DisplayDelegate( string message );
50
51 // method DisplayMessage sets displayTextBox's Text property
52 // in a thread-safe manner
53 internal void DisplayMessage( string message )
54 {
55     // if modifying displayTextBox is not thread safe
56     if ( displayTextBox.InvokeRequired )
57     {
58         // use inherited method Invoke to execute DisplayMessage
59         // via a delegate
60         Invoke( new DisplayDelegate( DisplayMessage ),
61             new object[] { message } );
62     } // end if
63     else // OK to modify displayTextBox in current thread
64         displayTextBox.Text += message;
65 } // end method DisplayMessage
66
67 // accepts connections from 2 players
68 public void SetUp()
69 {
70     DisplayMessage( "Waiting for players...\r\n" );
71
72     // set up Socket
73     listener =
74         new TcpListener( IPAddress.Parse( "127.0.0.1" ), 50000 );
75     listener.Start();

```

[Page 1255]

```

76
77 // accept first player and start a player thread
78 players[ 0 ] = new Player( listener.AcceptSocket(), this, 0 );
79 playerThreads[ 0 ] =
80     new Thread( new ThreadStart( players[ 0 ].Run ) );
81 playerThreads[ 0 ].Start();
82
83 // accept second player and start another player thread
84 players[ 1 ] = new Player( listener.AcceptSocket(), this, 1 );
85 playerThreads[ 1 ] =

```

```

86         new Thread( new ThreadStart( players[ 1 ].Run ) );
87     playerThreads[ 1 ].Start();
88
89     // let the first player know that the other player has connected
90     lock ( players[ 0 ] )
91     {
92         players[ 0 ].threadSuspended = false;
93         Monitor.Pulse( players[ 0 ] );
94     } // end lock
95 } // end method SetUp
96
97 // determine if a move is valid
98 public bool ValidMove( int location, int player )
99 {
100     // prevent another thread from making a move
101     lock ( this )
102     {
103         // while it is not the current player's turn, wait
104         while ( player != currentPlayer )
105             Monitor.Wait( this );
106
107         // if the desired square is not occupied
108         if ( !IsOccupied( location ) )
109         {
110             // set the board to contain the current player's mark
111             board[ location ] = ( byte ) ( currentPlayer == 0 ?
112                 'X' : 'O' );
113
114             // set the currentPlayer to be the other player
115             currentPlayer = ( currentPlayer + 1 ) % 2;
116
117             // notify the other player of the move
118             players[ currentPlayer ].OtherPlayerMoved( location );
119
120             // alert the other player that it's time to move
121             Monitor.Pulse( this );
122             return true;
123         } // end if
124         else
125             return false;
126     } // end lock
127 } // end method ValidMove
128

```

[Page 1256]

```

129 // determines whether the specified square is occupied
130 public bool IsOccupied( int location )
131 {
132     if ( board[ location ] == 'X' || board[ location ] == 'O' )
133         return true;
134     else
135         return false;
136 } // end method IsOccupied
137
138 // determines if the game is over
139 public bool GameOver()
140 {
141     // place code here to test for a winner of the game
142     return false;
143 } // end method GameOver
144 } // end class TicTacToeServerForm
145
146 // class Player represents a tic-tac-toe player

```

```

147 public class Player
148 {
149     internal Socket connection; // Socket for accepting a connection
150     private NetworkStream socketStream; // network data stream
151     private TicTacToeServerForm server; // reference to server
152     private BinaryWriter writer; // facilitates writing to the stream
153     private BinaryReader reader; // facilitates reading from the stream
154     private int number; // player number
155     private char mark; // player's mark on the board
156     internal bool threadSuspended = true; // if waiting for other player
157
158     // constructor requiring Socket, TicTacToeServerForm and int
159     // objects as arguments
160     public Player( Socket socket, TicTacToeServerForm serverValue,
161         int newNumber )
162     {
163         mark = (newNumber == 0 ? 'X' : 'O');
164         connection = socket;
165         server = serverValue;
166         number = newNumber;
167
168         // create NetworkStream object for Socket
169         socketStream = new NetworkStream( connection );
170
171         // create Streams for reading/writing bytes
172         writer = new BinaryWriter( socketStream );
173         reader = new BinaryReader( socketStream );
174     } // end constructor
175
176     // signal other player of move
177     public void OtherPlayerMoved( int location )
178     {
179         // signal that opponent moved
180         writer.Write( "Opponent moved." );

```

[Page 1257]

```

181         writer.Write( location ); // send location of move
182     } // end method OtherPlayerMoved
183
184     // allows the players to make moves and receive moves
185     // from the other player
186     public void Run()
187     {
188         bool done = false;
189
190         // display on the server that a connection was made
191         server.DisplayMessage( "Player " + ( number == 0 ? 'X' : 'O' )
192             + " connected\r\n" );
193
194         // send the current player's mark to the client
195         writer.Write( mark );
196
197         // if number equals 0 then this player is X,
198         // otherwise 0 must wait for X's first move
199         writer.Write( "Player " + ( number == 0 ?
200             "X connected.\r\n" : "O connected, please wait.\r\n" ) );
201
202         // X must wait for another player to arrive
203         if ( mark == 'X' )
204         {
205             writer.Write( "Waiting for another player." );
206
207             // wait for notification from server that another

```

```

208         // player has connected
209         lock ( this )
210         {
211             while ( threadSuspended )
212                 Monitor.Wait( this );
213         } // end lock
214
215         writer.Write( "Other player connected. Your move." );
216     } // end if
217
218     // play game
219     while ( !done )
220     {
221         // wait for data to become available
222         while ( connection.Available == 0 )
223         {
224             Thread.Sleep( 1000 );
225
226             if ( server.disconnected )
227                 return;
228         } // end while
229
230         // receive data
231         int location = reader.ReadInt32();
232


---


233         [Page 1258]
234         // if the move is valid, display the move on the
235         // server and signal that the move is valid
236         if ( server.ValidMove( location, number ) )
237         {
238             server.DisplayMessage( "loc: " + location + "\r\n" );
239             writer.Write( "Valid move." );
240         } // end if
241         else // signal that the move is invalid
242             writer.Write( "Invalid move, try again." );
243
244         // if game is over, set done to true to exit while loop
245         if ( server.GameOver() )
246             done = true;
247     } // end while loop
248
249     // close the socket connection
250     writer.Close();
251     reader.Close();
252     socketStream.Close();
253     connection.Close();
254 } // end method Run
255 } // end class Player

```

Figure 23.6. Client side of client/server Tic-Tac-Toe program.

(This item is displayed on pages 1260 - 1266 in the print version)

```

1 // Fig. 23.6: TicTacToeClient.cs
2 // Client for the TicTacToe program.
3 using System;
4 using System.Drawing;
5 using System.Windows.Forms;
6 using System.Net.Sockets;
7 using System.Threading;

```

```

8 using System.IO;
9
10 public partial class TicTacToeClientForm : Form
11 {
12     public TicTacToeClientForm()
13     {
14         InitializeComponent();
15     } // end constructor
16
17     private Square[ , ] board; // local representation of the game board
18     private Square currentSquare; // the Square that this player chose
19     private Thread outputThread; // Thread for receiving data from server
20     private TcpClient connection; // client to establish connection
21     private NetworkStream stream; // network data stream
22     private BinaryWriter writer; // facilitates writing to the stream
23     private BinaryReader reader; // facilitates reading from the stream
24     private char myMark; // player's mark on the board
25     private bool myTurn; // is it this player's turn?
26     private SolidBrush brush; // brush for drawing X's and O's
27     private bool done = false; // true when game is over
28
29     // initialize variables and thread for connecting to server
30     private void TicTacToeClientForm_Load( object sender, EventArgs e )
31     {
32         board = new Square[ 3, 3 ];
33
34         // create 9 Square objects and place them on the board
35         board[ 0, 0 ] = new Square( board0Panel, ' ', 0 );
36         board[ 0, 1 ] = new Square( board1Panel, ' ', 1 );
37         board[ 0, 2 ] = new Square( board2Panel, ' ', 2 );
38         board[ 1, 0 ] = new Square( board3Panel, ' ', 3 );
39         board[ 1, 1 ] = new Square( board4Panel, ' ', 4 );
40         board[ 1, 2 ] = new Square( board5Panel, ' ', 5 );
41         board[ 2, 0 ] = new Square( board6Panel, ' ', 6 );
42         board[ 2, 1 ] = new Square( board7Panel, ' ', 7 );
43         board[ 2, 2 ] = new Square( board8Panel, ' ', 8 );
44
45         // create a SolidBrush for writing on the Squares
46         brush = new SolidBrush( Color.Black );
47
48         // make connection to server and get the associated
49         // network stream
50         connection = new TcpClient( "127.0.0.1", 50000 );
51         stream = connection.GetStream();
52         writer = new BinaryWriter( stream );
53         reader = new BinaryReader( stream );

```

[Page 1261]

```

54
55     // start a new thread for sending and receiving messages
56     outputThread = new Thread( new ThreadStart( Run ) );
57     outputThread.Start();
58 } // end method TicTacToeClientForm_Load
59
60 // repaint the Squares
61 private void TicTacToeClientForm_Paint( object sender,
62     PaintEventArgs e )
63 {
64     PaintSquares();
65 } // end method TicTacToeClientForm_Load
66
67 // game is over
68 private void TicTacToeClientForm_FormClosing( object sender,

```

```

69     FormClosingEventArgs e )
70 {
71     done = true;
72     System.Environment.Exit( System.Environment.ExitCode );
73 } // end TicTacToeClientForm_FormClosing
74
75 // delegate that allows method DisplayMessage to be called
76 // in the thread that creates and maintains the GUI
77 private delegate void DisplayDelegate( string message );
78
79 // method DisplayMessage sets displayTextBox's Text property
80 // in a thread-safe manner
81 private void DisplayMessage( string message )
82 {
83     // if modifying displayTextBox is not thread safe
84     if ( displayTextBox.InvokeRequired )
85     {
86         // use inherited method Invoke to execute DisplayMessage
87         // via a delegate
88         Invoke( new DisplayDelegate( DisplayMessage ),
89             new object[] { message } );
90     } // end if
91     else // OK to modify displayTextBox in current thread
92         displayTextBox.Text += message;
93 } // end method DisplayMessage
94
95 // delegate that allows method ChangeIdLabel to be called
96 // in the thread that creates and maintains the GUI
97 private delegate void ChangeIdLabelDelegate( string message );
98
99 // method ChangeIdLabel sets displayTextBox's Text property
100 // in a thread-safe manner
101 private void ChangeIdLabel( string label )
102 {
103     // if modifying idLabel is not thread safe
104     if ( idLabel.InvokeRequired )
105     {
106         [Page 1262]
107         // use inherited method Invoke to execute ChangeIdLabel
108         // via a delegate
109         Invoke( new ChangeIdLabelDelegate( ChangeIdLabel ),
110             new object[] { label } );
111     } // end if
112     else // OK to modify idLabel in current thread
113         idLabel.Text = label;
114 } // end method ChangeIdLabel
115
116 // draws the mark of each square
117 public void PaintSquares()
118 {
119     Graphics g;
120
121     // draw the appropriate mark on each panel
122     for ( int row = 0; row < 3; row++ )
123     {
124         for ( int column = 0; column < 3; column++ )
125         {
126             // get the Graphics for each Panel
127             g = board[ row, column ].SquarePanel.CreateGraphics();
128
129             // draw the appropriate letter on the panel
130             g.DrawString( board[ row, column ].Mark.ToString(),

```

```

130         board[0][0].Font, brush, 10, 8 );
131     } // end for
132 } // end for
133 } // end method PaintSquares
134
135 // send location of the clicked square to server
136 private void square_MouseUp( object sender,
137     System.Windows.Forms.MouseEventArgs e )
138 {
139     // for each square check if that square was clicked
140     for ( int row = 0; row < 3; row++ )
141     {
142         for ( int column = 0; column < 3; column++ )
143         {
144             if ( board[ row, column ].SquarePanel == sender )
145             {
146                 CurrentSquare = board[ row, column ];
147
148                 // send the move to the server
149                 SendClickedSquare( board[ row, column ].Location );
150             } // end if
151         } // end for
152     } // end for
153 } // end method square_MouseUp
154
155 // control thread that allows continuous update of the
156 // TextBox display
157 public void Run()
158 {

```

[Page 1263]

```

159     // first get players's mark (X or O)
160     myMark = reader.ReadChar();
161     ChangeIdLabel( "You are player \"\" + myMark + "\"\" );
162     myTurn = ( myMark == 'X' ? true : false );
163
164     // process incoming messages
165     try
166     {
167         // receive messages sent to client
168         while ( !done )
169             ProcessMessage( reader.ReadString() );
170     } // end try
171     catch ( IOException )
172     {
173         MessageBox.Show( "Server is down, game over", "Error",
174             MessageBoxButtons.OK, MessageBoxIcon.Error );
175     } // end catch
176 } // end method Run
177
178 // process messages sent to client
179 public void ProcessMessage( string message )
180 {
181     // if the move the player sent to the server is valid
182     // update the display, set that square's mark to be
183     // the mark of the current player and repaint the board
184     if ( message == "Valid move." )
185     {
186         DisplayMessage( "Valid move, please wait.\r\n" );
187         currentSquare.Mark = myMark;
188         PaintSquares();
189     } // end if
190     else if ( message == "Invalid move, try again." )

```

```

191     {
192         // if the move is invalid, display that and it is now
193         // this player's turn again
194         DisplayMessage( message + "\r\n" );
195         myTurn = true;
196     } // end else if
197     else if ( message == "Opponent moved." )
198     {
199         // if opponent moved, find location of their move
200         int location = reader.ReadInt32();
201
202         // set that square to have the opponents mark and
203         // repaint the board
204         board[ location / 3, location % 3 ].Mark =
205             ( myMark == 'X' ? 'O' : 'X' );
206         PaintSquares();
207
208         DisplayMessage( "Opponent moved. Your turn.\r\n" );
209

```

[Page 1264]

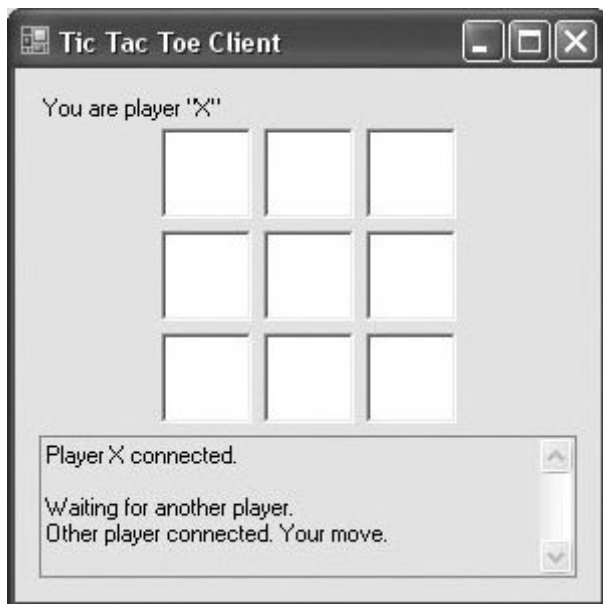
```

210         // it is now this player's turn
211         myTurn = true;
212     } // end else if
213     else
214         DisplayMessage( message + "\r\n" ); // display message
215 } // end method ProcessMessage
216
217 // sends the server the number of the clicked square
218 public void SendClickedSquare( int location )
219 {
220     // if it is the current player's move right now
221     if ( myTurn )
222     {
223         // send the location of the move to the server
224         writer.Write( location );
225
226         // it is now the other player's turn
227         myTurn = false;
228     } // end if
229 } // end method SendClickedSquare
230
231 // write-only property for the current square
232 public Square CurrentSquare
233 {
234     set
235     {
236         currentSquare = value;
237     } // end set
238 } // end property CurrentSquare
239 } // end class TicTacToeClientForm

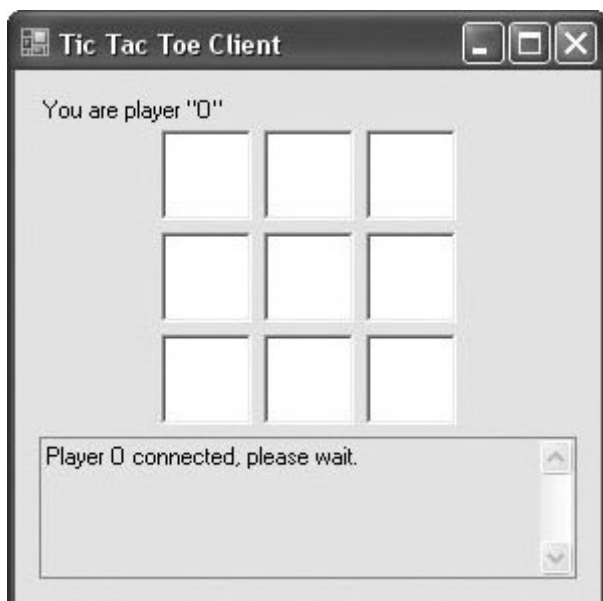
```

At the start of the game.

(a)

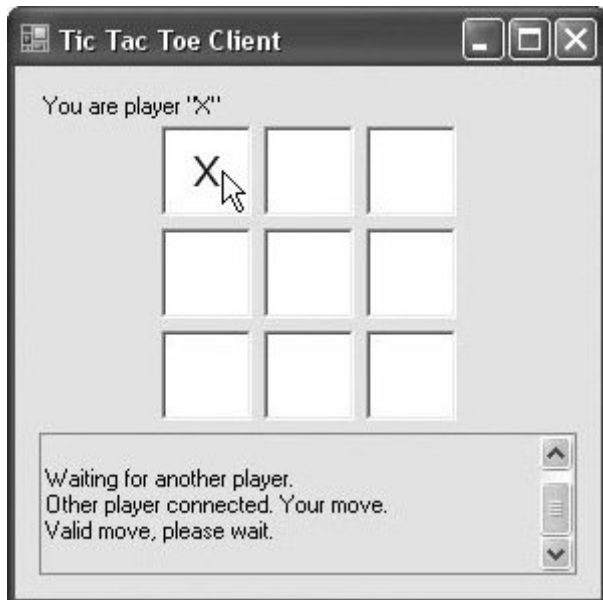


(b)

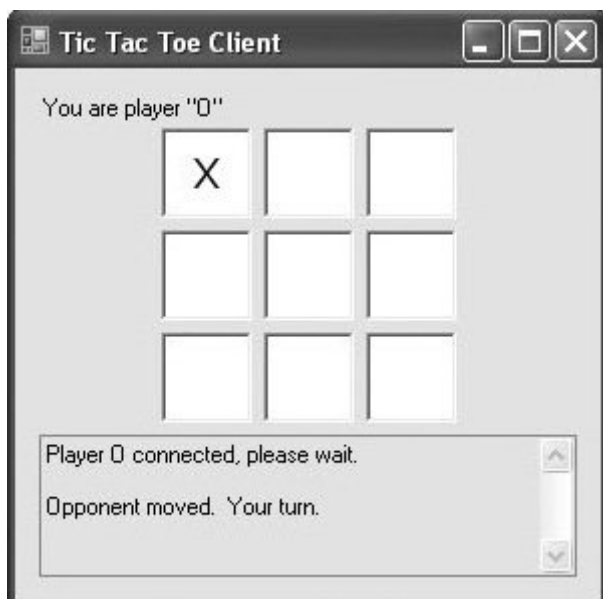


[Page 1265]**After Player X makes the first move.**

(c)

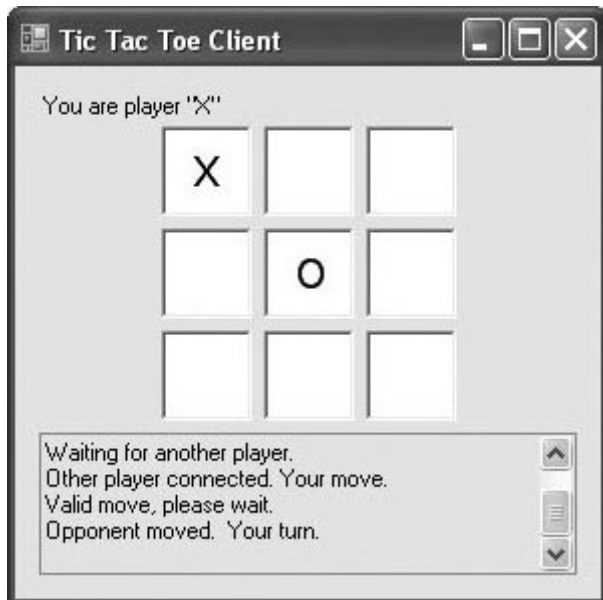


(d)

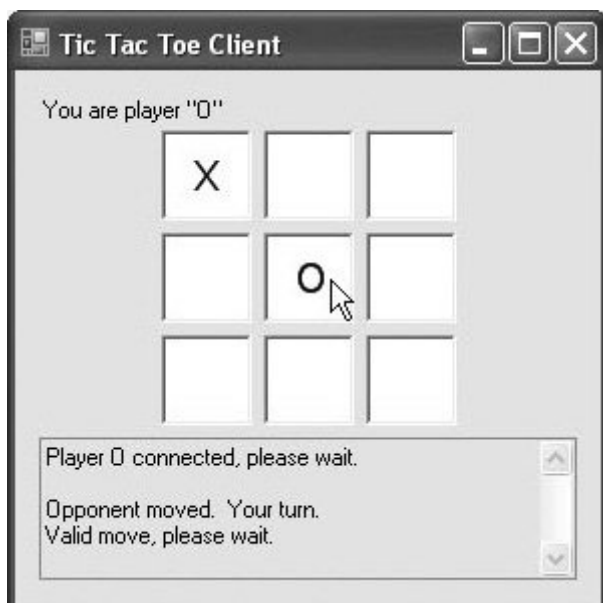


After Player O makes the second move.

(e)

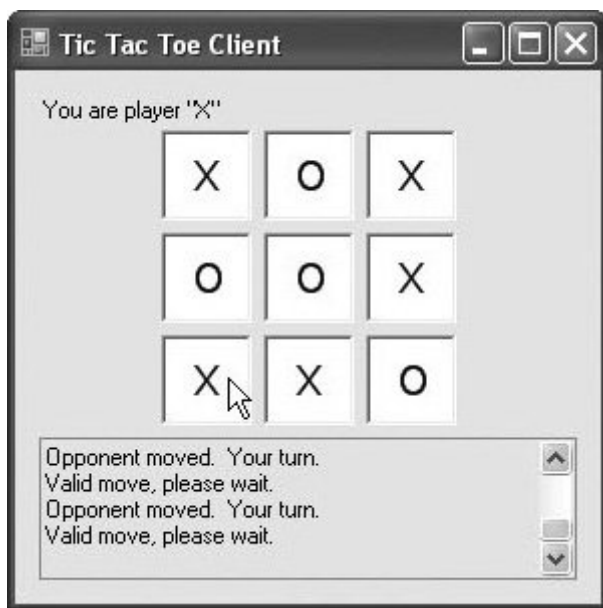


(f)

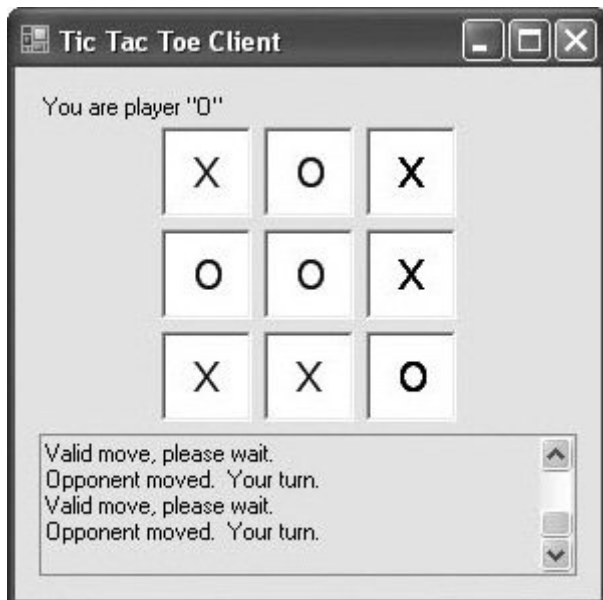


After Player X makes the final move.

(g)



(h)

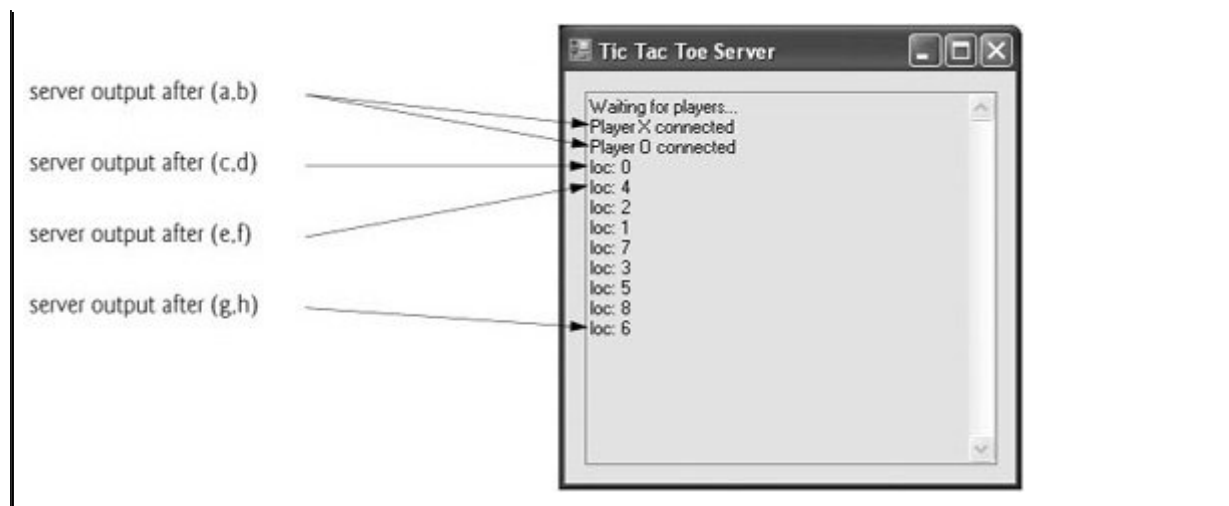


[Page 1266]

The Tie Tac Toe Server's output from the client interactions

(i)

[\[View full size image\]](#)

**Figure 23.7. Class Square.**

(This item is displayed on pages 1266 - 1267 in the print version)

```

1 // Fig. 23.7: Square.cs
2 // A Square on the TicTacToe board.
3 using System.Windows.Forms;
4
5 // the representation of a square in a tic-tac-toe grid
6 public class Square
7 {
8     private Panel panel; // GUI Panel that represents this Square
9     private char mark; // player's mark on this Square (if any)
10    private int location; // location on the board of this Square
11
12    // constructor
13    public Square( Panel newPanel, char newMark, int newLocation )
14    {
15        panel = newPanel;
16        mark = newMark;
17        location = newLocation;
18    } // end constructor
19
20    // property SquarePanel; the panel which the square represents
21    public Panel SquarePanel
22    {
23        get
24        {
25            return panel;
26        } // end get
27    } // end property SquarePanel
28
29    // property Mark; the mark on the square
30    public char Mark
31    {

```

[Page 1267]

```

32        get
33        {
34            return mark;
35        } // end get
36        set
37        {
38            mark = value;
39        } // end set
40    } // end property Mark

```

```

41
42 // property Location; the square's location on the board
43 public int Location
44 {
45     get
46     {
47         return location;
48     } // end get
49 } // end property Location
50 } // end class Square

```

TicTacToeServerForm Class

TicTacToeServerForm ([Fig. 23.5](#)) uses its Load event handler (lines 2737) to create a byte array to store the moves the players have made (line 29). The program creates an array of two references to Player objects (line 30) and an array of two references to Thread objects (line 31). Each element in both arrays corresponds to a Tic-Tac-Toe player. Variable currentPlayer is set to 0 (line 32), which corresponds to player "X". In our program, player "X" makes the first move. Lines 3536 create and start Thread getPlayers, which the TicTacToeServerForm uses to accept connections so that the current Thread does not block while awaiting players.

[Page 1258]

Lines 4965 define DisplayDelegate and DisplayMessage, allowing any thread to modify displayTextBox's Text property. This time, the DisplayMessage method is declared as internal, so it can be called inside a method of class Player through a TicTacToeServerForm reference.

Thread getPlayers executes method Setup (lines 6895), which creates a TcpListener object to listen for requests on port 50000 (lines 7375). This object then listens for connection requests from the first and second players. Lines 78 and 84 instantiate Player objects representing the players, and lines 7981 and 8587 create two Threads that execute the Run methods of each Player object.

The Player constructor ([Fig. 23.5](#), lines 160174) receives as arguments a reference to the Socket object (i.e., the connection to the client), a reference to the TicTacToeServerForm object and an int indicating the player number (from which the constructor infers the mark, "X" or "O" used by that player). In this case study, TicTacToeServerForm calls method Run (lines 186253) after instantiating a Player object. Lines 191200 notify the server of a successful connection and send to the client the char that the client will place on the board when making a move. If Run is executing for Player "X", lines 205215 execute, causing Player "X" to wait for a second player to connect. Lines 211212 define a while statement that suspends the Player "X" Thread until the server signals that Player "O" has connected. The server notifies the Player of the connection by setting the Player's threadSuspended variable to false (line 92). When threadSuspended becomes false, Player exits the while statement at lines 211212.

Method Run executes the while statement at lines 21924), enabling the user to play the game. Each iteration of this statement waits for the client to send an int specifying where on the board to place the "X" or "O" the Player then places the mark on the board, if the specified mark location is valid (e.g., that location does not already contain a mark). Note that the while statement continues execution only if bool variable done is false. This variable is set to true by event handler TicTacToeServerForm_FormClosing of class TicTacToeServerForm, which is invoked when the server closes the connection.

[Page 1259]

Line 222 of [Fig. 23.5](#) begins a while statement that loops until Socket property Available indicates that there is information to receive from the Socket (or until the server disconnects from the client). If there is no information, the Thread goes to sleep for one second. On awakening, the Thread uses property Disconnected to check whether server variable disconnected is true (line 226). If the value is true, the Thread exits the method (thus terminating the Thread); otherwise, the Thread loops again. However, if property Available indicates that there is data to receive, the while statement of lines 222228 terminates, enabling the information to be processed.

This information contains an int representing the location in which the client wants to place a mark. Line 231 calls method ReadInt32 of the BinaryReader object (which reads from the NetworkStream created with the Socket) to read this int. Line 235 then passes the int to TicTacToeServerForm method ValidMove. If this method validates the move, the Player places the mark in the desired location.

Method ValidMove (lines 98127) sends to the client a message indicating whether the move was valid. Locations on the board correspond to numbers from 08 (02 for the top row, 35 for the middle and 68 for the bottom). All statements in method ValidMove are enclosed in a lock statement that allows only one move to be attempted at a time. This prevents two players from modifying the game's state information simultaneously. If the Player attempting to validate a move is not the current player (i.e., the one allowed to make a move), that Player is placed in a Wait state until it is that Player's turn to move. If the user attempts to place a mark on a location that already contains a mark, method ValidMove returns false. However, if the user has selected an unoccupied location (line 108), lines 111112 place the mark on the local representation of the board. Line 118 notifies the other Player that a move has been made, and line 121 invokes the Pulse method so that the waiting Player can validate a move. The method then returns true to indicate that the move is valid.

When a TicTacToeClientForm application ([Fig. 23.6](#)) executes, it creates a TextBox to display messages from the server and the Tic-Tac-Toe board representation. The board is created out of nine Square objects ([Fig. 23.7](#)) that contain Panels on which the user can click, indicating the position on the board in which to place a mark. The TicTacToeClientForm's Load event handler (lines 3058) opens a connection to the server (line 50) and obtains a reference to the connection's associated NetworkStream object from TcpClient (line 51). Lines 5657 start a thread to read messages sent from the server to the client. The server passes messages (for example, whether each move is valid) to method ProcessMessage (lines 179215). If the message indicates that a move is valid (line 184), the client sets its Mark to the current square (the square that the user clicked) and repaints the board. If the message indicates that a move is invalid (line 190), the client notifies the user to click a different square. If the message indicates that the opponent made a move (line 197), line 200 reads an int from the server specifying where on the board the client should place the opponent's Mark. TicTacToeClientForm includes a delegate/method pair for allowing threads to modify idLabel's Text property (lines 97113), as well as DisplayDelegate and DisplayMessage for modifying displayTextBox's Text property (lines 7793).



[Page 1267]

23.9. WebBrowser Control

With FCL 2.0, Microsoft introduced the `WebBrowser` control, which enables applications to incorporate Web browsing capabilities. The control provides methods for navigating Web pages and maintains its own history of Web sites visited. It also generates events as the user interacts with the content displayed in the control, so your application can respond to events such as the user clicking the links displayed in the content.

[Figure 23.8](#) demonstrates the `WebBrowser` control's capabilities. Class `BrowserForm` provides the basic functionality of a Web browser, allowing the user to navigate to a URL, to move backward and forward through the history of visited sites and to reload the current Web page.

Figure 23.8. `WebBrowser` control example.

```

1 // Fig. 23.8: Browser.cs
2 // WebBrowser control example.
3 using System;
4 using System.Windows.Forms;
5
6 public partial class BrowserForm : Form
7 {
8     public BrowserForm()
9     {
10         InitializeComponent();
11     } // end constructor
12
13     // navigate back one page
14     private void backButton_Click( object sender, EventArgs e )
15     {
16
17         [Page 1268]
18         webBrowser.GoBack();
19     } // end method backButton_Click
20
21     // navigate forward one page
22     private void forwardButton_Click( object sender, EventArgs e )
23     {
24         webBrowser.GoForward();
25     } // end method forwardButton_Click
26
27     // stop loading the current page
28     private void stopButton_Click( object sender, EventArgs e )
29     {
30         webBrowser.Stop();
31     } // end method stopButton_Click
32
33     // reload the current page
34     private void reloadButton_Click( object sender, EventArgs e )
35     {
36         webBrowser.Refresh();
37     } // end method reloadButton_Click
38
39     // navigate to the user's home page
40     private void homeButton_Click( object sender, EventArgs e )
41     {
42         webBrowser.GoHome();
43     } // end method homeButton_Click
44
45     // if the user pressed enter, navigate to the specified URL
46     private void navigationTextBox_KeyDown( object sender,
47         KeyEventArgs e )
48     {
49         if ( e.KeyCode == Keys.Enter )

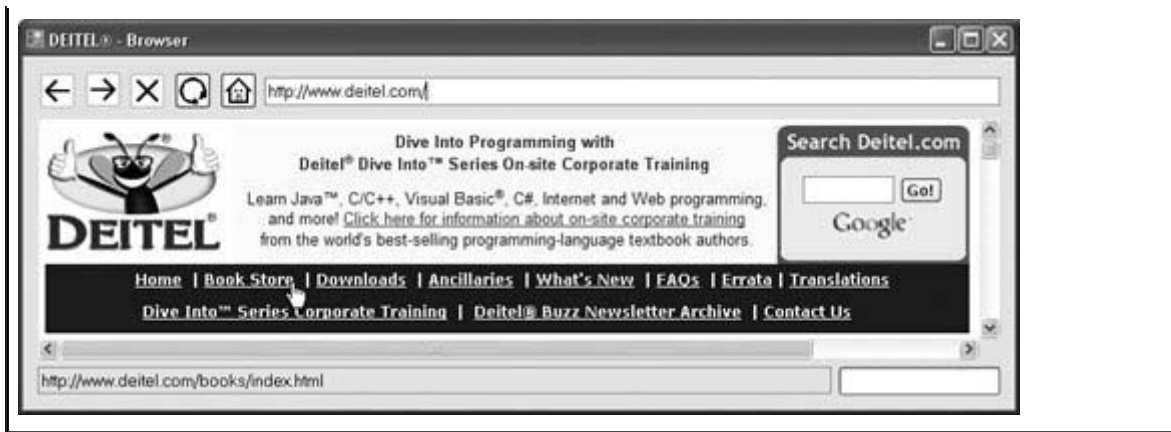
```

```
48         webBrowser.Navigate( navigationTextBox.Text );
49     } // end method navigationTextBox_KeyDown
50
51     // enable stopButton while the current page is loading
52     private void webBrowser_Navigating( object sender,
53         WebBrowserNavigatingEventArgs e )
54     {
55         stopButton.Enabled = true;
56     } // end method webBrowser_Navigating
57
58     // update the status text
59     private void webBrowser_StatusTextChanged( object sender,
60         EventArgs e )
61     {
62         statusTextBox.Text = webBrowser.StatusText;
63     } // end method webBrowser_StatusTextChanged
64
65     // update the ProgressBar for how much of the page has been loaded
66     private void webBrowser_ProgressChanged( object sender,
67         WebBrowserProgressChangedEventArgs e )
68     {
```

[Page 1269]

```
69         pageProgressBar.Value =
70             ( int ) ( ( 100 * e.CurrentProgress ) / e.MaximumProgress );
71     } // end method webBrowser_ProgressChanged
72
73     // update the web browser's controls appropriately
74     private void webBrowser_DocumentCompleted( object sender,
75         WebBrowserDocumentCompletedEventArgs e )
76     {
77         // set the text in navigationTextBox to the current page's URL
78         navigationTextBox.Text = webBrowser.Url.ToString();
79
80         // enable or disable backButton and forwardButton
81         backButton.Enabled = webBrowser.CanGoBack;
82         forwardButton.Enabled = webBrowser.CanGoForward;
83
84         // disable stopButton
85         stopButton.Enabled = false;
86
87         // clear the pageProgressBar
88         pageProgressBar.Value = 0;
89     } // end method webBrowser_DocumentCompleted
90
91     // update the title of the Browser
92     private void webBrowser_DocumentTitleChanged( object sender,
93         EventArgs e )
94     {
95         this.Text = webBrowser.DocumentTitle + " - Browser";
96     } // end method webBrowser_DocumentTitleChanged
97 } // end class BrowserForm
```

[\[View full size image\]](#)



Lines 1441 define five Click event handlers, one for each of the five navigation Buttons that appear at the top of the Form. Each event handler calls a corresponding WebBrowser method. WebBrowser method GoBack (line 16) causes the control to navigate back to the previous page in the navigation history. Method GoForward (line 22) causes the control to navigate forward to the next page in the navigation history. Method Stop (line 28) causes the control to stop loading the current page. Method Refresh (line 34) causes the control to reload the current page. Method GoHome (line 40) causes the control to navigate to the user's home page, as defined under Internet Explorer's settings (under Tools > Internet Options... in the Home page section).

[Page 1270]

The TextBox to the right of the navigation buttons allows the user to enter the URL of a Web site to browse. When the user types each keystroke in the TextBox, the event handler at lines 4449 executes. If the key pressed was Enter, line 48 calls WebBrowser method Navigate to retrieve the document at the specified URL.

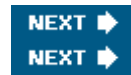
The WebBrowser control generates a Navigating event when the WebBrowser starts loading a new page. When this occurs, the event handler at lines 5256 executes, and line 55 enables stopButton so that the user can cancel the loading of the Web page.

Typically, a user can see the status of a loading Web page at the bottom of the browser window. For this reason, we include a TextBox control (named statusTextBox) and a ProgressBar control (named pageProgressBar) at the bottom of our Form. The WebBrowser control generates a StatusTextChanged event when the webBrowser's StatusText property changes. The event handler for this event (lines 5963) assigns the new contents of the control's StatusText property to statusTextBox's Text property (line 62), so the user can monitor the WebBrowser's status messages. The control generates a ProgressChanged event when the webBrowser control's page-loading progress is updated. The ProgressChanged event handler (lines 6671) updates pageProgressBar's Value (lines 6970) to reflect how much of the current document has been loaded.

When the WebBrowser finishes loading a document, the control generates a DocumentCompleted event. This executes the event handler at lines 7489. Line 78 updates the contents of navigationTextBox so that it shows the URL of the currently loaded page (WebBrowser property Url). This is particularly important if the user browses to another Web page by clicking a link in the existing page. Lines 8182 use properties CanGoBack and CanGoForward to determine whether the back and forward buttons should be enabled or disabled. Since the document is now loaded, line 85 disables stopButton. Line 88 sets pageProgressBar's Value to 0 to indicate that no content is currently being loaded.

Lines 9296 define an event handler for the DocumentTitleChanged event, which occurs when a new

document is loaded in the WebBrowser control. Line 95 sets BrowserForm's Text property (which is displayed in the Form's title bar) to the WebBrowser's current DocumentTitle.



[Page 1270 (continued)]

23.10. .NET Remoting

The .NET framework provides a distributed computing technology called .NET remoting that allows a program to access objects on another machine over a network. .NET remoting is similar in concept to RMI (remote method invocation) in Java and RPC (remote procedure call) in procedural programming languages. .NET remoting is also similar to Web services ([Chapter 22](#)) with a few key differences. With Web services, a client application communicates with a Web service that is hosted by a Web server. The client and the Web service can be written in any language, as long as they can transmit messages in SOAP. With .NET remoting, a client application communicates with a server application, both of which must be written in .NET languages. Using .NET remoting, a client and a server can communicate via method calls and objects can be transmitted between applications a process known as marshaling the objects.

[Page 1271]

Channels

The client and the server are able to communicate with one another through channels. Channels typically use either the HTTP protocol or the TCP protocol to transmit messages. The advantage of an HTTP channel is that firewalls usually permit HTTP connections by default, while they normally block unfamiliar TCP connections. The advantage of a TCP channel is better performance than an HTTP channel. In a .NET remoting application, the client and the server each create a channel, and both channels must use the same protocol to communicate with one another. In our example, we use HTTP channels.

Marshaling

There are two ways to marshal an object by value and by reference. Marshal by value requires that the object be serializable that is, capable of being represented as a formatted message that can be sent between applications through a channel. The receiving end of the channel deserializes the object to obtain a copy of the original object. To enable an object to be serialized and deserialized, its class must either be declared with attribute `[ Serializable ]` or must implement interface *ISerializable*.

Marshal by reference requires that the object's class extend class `MarshalByRefObject` of namespace `System`. An object that is marshaled by reference is referred to as a remote object, and its class is referred to as a remote class. When an object is marshaled by reference, the object itself is not transmitted. Instead, two proxy objects are created a transparent proxy and a real proxy. The transparent proxy provides all the public services of a remote object. Typically, a client calls the methods and properties of the transparent proxy as if it were the remote object. The transparent proxy then calls the `Invoke` method of the real proxy. This sends the appropriate message from the client channel to the server channel. The server receives this message and performs the specified method call or accesses the specified property on the actual object, which resides on the server. In our example, we marshal a remote object by reference.

Weather Information Application Using .NET Remoting

We now present a .NET remoting example that downloads the Traveler's Forecast weather information from the National Weather Service Web site:

<http://iwin.nws.noaa.gov/iwin/us/traveler.html>

[Note: As we developed this example, the National Weather Service indicated that the information provided on the Traveler's Forecast Web page would be provided by a Web service in the near future. The information we use in this example depends directly on the format of the Traveler's Forecast Web page. If you have trouble running this example, please refer to the FAQ page at www.deitel.com/faq.html. This potential problem demonstrates a benefit of using Web services or .NET remoting to implement distributed computing applications that may change in the future. Separating the server part of the application that depends on the format of an outside data source from the client part of the application allows the server implementation to be updated without requiring any changes to the client.]

Our .NET remoting application consists of five components:

1. Serializable class `CityWeather`, which represents the weather report for one city.
-
- [Page 1272]**
2. Interface `Report`, which declares a property `Reports` that the marshaled object provides to a client application to obtain a collection of `CityWeather` objects.
 3. Remote class `ReportInfo`, which extends class `MarshalByRefObject`, implements interface `Report` and will be instantiated only on the server.
 4. A `WeatherServer` application that sets up a server channel and makes the `ReportInfo` class available at a particular URI (uniform resource identifier).
 5. A `WeatherClient` application that sets up a client channel and requests a `ReportInfo` object from the `WeatherServer` to retrieve the day's weather report.

Class `CityWeather`

Class `CityWeather` ([Fig. 23.9](#)) contains weather information for one city. Class `CityWeather` is declared in namespace `weather` (line 5) for reusability and will be published in the `weather.dll` class library file that both the server application and the client application must reference. For this reason, you should place this class (and interface `Report` from [Fig. 23.10](#)) in a class library project. Class `CityWeather` is declared with attribute `Serializable` (line 7), which indicates that an object of class `CityWeather` can be marshaled by value. This is necessary because `CityWeather` objects will be returned by the `ReportInfo` object's `Reports` property, and the return values of the methods and properties declared by a remote class must themselves be marshaled by value from the server to the client. (The argument values in method calls will also be marshaled by value from the client to the server.) So when the client calls a get accessor that returns `CityWeather` objects, the server channel will serialize the `CityWeather` objects in a message that the client channel can deserialize to create copies of the original `CityWeather` objects. Class `CityWeather` also implements interface `IComparable` (line 8) so that an `ArrayList` of `CityWeather` objects can be sorted alphabetically.

Figure 23.9. Class `cityweather`.

```
1 // Fig. 23.9: CityWeather.cs
2 // Class representing the weather information for one city.
3 using System;
4
5 namespace Weather
6 {
7     [ Serializable ]
8     public class CityWeather : IComparable
9     {
10         private string cityName;
11         private string description;
12         private string temperature;
13
14         public CityWeather( string city, string information,
15             string degrees )
16         {
17             cityName = city;
18             description = information;
19             temperature = degrees;
20         } // end constructor
21
22         [Page 1273]
23         // read-only property that gets city's name
24         public string CityName
25         {
26             get
27             {
28                 return cityName;
29             } // end get
30         } // end property CityName
31
32         // read-only property that gets city's weather description
33         public string Description
34         {
35             get
36             {
37                 return description;
38             } // end get
39         } // end property Description
40
41         // read-only property that gets city's temperature
42         public string Temperature
43         {
44             get
45             {
46                 return temperature;
47             } // end get
48         } // end property Temperature
49
50         // implementation of CompareTo method for alphabetizing
51         public int CompareTo( object other )
52         {
53             return string.Compare(
54                 cityName, ( ( CityWeather ) other ).CityName );
55         } // end method Compare
56
57         // return string representation of this CityWeather object
58         // (used to display the weather report on the server console)
59         public override string ToString()
60         {
61             return cityName + " | " + temperature + " | " + description;
62         } // end method ToString
```

```

62     } // end class CityWeather
63 } // end namespace Weather

```

Figure 23.10. Interface Report in namespace Weather.

(This item is displayed on page 1274 in the print version)

```

1 // Fig. 23.10: Report.cs
2 // Interface that defines a property for getting
3 // the information in a weather report.
4 using System;
5 using System.Collections;
6
7 namespace Weather
8 {
9     public interface Report
10    {
11        ArrayList Reports
12        {
13            get;
14        } // end property Reports
15    } // end interface Report
16 } // end namespace Weather

```

[Page 1273]

CityWeather contains three instance variables (lines 1012) for storing the city's name, high/low temperatures and weather condition description. The CityWeather constructor (lines 1420) initializes the three instance variables. Lines 2347 declare three read-only properties that allow the values of the three instance variables to be retrieved. CityWeather implements IComparable, so it must declare a method called CompareTo that takes an object reference and returns an int (lines 5054). Also, we want to alphabetize CityWeather objects by their cityName, so CompareTo calls string method Compare with the cityName of the two CityWeather objects. Class CityWeather also overrides the ToString method to display information for this city (lines 5861). Method ToString is used by the server application to display the weather information retrieved from the Traveler's Forecast Web page in the console.

[Page 1274]

Interface Report

[Figure 23.10](#) shows the code for interface Report. Interface Report is also declared in namespace weather (line 7) and will be included with class CityWeather in the weather.dll class library file, so it can be used in both the client and server applications. Report declares a read-only property (lines 1114) with a get accessor that returns an ArrayList of CityWeather objects. The client application will use this property to retrieve the information in the weather report each city's name, high/low temperature and weather condition.

Class ReportInfo

Remote class `ReportInfo` ([Fig. 23.11](#)) implements interface `Report` (line 10) of namespace `Weather` (specified by the `using` directive in line 8). `ReportInfo` also extends base class `MarshalByRefObject`. Class `ReportInfo` is part of the remote `WeatherServer` application and will not be directly available to the client application.

Figure 23.11. Class `ReportInfo`, which implements interface `Report`, is marshaled by reference.

```

1 // Fig. 23.11: ReportInfo.cs
2 // Class that implements interface Report, retrieves
3 // and returns data on weather
4 using System;
5 using System.Collections;
6 using System.IO;
7 using System.Net;
8 using Weather;
9
10 public class ReportInfo : MarshalByRefObject, Report
11 {
12     private ArrayList cityList; // cities, temperatures, descriptions
13
14     public ReportInfo()
15     {
16         cityList = new ArrayList();
17
18         // create WebClient to get access to Web page
19         WebClient myClient = new WebClient();
20
21         // get StreamReader for response so we can read page
22         StreamReader input = new StreamReader( myClient.OpenRead(
23             "http://iwin.nws.noaa.gov/iwin/us/traveler.html" ) );
24
25         string separator1 = "TAV12"; // indicates first batch of cities
26         string separator2 = "TAV13"; // indicates second batch of cities
27
28         // locate separator1 in Web page
29         while ( !input.ReadLine().StartsWith( separator1 ) ); // do nothing
30         ReadCities( input ); // read the first batch of cities
31
32         // locate separator2 in Web page
33         while ( !input.ReadLine().StartsWith( separator2 ) ); // do nothing
34         ReadCities( input ); // read the second batch of cities
35
36         cityList.Sort(); // sort list of cities by alphabetical order
37         input.Close(); // close StreamReader to NWS server
38
39         // display the data on the server side
40         Console.WriteLine( "Data from NWS Web site:" );
41
42         foreach ( CityWeather city in cityList )
43         {
44             Console.WriteLine( city );
45         } // end foreach
46     } // end constructor
47
48     // utility method that reads a batch of cities
49     private void ReadCities( StreamReader input )
50     {
51         // day format and night format
52         string dayFormat =
53             "CITY          WEA      HI/LO   WEA      HI/LO";

```

```

54     string nightFormat =
55         "CITY          WEA          LO/HI    WEA          LO/HI";
56     string inputLine = "";
57
58     // locate header that begins weather information
59     do
60     {
61         [Page 1276]
62         inputLine = input.ReadLine();
63     } while ( !inputLine.Equals( dayFormat ) &&
64             !inputLine.Equals( nightFormat ) );
65     inputLine = input.ReadLine(); // get first city's data
66
67     // while there are more cities to read
68     while ( inputLine.Length > 28 )
69     {
70         // create CityWeather object for city
71         CityWeather weather = new CityWeather(
72             inputLine.Substring( 0, 16 ),
73             inputLine.Substring( 16, 7 ),
74             inputLine.Substring( 23, 7 ) );
75
76         cityList.Add( weather ); // add to ArrayList
77         inputLine = input.ReadLine(); // get next city's data
78     } // end while
79 } // end method ReadCities
80
81 // property for getting the cities' weather reports
82 public ArrayList Reports
83 {
84     get
85     {
86         return cityList;
87     } // end get
88 } // end property Reports
89 } // end class ReportInfo

```

Lines 1446 declare the ReportInfo constructor. Line 19 creates a WebClient (namespace System.Net) object to interact with a data source that is specified by a URL in this case, the URL for the NWS Traveler's Forecast page (<http://iwin.nws.noaa.gov/iwin/us/traveler.html>). Lines 2223 call WebClient method OpenRead, which returns a Stream that the program can use to read data containing the weather information from the specified URL. This Stream is used to create a StreamReader object, so the program can read the Web page's HTML markup line-by-line.

The section of the Web page in which we are interested consists of two batches of cities Albany through Reno, and Salt Lake City through Washington, D.C. The first batch occurs in a section that starts with the string "TAV12," while the second batch occurs in a section that starts with the string "TAV13." We declare variables separator1 and separator2 to store these strings. Line 29 reads the HTML markup one line at a time until "TAV12" is encountered. Once "TAV12" is reached, the program calls utility method ReadCities to read a batch of cities into ArrayList cityList. Next, line 33 reads the HTML markup one line at a time until "TAV13" is encountered, and line 34 makes another call to method ReadCities to read the second batch of cities. Line 36 calls method Sort of class ArrayList to sort the CityWeather objects in alphabetical order by city name. Line 37 closes the StreamReader connection to the Web site. Lines 4245 output the weather information for each city to the server application's console display.

[Page 1277]

Lines 4979 declare utility method `ReadCities`, which takes a `StreamReader` object and reads the information for each city, creates a `CityWeather` object for it and places the `CityWeather` object in `cityList`. The `do...while` statement (lines 5963) continues to read the page one line at a time until it finds the header line that begins the weather forecast table. This line starts with either `dayFormat` (lines 5253), indicating the header for the daytime information, or `nightFormat` (lines 5455), indicating the header for the nighttime information. Because the line could be in either format based on the time of day, the loop-continuation condition checks for both. Line 65 reads the next line from the Web page, which is the first line containing temperature information.

The `while` statement (lines 6878) creates a new `CityWeather` object to represent the current city. It parses the string containing the current weather data, separating the city name, the weather condition and the temperature. The `CityWeather` object is added to `cityList`. Then the next line from the page is read and stored in `inputLine` for the next iteration. This process continues while the length of the string read from the Web page is greater than 28 (the lines containing weather data are all longer than 28 characters). The first line shorter than this signals the end of that forecast section in the Web page.

Read-only property `Reports` (lines 8288) implements the `Report` interface's `Reports` property to return `cityList`. The client application will remotely call this property to retrieve the day's weather report.

Class `WeatherServer`

[Figure 23.12](#) contains the code for the server application. The `using` directives in lines 57 specify .NET remoting namespaces `System.Runtime.Remoting`, `System.Runtime.Remoting.Channels` and `System.Runtime.Remoting.Channels.Http`. The first two namespaces are required for .NET remoting, and the third is required for HTTP channels. Namespace `System.Runtime.Remoting.Channels.Http` requires the project to reference the `System.Runtime.Remoting` assembly, which can be found under the .NET tab in the Add References menu. The `using` directive at line 8 specifies namespace `Weather`, which contains interface `Report`. Remember to add a reference to `Weather.dll` in this project.

Figure 23.12. Class `WeatherServer` exposes remote class `ReportInfo`.

```

1 // Fig. 23.12: WeatherServer.cs
2 // Server application that uses .NET remoting to send
3 // weather report information to a client
4 using System;
5 using System.Runtime.Remoting;
6 using System.Runtime.Remoting.Channels;
7 using System.Runtime.Remoting.Channels.Http;
8 using Weather;
9
10 class WeatherServer
11 {
12     static void Main( string[] args )
13     {
14         // establish HTTP channel
15         HttpChannel channel = new HttpChannel( 50000 );
16
17         ChannelServices.RegisterChannel( channel, false );

```

[Page 1278]

```

18      // register ReportInfo class
19      RemotingConfiguration.RegisterWellKnownServiceType(
20          typeof( ReportInfo ), "Report",
21          WellKnownObjectMode.Singleton );
22
23      Console.WriteLine( "Press Enter to terminate server." );
24      Console.ReadLine();
25  } // end Main
26 } // end class WeatherServer

```

Lines 1516 in Main register an HTTP channel on the current machine at port 50000. This is the port number that clients will use to connect to the WeatherServer remotely. The argument false at line 16 indicates that we do not wish to enable security, which is beyond the scope of this introduction. Lines 1921 register the ReportInfo class type at the "Report" URI as a *Singleton* remote class. If a remote class is registered as Singleton, one remote object will be created when the first client requests that remote class, and that remote object will service all clients. The alternative mode is SingleCall, where one remote object is created for each individual remote method call to the remote class. [Note: A Singleton remote object does not have an infinite lifetime; it will be garbage collected after being idle for 5 minutes. A new Singleton remote object will be created if another client requests one later.] The ReportInfo remote class is now available to clients at the URI "http://IPAddress:50000/Report" where IPAddress is the IP address of the computer on which the server is running. The channel remains open as long as the server application continues running, so line 24 waits for the user running the server application to press Enter before terminating the application.

Class WeatherClientForm

WeatherClientForm ([Fig. 23.13](#)) is a Windows application that uses .NET remoting to retrieve weather information from WeatherServer and displays the information in a graphical, easy-to-read manner. The GUI contains 43 Labels one for each city in the Traveler's Forecast. All the Labels are placed in a Panel with a vertical scrollbar. Each Label displays the weather information for one city. Lines 79 are using directives for the namespaces that are required to perform .NET remoting. For this project, you must add references to the System.Runtime.Remoting assembly and the weather.dll file we created earlier.

[Page 1281]

Figure 23.13. The Weather Client accesses a ReportInfo object remotely and displays the weather report.
(This item is displayed on pages 1278 - 1280 in the print version)

```

1 // Fig. 23.13: WeatherClient.cs
2 // Client that uses .NET remoting to retrieve a weather report.
3 using System;
4 using System.Collections;
5 using System.Drawing;
6 using System.Windows.Forms;
7 using System.Runtime.Remoting;
8 using System.Runtime.Remoting.Channels;
9 using System.Runtime.Remoting.Channels.Http;
10 using Weather;

```

[Page 1279]

11

```

12 public partial class WeatherClientForm : Form
13 {
14     public WeatherClientForm()
15     {
16         InitializeComponent();
17     } // end constructor
18
19     // retrieve weather data
20     private void WeatherClientForm_Load( object sender, EventArgs e )
21     {
22         // setup HTTP channel, does not need to provide a port number
23         HttpChannel channel = new HttpChannel();
24         ChannelServices.RegisterChannel( channel, false );
25
26         // obtain a proxy for an object that implements interface Report
27         Report info = ( Report ) RemotingServices.Connect(
28             typeof( Report ), "http://localhost:50000/Report" );
29
30         // retrieve an ArrayList of CityWeather objects
31         ArrayList cities = info.Reports;
32
33         // create array and populate it with every Label
34         Label[] cityLabels = new Label[ 43 ];
35         int labelCounter = 0;
36
37         foreach ( Control control in displayPanel.Controls )
38         {
39             if ( control is Label )
40             {
41                 cityLabels[ labelCounter ] = ( Label ) control;
42                 ++labelCounter; // increment Label counter
43             } // end if
44         } // end foreach
45
46         // create Hashtable and populate with all weather conditions
47         Hashtable weather = new Hashtable();
48         weather.Add( "SUNNY", "sunny" );
49         weather.Add( "PTCLDY", "pcloudy" );
50         weather.Add( "CLOUDY", "mcloudy" );
51         weather.Add( "MOCLDY", "mcloudy" );
52         weather.Add( "TSTRMS", "rain" );
53         weather.Add( "RAIN", "rain" );
54         weather.Add( "SNOW", "snow" );
55         weather.Add( "VRYHOT", "vryhot" );
56         weather.Add( "FAIR", "fair" );
57         weather.Add( "RNSNOW", "rnsnow" );
58         weather.Add( "SHWRS", "showers" );
59         weather.Add( "WINDY", "windy" );
60         weather.Add( "NOINFO", "noinfo" );
61         weather.Add( "MISG", "noinfo" );

```

[Page 1280]

```

62         weather.Add( "DRZL", "rain" );
63         weather.Add( "HAZE", "noinfo" );
64         weather.Add( "SMOKE", "mcloudy" );
65         weather.Add( "SNOWSHWRS", "snow" );
66         weather.Add( "FLRRYS", "snow" );
67         weather.Add( "FOG", "noinfo" );
68
69         // create the font for the text output
70         Font font = new Font( "Courier New", 8, FontStyle.Bold );
71
72         // for every city

```

```

73     for ( int i = 0; i < cities.Count; i++ )
74     {
75         // use array cityLabels to find the next Label
76         Label currentCity = cityLabels[ i ];
77
78         // use ArrayList cities to find the next CityWeather object
79         CityWeather city = ( CityWeather ) cities[ i ];
80
81         // set current Label's image to image
82         // corresponding to the city's weather condition -
83         // find correct image name in Hashtable weather
84         currentCity.Image = new Bitmap( @"images\" +
85             weather[ city.Description.Trim() ] + ".png" );
86         currentCity.Font = font; // set font of Label
87         currentCity.ForeColor = Color.White; // set text color of Label
88
89         // set Label's text to city name and temperature
90         currentCity.Text = "\r\n " + city.CityName + city.Temperature;
91     } // end for
92 } // end method WeatherClientForm_Load
93 } // end class WeatherClientForm

```

[\[View full size image\]](#)



Method `WeatherClientForm_Load` (lines 2092) retrieves the weather information when this Windows application loads. Line 23 creates an `HTTP` channel without specifying a port number. This causes the `HttpChannel` constructor to choose any available port number on the client computer. A specific port number is not necessary because this application does not have its own clients that need to know the port number in advance. Line 24 registers the channel on the client computer. This will allow the server to send information back to the client. Lines 2728 declare a `Report` variable and assign to it a proxy for a `Report` object instantiated by the server. This proxy allows the client to remotely call `ReportInfo`'s properties by redirecting method calls to the server. *RemotingServices* method `Connect` connects to the server and returns a reference to the proxy for the `Report` object. Note that we are executing the client and the server on the same computer, so we use `localhost` in the URL that represents the server application. To connect to a `WeatherServer` on a different computer, you must modify this URL accordingly. Line 31 retrieves the `ArrayList` of `CityWeather` objects generated by the `ReportInfo` constructor (lines 1446 of [Fig. 23.11](#)). Variable `cities` now refers to an `ArrayList` of `CityWeather` objects that contains the information taken from the Traveler's Forecast Web page.

Because the application presents weather data for so many cities, we must establish a way to

organize the information in the `Labels` and to ensure that each weather description is accompanied by an appropriate image. The program uses an array to store all the `Labels`, and a *Hashtable* (discussed further in [Chapter 27](#), Collections) to store weather descriptions and the names of their corresponding images. A *Hashtable* stores keyvalue pairs, in which both the key and the value can be any type of object. Method `Add` adds keyvalue pairs to a *Hashtable*. The class also provides an indexer to return the value for a particular key in the *Hashtable*. Line 34 creates an array of `Label` references, and lines 35-44 place the `Labels` we created in the designer in the array so that they can be accessed programmatically to display weather information for individual cities. Line 47 creates *Hashtable* object `weather` to store pairs of weather conditions and the names for images associated with those conditions. Note that a given weather description name does not necessarily correspond to the name of the PNG file containing the correct image. For example, both "TSTRMS" and "RAIN" weather conditions use the `rain.png` image file.

Lines 73-91 set each `Label` so that it contains a city name, the current temperature in the city and an image corresponding to the weather conditions for that city. Line 76 retrieves the `Label` that will display the weather information for the next city. Line 79 uses `ArrayList cities` to retrieve the *CityWeather* object that contains the weather information for the city. Lines 84-85 set the `Label`'s image to the PNG image that corresponds to the city's weather conditions. This is done by eliminating any spaces in the description string by calling `string` method `trim` and retrieving the name of the PNG image from the `weather Hashtable`. Lines 86-87 set `Label` properties to achieve the visual effect seen in the sample output. Line 90 sets the `Text` property to display the city's name and high/low temperatures. [Note: To preserve the layout of the client application's window, we set the `MaximumSize` and `MinimumSize` properties of the `Windows Form` to the same value so that the user cannot resize the window.]

[Page 1282]

Web Resources for .NET Remoting

This section provided a basic introduction to .NET remoting. There is much more to this powerful .NET framework capability. The following Web sites provide additional information for readers who wish to investigate these capabilities further. In addition, searching for ".NET remoting" with most search engines yields many additional resources.

msdn.microsoft.com/library/en-us/cpguide/html/cpconaccessingobjectsinotherrapplicationdomainsusingnetremoting.asp

The .NET Framework Developer's Guide on the MSDN Web site provides detailed information on .NET remoting, including articles that include choosing between ASP.NET and .NET remoting, an overview of .NET remoting, advanced .NET remoting techniques and .NET remoting examples.

msdn.microsoft.com/library/en-us/dndotnet/html/introremoting.asp

This site offers a general overview of .NET remoting capabilities.

search.microsoft.com/search/results.aspx?qu=.net+remoting

This `microsoft.com` search provides links to many .NET remoting articles and resources.



[Page 1282 (continued)]

23.11. Wrap-Up

In this chapter, we presented both connection-oriented and connectionless networking techniques. You learned that the Internet is an "unreliable" network that simply transmits packets of data. We discussed two protocols for transmitting packets over the Internet: Transmission Control Protocol (TCP) and User Datagram Protocol (UDP). You learned that TCP is a connection-oriented communication protocol that guarantees that sent packets will arrive at the intended receiver undamaged and in the correct sequence. You also learned that UDP is typically used in performance-oriented applications because it incurs minimum overhead for communicating between applications. We presented some of C#'s capabilities for implementing communications with TCP and UDP. We showed how to create a simple client/server chat application using stream sockets. We then showed how to send datagrams between a client and a server. You also saw a multithreaded Tic-Tac-Toe server that allowed two clients to connect simultaneously to the server and play Tic-Tac-Toe against one another. We presented the new `WebBrowser` control, which allows you to add Web browsing capabilities to your Windows applications. Finally, we demonstrated .NET remoting, a technology that allows a client application to remotely access the properties and methods of an object instantiated by a server application. In [Chapter 24](#), Searching and Sorting, we discuss how to create methods that can order array elements in ascending or descending order, and that can search for values in arrays.



[Page 1282 (continued)]

Summary

Section 23.1 Introduction

- The Internet ties the information world together.
- The Web makes the Internet easy to use and gives it the flair and sizzle of multimedia.

Section 23.2 Connection-Oriented vs. Connectionless Communication

- There are two primary approaches to communicating between applications: connection oriented and connectionless.

[Page 1283]

- Connection-oriented communications are similar to the telephone system, in which a connection is established and held for the length of the session.
- Connectionless services are similar to the postal service, in which two letters mailed from one place to the same destination may actually take two dramatically different paths through the system and even arrive at different times, or not at all.
- Data is sent by TCP in packets, which contain pieces of the data, along with information that helps the Internet route the packets to the proper destination.
- The Internet does not guarantee anything about the packets sent; they could arrive corrupted or out of order, as duplicates or not at all. The Internet makes only a "best effort" to deliver

packets.

- A connection-oriented approach guarantees that sent packets will arrive at the intended receiver undamaged and be reassembled into the correct sequence.
- In a connectionless approach, data sent may never reach the intended recipient. A connectionless approach, however, avoids the overhead associated with handshaking and enforcing reliabilityless information often needs to be passed between the hosts.

Section 23.3 Protocols for Transporting Data

- Protocols are sets of rules that govern how two entities should interact.
- C#'s Transmission Control Protocol (TCP) and User Datagram Protocol (UDP) networking capabilities are defined by classes of the `System.Net.Sockets` namespace.
- Transmission Control Protocol (TCP) is a connection-oriented communication protocol that guarantees that sent packets will arrive at the intended receiver undamaged and in the correct sequence.
- If packets of information don't arrive at the recipient, TCP ensures that the packets are sent again. If packets arrive out of order, TCP reassembles them into the correct order transparently to the application receiving the data. If duplicate packets arrive, TCP discards the duplicates.
- Applications that do not require reliable end-to-end transmission typically use the connectionless User Datagram Protocol (UDP).
- UDP makes no guarantees that packets, called datagrams, will reach their destination or arrive in their original order.
- UDP has little overhead because UDP datagrams do not need to carry the information that TCP packets carry to ensure reliability.

Section 23.4 Establishing a Simple TCP Server (Using Stream Sockets)

- Class `IPAddress` represents an Internet Protocol (IP) address.
- Establishing a simple server with TCP and stream sockets in C# requires several steps. Step 1 is to create a `TcpListener` object. This class represents a TCP stream socket that a server can use to receive connections. To receive connections, the `TcpListener` must be listening for them. For the `TcpListener` to listen for client connections, its `Start` method must be called (Step 2). `TcpListener` method `AcceptSocket` blocks indefinitely until a connection is established, at which point it returns a `Socket` (Step 3). Step 4 is the processing phase, in which the server and the client communicate via streams maintained by a `NetworkStream` object. When the client and server have finished communicating, the server closes the connection (Step 5). The server can then wait for another client's connection.
- A port number is a numeric ID number that an application uses to identify itself at a given network address.
- An individual application running on a computer is identified by an IP address/port number pair. Hence, no two processes can have the same port number at a given IP address.

Section 23.5 Establishing a Simple TCP Client (Using Stream Sockets)

- Class `IPEndPoint` represents an endpoint on a network.
- Establishing a simple client requires four steps. In Step 1, we create a `TcpClient` to connect to the server. This connection is established through a call to the `TcpClient` method `Connect` containing two arguments the server's IP address and the port number. In Step 2, the `TcpClient` uses method `GetStream` to get a `NetworkStream` to write to and read from the server. Step 3 is the processing phase, in which the client and the server communicate. In Step 4, the client closes the connection.

Section 23.7 Connectionless Client/Server Interaction with Datagrams

- Class `UdpClient` is provided for connectionless transmission of data.
- Class `UdpClient` methods `Send` and `Receive` are used to transmit data.

Section 23.8 Client/Server Tic-Tac-Toe Using a Multithreaded Server

- Multithreaded servers can manage many simultaneous connections with multiple clients.

Section 23.9 WebBrowser Control

- The `WebBrowser` control provides Web browsing functionality and maintains its own navigation history.
- `WebBrowser` methods `GoBack`, `GoForward`, `GoHome` and `Navigate` are used for Web navigation. Method `Stop` cancels loading of the current page, and method `Refresh` reloads the current page.
- Many `WebBrowser` events, including `StatusTextChanged`, `ProgressChanged` and `DocumentTitleChanged`, indicate changes in `WebBrowser` properties. The `Navigating` event occurs when the `WebBrowser` starts loading a page and the `DocumentCompleted` event occurs when the page is finished loading.

Section 23.10 .NET Remoting

- .NET remoting is a distributed computing technology that allows a program to access objects on another machine over a network.
- .NET remoting is similar in concept to RMI (remote method invocation) in Java and RPC (remote procedure calls) in procedural programming languages.
- Marshal by value requires that the object be serializable, capable of being represented as a formatted message which the server can send through a channel to the client. To enable an object to be serialized and deserialized, its class must either be declared with the attribute `[ Serializable ]` or must implement interface `ISerializable`.
- Marshal by reference requires that the object's class extend class `MarshalByRefObject`. An object that is marshaled by reference is referred to as a remote object, and its class is referred to as a remote class.

- When an object is marshaled by reference, the client creates two proxy objects, a transparent proxy and a real proxy. The client application calls the methods and properties of the transparent proxy as if it were the remote object itself. The transparent proxy calls the real proxy to send a message to the server to perform the appropriate actions on the actual object.
- The parameters and return values of the methods and properties declared by a remote class will themselves be marshaled by value.
- Namespaces `System.Runtime.Remoting` and `System.Runtime.Remoting.Channels` are required for .NET remoting. Namespace `System.Runtime.Remoting.Channels.Http` is required for HTTP channels and is found in reference `System.Runtime.Remoting`.

[Page 1285]

- Class `HttpChannel` represents an HTTP channel. Its constructor takes an `int` port number. static method `RegisterChannel` of class `ChannelServices` takes an object of a class that implements interface `Channel`, such as `HttpChannel`, and registers it on the local machine.
- static method `RegisterWellKnownServiceType` of class `RemotingConfiguration` takes a `Type`, a URI extension and a mode, `Singleton` or `SingleCall`, as arguments and makes the class type available to respond to clients that clients access the class at that URI.
- A `Singleton` remote class instantiates one remote object to service all clients. A `SingleCall` remote class instantiates one remote object for each individual method call.
- static method `Connect` of class `RemotingServices` takes a `Type` and a URI as arguments and returns a reference to the transparent proxy that enables method calls to the remote object.
- A `Hashtable` stores keyvalue pairs, in which both the key and the value can be any type of object. Method `Add` adds keyvalue pairs to a `Hashtable`. The indexer returns the value with which a particular key was placed in the `Hashtable`.



[Page 1285 (continued)]

Terminology

[127.0.0.1](#)

[AcceptSocket method of class TcpListener](#)

[Add method of class Hashtable](#)

[BinaryReader class](#)

[BinaryWriter class](#)

[Bind method of class Socket](#)

[binding a server to a port](#)

blocking until connection received

[CanGoBack property of WebBrowser](#)

[CanGoForward property of WebBrowser](#)

[channel](#)

[client](#)

[client/server chat application](#)

[client/server relationship](#)

[Close method of class Socket](#)

[Close method of class TcpClient](#)

[collaborative applications](#)

[Connect method of class RemotingService](#)

[Connect method of class TcpClient](#)

[connection between applications](#)

connection attempt

[connection between client and server terminates](#)

[connection port](#)

[connection to a client](#)

[connectionless communication](#)

[connectionless transmission via datagrams](#)

[connection-oriented communication](#)

[datagram](#)

[deserialize](#)

[DocumentCompleted event of WebBrowser](#)

[DocumentTitle property of WebBrowser](#)

[DocumentTitleChanged event of WebBrowser](#)

[echo a packet](#)

[Exit method of class Environment](#)

[ExitCode property of class Environment](#)

[GetStream method of class Socket](#)

[GoBack method of WebBrowser](#)

[GoForward method of WebBrowser](#)

[GoHome method of WebBrowser](#)

[handshaking](#)

[HashTable class](#)

[HTTP channel](#)

[IP \(Internet Protocol\) address](#)

[IPAddress class](#)

[IPEndPoint class](#)

[Invoke method](#)

InvokeRequired property

[ISerializable interface](#)

[localhost](#)

[Loopback static member of class IPAddress](#)

[marshal by reference](#)

[marshal by value](#)

[MarshalByRefObject class Marshaling](#)

[Navigate method of WebBrowser](#)

[Navigating event of WebBrowser](#)

[.NET remoting](#)

[network address](#)

[NetworkStream class](#)

[packet](#)

[port number](#)

[ProgressChanged event of WebBrowser](#)

[protocol](#)

[proxy](#)

[ReadString method of BinaryReader](#)

[real proxy](#)

[receive a request for connection](#)

[Receive method of class UdpClient](#)

[ReceiveFrom method of class Socket](#)

[Page 1286]

[Refresh method of WebBrowser](#)

[remote class](#)

[RemotingService class](#)

[Send method of class UdpClient](#)

[SendTo method of class Socket](#)

[Serializable attribute](#)

[serialize](#)

[server](#)

[SingleCall](#)

[Singleton](#)

[socket](#)

[Socket class](#)

[Sort method of class ArrayList](#)

[Start method of class TcpListener](#)

[StatusText property of WebBrowser](#)

[StatusTextChanged event of WebBrowser](#)

[Stop method of WebBrowser](#)

[stream socket](#)

[streams-based transmission](#)

[System.Net namespace](#)

[System.Net.Sockets namespace](#)

[SystemRuntime.Remoting namespace](#)

[SystemRuntime.Remoting.Channels](#)

[System.RuntimeRemoting.Channels HTTP namespace](#)

[TCP channel](#)

[TcpClient class](#)

[TcpListener class](#)

[Transmission Control Protocol \(TCP\)](#)

[transparent proxy](#)

[UdpClient class](#)

[unreliable network](#)

[Uniform Resource Identifier \(URI\)](#)

[User Datagram Protocol \(UDP\)](#)

[WebBrowser control](#)

[WebClient class](#)

[Write method of class BinaryWriter](#)



[Page 1286 (continued)]

Self-Review Exercises

23.1 State whether each of the following is true or false. If false, explain why.

- a. UDP is a connection-oriented protocol.
- b. Packet transmission over a network is reliablepackets are guaranteed to arrive in sequence.

- c. TCP is preferred over UDP when applications require reliable packet transmission.
- d. Each `TcpListener` can accept only one connection.
- e. A `TcpListener` can listen for connections at more than one port at a time.
- f. A `UdpClient` can send information only to one particular port.
- g. Clients must know the port number at which the server is waiting for connections in order to connect to the server.
- h. The `WebBrowser` control can be used in any Windows application.
- i. When an object is marshaled by reference, a copy of that object is created and sent to the client.
- j. A `Singleton` remote class instantiates one remote object for each client.

23.2 Fill in the blanks in each of the following statements:

- a. Many of C#'s networking classes are contained in namespaces _____ and _____.
- b. Class _____ is used for unreliable but fast datagram transmission.
- c. An object of class _____ represents an Internet Protocol (IP) address.
- d. The two socket types discussed in this chapter are _____ and _____ sockets.
- e. The acronym TCP stands for _____.
- f. Class _____ listens for connections from clients.
- g. Class _____ connects to servers.
- h. Class _____ provides access to stream data on a network.
- i. Method _____ causes a `WebBrowser` to go to a particular URL.
- j. A serializable class must either be declared with attribute _____ or extend class _____.
- k. The _____ provides all the public services of the remote object being marshaled by reference.

Answers to Self-Review Exercises

23.1 a) False. UDP is a connectionless protocol, and TCP is a connection-oriented protocol. b) False. Packets can be lost, arrive out of order or even be duplicated. c) True. d) False. TcpListener's AcceptSocket method may be called as often as necessary each call will accept a new connection. e) False. A TcpListener can listen for connections at only one port at a time. f) False. A UdpClient can send information to any port represented by an IPEndPoint. g) True. h) True. i) False. When an object is marshaled by reference, the client application accesses the original remote object on the server, not a local copy. j) False. A Singleton remote class instantiates one remote object to service all clients.

23.2 a) System.Net, System.Net.Sockets. b) UdpClient. c) IPAddress. d) stream, datagram. e) Transmission Control Protocol. f) TcpListener. g) TcpClient. h) NetworkStream. i) Navigate. j) Serializable, ISerializable. k) transparent proxy.



[Page 1287 (continued)]

Exercises

- 23.3** Use a socket connection to allow a client to specify a text file's name and have the server send the contents of the file or indicate that the file does not exist.
- 23.4** Modify [Exercise 23.3](#) to allow the client to modify the contents of the file and send the file back to the server for storage. The user can edit the file in a TextBox, then click an Update file on server button to send the file back to the server.
- 23.5** Multithreaded servers are quite popular today, especially because of the increasing use of multiprocessing servers. Modify the simple server application presented in [Section 23.6](#) to be a multithreaded server. Then use several client applications and have each of them connect to the server simultaneously. Use a Hashtable (namespace System.Collections) to store the client threads. Hashtable provides several properties and methods of use in this exercise. Property Keys returns an ICollection of keys currently found in the Hashtable. Each key can then be used in the Hashtable's indexer to retrieve the corresponding value. Method Add places its arguments a key and a value into the Hashtable. Method Remove deletes its argument the key from the Hashtable.
- 23.6** Create a client/server application for the game of Hangman, using socket connections. The server should randomly pick a word or phrase from a file. After connecting, the client should be allowed to begin guessing. If a client guesses incorrectly five times, the game is over. Display the original phrase or word on the server. Display dashes (for letters that have not been guessed yet) and the letters that have been guessed in the word or phrase on the client.
- 23.7** Modify the previous exercise to be a connectionless game using datagrams.

- 23.8** (Checkers Game) In the text, we presented a Tic-Tac-Toe program controlled by a multithreaded server. Develop a checkers program modeled after the Tic-Tac-Toe program. The two users should alternate making moves. Your program should mediate the players' moves, determining whose turn it is and allowing only valid moves. The players themselves will determine when the game is over.
- 23.9** (Networked Morse Code) Perhaps the most famous of all coding schemes is the Morse code, developed by Samuel Morse in the 1830s for use with the telegraph system. The Morse code assigns a series of dots and dashes to each letter of the alphabet, each digit and a few special characters (such as period, comma, colon and semicolon). In sound-oriented systems, the dot represents a short sound and the dash represents a long sound. Other representations of dots and dashes are used with light-oriented systems and signal-flag systems.

Separation between words is indicated by a space, or quite simply, the absence of a dot or dash. In a sound-oriented system, a space is indicated by a short period of time during which no sound is transmitted. The international version of the Morse code appears in [Fig. 23.14](#).

[Page 1288]

Figure 23.14. English letters of the alphabet and decimal digits as expressed in international Morse code.

Character	Code
A	. -
B	- . . .
C	- . - .
D	- . .
E	.
F	. . - .
G	- - .
H	
I	. .
J	. - - -
K	- . -
L	. - . .
M	- -
N	- .
O	- - -
P	. - - .
Q	- - . -

R	. - .
S	. . .
T	-
U	. . -
V	. . . -
W	. - -
X	- . . -
Y	- . - -
Z	- - . .
Digits	
1	. - - - -
2	. . - - -
3	. . . - -
4	 -
5	
6	-
7	- - . . .
8	- - - . .
9	- - - - .
0	- - - - -

Write a client/server application in which two clients can send Morse code messages to each other through a multithreaded server application. The client application should allow the user to type English-language phrases in a `TextBox`. When the user sends the message, the client application encodes the text in Morse code and sends the coded message through the server to the other client. Use one blank between each Morse-coded letter and three blanks between each Morsecoded word. When messages are received, they should be decoded and displayed as normal characters and as Morse code. The client should have one `TextBox` for typing and one `TextBox` for displaying the other client's messages.

- 23.10** Write a bulletin board application that uses .NET remoting. Create a remote object that will allow the client application to list all available bulletins, retrieve the text of any bulletin and post a new bulletin. Each bulletin should be stored in a .txt file in the server's directory. Refer to [Chapter 18](#), Files and Streams.
- 23.11** Write a quiz application that uses .NET remoting. Store one quiz question in a text file on the server machine. Allow the client to retrieve the question from the server and send an answer back to the server. You may wish to store records of who answered the quiz question correctly so the server operator can keep score.

[◀ PREV](#)

[NEXT ▶](#)