JOURNAL OF OBJECT TECHNOLOGY

Online at http://www.jot.fm. Published by ETH Zurich, Chair of Software Engineering ©**JOT, 2004**

Vol. 3, No. 1, January-February 2004

Remoting in C# and .NET

Dr. Richard Wiener, Editor-in-Chief, JOT**,** Associate Professor of Computer Science**,** University of Colorado at Colorado Springs

This is the first in a series of tutorial columns that shall deal with specialized aspects of C# programming and the .NET framework.

The .NET Remoting API is the equivalent of the Java Remote Method Invocation (RMI) API. Both frameworks allow objects on a client machine to communicate with remote objects on a server. To me, the infrastructure required in .NET appears simpler than in Java's RMI.

What makes Remoting (or equivalently RMI) so attractive is that the low-level socket protocol that the programmer must normally manage is abstracted out. The programmer is able to operate at a much higher and simpler level of abstraction. In both languages there is some overhead in the form of boilerplate protocols that must be observed in order to setup the handshaking between the client and server machines. Once this is done, sending a message from a client machine to a server object uses the same syntax as sending a message to a local object. The metaphor of object-oriented programming remains central to this distributed programming.

My last column presented a distributed solution to the Traveling Salesperson Problem (see this column in the Nov/Dec, 2003 issue of JOT) using Java's RMI.

Although it is tempting to present a detailed and exhaustive presentation of .NET's distributed computing protocol, space does not permit this here. In this column, only an introduction and several simple examples of .NET Remoting using C# are presented.

The namespaces that one typically uses in C# distributed object applications are the following:

```
using System;
using System.Runtime.Remoting;
using System.Runtime.Remoting.Channels;
using System.Runtime.Remoting.Channels.Http;
```

When using one of the major IDE's (Visual Studio or C# Builder), it is important to add the reference *system.remoting.dll* to your build if it is not already present.

Cite this column as follows: Richard Wiener: "Remoting in C# and .NET", in *Journal of Object Technology*, vol. 3, no. 1, January-February 2004, pp. 83-100.
http://www.jot.fm/issues/issue_2004_01/column8

In C#, using distributed objects does not require stubs or interfaces as in Java. The CLR (common language runtime) provides full support for remote object calls. Using distributed objects does not depend on the system registry for information about the remote classes. This information is encapsulated in a .DLL file that must be added as a reference when compiling the client code.

Classes derived from *System.MarshalByRefObject* cause the distributed object system to generate proxy objects on the client that encapsulate the low-level socket protocol. When the client sends a message to a remote object, it is the proxy that processes this message and sends serialized information across the network. The same works in reverse when proxy objects de-serialize information that is returned from the server.

Channel objects are the mechanism used to transfer messages between client and server. The .NET framework provides two bidirectional channels:

System.Runtime.Remoting.Channels.http.HttpChannel and
System.Runtime.Remoting.Channels.Tcp.TcpChannel.

The http channel uses SOAP (Simple Object Access Protocol) and the tcp channel uses a binary stream. This latter method is more efficient because it avoids the need to encode and decode SOAP messages.

A channel must be registered before it can be used. The *ChannelServices* class is used to accomplish this as follows:

```
ChannelServices.RegisterChannel(someChannel);
```

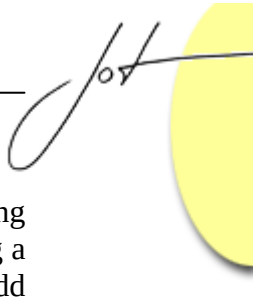
The general steps involved in writing a distributed application are summarized below.

Writing the Server

1. Construct the server class.
2. Select a method for hosting the server object(s) on the server. Typically a short application is created that launches the server and makes the server object available to the client(s).
3. The server object typically waits for one or more client objects to communicate with it.

Writing the Client

1. Identify the remote server object to the client.
2. Connect the server to the client through a channel.
3. The client must activate the remote object and create a reference to it.
4. Communication to the remote object(s), once activated, is similar to sending messages to local objects.



Let us consider a simple client server application. The server uses a supporting *NameHolder* class that encapsulates an *ArrayList* of *String* objects, *names*, each holding a person's name. The server has an *AddName* method that uses a *String* parameter to add another object to the *names* field within the *NameHolder* object.

The client application takes Console input as it is invoked and passes the string representing a person's name to the server object serving as a proxy. It will be clear from the code how this is all accomplished. Let us examine the details of Listing 1.

Listing 1 – Simple Client/Server application using Remoting

```
using System;
using System.Runtime.Remoting;
using System.Runtime.Remoting.Channels;
using System.Collections;

namespace Remoting {

   // Server class
   public class SaveNamesServer : MarshalByRefObject {

      // Fields
      private NameHolder holder; // Must be serializable

      // Constructor
      public SaveNamesServer() {
         holder = new NameHolder();
      }

      // Commands
      public void AddName(String name) {
         holder.AddName(name);
      }

      // Queries
      public ArrayList GetNames() {
         return holder.GetNames();
      }
   }
}

using System;
using System.Collections;

namespace Remoting {

   [Serializable]
   public class NameHolder {

      // Fields
      private ArrayList names = new ArrayList();
```

```
      // Commands
      public void AddName(String newName) {
         names.Add(newName);
      }

      // Queries
      public ArrayList GetNames() {
         return names;
      }
   }
}

using System;
using System.Runtime.Remoting;
using System.Runtime.Remoting.Channels;
using System.Runtime.Remoting.Channels.Http;
using System.Collections;

namespace Remoting {

   public class Client {

      // Constructor
      public Client(String newName) {
         // Create and register the remoting channel
         HttpChannel channel = new HttpChannel();
         ChannelServices.RegisterChannel(channel);

         // Initialize the remoting system
         InitializeRemoteServer();

         this.AddName(new SaveNamesServer(), newName);
      }

      private void InitializeRemoteServer() {
         RemotingConfiguration.RegisterWellKnownClientType(
            typeof(SaveNamesServer),
            "http://localhost:12345/SaveNamesServer");
      }

      private void AddName(SaveNamesServer server, String newName) {
         server.AddName(newName);
         ArrayList names = server.GetNames();
            foreach (String name in names) {
                Console.WriteLine(name);
            }
            Console.WriteLine();
      }

      static void Main(String [] args) {
         Console.WriteLine("Adding " + args[0] + " to name list.");
         new Client(args[0]);
```

```
      }
   }
}

using System;
using System.Runtime.Remoting;
using System.Runtime.Remoting.Channels;
using System.Runtime.Remoting.Channels.Http;

namespace Remoting {

   public class StartServer {

      static void Main() {
         // Create and register the channel
         HttpChannel channel = new HttpChannel(12345);

         ChannelServices.RegisterChannel(channel);

         Console.WriteLine("Starting SaveNamesServer");

         // Register the SaveNamesServer for remoting
         RemotingConfiguration.RegisterWellKnownServiceType(
                              typeof(SaveNamesServer),
                              "SaveNamesServer",
                              WellKnownObjectMode.Singleton);
         Console.WriteLine("Press return to exit SaveNamesServer.");
         Console.ReadLine();
      }
   }
}
```

Discussion and Analysis of Listing 1

The *Client* class uses the *RemotingConfiguration.RegisterWellKnownClientType* method as follows to connect itself to the server.

```
RemotingConfiguration.RegisterWellKnownClientType(
            typeof(SaveNamesServer),
            "http://localhost:12345/SaveNamesServer");
```

Here one computer is being used as a client and server.

The *Client* constructor creates an instance of the *SaveNamesServer* and passes this instance to its *AddName* method that uses the *AddName* method of the server (proxy) object to increase the size of the *ArrayList* in the *NameHolder* field by one.

The *StartServer* is used to launch the server process. It uses the same localhost socket as the *Client*. Using the *Singleton* mode ensures that the server object maintains state between client calls to this single server object.

It is essential that the NameHolder class be tagged as [Serializable] in order for the SaveNamesServer to be able to marshal its information properly.

Deployment

To deploy this distributed application, the following sequence of steps must be followed:

1. Compile the server classes into a DLL as follows:

   csc /target:library SaveNamesServer.cs NameHolder.cs → **SaveNamesServer.dll**

2. Compile the StartServer application as follows:

   csc StartServer.cs /reference:SaveNamesServer.dll -> **StartServer.exe**

3. Compile the Client application as follows:

   csc Client.cs /reference:SaveNamesServer.dll -> **Client.exe**

4. Launch the StartServer application.

5. Launch the Client application.

Each time the client is launched and a new name is specified on the command line, the previous names that were entered will be output.

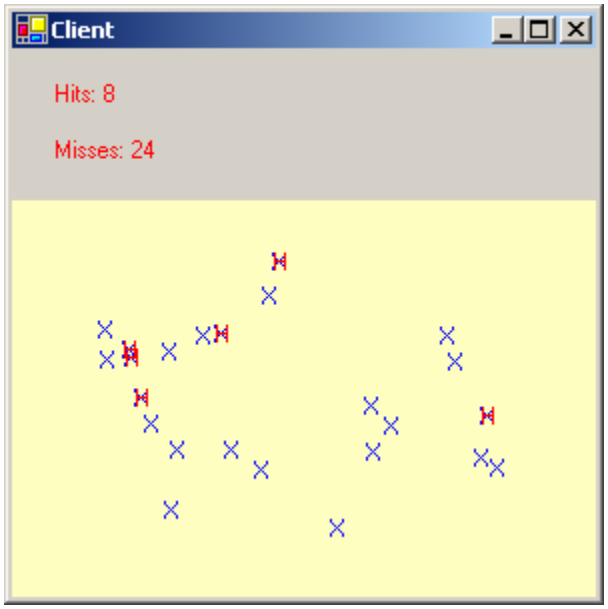
Two GUI Clients and a Server Application

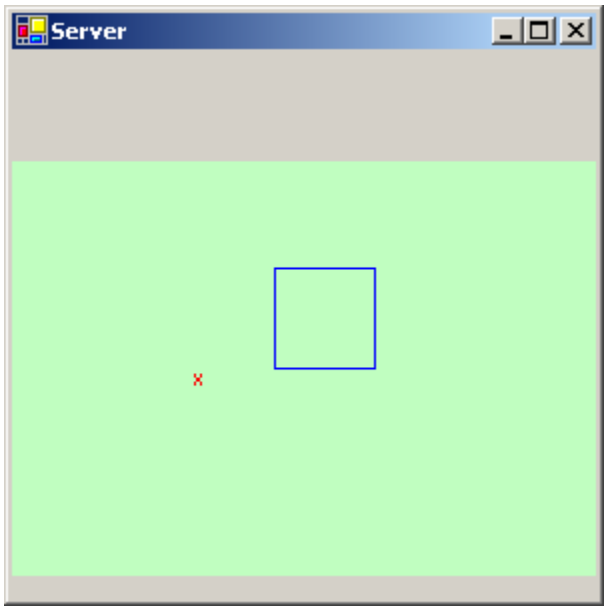
The next application of Remoting involves two client GUI's running in separate processes that communicate with each other through a server object.

One of the GUI applications, *Client2*, periodically takes the coordinates of a square of size 50 from the server and moves a rectangle to this position (upper-left corner of the rectangle). The server changes this coordinate every two seconds so the square jumps from one location to another every two seconds.

The other GUI application, *Client1*, waits for the user to click the mouse button in a panel. A blue "X" marks the spot of the mouse click. Simultaneously, the spot at which the user clicked the mouse is marked with a small red "x" in *Client2*. The communication is done through the server. If the *Client1* user clicks within the boundaries of the square that is slowing dancing around in *Client2*, a red "H" is shown in the panel of *Client1*. The cumulative hits and misses are also updated after each mouse click in *Client1*.

A screenshot of both client applications running and the server providing the communication channel as well as coordinates for the moving square in *Client2* is shown below.





Listing 2 contains the C# classes that implement this Remoting application.

Listing 2 - Remoting application with a server and two GUI clients

```
using System;
using System.Runtime.Remoting;
using System.Runtime.Remoting.Channels;
using System.Runtime.Remoting.Channels.Http;
using System.Threading;

namespace Remoting {

   public class TargetServer: MarshalByRefObject {

      // Fields
      private int xPos, yPos; // position of moving rectangle
      private int x, y;       // position of shot
      private Thread thrd;
      private Random rnd;

      // Constructor
      public TargetServer() {
         rnd = new Random();
         for (int i = 0; i < 50000; i++) {
            rnd.Next(100000);
         }
         thrd = new Thread(new ThreadStart(MovePosition));
         thrd.Start();
      }

      // Read-only properties
      public int XPOS {
         get {
            return xPos;
         }
      }

      public int YPOS {
         get {
            return yPos;
         }
      }

      public int X {
         get {
            return x;
         }
      }

      public int Y {
         get {
            return y;
         }
      }
```

```
      // Commands
      public void Record(int x, int y) {
         this.x = x;
         this.y = y;
      }

      // Queries
      public bool IsTargetHit(int x, int y) {
         return (x >= XPOS && x <= XPOS + 50) &&
                (y >= YPOS && y <= YPOS + 50);
      }

      private void MovePosition() {
          for (;;) {
            Thread.Sleep(2000);
            xPos = rnd.Next(225);
            yPos = rnd.Next(150);
         }
      }
   }
}

using System;
using System.Drawing;
using System.Collections;
using System.ComponentModel;
using System.Windows.Forms;
using System.Data;
using System.Runtime.Remoting;
using System.Runtime.Remoting.Channels;
using System.Runtime.Remoting.Channels.Http;

namespace Remoting {

   public class Client1 : Form {
      private System.ComponentModel.Container components = null;
      private Panel panel;
      private Label hitsLbl;
      private Label missesLbl;

      // Fields
      private TargetServer server;
      private int hits, misses;

      public Client1() {
         InitializeComponent();
         // Center form on the user's screen
         this.SetBounds((Screen.GetBounds(this).Width / 2) -
                        (this.Width / 2),
                     (Screen.GetBounds(this).Height / 2) –
                        (this.Height / 2),
                     this.Width, this.Height,
                        BoundsSpecified.Location);
```

```
            panel.MouseDown += new MouseEventHandler( HandleMouseClick);
            HttpChannel channel = new HttpChannel();
            ChannelServices.RegisterChannel(channel);
            InitializeRemoteServer();
            server = new TargetServer();
        }

        protected override void Dispose (bool disposing) {
            if (disposing) {
                if (components != null)
                {
                    components.Dispose();
                }
            }
            base.Dispose(disposing);
        }

        private void InitializeRemoteServer() {
            RemotingConfiguration.RegisterWellKnownClientType(
                typeof(TargetServer),
                "http://localhost:12345/TargetServer");
        }

        private void HandleMouseClick(Object sender, MouseEventArgs e) {
            Graphics g = panel.CreateGraphics();
            g.DrawString("X", Font, new SolidBrush(Color.Blue), e.X, e.Y);
            server.Record(e.X, e.Y);
            if (server.IsTargetHit(e.X, e.Y)) {
                g.DrawString("H", Font, new SolidBrush(Color.Red),
                                        e.X, e.Y);
                hits++;
            } else {
                misses++;
            }
            hitsLbl.Text = "Hits: " + hits;
            missesLbl.Text = "Misses: " + misses;
          }

        private void InitializeComponent() {
            // Details not shown
        }

        [STAThread]
        static void Main() {
            Application.Run(new Client1());
        }
    }
}
```

```
using System;
using System.Drawing;
using System.Collections;
using System.ComponentModel;
using System.Windows.Forms;
using System.Data;
using System.Threading;
using System.Runtime.Remoting;
using System.Runtime.Remoting.Channels;
using System.Runtime.Remoting.Channels.Http;

namespace Remoting {

    public class Client2 : Form {

      // Controls
      private Container components = null;
      private Panel panel;

      // Fields
      private TargetServer server;
      private Thread thrd;

      // Constructor
      public Client2() {
         InitializeComponent();

         // Set position on screen
         this.SetBounds(0, 0, this.Width, this.Height);

         panel.Paint += new PaintEventHandler(OnPaint);
         HttpChannel channel = new HttpChannel();
         ChannelServices.RegisterChannel(channel);
         InitializeRemoteServer();

         server = new TargetServer();
         thrd = new Thread(new ThreadStart(Pulse));
         thrd.Start();
      }

      private void Pulse() {
         for (;;) {
            Thread.Sleep(500);
            panel.Invalidate();
         }
      }

      private void InitializeRemoteServer() {
         RemotingConfiguration.RegisterWellKnownClientType(
               typeof(TargetServer),
               "http://localhost:12345/TargetServer");
      }

      protected override void Dispose (bool disposing) {
```

```
         if (disposing) {
            if (components != null) {
               components.Dispose();
            }
         }
         base.Dispose(disposing);
      }

      private void OnPaint(Object sender, PaintEventArgs e) {
            Graphics g = e.Graphics;
         g.DrawRectangle(new Pen(new SolidBrush(Color.Blue)),
                              server.XPOS, server.YPOS, 50, 50);
        if (server.X != 0 && server.Y != 0) {
            g.DrawString("x", Font,
               new SolidBrush(Color.Red), server.X, server.Y);
         }
      }

      private void InitializeComponent() {
         // Details not shown
      }

      [STAThread]
      static void Main() {
         Application.Run(new Client2());
      }
   }
}
```

Discussion and Analysis of Listing 2

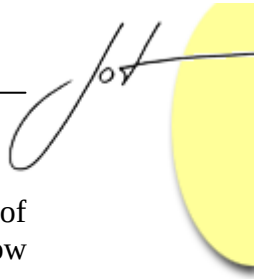
The *TargetServer* spawns a thread and starts creating *xPos* and *yPos* values every two seconds. The *Client1* application communicates with this server through the *Record* method. Because the *TargetServer* is declared a subclass of *MarshalByRefObject*, communication through proxy objects is accomplished for both *Client1* and *Client2*. Each of these GUI applications holds a reference to this proxy object as a *server* field.

Chat Session Application

The final application is a simple chat client/server application.

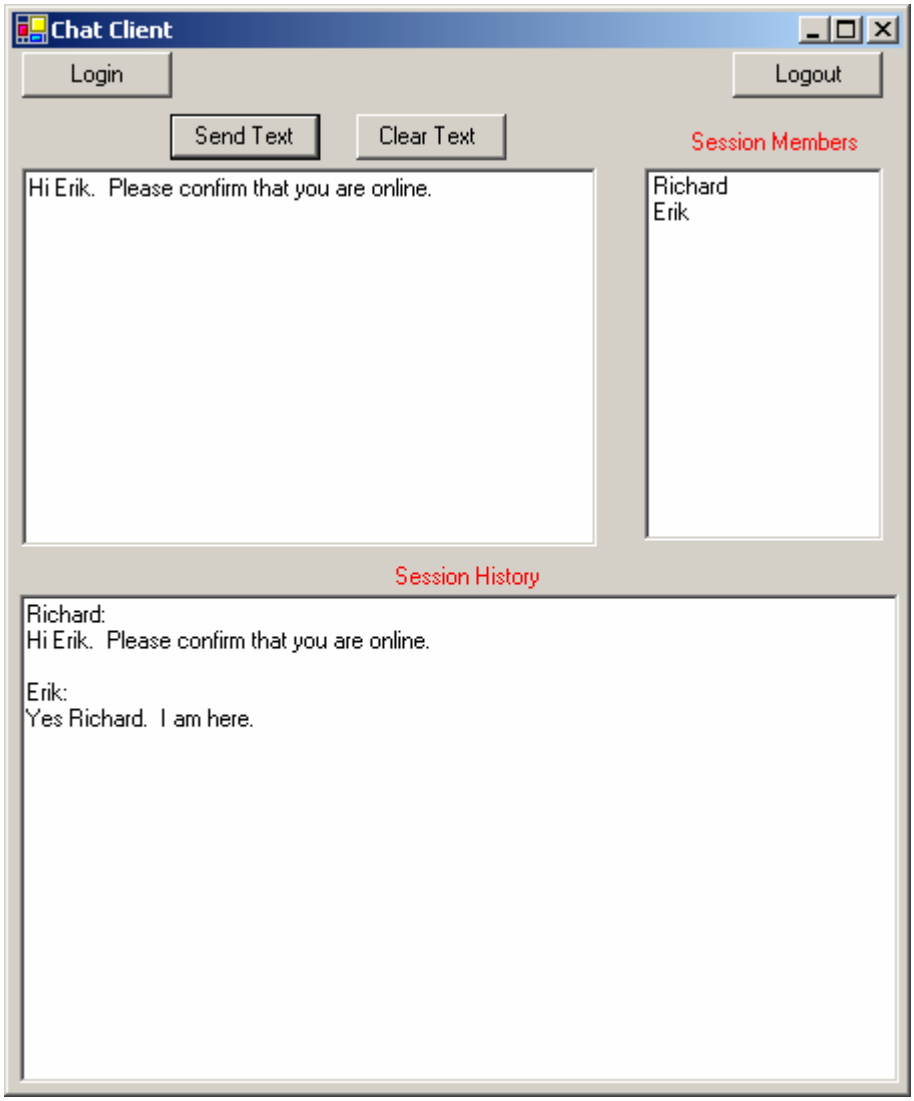
Any number of client applications, each containing a GUI that allows text to be added to the existing session, should permit communication among all clients that have logged in. All clients are updated on all communications every second.

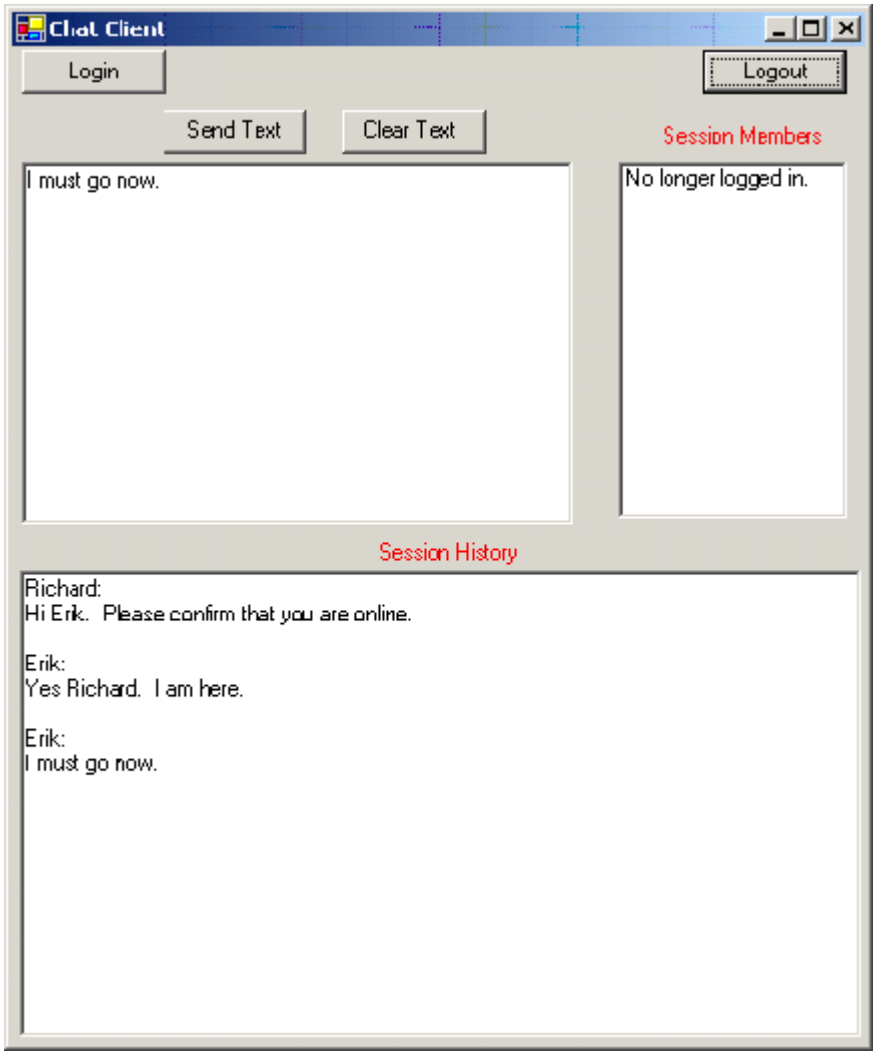
The application is designed so that each client must know about the server, but the server does not know about any of the clients. The server's responsibility is to maintain a centralized store of the clients that are logged in as well as the text for the entire chat session by adding text to it whenever a client posts a new message by clicking the "Send Text" button.



Each client, through a thread, polls the server every second to update the list of clients and to update the overall session text. Therefore the server does not need to know about any of the clients.

Two screen shots that depict a session between two clients before and after one has logged off are shown below.





The C# code for this application is given in Listing 3.

Listing 3 – Chat Session Client/Server Application Using Remoting

```
using System;
using System.Runtime.Remoting;
using System.Runtime.Remoting.Channels;
using System.Runtime.Remoting.Channels.Http;
using System.Collections;

namespace Remoting {

   public class ChatServer : MarshalByRefObject {

      // Fields
      private ArrayList clients = new ArrayList(); // Names of clients
```

```
      private String chatSession = ""; // Holds text for chat session

      // Commands
      public void AddClient(String name) {
         if (name != null) {
            lock (clients) {
               clients.Add(name);
            }
         }
      }

      public void RemoveClient(String name) {
         lock (clients) {
            clients.Remove(name);
         }
      }

      public void AddText(String newText) {
         if (newText != null) {
            lock (chatSession) {
               chatSession += newText;
            }
         }
      }

      // Queries
      public ArrayList Clients() {
         return clients;
      }

      public String ChatSession() {
         return chatSession;
      }
   }
}

using System;
using System.Drawing;
using System.Collections;
using System.ComponentModel;
using System.Windows.Forms;
using System.Data;
using System.Runtime.Remoting;
using System.Runtime.Remoting.Channels;
using System.Runtime.Remoting.Channels.Http;
using System.Threading;

namespace Remoting {

   public class ChatClient: Form {
      // Controls
      private Button sendBtn;
      private Button logoutBtn;
      private Button loginBtn;
```

```
private ListBox listBox;
private Container components = null;
private RichTextBox sendTextBox;
private RichTextBox sessionTextBox;
private Button clearTextBtn;
private Label label1;
private Label label2;

// Fields
private ChatServer server;
private String name;
private Thread thrd; // Used to poll the server for client names

// Constructor
public ChatClient(String name) {
   InitializeComponent();
   this.name = name;
}

protected override void Dispose( bool disposing ) {
   if( disposing ) {
      if (components != null) {
         components.Dispose();
      }
   }
   base.Dispose( disposing );
}

private void InitializeComponent() {
   // Details not shown
}

private void InitializeRemoteServer() {
   RemotingConfiguration.RegisterWellKnownClientType(
      typeof(ChatServer), "http://localhost:12345/ChatServer");
}


[STAThread]
static void Main(String [] args) {
   if (args.Length != 1) {
      MessageBox.Show("Must supply name in command line.");
      return;
   }
   Application.EnableVisualStyles();
   Application.Run(new ChatClient(args[0]));
}

private void loginBtn_Click(object sender, System.EventArgs e) {
   HttpChannel channel = new HttpChannel();
   ChannelServices.RegisterChannel(channel);
   InitializeRemoteServer();
   server = new ChatServer();
   server.AddClient(this.name);
```

```
         thrd = new Thread(new ThreadStart(PollServer));
         thrd.Start();
      }

      private void logoutBtn_Click(object sender, System.EventArgs e) {
         server.RemoveClient(this.name);
         listBox.Items.Clear();
         listBox.Items.Add("No longer logged in.");
         thrd.Abort();
      }

      private void PollServer() {
         for (; ; ) {
            Thread.Sleep(1000);
            ArrayList clients = server.Clients();
            listBox.Items.Clear();
            foreach (String clientName in clients) {
               listBox.Items.Add(clientName);
            }
            String sessionText = server.ChatSession();
            sessionTextBox.Clear();
            sessionTextBox.Text = sessionText;
         }
      }

      private void clearTextBtn_Click(object sender,
                                      System.EventArgs e) {
         sendTextBox.Clear();
      }

      private void sendBtn_Click(object sender, System.EventArgs e) {
         String toSend = name + ": ";
         server.AddText(name + ":\n" + sendTextBox.Text + "\n\n");
      }
   }
}


using System;
using System.Runtime.Remoting;
using System.Runtime.Remoting.Channels;
using System.Runtime.Remoting.Channels.Http;

namespace Remoting {

   public class StartServer {

      static void Main() {
         // Create and register the channel
         HttpChannel channel = new HttpChannel(12345);
         ChannelServices.RegisterChannel(channel);
         Console.WriteLine("Starting ChatServer");

         // Register the ChatServer for remoting
```

```
            RemotingConfiguration.RegisterWellKnownServiceType(
                                  typeof(ChatServer),
                                  "ChatServer",
                                  WellKnownObjectMode.Singleton);
               Console.WriteLine("Press return to exit ChatServer.");
            Console.ReadLine();
         }
      }
   }
```

Discussion and Analysis of Listing 3

The *ChatServer* class holds two serializable fields, *clients*, an *ArrayList*, and *chatSession*, a *String*.

The *ChatClient* class is more complex because of the GUI. It contains a server field of type *ChatServer*. When each client application is launched, a command-line argument (string) that contains the name of the client (Richard and Erik in the example above) must be supplied to the program. This important identification is the only means that clients have to know who is logged-on and who they can "talk" to.

When the user clicks the "Login" button, code that enables the client to be connected to the server and the server object activated is executed. Since the server object is a singleton, its state is maintained between calls from various clients.

All calls to the remote *server* object are shown in blue. *MarshalByRefObject* is used to ensure that information is transmitted to the server through a proxy through serialization and not by value. If marshal by value were used (by removing the inheritance from class *MarshalByRefObject*), each client would have its own independent copy of the server object and would not be updated when another client added text or removed itself from the system.

About the author

Richard Wiener is Associate Professor of Computer Science at the University of Colorado at Colorado Springs. He is also the Editor-in-Chief of JOT and former Editor-in-Chief of the Journal of Object Oriented Programming. In addition to University work, Dr. Wiener has authored or co-authored 21 books and works actively as a consultant and software contractor whenever the possibility arises.