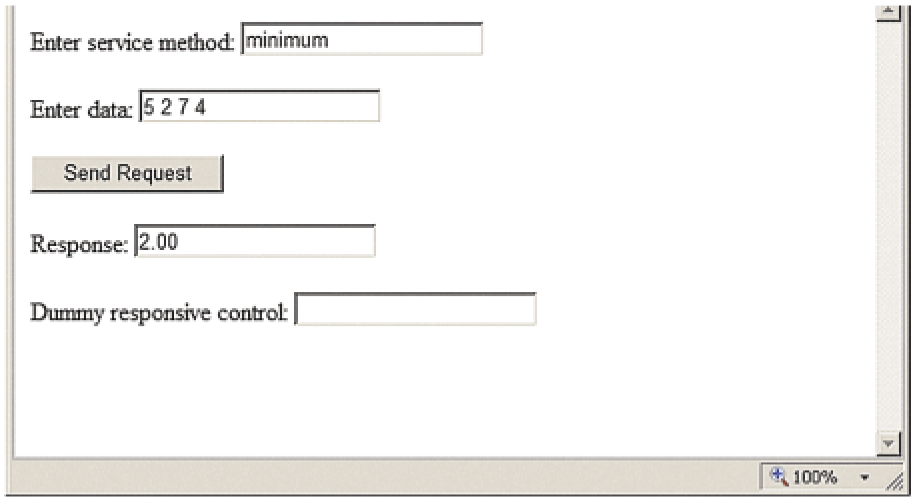
Enter service method: minimum

Enter data: 5 2 7 4

Response: 2.00

Dummy responsive control:

Figure 1 Demo TCP-Based Service with Two Clients

The bottom part of **Figure 1** shows a Web application client in action. The client has sent an asynchronous request to find the minimum value of (5, 2, 7, 4). Although it's not apparent from the screenshot, while the Web application is waiting for the service response, the application is responsive to user input.

In the sections that follow, I'll show how to code the service, the WinForm client and the Web application client. Along the way I'll discuss the pros and cons of using sockets. This article assumes you have at least intermediate-level C# programming skill, but does not

assume you have deep understanding or significant experience with asynchronous programming. The code download that accompanies this article has the complete source code for the three programs shown in **Figure 1**. I have removed most normal error checking to keep the main ideas as clear as possible.

Creating the Service

The overall structure of the demo service, with a few minor edits to save space, is presented in **Figure 2**. To create the service, I launched Visual Studio 2012, which has the required .NET Framework 4.5, and created a new C# console application named DemoService. Because socket-based services tend to have specific, limited functionality, using a more descriptive name would be preferable in a real-life scenario.

Figure 2 The Demo Service Program Structure

XML

```
using System;
using System.Net;
using System.Net.Sockets;
using System.IO;
using System.Threading.Tasks;
namespace DemoService
{
  class ServiceProgram
  {
    static void Main(string[] args)
    {
      try
      {
        int port = 50000;
        AsyncService service = new AsyncService(port);
        service.Run();
        Console.ReadLine();
      }
      catch (Exception ex)
      {
        Console.WriteLine(ex.Message);
        Console.ReadLine();
      }
    }
  }
  public class AsyncService
  {
    private IPAddress ipAddress;
    private int port;

    public AsyncService(int port) { . . }
    public async void Run() { . . }
    private async Task Process(TcpClient tcpClient) { . . }
    private static string Response(string request)
    private static double Average(double[] vals) { . . }
    private static double Minimum(double[] vals) { . . }
  }
}
```

After the template code loaded into the editor, I modified the using statements at the top of the source code to include System.Net and System.Net.Sockets. In the Solution Explorer window, I renamed file Program.cs to ServiceProgram.cs and Visual Studio automatically renamed class Program for me. Starting the service is simple:

XML

```
int port = 50000;
AsyncService service = new AsyncService(port);
service.Run();
```

Each custom socket-based service on a server must use a unique port. Port numbers between 49152 and 65535 are generally used for custom services. Avoiding port number collisions can be tricky. It's possible to reserve port numbers on a server using the system registry ReservedPorts entry. The service uses an object-oriented programming (OOP) design and is instantiated via a constructor that accepts the port number. Because service port numbers are fixed, the port number can be hardcoded rather than passed as a parameter. The Run method contains a while loop that will accept and process client requests until the console shell receives an <enter> key press.

The AsyncService class has two private members, ipAddress and port. These two values essentially define a socket. The constructor accepts a port number and programmatically determines the IP address of the server. Public method Run does all the work of accepting requests, then computing and sending responses. The Run method calls helper method Process, which in turn calls helper Response. Method Response calls helpers Average and Minimum.

There are many ways to organize a socket-based server. The structure used in the demo tries to strike a balance between modularity and simplicity, and has worked well for me in practice.

The Service Constructor and Run Methods

The two public methods of the socket-based demo service are presented in **Figure 3**. After storing the port name, the constructor uses method GetHostName to determine the name of the server, and to then fetch a structure that contains information about the server. The AddressList collection holds different machine addresses, including IPv4 and IPv6 addresses. The InterNetwork enum value means an IPv4 address.

Figure 3 Service Constructor and Run Methods

XML

```
public AsyncService(int port)
{
  this.port = port;
  string hostName = Dns.GetHostName();
  IPHostEntry ipHostInfo = Dns.GetHostEntry(hostName);
  this.ipAddress = null;
  for (int i = 0; i < ipHostInfo.AddressList.Length; ++i) {
    if (ipHostInfo.AddressList[i].AddressFamily ==
      AddressFamily.InterNetwork)
    {
      this.ipAddress = ipHostInfo.AddressList[i];
      break;
    }
  }
  if (this.ipAddress == null)
    throw new Exception("No IPv4 address for server");
}
public async void Run()
{
  TcpListener listener = new TcpListener(this.ipAddress, this.port);
  listener.Start();
  Console.Write("Array Min and Avg service is now running"
  Console.WriteLine(" on port " + this.port);
  Console.WriteLine("Hit <enter> to stop service\n");
  while (true) {
    try {
      TcpClient tcpClient = await listener.AcceptTcpClientAsync();
      Task t = Process(tcpClient);
      await t;
    }
    catch (Exception ex) {
      Console.WriteLine(ex.Message);
    }
  }
}
```

This approach restricts the server to listen to requests using only the server's first assigned IPv4 address. A simpler alternative could allow the server to accept requests sent to any of its addresses by just assigning the member field as this.ipAddress = IPAddress.Any.

Notice the service's Run method signature uses the async modifier, indicating that in the body of the method some asynchronous method will be called in conjunction with the await keyword. The method returns void rather than the more usual Task because Run is called by the Main method, which, as a special case, does not allow the async modifier. An alternative is to define method Run to return type Task and then call the method as service.Run().Wait.

The service's Run method instantiates a TcpListener object using the server's IP address and port number. The listener's Start method begins monitoring the specified port, waiting for a connection request.

Inside the main processing while loop, a TcpClient object, which you can think of as an intelligent socket, is created and waits for a connection via the AcceptTcpClientAsync method. Prior to the .NET Framework 4.5, you'd have to use BeginAcceptTcpClient and then write custom asynchronous coordination code, which, believe me, is not simple. The .NET

Framework 4.5 adds many new methods that, by convention, end with "Async." These new methods, combined with the async and await keywords, make asynchronous programming much, much easier.

Method Run calls method Process using two statements. An alternative is to use shortcut syntax and call method Process in a single statement: await Process(tcpClient).

To summarize, the service uses TcpListener and TcpClient objects to hide the complexity of raw socket programming, and uses the new AcceptTcpClientAsync method in conjunction with the new async and await keywords to hide the complexity of asynchronous programming. Method Run sets up and coordinates connection activities, and calls method Process to process requests and then a second statement to await on the return Task.

The Service Process and Response Methods

The Process and Response methods of the service object are presented in **Figure 4**. The Process method's signature uses the async modifier and returns type Task.

Figure 4 The Demo Service Process and Response Methods

XML

```
private async Task Process(TcpClient tcpClient)
{
  string clientEndPoint =
    tcpClient.Client.RemoteEndPoint.ToString();
  Console.WriteLine("Received connection request from "
    + clientEndPoint);
  try {
    NetworkStream networkStream = tcpClient.GetStream();
    StreamReader reader = new StreamReader(networkStream);
    StreamWriter writer = new StreamWriter(networkStream);
    writer.AutoFlush = true;
    while (true) {
      string request = await reader.ReadLineAsync();
      if (request != null) {
        Console.WriteLine("Received service request: " + request);
        string response = Response(request);
        Console.WriteLine("Computed response is: " + response + "\n");
        await writer.WriteLineAsync(response);
      }
      else
        break; // Client closed connection
    }
    tcpClient.Close();
  }
```

```
    catch (Exception ex) {
      Console.WriteLine(ex.Message);
      if (tcpClient.Connected)
        tcpClient.Close();
    }
  }
  private static string Response(string request)
  {
    string[] pairs = request.Split('&');
    string methodName = pairs[0].Split('=')[1];
    string valueString = pairs[1].Split('=')[1];
    string[] values = valueString.Split(' ');
    double[] vals = new double[values.Length];
    for (int i = 0; i < values.Length; ++i)
      vals[i] = double.Parse(values[i]);
    string response = "";
    if (methodName == "average") response += Average(vals);
    else if (methodName == "minimum") response += Minimum(vals);
    else response += "BAD methodName: " + methodName;
    int delay = ((int)vals[0]) * 1000; // Dummy delay
    System.Threading.Thread.Sleep(delay);
    return response;
  }
```

One of the advantages of using low-level sockets instead of Windows Communication Foundation (WCF) is that you can easily insert diagnostic WriteLine statements anywhere you choose. In the demo, I replaced clientEndPoint with the dummy IP address value 123.45.678.999 for security reasons.

The three key lines in method Process are:

XML

```
string request = await reader.ReadLineAsync();
...
string response = Response(request);
...
await writer.WriteLineAsync(response);
```

You can interpret the first statement to mean, “read a line of the request asynchronously, allowing other statements to execute if necessary.” Once the request string is obtained, it’s passed to the Response helper. Then the response is sent back to the requesting client asynchronously.

The server is using a read-request, write-response cycle. It's simple, but there are several caveats of which you should be aware. If the server reads without writing, it can't detect a half-open situation. If the server writes without reading (for example, responding with a large amount of data), it could deadlock with the client. A read-write design is acceptable for simple in-house services but shouldn't be used for services that are critical or public-facing.

The Response method accepts the request string, parses the request and computes a response string. A simultaneous strength and weakness of a socket-based service is that you must craft some sort of custom protocol. In this case, requests are assumed to look like:

XML

```
method=average&data=1.1 2.2 3.3&eor
```

In other words, the service expects the literal "method=" followed by the string "average" or "minimum," then an ampersand character ("&") followed by the literal "data=". The actual input data must be in space-delimited form. The request is terminated by an "&" followed by the literal "eor," which stands for end-of-request. A disadvantage of socket-based services compared to WCF is that serializing complex parameter types can be a bit tricky sometimes.

In this demo example, the service response is simple, just a string representation of the average or minimum of an array of numeric values. In many custom client-server situations, you'll have to design some protocol for the service response. For example, instead of sending a response just as "4.00," you might want to send the response as "average=4.00."

Method Process uses a relatively crude approach to close a connection if an Exception occurs. An alternative is to use the C# using statement (which will automatically close any connection) and remove the explicit call to method Close.

Helper methods Average and Minimum are defined as:

XML

```
private static double Average(double[] vals)
{
  double sum = 0.0;
  for (int i = 0; i < vals.Length; ++i)
    sum += vals[i];
  return sum / vals.Length;
}
private static double Minimum(double[] vals)
{
  double min = vals[0]; ;
  for (int i = 0; i < vals.Length; ++i)
    if (vals[i] < min) min = vals[i];
  return min;
}
```

In most situations, if you're using a program structure similar to the demo service, your helper methods at this point would connect to some data source and fetch some data. An advantage of low-level services is that you have greater control over your data-access approach. For example, if you're getting data from SQL, you can use classic ADO.NET, the Entity Framework or any other data access method.

A disadvantage of a low-level approach is you must explicitly determine how to handle errors in your system. Here, if the demo service is unable to satisfactorily parse the request string, instead of returning a valid response (as a string), the service returns an error message. Based on my experience, there are very few general principles on which to rely. Each service requires custom error handling.

Notice the Response method has a dummy delay:

XML

```
int delay = ((int)vals[0]) * 1000;
System.Threading.Thread.Sleep(delay);
```

This response delay, arbitrarily based on the first numeric value of the request, was inserted to slow the service down so that the WinForm and Web application clients could demonstrate UI responsiveness while waiting for a response.

The WinForm Application Demo Client

To create the WinForm client shown in **Figure 1**, I launched Visual Studio 2012 and created a new C# WinForm application named DemoFormClient. Note that, by default, Visual Studio modularizes a WinForm application into several files that separate the UI code from the logic code. For the code download that accompanies this article, I refactored the modularized Visual Studio code into a single source code file. You can compile the application by launching a Visual Studio command shell (which knows where the C#

compiler is), and executing the command: csc.exe /target:winexe DemoFormClient.cs.

Using the Visual Studio design tools, I added a ComboBox control, a TextBox control, two Button controls and a ListBox control, along with four Label controls. For the ComboBox control, I added strings “average” and “minimum” to the control’s Items collection property. I changed the Text properties of button1 and button2 to Send Async and Say Hello, respectively. Then, in design view, I double-clicked on the button1 and button2 controls to register their event handlers. I edited the click handlers as shown in **Figure 5**.

Figure 5 WinForm Demo Client Button Click Handlers

XML

```
private async void button1_Click(object sender, EventArgs e)
{
  try {
    string server = "mymachine.network.microsoft.com";
    int port = 50000;
    string method = (string)comboBox1.SelectedItem;
    string data = textBox1.Text;
    Task<string> tsResponse =
      SendRequest(server, port, method, data);
    listBox1.Items.Add("Sent request, waiting for response");

    await tsResponse;
    double dResponse = double.Parse(tsResponse.Result);
    listBox1.Items.Add("Received response: " +
     dResponse.ToString("F2"));
  }
  catch (Exception ex) {
    listBox1.Items.Add(ex.Message);
  }
}
private void button2_Click(object sender, EventArgs e)
{
  listBox1.Items.Add("Hello");
}
```

Notice the signature of the button1 control’s click handler was changed to include the async modifier. The handler sets up a hardcoded server machine name as a string and port number. When using low-level socket-based services, there’s no automatic discovery mechanism and so clients must have access to the server name or IP address and port information.

The key lines of code are:

XML

```
Task<string> tsResponse = SendRequest(server, port, method, data);
// Perform some actions here if necessary
await tsResponse;
double dResponse = double.Parse(tsResponse.Result);
```

SendRequest is a program-defined asynchronous method. The call can be loosely interpreted as “send an asynchronous request that will return a string, and when finished continue execution at the statement ‘await tsResponse,’ which occurs later.” This allows the application to perform other actions while waiting for the response. Because the response is encapsulated in a Task, the actual string result must be extracted using the Result property. That string result is converted to type double so that it can be formatted nicely to two decimal places.

An alternative calling approach is:

XML

```
string sResponse = await SendRequest(server, port, method, data);
double dResponse = double.Parse(sResponse);


listBox1.Items.Add("Received response: " + dResponse.ToString("F2"));
```

Here, the await keyword is placed in-line with the asynchronous call to SendRequest. This simplifies the calling code a bit and also allows the return string to be fetched without a call to Task.Result. The choice of using an inline await call or using a separate-statement await call will vary from situation to situation, but as a general rule of thumb, it’s better to avoid the explicit use of a Task object’s Result property.

Most of the asynchronous work is performed in the SendRequest method, which is listed in **Figure 6**. Because SendRequest is asynchronous, it might better be named SendRequestAsync or MySendRequestAsync.

Figure 6 WinForm Demo Client SendRequest Method

XML

```
private static async Task<string> SendRequest(string server,
  int port, string method, string data)
{
  try {
    IPAddress ipAddress = null;
```

```
      IPHostEntry ipHostInfo = Dns.GetHostEntry(server);
      for (int i = 0; i < ipHostInfo.AddressList.Length; ++i) {
        if (ipHostInfo.AddressList[i].AddressFamily ==
          AddressFamily.InterNetwork)
        {
          ipAddress = ipHostInfo.AddressList[i];
          break;
        }
      }
      if (ipAddress == null)
        throw new Exception("No IPv4 address for server");
      TcpClient client = new TcpClient();
      await client.ConnectAsync(ipAddress, port); // Connect
      NetworkStream networkStream = client.GetStream();
      StreamWriter writer = new StreamWriter(networkStream);
      StreamReader reader = new StreamReader(networkStream);
      writer.AutoFlush = true;
      string requestData = "method=" + method + "&" + "data=" +
        data + "&eor"; // 'End-of-request'
      await writer.WriteLineAsync(requestData);
      string response = await reader.ReadLineAsync();
      client.Close();
      return response;
    }
    catch (Exception ex) {
      return ex.Message;

    }
  }
```

SendRequest accepts a string representing the server name and begins by resolving that name to an IP address using the same code logic that was used in the service class constructor. A simpler alternative is to just pass the name of the server: await client.ConnectAsync(server, port).

After the server's IP address is determined, a TcpClient intelligent-socket object is instantiated and the object's ConnectAsync method is used to send a connection request to the server. After setting up a network StreamWriter object to send data to the server and a StreamReader object to receive data from the server, a request string is created using the formatting expected by the server. The request is sent and received asynchronously and returned by the method as a string.

The Web Application Demo Client

I created the demo Web application client shown in **Figure 1** in two steps. First, I used Visual Studio to create a Web site to host the application, and then I coded the Web

Visual Studio to create a Web site to host the application, and then I coded the Web application using Notepad. I launched Visual Studio 2012 and created a new C# Empty Web Site named DemoClient at https://localhost/. This set up all the necessary IIS plumbing to host an application and created the physical location associated with the Web site at C:\inetpub\wwwroot\DemoClient\. The process also created a basic configuration file, Web.config, which contains information to allow applications in the site to access async functionality in the .NET Framework 4.5:

XML

```xml
<?xml version="1.0"?>
<configuration>
  <system.web>
    <compilation debug="false" targetFramework="4.5" />
    <httpRuntime targetFramework="4.5" />
  </system.web>
</configuration>
```

Next, I launched Notepad with administrative privileges. When creating simple ASP.NET applications, I sometimes prefer using Notepad instead of Visual Studio so I can keep all application code in a single .aspx file, rather than generating multiple files and unwanted example code. I saved the empty file as DemoWebClient.aspx at C:\inetpub\wwwroot\DemoClient.

The overall structure of the Web application is shown in **Figure 7**.

Figure 7 Web Application Demo Client Structure

XML

```xml
<%@ Page Language="C#" Async="true" AutoEventWireup="true"%>
<%@ Import Namespace="System.Threading.Tasks" %>
<%@ Import Namespace="System.Net" %>
<%@ Import Namespace="System.Net.Sockets" %>
<%@ Import Namespace="System.IO" %>
<script runat="server" language="C#">
  private static async Task<string> SendRequest(string server,
  private async void Button1_Click(object sender, System.EventArgs e) { . . }
</script>
<head>
  <title>Demo</title>
</head>
<body>
  <form id="form1" runat="server">
```

```
    <div>
    <p>Enter service method:
      <asp:TextBox ID="TextBox1" runat="server"></asp:TextBox></p>
    <p>Enter data:
      <asp:TextBox ID="TextBox2" runat="server"></asp:TextBox></p>
    <p><asp:Button Text="Send Request" id="Button1"
      runat="server" OnClick="Button1_Click"> </asp:Button> </p>
    <p>Response:
      <asp:TextBox ID="TextBox3" runat="server"></asp:TextBox></p>
    <p>Dummy responsive control:
      <asp:TextBox ID="TextBox4" runat="server"></asp:TextBox></p>
    </div>
    </form>
  </body>
  </html>
```

At the top of the page I added Import statements to bring the relevant .NET namespaces into scope, and a Page directive that includes the Async=true attribute.

The C# script region contains two methods, SendRequest and Button1_Click. The application page body has two TextBox controls and one Button control for input, an output TextBox control to hold the service response, and a dummy, unused TextBox control to demonstrate UI responsiveness while the application waits for the service to respond to a request.
The code for the Web application's SendRequest method is exactly the same as the code in the WinForm application's SendRequest. The code for the Web application's Button1_Click handler differs only slightly from the WinForm's button1_Click handler to accommodate the different UI:

XML

```
try {
  string server = "mymachine.network.microsoft.com";
  int port = 50000;
  string method = TextBox1.Text;
  string data = TextBox2.Text;
  string sResponse = await SendRequest(server, port, method, data);
  double dResponse = double.Parse(sResponse);
  TextBox3.Text = dResponse.ToString("F2");
}
catch (Exception ex) {
  TextBox3.Text = ex.Message;
}
```

Even though the code for the Web application is essentially the same as the code for the

WinForm application, the calling mechanism is quite a bit different. When a user makes a request using the WinForm, the WinForm issues the call directly to the service and the service responds directly to the WinForm. When a user makes a request from the Web application, the Web application sends the request information to the Web server that's hosting the application, the Web server makes the call to the service, the service responds to the Web server, the Web server constructs a response page that includes the response and the response page is sent back to the client browser.

Wrapping Up

So, when should you consider using asynchronous TCP sockets instead of WCF? Roughly 10 years ago, before the creation of WCF and its predecessor technology ASP.NET Web Services, if you wanted to create a client-server system, using sockets was often the most logical option. The introduction of WCF was a big advance, but because of the huge number of scenarios WCF is designed to handle, using it for simple client-server systems might be overkill in some situations. Although the latest version of WCF is easier to configure than previous versions, it can still be tricky to work with WCF.

For situations where the client and server are on different networks, making security a major consideration, I always use WCF. But for many client-server systems where client and server are located on a single secure enterprise network, I often prefer using TCP sockets.

A relatively new approach for implementing client-server systems is to use the ASP.NET Web API framework for HTTP-based services combined with the ASP.NET SignalR library for asynchronous methods. This approach, in many cases, is simpler to implement than using WCF and avoids many of the low-level details involved with a socket approach.

Dr. James McCaffrey *works for Microsoft Research in Redmond, Wash. He has worked on several Microsoft products including Internet Explorer and Bing. He can be reached at jammc@microsoft.com.*

Thanks to the following technical experts for their advice and for reviewing this article: Piali Choudhury (MS Research), Stephen Cleary (consultant), Adam Eversole (MS Research) Lynn Powers (MS Research) and Stephen Toub (Microsoft)