webMethods for Microsoft Package Client API Programmer's Guide

webMethods for Microsoft Package

Version 7.1

August 2007

webMethods

This document applies to webMethods for Microsoft Package Version 7.1 and to all subsequent releases.

Specifications contained herein are subject to change and these changes will be reported in subsequent release notes or new editions.

Copyright © 2004—2007 Software AG, Darmstadt, Germany and/or Software AG USA, Inc., Reston, VA, United States of America, and/or their licensors.

Detailed information on trademarks and patents owned by Software AG and/or its subsidiaries is located at http://documentation.softwareag.com/legal/.

Use of this software is subject to adherence to Software AG's licensing conditions and terms. These terms are part of the product documentation, located at http://documentation.softwareag.com/legal/ and/or in the root installation directory of the licensed product(s).

This software may include portions of third-party products. For third-party copyright notices and license terms, please refer to "License Texts, Copyright Notices and Disclaimers of Third Party Products." This document is part of the product documentation, located at http://documentation.softwareag.com/legal/ and/or in the root installation directory of the licensed product(s).

Document ID: MSP-CLIENTAPI-PG-71-20121008

Contents

About This Guide 5
Document Conventions 5
Additional Information 5

Chapter 1. Invoking a Service using a Microsoft .NET Client 7
Generating Microsoft .NET Clients 8
The Structure of Client Code 8
Description of the Generated Code Sample 9
Generating a C# Class in Visual Studio .NET 9
List Packages to be Used (Generated Code) 11
Class Declaration (Generated Code) 11
Connect to Integration Server (Generated Code) 11
Populate the Service Inputs (Generated Code) 12
Invoke the Service and Retrieve the Output (Generated Code) 12
Disconnect from Integration Server (Generated Code) 12
Complete Generated Code Sample 13
Description of the IData Sample 14
List Packages to be Used (IData) 15
Class Declaration (IData) 15
Connect to Integration Server (IData) 16
Populate the Service Inputs (IData) 16
Invoke the Service and Retrieve the Output (IData) 17
Disconnect from Integration Server (IData) 17
Complete IData Code Sample 18
Notes About IData and IDataUtil 20
Microsoft .NET Client API Documentation 21

Chapter 2. Using the .NET Remoting API 23
Overview of Remoting for Microsoft .NET 24
Alternate Solutions 24
Remoting Solution 24
Sample Code for Remoting 25
The Remote Server 25
The Remote Client 26
The Bat File 30

About This Guide

This guide provides some examples for using the C# Client API that is part of the webMethods for Microsoft Package. The examples show how to invoke a service using the client API and how to expose, as webMethods services, those services provided by remote .NET components and accessible by means of the Microsoft .NET Remoting API.

Document Conventions

Convention	Description
Bold	Identifies elements on a screen.
Italic	Identifies variable information that you must supply or change based on your specific situation or environment. Identifies terms the first time they are defined in text. Also identifies service input and output variables.
Narrow font	Identifies storage locations for services on the webMethods Integration Server using the convention *folder.subfolder:service*.
`Typewriter font`	Identifies characters and values that you must type exactly or messages that the system displays on the console.
UPPERCASE	Identifies keyboard keys. Keys that you must press simultaneously are joined with the "+" symbol.
\	Directory paths use the "\" directory delimiter unless the subject is UNIX-specific.
[ ]	Optional keywords or values are enclosed in [ ]. Do not type the [ ] symbols in your own code.

Additional Information

The webMethods Advantage Web site at http://advantage.webmethods.com provides you with important sources of information about webMethods products:

- **Troubleshooting Information.** The webMethods Knowledge Base provides troubleshooting information for many webMethods products.
- **Documentation Feedback.** To provide feedback on webMethods documentation, go to the Documentation Feedback Form on the webMethods Bookshelf.

- **Additional Documentation.** Starting with 7.0, you have the option of downloading the documentation during product installation to a single directory called "_documentation," located by default under webMethods installation directory. In addition, you can find documentation for all webMethods products on the webMethods Bookshelf.

Chapter 1. Invoking a Service using a Microsoft .NET Client

- Generating Microsoft .NET Clients 8
- The Structure of Client Code 8
- Description of the Generated Code Sample 9
- Description of the IData Sample 14
- Microsoft .NET Client API Documentation 21

Generating Microsoft .NET Clients

The code samples in this chapter show two different methods of invoking a selected webMethods service from the context of the Microsoft .NET environment. The invocation uses the webMethods .NET API. The webMethods .NET Client API functionality is supplied in a set of DLLs. You need to reference these DLLs by means of a `using` statement. The examples in this chapter do not create whole executables; rather, they are show how to generate objects you can use in your own code.

For the first example, the webMethods add-in for Microsoft Visual Studio .NET allows you to select a service from a running Integration Server and generate a C# or Visual Basic class. This class encapsulates the integration service, allowing for interaction with the service in an object oriented way. This example is described in "Description of the Generated Code Sample" on page 9.

If you are familiar with the Java Client API for Integration Server, you may prefer to use the standard IData pipeline interface for invoking Integration Services. For more on using Idata, see "Description of the IData Sample" on page 14.

The Structure of Client Code

The .NET sample code is generated using the same structure as is used for generation of other language samples. This structure breaks essential elements out into separate routines. For purposes of simplification, we condense the essential elements of those samples into a more concise form. The essential elements of a .NET webMethods client are:

1. Import library references by means of the `using` statement.
2. Create a server context object.
3. Connect the server context to a running Integration Server.
4. Populate the service inputs.
5. Invoke one or more services.
6. Retrieve the service outputs.
7. Disconnect from Integration Server.

This same set of steps applies to both examples described in this chapter.

Description of the Generated Code Sample

Using the webMethods add-in for Microsoft Visual Studio .NET, you can select a service from a running Integration Server and generate a C# or Visual Basic class, as shown in "Generating a C# Class in Visual Studio .NET" on page 9. This sample shows how you would go about creating a console application, but does not attempt to depict the end-to-end actions needed to create an executable.

This sample requires that you add two references to the .NET project, which you can find in the webMethods Add-In for Microsoft Visual Studio .NET installation directory.

The files are CGUTIL.dll and wmClientAPI.dll.

This sample is based on the WmPublic.pub.string:concat service delivered with Integration Server.

The following sections provide brief descriptions of the sample code as they relate to "The Structure of Client Code" on page 8:

- "List Packages to be Used (Generated Code)" on page 11
- "Class Declaration (Generated Code)" on page 11
- "Connect to Integration Server (Generated Code)" on page 11
- "Invoke the Service and Retrieve the Output (Generated Code)" on page 12
- "Disconnect from Integration Server (Generated Code)" on page 12

You can find the complete code sample in "Complete Generated Code Sample" on page 13.

Generating a C# Class in Visual Studio .NET

Using the webMethods add-in for Microsoft Visual Studio .NET, you can connect to a running Integration Server and generate an object from an integration service. This sample uses the concat service.

To create a C# file from the concat service

1. Open Visual Studio .NET.
2. Open a new .NET project using the **Console Application** template.
3. On the **Tools** menu, click **webMethods add-in for Microsoft Visual Studio .NET**.

Note: If this menu item is not visible, make sure you have installed the add-in. For information, see the *webMethods for Microsoft Package Installation and User's Guide*.

4 In the **webMethods add-in for Microsoft Visual Studio .NET** dialog box, provide the following information about the instance of Integration Server to which to connect.

In this field...	Type this...
webMethods Integration Server	Integration Server host name and port in the format *host:port*. The default is localhost:5555.
UserID	Name of a valid user account on this Integration Server.
Password	Password for the user account. Passwords are case-sensitive.

5 Click **Connect**. The **webMethods add-in for Microsoft Visual Studio .NET** window is displayed. This window contains a tree view of packages and services on the Integration Server to which you are connected.

Tip! The add-in opens as a floating window. You can dock the window by dragging it to the Visual Studio .NET toolbar.

6 In the tree view of packages, go to the WmPublic.pub.string:concat service.

7 Right-click the concat service and, click **Generate Client Code**, and then click **C# Code**.

8 If the Solution Explorer is not already open, in the **View** menu, click **Solution Explorer**. The concat.cs file should be visible in the **Solution Explorer** panel.

9 Add the references by doing the following:

   a In the **Solution Explorer** panel, right-click the **References** node, and then click **Add Reference**.

   b With the **.NET** tab selected in the Add References window, click **Browse** and go to the *webMethods_install_dir*\webMethods Add-In for Microsoft Visual Studio .NET directory.

   c Select CGUTIL.dll and wmClientAPI.dll and click **Open**.

   d With the two DLLs in the **Selected Components** panel, click **OK**. The two DLLs should now appear under the **References** node in the **Solution Explorer** panel.

The following sections describe how the generated C# class might be used in the code sample.

List Packages to be Used (Generated Code)

The first stage of the code sample contains `using` statements that specify the .NET packages to use:

```
using System;
using webMethods.ClientAPI;
```

Class Declaration (Generated Code)

The second stage of the code creates the context in which the client operates:

```
namespace CodeGenClientSample
{
  /// <summary>
  /// Sample client demonstrating invocation of the 'concat' Integration
  /// Service using an object generated by the webMethods Visual
  /// Studio Add-in.
  /// </summary>
  class ClientSample2
```

Connect to Integration Server (Generated Code)

The third stage creates the connection to Integration Server:

```
{
  String returnString = null;

  // create our connection context with the Integration Server
  Context serverContext = new Context();

  // connect to the server
  try
  {
    serverContext.connect( "localhost:5555", "Administrator", "manage" );
  }
  catch( Exception ex )
  {
    Console.WriteLine("Connection to server failed, reason=" + ex );
    return null;
  }
```

Populate the Service Inputs (Generated Code)

The fourth stage of the code creates input data variables:

```
// create the Concat service object
//(created by the Add-in for the pub.string:concat service)
Concat concatService = new Concat();

// populate the inputs
concatService.in_inString1 = string1;
concatService.in_inString2 = string2;
```

Invoke the Service and Retrieve the Output (Generated Code)

The fifth and sixth stages are to invoke the service and retrieve the output:

```
// invoke the service
concatService.invoke( serverContext );

// extract the output
returnString = concatService.out_value;
```

Disconnect from Integration Server (Generated Code)

The seventh stage is to disconnect from Integration Server:

```
   // disconnect from the server
   serverContext.disconnect();
```

Complete Generated Code Sample

The following sample shows the all of the C# code used in "Description of the Generated Code Sample" but does not attempt to depict the end-to-end actions needed to create an executable:

```
using System;
using webMethods.ClientAPI;


namespace CodeGenClientSample
{
  /// <summary>
  /// Sample client demonstrating invocation of the 'concat' Integration   ///
Service using an object generated by the webMethods Visual
  /// Studio Add-in.
  /// </summary>
  class ClientSample2
  {
    /// <summary>
    /// The main entry point for the application.
    /// </summary>
    [STAThread]
    static void Main(string[] args)
    {
      // execute the generated service object
      String result = concatStrings( "testString1", "testString2" );
      Console.WriteLine( "concat=" + result );
    }


    /**
     * Concatentate strings using the Add-in generated class for the
     * pub.strings:concat service
     */
    public static String concatStrings( String string1, String string2 )
    {
      String returnString = null;

      // create our connection context with the Integration Server
      Context serverContext = new Context();
```

```
      // connect to the server
      try
      {
        serverContext.connect( "localhost:5555", "Administrator", "manage" );
      }
      catch( Exception ex )
      {
        Console.WriteLine("Connection to server failed, reason=" + ex );
        return null;
      }


      // create the Concat service object
      //(created by the Add-in for the pub.string:concat service)
      Concat concatService = new Concat();


      // populate the inputs
      concatService.in_inString1 = string1;
      concatService.in_inString2 = string2;


      // invoke the service
      concatService.invoke( serverContext );

      // extract the output
      returnString = concatService.out_value;


      // disconnect from the server
      serverContext.disconnect();

      return returnString;

    }

  }
}
```

Description of the IData Sample

Using the Java Client API for Integration Server, you can use the standard IData pipeline interface for invoking Integration Services. Using the webMethods add-in for Microsoft Visual Studio .NET, you can select a service from a runningIntegration Server and generate a C# or Visual Basic class. This sample does not attempt to depict the end-to-end actions needed to create an executable.

This sample requires that you add one reference to the .NET project, which you can find in the webMethods Add-In for Microsoft Visual Studio .NET installation directory or the *Integration Server_directory*\lib directory.

The file is wmClientAPI.dll.

This sample is based on the Integration Server built-in service WmPublic.pub.math:addInts.

The following sections provide brief descriptions of the sample code as they relate to "The Structure of Client Code" on page 8:

- "List Packages to be Used (IData)" on page 15
- "Class Declaration (IData)" on page 15
- "Connect to Integration Server (IData)" on page 16
- "Populate the Service Inputs (IData)" on page 16
- "Invoke the Service and Retrieve the Output (IData)" on page 17
- "Disconnect from Integration Server (IData)" on page 17

You can find the complete code sample in "Complete IData Code Sample" on page 18.

List Packages to be Used (IData)

The first stage of the code sample contains `using` statements that specify the .NET packages to be used:

```
using System;
using webMethods.ClientAPI;
using webMethods.ClientAPI.Data;
```

Class Declaration (IData)

The second stage of the code creates the context in which the client operates:

```
namespace IDataClientSample
{
  /// <summary>
  /// Sample client demonstrating invocation of an Integration Service
  /// using the standard IData pipeline mechanism
  /// </summary>
```

Connect to Integration Server (IData)

The third stage creates the connection to Integration Server:

```
{
  String returnString = null;

  // create our connection context with the Integration Server
  Context serverContext = new Context();

  // connect to the server
  try
  {
    serverContext.connect( "localhost:5555", "Administrator", "manage" );
  }
  catch( Exception ex )
  {
    Console.WriteLine("Connection to server failed, reason=" + ex );
    return null;
  }
```

Populate the Service Inputs (IData)

The fourth stage of the code creates input data variables. Service inputs are passed to Integration Server as an IData object. This code fragment demonstrates construction of the input IData pipeline:

```
      // create the service inputs as an IData object
      IData inputIData = IDataFactory.create();
      // get a data cursor for our input IData
      IDataCursor inputCursor = inputIData.getCursor();

      // the addInts service takes 2 inputs: num1 and num2 (as strings)
      IDataUtil.put( inputCursor, "num1" , num1.ToString() );
      IDataUtil.put( inputCursor, "num2" , num2.ToString() );

      // done with the input cursor, destroying it only cleans up the
      // cursor resources, the IData object is left unaffected
      inputCursor.destroy();
```

Invoke the Service and Retrieve the Output (IData)

The fifth and sixth stages are combined in the serverContext.invoke method, which invokes the service with the IData input object and receives the reply as an Idata output object:

```
{
  // Invoke the service
  IData outputIData = serverContext.invoke( serviceFolder,
    serviceName, inputIData );
  // get a data cursor for out output IData
  IDataCursor outputCursor = outputIData.getCursor();

  // the addInts service has one return variable, "value" as string
  // however we can retrieve the value as an int using the IDataUtil
  // helper method getInteger();
  sum = IDataUtil.getInt( outputCursor, "value", 0 );

}

catch( Exception ex )
{
  Console.WriteLine("The service invoke failed, reason=" + ex );
}
```

Disconnect from Integration Server (IData)

The seventh stage is to disconnect from Integration Server, as shown in this fragment:

```
// disconnect from the server
serverContext.disconnect();
```

Complete IData Code Sample

The following sample shows the all of the C# code used in "Description of the IData Sample" but does not attempt to depict the end-to-end actions needed to create an executable:

```
using System;
using webMethods.ClientAPI;
using webMethods.ClientAPI.Data;


namespace IDataClientSample
{
  /// <summary>
  /// Sample client demonstrating invocation of an Integration Service
  /// using the standard IData pipeline mechanism
  /// </summary>
  class ClientSample1
  {
    /// <summary>
    /// The main entry point for the application.
    /// </summary>
    [STAThread]
    static void Main(string[] args)
    {
      // invoke the sample service using an IData pipeline
      int sum = addInts( 11, 22 );
      Console.WriteLine( "sum=" + sum );
    }


    /**
     * Add two integers together by invoking the pub.math:addInts service
     * using an IData pipeline
     */
    public static int addInts( int num1, int num2 )
    {
      // the resulting sum of the 2 integers
      int sum = 0;
```

```
// create our connection context with the Integration Server
Context serverContext = new Context();

// connect to the server
try
{
  serverContext.connect( "localhost:5555", "Administrator", "manage" );
}
catch( Exception ex )
{
  Console.WriteLine("Connection to server failed, reason=" + ex );
  return 0;
}


// create the service inputs as an IData object
IData inputIData = IDataFactory.create();
// get a data cursor for our input IData
IDataCursor inputCursor = inputIData.getCursor();


// the addInts service takes 2 inputs: num1 and num2 (as strings)
IDataUtil.put( inputCursor, "num1" , num1.ToString() );
IDataUtil.put( inputCursor, "num2" , num2.ToString() );

// done with the input cursor, destroying it only cleans up the
// cursor resources, the IData object is left unaffected
inputCursor.destroy();


try
{
  // Invoke the service
  IData outputIData = serverContext.invoke( pub.math,
    addInts, inputIData );
  // get a data cursor for out output IData
  IDataCursor outputCursor = outputIData.getCursor();


  // the addInts service has one return variable, "value" as string
  // however we can retrieve the value as an int using the IDataUtil
  // helper method getInteger();
  sum = IDataUtil.getInt( outputCursor, "value", 0 );

}
```

```
      catch( Exception ex )
      {
        Console.WriteLine("The service invoke failed, reason=" + ex );
      }

      // disconnect from the server
      serverContext.disconnect();


      return sum;
    }

  }
}
```

Notes About IData and IDataUtil

IData can store any data object that is an instance of the Common Data Types, arrays with two or less dimensions, or instances of any class derived from System.Object. However, instances of custom classes may cause errors when used to invoke an integration service. Integration Server runs as a Java process; any custom objects passed to an integration service are meaningless because Integration Server does not have a definition of that class. The same holds true for services that attempt to return an instance of a custom Java class. The .NET client does not have a class definition with which to create the Java class.

The Client API includes the helper class IDataUtil for extracting variables from IData with common data types. For example, getString() extracts an object from an IData object and casts it as a String. Similar methods exist for integers, arrays, and so forth. Consult the IDataUtil class documentation for details.

Output variables that are native arrays are retrieved from the output IData as System.Array instances, not as their native array type. For example, input data can be entered as a native array type:

```
IDataUtil.put( cursor, "testArray", new int[2] {1 ,2 } );
```

However, when retrieved from IData, the array is of type System.Array.

```
System.array testArray = IDataUtil.getArray( cursor, "testArray");
```

Microsoft .NET Client API Documentation

The Client API documentation is installed as part of the webMethods for Microsoft Package. These documents are available in a format similar to Javadoc and can be found in the Resources section of the webMethods for Microsoft Package administration page.

To access the webMethods for Microsoft Package from a connection to Integration Server

1. Start your browser.
2. Point your browser to the host and port for an instance of Integration Server where the webMethods for Microsoft Package is installed. Use the syntax http://*host*:*port*. For example, if Integration Server is running on the default port on a machine named myhost, you would type http://myhost:5555.

   If Integration Server is running, you are prompted for a user name and password.
3. Log on with a user name that has administrator privileges. For information on maintaining administrator privileges, see the *webMethods Integration Server Administrator's Guide*.
4. In the **Solutions** menu of the navigation frame, click **Microsoft Package**.
5. Expand the Resources node and click Client API Doc.

Chapter 2. Using the .NET Remoting API

- Overview of Remoting for Microsoft .NET 24
- Sample Code for Remoting 25

Overview of Remoting for Microsoft .NET

This chapter describes one approach to exposing, as webMethods services, those services provided by remote .NET components and accessible by means of the Microsoft .NET Remoting API.

Alternate Solutions

The .NET remoting solution described here is only one solution. Three other solutions are readily available, which you may want to consider:

- Use two Integration Servers. This solution involves having instances of Integration Server installed at both the primary and remote locations. Assembly methods would be exposed as webMethods services at the remote location and those services can be remotely invoked from the primary IS.
- Use the facilities of other webMethods components to expose the assembly methods as Web services and invoke them from Integration Server as Web services.
- Attribute the assembly methods using the Microsoft [webMethods] attribute, create an ASP to access that assembly, deploy the assembly and ASP to IIS, generate the WSDL to describe the object, use the WSDL to create services in Integration Server, then invoke those methods as Web services from Integration Server.

Any of these techniques will provide a functionally equivalent solution to remoting and should be considered as an alternative.

Remoting Solution

Any .NET assembly class that inherits from MarshalByRefObject is a candidate for remoting. To access an assembly by means of .NET remoting, the remote object classes whose methods you want to use must inherit from MarshalByRefObject. If this is the case, the methods in those classes can be invoked either via HttpChannel (a raw SOAP-based invocation useful when tunneling through a firewall) or TcpChannel (a much more efficient binary protocol).

There are a number of implementation possibilities in the .NET remoting environment. The remote object can be client activated or it can be server activated. If server activated, the client can operate as a singleton or a single call object. Server objects must be registered by an encapsulating application before they can be accessed by a client. You must be aware of the default or configured lifetime of the server objects you are dealing with and ensure that the remote lifetime matches the configured local lifetime for the client object. Finally the object methods can be accessed by means of multiple protocols. This discussion is outside the scope of the webMethods for Microsoft Package but it is worthwhile to understand the range of possibilities and their application before venturing into the remoting environment. Microsoft documentation covers these issues thoroughly.

Although it appears complicated, the technique is actually simple. The example included here serves to illustrate one of the available techniques.

In this release we assume that the remoting application already exists and, as such, the client of the remote object is present. We further assume that this client is a DLL that contains public methods. If this is the case, the client can be introspected to generate services in the webMethods namespace. Those webMethods services represent the remote services provided by the remote object. If the remote application is new, you only have to create a wrapper client DLL to access the remote application, which can then be introspected and used to generate webMethods services.

Sample Code for Remoting

The sample code for remoting provided here consists of three components that will illustrate the technique. One source file is a remote server and the other is a client of that server.The should be named server.cs and client.cs. A bat file is provided for compiling the two source files. These files are described in the following sections.

In brief, after the source files are compiled, the execution sequence is:

1. Start the server
2. Set the host location (optional)
3. Connect to the remote server
4. Send messages.

The Remote Server

The module server.cs is a remote server implementation that compiles to an executable. Once started in a command window, the executable waits for incoming connections.

The server.cs module registers its class, provides the remote public methods and makes those methods available by means of the TcpChannel.The following example shows the complete server.cs file.

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

```
public class CoHello : MarshalByRefObject
{
    public static void Main()
    {
        TcpChannel channel = new TcpChannel( 4000 );
        ChannelServices.RegisterChannel( channel );
        RemotingConfiguration.RegisterWellKnownServiceType(
            typeof( CoHello ),              // type name
            "HelloDotNet",                  // URI
            WellKnownObjectMode.Singleton ); // operating mode


        System.Console.WriteLine( "Hit <enter> to exit..." );
        System.Console.ReadLine();
    }


    public void SayHello()
    {
        Console.WriteLine( "Hello from afar" );
    }
    public String SayHello(String msg)
    {
        Console.WriteLine( msg );
        return( "Hello from afar" );
    }
    public int SayHello(int msg)
    {
        Console.WriteLine( msg.ToString() );
        return( 1 );
    }
    public void SayHello(String[] msg)

    {
        for( int i = msg.Length; i < msg.Length; i++ )
        {
            Console.WriteLine( msg[i] );
        }
    }
}
```

The Remote Client

The module client.cs is the component used to communicate with the remote server. When compiled it generates a DLL that you can introspect from Developer to generate webMethods services for the application. The remote client has three methods.

The client.cs module creates a TcpChannel, creates an instance of the remote object, and drives the remote object methods. When client.cs is compiled into a DLL, that DLL can be introspected by Developer, and services for each of its methods can be created in the

webMethods namespace. The client.cs module has three methods. One method allows you to set the host location (remote object location). The host location specifies a local host so you can omit running that method if your server is local.

```
private static String remoteServer =
    "tcp://127.0.0.1:4000/HelloDotNet";
private Object obj = null;
CoHello h = null;

public String setRemoteServer( String server )
{
    remoteServer = server;
    return( remoteServer );
}
```

Another method connects to the remote server and caches the object reference. You need to run this method at least once after setting the host location (if you choose to do so).

```
public String connect()
{
    String reply = null;
    try
    {
        TcpChannel channel = new TcpChannel();
        ChannelServices.RegisterChannel( channel );
        obj = Activator.GetObject(
            typeof( CoHello ),        // type
            remoteServer );           //location
        h = (CoHello)obj;
        reply = "connected";
    }
    catch( Exception e )
    {
        Console.WriteLine( e.ToString() );
        reply = "server connection failed";
    }
    return( reply );
}
```

The last method sends a message that is displayed on the remote console window and returns a reply parameter in the service invoke pipeline. Remember that if you have this configured as a session lifetime object, you need to use the instance ID returned from the

first method invocation in all subsequent method invocations (see the *webMethods for Microsoft Package Installation and User's Guide* for more information).

```
    public String talkToRemoteServer( String msg )
    {
        String reply = null;
        try
        {
            reply = h.SayHello( msg );
        }
        catch( Exception e )
        {
            Console.WriteLine( e.ToString() );
            reply = "server invocation failed";
        }
        return( reply );
    }
```

The following example shows the complete client.cs file.

```
using System;
using System.Runtime.Remoting;
using System.Runtime.Remoting.Channels;
using System.Runtime.Remoting.Channels.Tcp;
using System.Reflection;

public class Client
{
    private static String remoteServer =
        "tcp://127.0.0.1:4000/HelloDotNet";
    private Object obj = null;
    CoHello h = null;

    public String setRemoteServer( String server )
    {
        remoteServer = server;
        return( remoteServer );
    }

    public String connect()
    {
        String reply = null;
        try
        {
```

```
            TcpChannel channel = new TcpChannel();
            ChannelServices.RegisterChannel( channel );
            obj = Activator.GetObject(
                typeof( CoHello ),        // type
                remoteServer );           //location
            h = (CoHello)obj;
            reply = "connected";
        }
        catch( Exception e )
        {
            Console.WriteLine( e.ToString() );
            reply = "server connection failed";
        }
        return( reply );
    }

    public String talkToRemoteServer( String msg )
    {
        String reply = null;
        try
        {
            reply = h.SayHello( msg );
        }
        catch( Exception e )
        {
            Console.WriteLine( e.ToString() );
            reply = "server invocation failed";
        }
        return( reply );
    }
}
```

To summarize the execution sequence:

1. Start the server in a command window
2. Set the host location (optional)
3. Connect to the remote server
4. Send messages.

You can accomplish steps 2, 3 and 4 using Developer after the DLL has been introspected and services have been created.

The Bat File

The module CSbake.bat is a bat file that compiles the source files.

```
csc server.cs
csc /t:library /r:server.exe client.cs
```