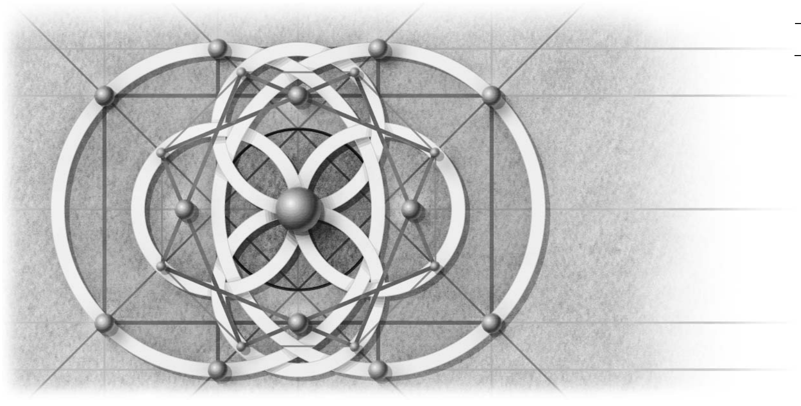


21

Upgrading Distributed Applications

Until now we haven't really talked about upgrading large-scale distributed applications. But building these types of applications is, after all, the cornerstone of the entire Microsoft .NET initiative. The .NET Framework is designed from the ground up to support new first-class distributed technologies. Knowing how these technologies are implemented is critical in assessing how to upgrade a given application. Unfortunately, so many possibilities and variations exist that it is impossible to address all or even a significant portion of them in a meaningful way.

This chapter gives an overview of the core concepts behind large-scale distributed applications. It then discusses how these concepts are implemented in Visual Basic .NET. We highlight the use of these technologies and describe how to implement them in your applications. Although this chapter covers some interoperability issues (such as how to build wrappers and marshal objects such as ADO *Recordset*s from the server to the client), there are others you may face if you get creative with your application. Consult Chapter 13 for more information regarding COM interoperability.

Note Some of the examples in this chapter use the Northwind database Northwind.mdb (included on the companion CD) as a basis for demonstrating various forms of data access.

Important Concepts for Distributed Applications

Since our topic is building large-scale distributed applications in Visual Basic .NET, now would be a good time to take a step back and provide an overview of the concepts that are important for understanding the benefits and trade-offs of the possible approaches. Writing smaller applications usually does not require a great deal of planning or design work. Issues such as calling overhead and latency don't typically rear their heads in smaller applications. Larger-scale applications, however, are a far different beast. For the purposes of this chapter, we define a **distributed application** as an application in which one or more of the logical tiers are located on remote machines. It is critically important to be aware of the design issues that can potentially make or break such an application. To this end, this section looks at three important concepts in large-scale distributed application development:

- Loosely coupled and tightly coupled applications
- Overhead involved in method calls
- Componentization and logical organization

Loosely Coupled vs. Tightly Coupled Applications

Imagine using a standard platform-independent communication mechanism between remote (network-separated) application segments (think SOAP). What this does is enable you to develop applications that have no knowledge or understanding of either participating system. In this scenario, you agree only on the communications mechanism. The platforms can vary wildly. In .NET, two systems are considered **loosely coupled** if the only mandate imposed on them both is to understand self-describing, text-based messages.

Tightly coupled systems, on the other hand, impose a significant amount of customized overhead to enable communication and require a greater understanding between the systems (think Distributed Common Object Model [DCOM] or remoting). A tightly coupled application requires that there be a greater level of system integration and dependence. A loosely coupled system is far less prone to problems with system setup, as long as the minimum communication requirements are met.

The other side to coupling is the overhead requirement. A loosely coupled system imposes greater marshaling overhead. The native binary format must be translated to something that both systems can understand. It is then the responsibility of the target system to parse that message and convert it to a format that it understands. A tightly coupled application does not need to translate to and from a universal format. Typically the network format is binary and is well understood by both parties, requiring less translation and, by extension, the use of fewer processing resources.

Overhead in Method Invocation

When you are developing a distributed application, you need to be aware of the limitations of this type of application. In a standard application, you typically don't worry about the processing overhead of function calls. These calls are typically limited only by the speed of your computer. Moving to the network introduces more variables into the equation.

Figure 21-1 demonstrates the relative differences in the overhead of method calls, depending on where the target resides. As you can see, in-process method invocation has very little calling overhead. Out-of-process method invocation introduces more overhead, due to marshaling across the process boundaries. This type of invocation involves communicating with another application on the local computer (such as a Microsoft Windows service or a COM+ server application). Remote process method invocation brings into play not just process-boundary marshaling but also network latency. This typically means a vastly greater calling overhead than any of the other options and implies the need for a different approach to the application's architecture.

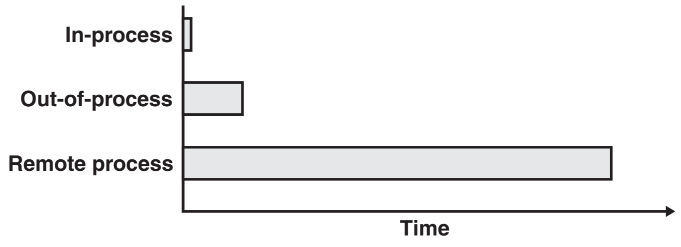


Figure 21-1 Overhead of method calls in different contexts.

So while we can call into local methods without any concern for overhead, and even out-of-process overhead isn't too bad, we need to be extra careful of not only how often we call remote methods but also how much work these methods do. Think of it this way: since the overhead is greater, we need to make the method calls worthwhile. Consider the following example to help demonstrate what we mean. It shows the difference between setting properties on an object—a typically simple operation with low overhead—and using a single method to set all the properties at once.

```
' Approach 1 - Don't do this for a remote or out-of-process object
Obj.Property1 = val1
Obj.Property2 = val2
Obj.Property3 = val3
Obj.Property4 = val4
Obj.Property5 = val5

' Approach 2 - Better for a remote or out-of-process object
Obj.SetValues(val1, val2, val3, val4, val5)
```

This example uses two different approaches, but it does not really clarify why one is better than the other. For this, look to Figure 21-2. It should help you visualize the differences between the approaches and understand how they affect an application's performance. Approach 1 is "chatty" in that each property access involves a separate network operation, whereas approach 2 is "chunky" because it accomplishes its task with a single network operation.

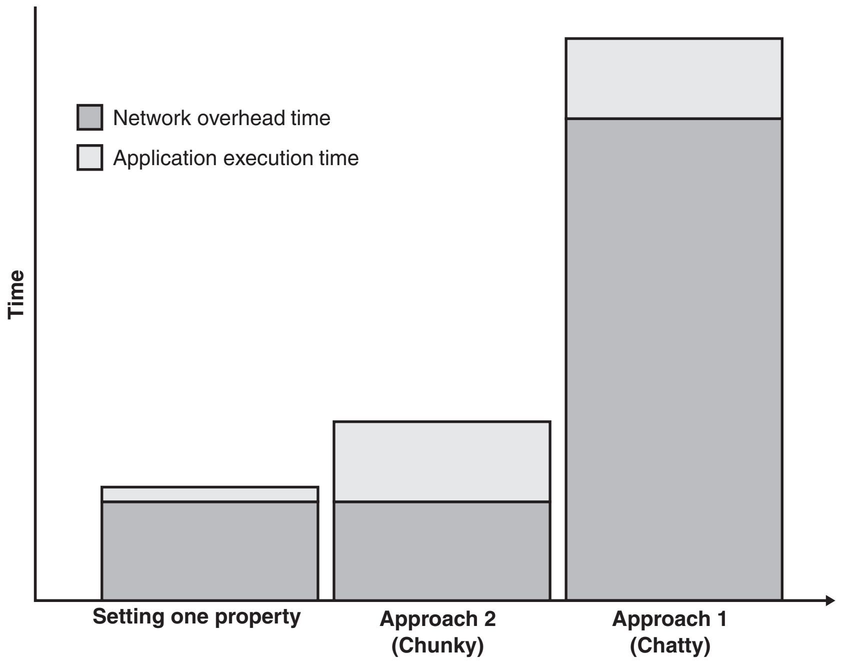


Figure 21-2 Remote objects and the effect of network latency.

At this point, it should be clear that the design of your remote objects and interface is vitally important to your application's scalability and performance. When you look at the performance of approach 1, you should also understand that when you access any remote object you are consuming resources on the remote server, which means that fewer connections are available to other applications. The more efficient your design, the better your application will perform and the better it will scale up in an enterprise environment.

Componentization and Logical Organization

Componentization is an extremely important concept for any large-scale application. It is more than just a separation based on the functional tiers of your application (such as user interface, business logic, and the data tier). It is also the separation of functional areas within individual tiers into specific components. This separation has several benefits. Imagine a customer management application. You might divide the server-side components into database, business logic, and general utility components, as shown in Figure 21-3. This structure gives you the ability to concentrate all of the database logic into a single area of the application and manage database changes through versions in a more feasible fashion.

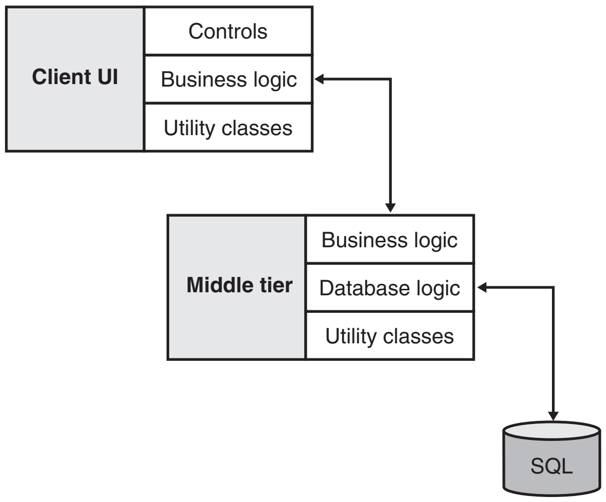


Figure 21-3 Hypothetical componentized application.

From the standpoint of upgrading, logical separation of code enables a smoother approach to upgrading larger applications. It gives you more choices regarding what to upgrade and how to upgrade your application within a given application. So before you undertake any upgrading tasks, it makes a lot of sense to analyze your application and look at ways to partition the logic into separate containers and to consider the option of leaving some code in Visual Basic 6 while upgrading elements that could derive immediate benefits.

Distributed Technologies in .NET

Visual Basic .NET uses three main distributed technologies (ignoring the ADO.NET data access technologies). The first is XML Web services. This is an open standard that enables developers to expose a set of software services over the Internet and to do so simply. The second is another new technology, unique to the .NET Framework, called .NET remoting. You can think of remoting as the .NET equivalent of DCOM. It is a method of communication between different operating system processes, regardless of whether they are on the same computer. The .NET **remoting** system is an architecture designed to simplify communication between objects living in different application domains, whether they're on the same computer or not, and between different contexts, whether they're in the same application domain or not. The last of the distributed technologies is COM+ application proxies. If you have worked with COM+ in a distributed environment, you are probably already familiar with application proxies. Let's look at each of these technologies in turn.

XML Web Services

The Internet is quickly evolving from the Web sites of today that deliver only pages to Web browsers to the next generation of programmable Web-based technologies that directly link organizations, applications, services, and devices with one another. These programmable Web sites are not passive—they are reusable, intelligent Web services. The common language runtime provides built-in support for creating and exposing XML Web services, using a programming abstraction that is consistent and familiar to both ASP.NET Web Forms developers and existing Visual Basic users. The resulting model is both scalable and extensible and embraces open Internet standards—HTTP, XML, SOAP, and Web Services Description Language (WSDL)—so that it can be accessed and consumed from any client or Internet-enabled device.

Support for XML Web services is provided by ASP.NET, using the .asmx file extension. An .asmx file contains code similar to an .aspx file. In fact, much of the ASP.NET programming model is available to XML Web service developers, with the exception of the user interface classes. For a Web service, the user interface has no real meaning. It is, after all, a middle-tier technology. There is a caveat, however. ASP.NET projects and ASP.NET Web service projects are not fundamentally different, and there is no required separation. You can add an XML Web service file to a Web project, and you can also add a Web form to an XML Web service project. These files are then URI addressable, in the same way that .aspx files are. Also like .aspx files, .asmx files are compiled automatically by the ASP.NET runtime when a request to the service is made. (Subsequent requests are serviced by a cached precompiled object.)

On a more fundamental level, XML Web services enable the exchange of data and the remote invocation of application logic using XML messaging to move data through firewalls and between heterogeneous systems. Although remote access of data and application logic is not a new concept, XML Web services enable it to take place in a loosely coupled fashion. The only assumption between the XML Web service client and the XML Web service is that recipients will understand the messages they receive. As a result, programs written in any language, using any component model, and running on any operating system can access XML Web services.

When you create and/or use an XML Web service, you are taking advantage of SOAP. You may be aware of the SOAP Toolkit that was published for Visual Basic developers. While you can think of XML Web services as an alternative to the SOAP Toolkit, they are far more powerful, thanks to the .NET Framework. The SOAP Toolkit required a lot of work on the part of the developer, and Visual Basic imposed limitations related to serialization of objects. Visual Basic .NET removes these limitations and provides a far more powerful framework for developing SOAP applications by eliminating the need to implement features such as serialization and proxy generation. These features are already implemented in Visual Basic .NET and can be leveraged without any additional effort. It is possible, of course, to implement serialization yourself to provide any custom features that your application requires. For the vast majority of applications, however, the built-in serialization capabilities of the .NET Framework are more than sufficient.

Creating a Simple XML Web Service

Implementing a basic XML Web service in any application is a straightforward process. The XML Web Service project type (selected in the New Project dialog box) contains all the necessary elements for implementing an XML Web service. If you are running Visual Basic .NET on a machine with Internet Information Services (IIS) installed, it is a trivial task to create a new project and implement an XML Web service. The companion CD contains a sample called Simple Web Service, as well as a setup document that discusses the steps necessary to get the example up and running.

The basic steps for creating an XML Web service are as follows:

1. Create a new ASP.NET Web service project.
2. Add a new XML Web service file to the project.
3. Add a method to the XML Web service class, and mark the method with the *WebMethod* attribute.

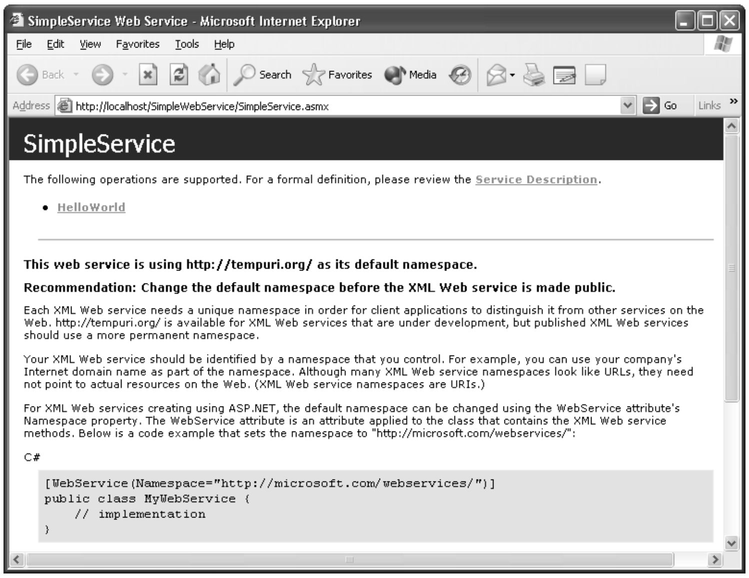
The most important part here is the use of the *WebMethod* attribute. This attribute tells the compiler to generate the XML contract for the Web service, register it for discovery, and produce the implementation code necessary for marshaling parameters and return objects. It also enables you to filter the methods of your XML Web service class and publish only the methods you want to make publicly available over the Internet. Think of this filtering as taking access protection a step further. You're already familiar with creating public, private, and friend methods. But there will probably be situations in which your application architecture requires a public method on your XML Web service class that you do not want to make available through the XML Web service itself. In such cases, the power of the *WebMethod* attribute really shines. It gives you absolute control over what you publish over the Internet without requiring you to sacrifice good component design in your XML Web service classes. The following excerpt from the SimpleService.asmx.vb file (part of the SimpleWebService sample project) demonstrates the *WebMethod* attribute in action:

```
<WebMethod()> Public Function HelloWorld() As String
   HelloWorld = "Hello World. The current time is " & Now()
End Function
```

You can do much more with the *WebMethod* attribute, such as manipulate additional settings, but we will use it in its simplest, most basic form, as demonstrated here. The MSDN documentation gives a more thorough treatment of the use of this attribute, above and beyond its default behavior.

Now is a good time to talk about restrictions on parameters and return values for XML Web services. An XML Web service is not a bidirectional object. This means that you cannot pass objects by reference; they must always be passed by value. In other words, your parameters and return values must be serializable. Most, if not all, of the intrinsic data types in Visual Basic .NET and the .NET Framework support serialization as long as the objects themselves are self-contained—that is, as long as they don't represent or contain any physical system resources, such as database connections or file handles. For your custom classes you have two options. You can implement the *ISerializable* interface for your class, but this is the more manual process. Alternatively, you can use the *Serialize* class attribute. This option enables your class to serialize itself. No work is necessary on your part unless one or more of the types contained in your class do not support serialization. In such cases, you have the choice of adding the *Serializable* attribute to those classes (if they are under your control) or implementing the serialization yourself with the *ISerializable* interface.

The Simple Web Service example demonstrates the basic elements of an XML Web service. Figure 21-4 illustrates what the compiled service looks like in a Web browser. You can see all the methods exposed by the service and view the complete XML service description. We will leave the latter for you to pursue if you're interested. It is really used only by the Web reference mechanism in Visual Basic .NET, and you don't need to be familiar with the contents of the service description to make use of an XML Web service.


SimpleService Web Service - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Search Favorites Media
Address http://localhost/SimpleWebService/SimpleService.asmx Go Links

SimpleService

The following operations are supported. For a formal definition, please review the Service Description.

- HelloWorld

This web service is using http://tempuri.org/ as its default namespace.

Recommendation: Change the default namespace before the XML Web service is made public.

Each XML Web service needs a unique namespace in order for client applications to distinguish it from other services on the Web. http://tempuri.org/ is available for XML Web services that are under development, but published XML Web services should use a more permanent namespace.

Your XML Web service should be identified by a namespace that you control. For example, you can use your company's Internet domain name as part of the namespace. Although many XML Web service namespaces look like URLs, they need not point to actual resources on the Web. (XML Web service namespaces are URIs.)

For XML Web services creating using ASP.NET, the default namespace can be changed using the WebService attribute's Namespace property. The WebService attribute is an attribute applied to the class that contains the XML Web service methods. Below is a code example that sets the namespace to "http://microsoft.com/webservices/":

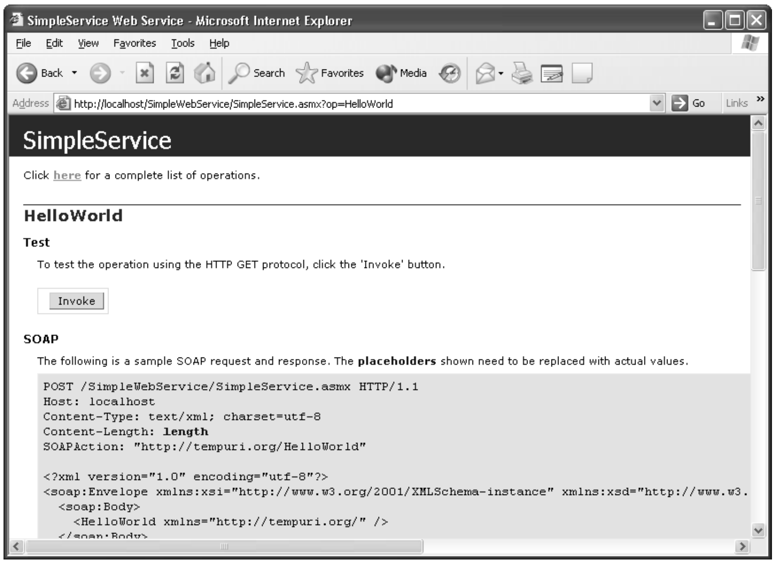
C#

```
[WebService(Namespace="http://microsoft.com/webservices/")]
public class MyWebService {
    // implementation
}
```

Figure 21-4 The SimpleService.asmx XML Web service as it appears in a Web browser.

Testing Your XML Web Service

As you have already seen, when you build an XML Web service, the Visual Basic IDE creates a test page for that service. This page allows you to inspect all the methods that are exposed, and it also gives you the ability to invoke the methods manually. This should be your first step in testing the service, to ensure that it is doing what you expect. In our sample service, you can click the *HelloWorld* method in the service description page to get more information about it. Figure 21-5 shows what this method looks like.


SimpleService Web Service - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Search Favorites Media
Address http://localhost/SimpleWebService/SimpleService.asmx?op=HelloWorld
Go Links

SimpleService

Click here for a complete list of operations.

HelloWorld

Test

To test the operation using the HTTP GET protocol, click the 'Invoke' button.

Invoke

SOAP

The following is a sample SOAP request and response. The **placeholders** shown need to be replaced with actual values.

```
POST /SimpleWebService/SimpleService.asmx HTTP/1.1
Host: localhost
Content-Type: text/xml; charset=utf-8
Content-Length: length
SOAPAction: "http://tempuri.org/HelloWorld"

<?xml version="1.0" encoding="utf-8"?>
<soap:Envelope xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:xsd="http://www.w3.
  <soap:Body>
    <HelloWorld xmlns="http://tempuri.org/" />
```

Figure 21-5 The SimpleWebService's *HelloWorld* method.

As you can see, a great deal of information is available through this page. You can also invoke the method (using the Invoke button) and view the SOAP result message. Additionally, if the service accepts parameters, this page will have input boxes for each of the parameters. There are some limitations to this, however. If your methods require parameters in a binary format (a *Byte* array, for example), you will need to invoke the method programmatically.

The result of invoking the *HelloWorld* method is displayed in a separate window and looks like this:

```
<?xml version="1.0" encoding="utf-8" ?>
<string xmlns="http://tempuri.org/">Hello World. The current time is
10/13/2001 11:13:21 AM</string>
```

Pretty neat, huh? Now we need to look at how you can use this method in an application.

Consuming a Simple Web Service

Now that an XML Web service exists on the target machine, it is possible to add a Web reference to a client project. An XML Web service client can be any kind of project, from a command-line or form-based application to a full-blown Web site or COM+ component. In this case, we have gone with a simple form-based application, but feel free to experiment by adding Web references to other kinds of projects.

Recall our discussion of different types of references in Chapter 6. A Web reference is simply a process by which the Visual Basic IDE goes out to the specified XML Web service, gets the XML contract for the service, and builds a proxy class that implements the contract. This process occurs transparently, and it is flexible enough that references can be "refreshed" in the IDE. In other words, it is possible to have the IDE regenerate the proxy class (which is usually desirable if there are new features that you want to take advantage of or if there have been interface implementation changes).

This proxy is not tightly coupled to the complete interface of an XML Web service. It will use methods and properties according to how they were defined the last time the proxy was generated. If the specified methods have not changed their signature—that is, the parameter types, parameter count, or return type—your application will not break, regardless of any other changes that might have been made to the XML Web service itself.

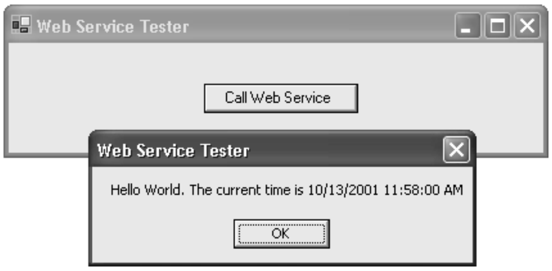
Creating this proxy is simple, as we have already mentioned. You create a simple Windows Forms project and add a Web reference by right-clicking the References collection in the Solution Explorer, specifying the path to the XML Web service that you have already created, and clicking OK. After a short delay, a new XML Web service proxy class should be added to your application. Now all that is necessary is to use it. Our sample solution has a Web Service Tester project that implements this sample XML Web service. This is a one-button application that does the following two things in response to a button click:

1. Create the XML Web service proxy object.
2. Call the *HelloWorld* method, and display a message box with the result.

The code itself is simple:

```
Private Sub CallButton_Click(ByVal sender As System.Object, _
   ByVal e As System.EventArgs) Handles CallButton.Click
   Dim ws As New localhost.SimpleService()
   MsgBox(ws.HelloWorld())
End Sub
```

You can see the results in Figure 21-6. This sample project helps demonstrate how XML Web services simplify the development of distributed applications. XML Web Services are a powerful tool that enable the development of fundamentally different kinds of applications, due to its platform- and language-independent nature.



Hello World. The current time is 10/13/2001 11:58:00 AM

Figure 21-6 Web Service Tester application in action.

Moving On

This is all well and good, you may be thinking, but what about my existing applications? How should I implement XML Web services in an existing COM or managed application? Does it require redesign, or are there alternatives? Read on….

Supporting Web Services in Your Existing Applications

If you have an existing application for which you want to implement XML Web services without a major rewrite, you can take the approach of implementing the service as a wrapper around your existing code. Doing so allows you to handle the necessary serialization work in the service without having to modify the original application's code. This can be an effective way to add support for XML Web services to existing COM applications without making any radical changes. It also gives the developer the opportunity to rethink the application's interface to the outside by hiding methods or doing additional work that was not previously done on the server side.

This approach does raise some questions. If you are wrapping existing COM applications, you need to be aware of return types and whether you will need to implement serialization for them. This concern arises as a potential problem only when you need to pass variables that are specific COM types. Intrinsic data types will have no problems; it gets tricky only when you have complex custom data types, such as COM objects. What it ultimately boils down to is that you cannot use a COM component as either a Web service parameter or a return type. You must first serialize a COM component to an intrinsic data type (an XML string or a Byte array). Of the commonly used Microsoft COM objects, only ADO supports serialization to and from XML, and it is a manual process—there is no such thing as implicit serialization for COM objects. If you create your own COM components, you may wish to add support for XML serialization. Including support is essential if you want to support passing such a component to and from an XML Web service.

There is also the possibility that you cannot modify the original COM object to add serialization capabilities. In this instance, you would be required to create code to perform the necessary serialization work in Visual Basic .NET, using it on both the server and client tiers. However, it may not be possible to work around some issues and you may be required to make some modifications to your original application. From a middle-tier standpoint, one of the most common objects passed around, and thus a likely candidate for inclusion in an XML Web service, is the ADO *Recordset* object, which is the subject of the next section.

Using *Recordset* with XML Web Services

As we mentioned earlier, it is possible to wrap an XML Web service around an existing COM object. If you are passing only the intrinsic types (such as strings, integers, and arrays of variables), you simply need to call the methods directly. The .NET Framework already knows how to serialize these types of variables. If you need to pass ADO *Recordset*s back and forth, however, you have some work to do.

In ADO 2.1, Microsoft added XML support to the ADO *Recordset* object. This object provides support for serialization to and from XML. The only caveat is that the support for serialization is not implicit. In other words, the *Recordset* does not support automatic or implicit serialization, but you can do the work explicitly. The key here is the *Save* method on the *Recordset* object and the use of the ADODB *Stream* object. Figure 21-7 illustrates the process of wrapping a COM object with an XML Web service, where the result is an ADO *Recordset*, and shows how the *Recordset* is marshaled from server to client.

Essentially, the figure depicts the process by which the *Recordset* is persisted to XML on the server and is returned to the client. The client then reconstructs the original *Recordset*. Notice that the reconstructed *Recordset* is not exactly the same as the original. The new *Recordset* does not contain any connections to a database, nor does it understand where it came from. For this reason, we say that this *Recordset* is disconnected.

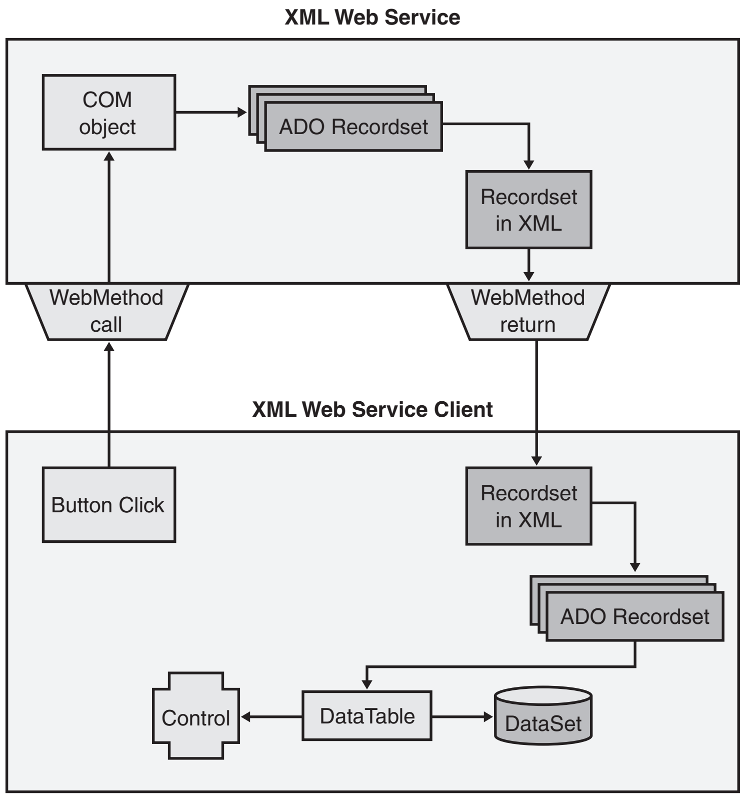


Figure 21-7 Passing a *Recordset* in an XML Web service.

To give you a better feel for how this process works, we've provided a sample on the companion CD called Web Services and ADODB. The sample consists of two projects: an XML Web service (NorthwindDatabase) and a Windows Forms client application (Northwind Viewer). The design of the sample is simple. One method is available through the XML Web service that allows the client to specify a SQL query string and returns the result as an XML representation of the resulting *Recordset*.

The following is the implementation code for the XML Web service. The *Query* method does all the work. You can see that this method takes a query string as a parameter and returns a string. As we have discussed previously, this return string contains the XML representation of the *Recordset* returned by the specified query. This project references ADODB and uses the ADO *Connection* object directly to execute the query. It would also work just fine with a regular COM object that returns a *Recordset*; we are just trying to make the sample as clear and concise as possible.

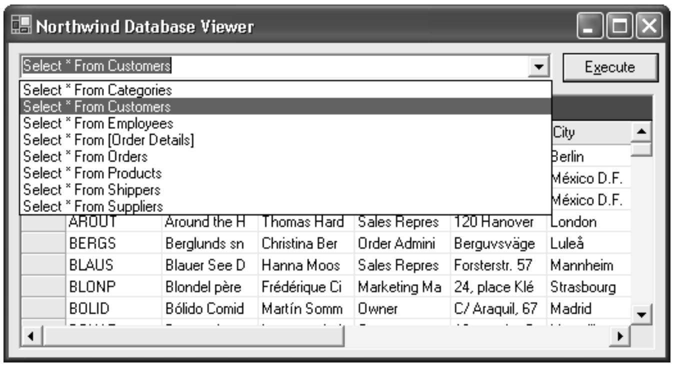
```
Imports ADODB
Imports System.Web.Services

<WebService(Namespace:="http://tempuri.org/")> _
Public Class NorthwindDatabase
   Inherits System.Web.Services.WebService

#Region " Web Services Designer Generated Code "

   <WebMethod()> Public Function Query(ByVal queryString As String) _
      As String
      Dim conn As New Connection()

      Try
         ' You will need to specify the local path to your copy
         ' of the Northwind.mdb file. This is just an example.
         conn.Open( "Provider=Microsoft.Jet.OLEDB.4.0;" & _
            "Data Source=c:\Northwind.mdb")
      Catch e As Exception
         Return Nothing
      End Try

      Dim rs As Recordset
      Try
         rs = conn.Execute(queryString)
      Catch e As Exception
         conn.Close()
         Return Nothing
      End Try

      Dim str As New StreamClass()
      rs.Save(str, PersistFormatEnum.adPersistXML)
      Query = str.ReadText()

      str.Close()
      rs.Close()
      conn.Close()
   End Function
End Class
```

After the query has been executed, we use the ADODB *Stream* class to serialize the contents of the *Recordset*. (In Chapter 20, we touched on XML serialization for *Recordset*s and introduced the concept with the *RsToString* and *StringToRS* methods.) Notice that there are no particular requirements as to the type of *Recordset* used. This example uses a forward-only *Recordset* (which is more than sufficient), but it would work equally well with any kind of client or server-side cursor. Also notice that we close out all the ADO classes (*Stream*, *Recordset*, and *Connection*). Chapter 10 discusses why this step is crucial.

Now that we have the XML Web service, we need something to test it. The test sample, Northwind Viewer, has a couple of interesting features. Not only does it deserialize a *Recordset* from XML, but it also uses the *OleDbDataAdapter* to convert the *Recordset* to an ADO.NET *DataTable*. It then sets the *DataSource* property of the *QueryDataGrid* using the newly created table. That's it, aside from additional initialization code in the form's *Load* event. We should point out one thing here. We instantiate the XML Web service proxy only once in the *Load* event, rather than on every button click. This practice is perfectly acceptable because the proxy class does not represent an active network connection. Connections are made only when a method on the class is invoked, so we don't bother creating the same proxy over and over. Creating it once is sufficient. Again, notice that we clean up the ADO objects by calling *Close*. Always play nice with your COM objects and clean up afterward.

```
Imports ADODB
Imports System.Data
Imports System.Data.OleDb

Public Class Form1
   Inherits System.Windows.Forms.Form

#Region " Windows Form Designer generated code "

   Dim nwdb As localhost.NorthwindDatabase
   Private Sub Form1_Load(ByVal sender As System.Object, _
      ByVal e As System.EventArgs) Handles MyBase.Load
      nwdb = New localhost.NorthwindDatabase()

      ' Add the query strings for the database
      QueryComboBox.Items.Add("Select * From Categories")
      QueryComboBox.Items.Add("Select * From Customers")
      QueryComboBox.Items.Add("Select * From Employees")
      QueryComboBox.Items.Add("Select * From [Order Details]")
      QueryComboBox.Items.Add("Select * From Orders")
      QueryComboBox.Items.Add("Select * From Products")
      QueryComboBox.Items.Add("Select * From Shippers")
      QueryComboBox.Items.Add("Select * From Suppliers")
   End Sub

   Private Sub ExecuteButton_Click(ByVal sender As System.Object, _
      ByVal e As System.EventArgs) Handles ExecuteButton.Click
      Dim xml As String

      xml = nwdb.Query(QueryComboBox.Text)
      If xml Is Nothing Then
         QueryDataGrid.DataSource = Nothing
         MsgBox("The query failed!")
         Exit Sub
      End If
      Dim str As New Stream()
      str.Open()
      str.WriteText(xml)
      str.Position = 0

      Dim rs As New Recordset()
      rs.Open(str)

      Dim da As New OleDbDataAdapter()
      Dim table As New DataTable()

      da.Fill(table, rs)

      QueryDataGrid.DataSource = table

      rs.Close()
      str.Close()
   End Sub
End Class
```

Figure 21-8 demonstrates the test application in action. You can see that the user selects a query from the ComboBox and then clicks the Execute button. Another neat feature of this application is that the ComboBox is editable at run time, so you can further customize the query.



					City
					Berlin
					México D.F.
					México D.F.
AROUT	Around the H	Thomas Hard	Sales Repres	120 Hanover	London
BERGS	Berglunds sn	Christina Ber	Order Admini	Berguvsväge	Luleå
BLAUS	Blauer See D	Hanna Moos	Sales Repres	Forsterstr. 57	Mannheim
BLONP	Blondel père	Frédérique Ci	Marketing Ma	24, place Klé	Strasbourg
BOLID	Bólido Comid	Martín Somm	Owner	C/ Araquil, 67	Madrid

Figure 21-8 Northwind Database Viewer application in action.

You should now be able to see how you might get started wrapping your application with XML Web services. Depending on how your application is implemented, it is quite possible to handle the serialization issues without a great deal of disruption. An advantage is that it is extremely easy to create a test XML Web service to discover how your application will behave and what is feasible.

C2161587x.fm Page 452 Friday, November 16, 2001 9:08 AM

Remoting

As we mentioned earlier, remoting is the process of communication between different operating system processes, regardless of whether they are on the same computer. It simplifies communication between objects living in different application domains, possibly on different computers, and between different contexts, possibly in different application domains. The remoting framework is built into the common language runtime and can be used to build sophisticated distributed applications. Some of the features provided by .NET remoting are as follows:

- Proxy objects
- Object passing
- Activation models
- Stateless and stateful objects
- Channels and serialization
- Lease-based lifetime
- Hosting objects in IIS

The most basic remoting scenario requires a client application and a minimal host process. The host process must register specific object types with the runtime and specify the desired configuration details. Remoting itself can be thought of as an abstraction layer over a transport protocol that enables applications to instantiate objects that may reside on remote machines and work with them as if they were local. It is an interesting technology because it is removed from the underlying transport protocol. What this means is that, unlike DCOM, you have choices as to how your client and server processes talk to each other.

Currently, remoting supports two transport protocols: TCP/IP and HTTP. Each has its advantages, depending on your requirements. TCP/IP offers the better performance by far. It is basically a low-level binary protocol that enables virtually direct communication with the remote objects. This protocol tends to work well in an intranet environment in which you have unrestricted TCP port usage. Using TCP/IP becomes problematic when you need to support clients across multiple sites or over the Internet. Most corporations have firewalls that block anything but the standard TCP/IP ports, which would render a remoting application using a custom TCP/IP port unusable.

The solution is the HTTP protocol. Firewalls are almost always configured to allow traffic over TCP/IP port 80 (the port used by the World Wide Web). Using HTTP, your remoting applications can communicate freely over the span of the Internet. The downside is that HTTP is nowhere near as efficient as TCP/IP. When you specify HTTP as your remoting transport protocol, you basically end up with XML over HTTP. Serializing your objects into XML is not as compact as using a binary format. It increases the size of the payload, increases processing resources on both the client and server, and takes longer to transmit. This extra overhead means that a method call over HTTP costs more than one over TCP/IP. Will you notice the difference in casual use? Not really. But it does have implications for larger-scale applications. It's nothing to worry about, just something to keep in mind.

As far as requirements for objects capable of remoting go, there is only one that is significant: they must derive the *MarshalByRefObject* class. Notice that this is a built-in feature of COM+ classes because the *ServicedComponent* class has *MarshalByRefObject* as a base class. Now is a good time to look at an example.

A Simple Remoting Example

The simple example in this section helps demonstrate the lifetime of remoting within your applications and illustrates some of the architectural considerations. Called Simple Remoting and included on the companion CD, it consists of a host process and a client process. The host process is a lightweight application that opens a TCP channel for listening to remoting requests and registers a type with the common language runtime. Once it is registered, the type is available for remote instantiation through the specified channel. The example uses the type *SimpleClass* (derived from *MarshalByRefObject*) as the remoting object and contains two separate projects (a server project and a client project). Before we go any further, let's take a look at *SimpleClass*:

```
Public Class SimpleClass
   Inherits MarshalByRefObject

   Public Function GetDateTime() As Date
      Return Now()
   End Function

   Public Function Hello() As String
      Return "Hello World!"
   End Function
End Class
```

From the application's perspective, the critical part is the server project, which is responsible for registering the remoting object with the host runtime. The code for doing this is fairly simple and looks like the following:

```
Dim channel As New TcpChannel(9999)
ChannelServices.RegisterChannel(channel)
RemotingConfiguration.ApplicationName = "SimpleRemotingServer"
Dim t As Type = Type.GetType("Server.SimpleClass")
RemotingConfiguration.RegisterWellKnownServiceType(t, "SimpleClass", _
   WellKnownObjectMode.SingleCall)
```

It is important to note that the type is available through remoting only when the host process is active. If the host process terminates at any time, all types registered by that process are immediately unregistered and cease to respond to any incoming requests. This includes objects that have already been instantiated. Thus, if the process terminates while a client is still working with the proxy, the client will experience a network failure on the next call through the proxy object. Figure 21-9 illustrates the lifetime of the Simple Remoting example. It emphasizes that once the *SimpleClass* type has been registered, it is available only during the lifetime of the server process.

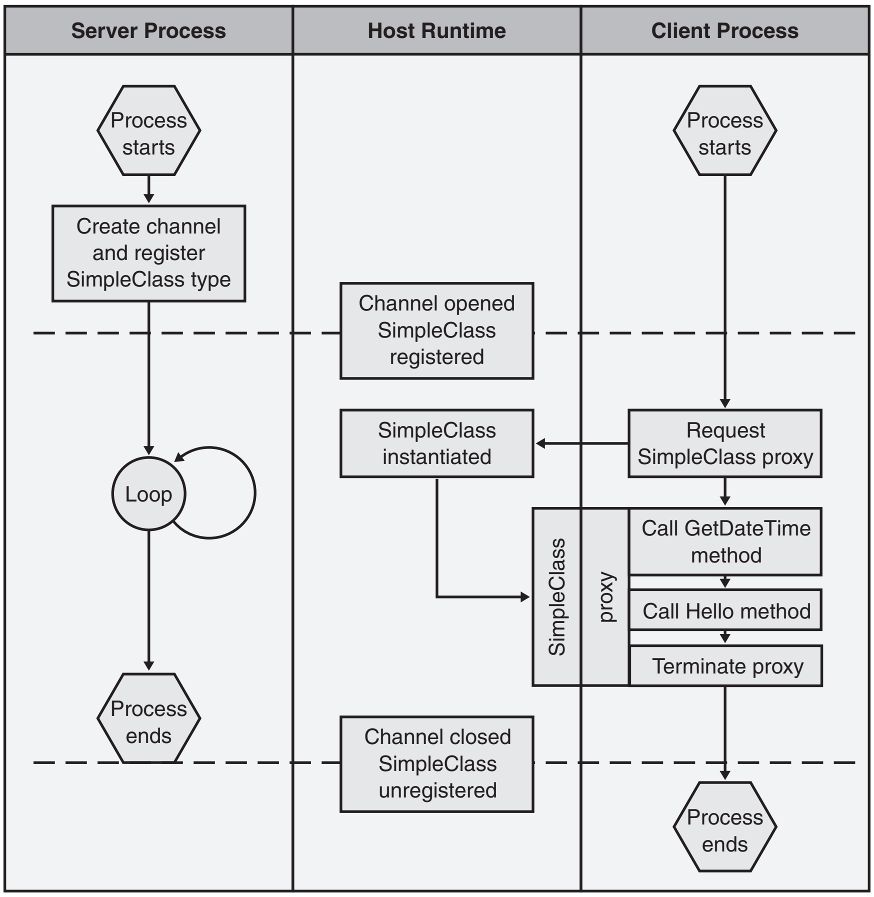
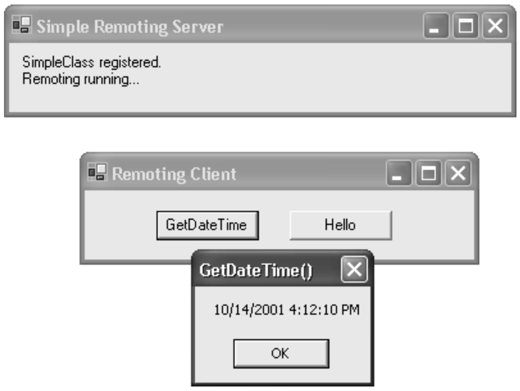


Figure 21-9 Life cycle of the Simple Remoting example.

As you can see, we need to keep this process running in some form or another; otherwise, our client is out of luck. Speaking of the client, here is what it looks like.

```
Imports System.Runtime.Remoting
Imports System.Runtime.Remoting.Channels
Imports System.Runtime.Remoting.Channels.Tcp

Public Class Form1
   Inherits System.Windows.Forms.Form

#Region " Windows Form Designer generated code "

   Dim sc As SimpleClass
   Private Sub DateTimeButton_Click(ByVal sender As System.Object, _
   ByVal e As System.EventArgs) Handles DateTimeButton.Click
      If Not Initialize() Then Return

      MsgBox(sc.GetDateTime(), MsgBoxStyle.DefaultButton1, _
         "GetDateTime()")
   End Sub

   Private Sub HelloButton_Click(ByVal sender As System.Object, _
   ByVal e As System.EventArgs) Handles HelloButton.Click
      If Not Initialize() Then Return

      MsgBox(sc.Hello(), MsgBoxStyle.DefaultButton1, "Hello()")
   End Sub

   Public Function Initialize() As Boolean
      If sc Is Nothing Then
         Dim chan As New TcpChannel()
         ChannelServices.RegisterChannel(chan)
         Dim t As Type = Type.GetType("Server.SimpleClass")
         sc = Activator.GetObject(t, _
            "tcp://localhost:9999/SimpleClass")

         If sc Is Nothing Then
            MsgBox("Could not initialize the remoting client. " & _
               "Check your configuration.")
            Return False
         End If
      End If

      Return True
   End Function
End Class
```

From this example, you can see how the client initializes the *SimpleClass* proxy by making the request of the *Activator* class. An important configuration requirement that is not obvious here is that both the client class and the server class must reside in the same namespace. That is why both the server code and the client code refer to *Server.SimpleClass*. Both projects have a copy of the class, and their namespaces match. This allows the client to know the fully qualified type of the *SimpleClass* that is also known to the server process. The application itself should look like Figure 21-10.



SimpleClass registered.
Remoting running...

10/14/2001 4:12:10 PM

Figure 21-10 Simple Remoting sample in action.

While this example will work, it is not the most desirable way to go about implementing remoting in an application—especially in a larger-scale application. This observation leads us to a discussion of the best architecture for an application that uses remoting.

Architecture for Remoting

The .NET Framework SDK has many decent remoting examples, but they are not in-depth enough to help you decide on an implementation strategy. Calling remote objects is by definition an expensive process, requiring you to think very carefully about how a remote object is used. Often when we use objects in our applications, we don't worry about whether we are using properties or methods, and we are not usually concerned with when and how often we manipulate those objects. In a distributed application, however, these are extremely important considerations. With XML Web services, it is possible to make methods available selectively. With remoting, it is not quite so simple. You are working with a proxy for the original class, and by definition all public methods are available to the client process. However, there is a way around this issue: interfaces.

Interfaces are an important way to implement a class for remoting. Programming against a limited interface prevents unintentional use of an object. You can do this by creating a separate assembly that contains the public interfaces intended only for use by remoting clients. Figure 21-11 illustrates this concept.

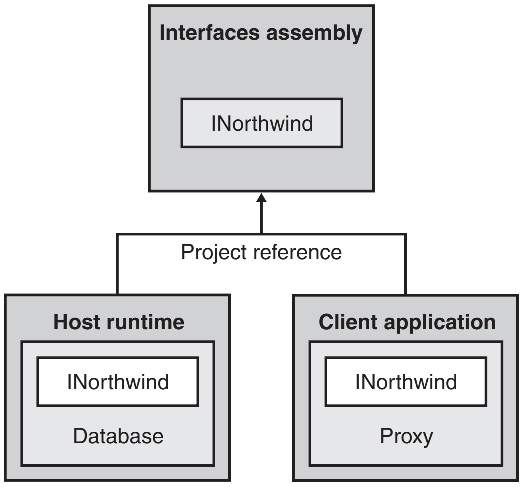


Figure 21-11. Suggested remoting architecture.

Programming against interfaces is important for larger-scale applications because it allows you to further control which methods are exposed to down-level clients. This technique gives you a greater level of control than just access protection for methods on any given class. We've provided another sample program on the companion CD, called Architecting for Remoting, that demonstrates how to use this technique. There are three main components to the application, as outlined in Figure 21-11: the *INorthwindDatabase* interface, the *Database* class (which implements the *INorthwindDatabase* interface), and the *MyClient* project. The critical difference between this example and the previous one is that the client application has awareness only of the *INorthwindDatabase* interface. It does not have a copy of the target class and does not know how the class is implemented. Here is the interface we are talking about:

```
Public Interface INorthwindDatabase
   ReadOnly Property Categories() As DataTable
   ReadOnly Property Customers() As DataTable
   ReadOnly Property Employees() As DataTable
   ReadOnly Property OrderDetails() As DataTable
   ReadOnly Property Orders() As DataTable
   ReadOnly Property Products() As DataTable
   ReadOnly Property Shippers() As DataTable
   ReadOnly Property Suppliers() As DataTable
End Interface
```

The next code example is the actual database class that was created to implement the *INorthwindDatabase* interface. Notice that the class contains other public methods that do not exist in the interface. As far as the class is concerned, these are utility methods that the client should not be able to call (the *ExecuteQuery* method, for example).

Requiring the client to use the interface, instead of the actual class definition, has two effects. First, it prevents the client from using methods it should not have access to. Second, it allows you to create other database classes based on the same interface and use them all interchangeably without having to make any modifications to the client application. The implementation class, in this example database, is derived from *ServicedComponent* instead of directly from *MarshalByRefObject*, for the sake of demonstrating how easy it is to support COM+ and remoting simultaneously.

```
Imports MyInterfaces
Imports System.Data
Imports System.Data.OleDb
Imports System.EnterpriseServices

' The database class containing the implementation for the
' INorthwindDatabase interface
Public Class Database
   Inherits ServicedComponent
   Implements MyInterfaces.INorthwindDatabase

   Dim conn As OleDbConnection
   Dim cmd As OleDbCommand
   Dim da As OleDbDataAdapter

   Public Sub New()
      conn = New OleDbConnection( _
         "Provider=Microsoft.Jet.OLEDB.4.0;" &
         "Data Source=C:\Northwind.mdb")
      cmd = New OleDbCommand("", conn)
      da = New OleDbDataAdapter(cmd)
   End Sub

   Private Function ExecuteQuery(ByVal query As String) As DataTable
      Dim t As New DataTable()
      conn.Open()
      cmd.CommandText = query
      da.Fill(t)
      conn.Close()
      Return t
   End Function

   Public ReadOnly Property Categories() As DataTable _
         Implements INorthwindDatabase.Categories
      Get
         Return ExecuteQuery("Select * From Categories")
      End Get
   End Property
   Public ReadOnly Property Customers() As DataTable _
         Implements INorthwindDatabase.Customers
      Get
         Return ExecuteQuery("Select * From Customers")
      End Get
   End Property

   Public ReadOnly Property Employees() As DataTable _
         Implements INorthwindDatabase.Employees
      Get
         Return ExecuteQuery("Select * From Employees")
      End Get
   End Property

   Public ReadOnly Property OrderDetails() As DataTable _
         Implements INorthwindDatabase.OrderDetails
      Get
         Return ExecuteQuery("Select * From [Order Details]")
      End Get
   End Property

   Public ReadOnly Property Orders() As DataTable _
         Implements INorthwindDatabase.Orders
      Get
         Return ExecuteQuery("Select * From Orders")
      End Get
   End Property

   Public ReadOnly Property Products() As DataTable _
         Implements INorthwindDatabase.Products
      Get
         Return ExecuteQuery("Select * From Products")
      End Get
   End Property

   Public ReadOnly Property Shippers() As DataTable _
         Implements INorthwindDatabase.Shippers
      Get
         Return ExecuteQuery("Select * From Shippers")
      End Get
   End Property

   Public ReadOnly Property Suppliers() As DataTable _
         Implements INorthwindDatabase.Suppliers
      Get
         Return ExecuteQuery("Select * From Suppliers")
      End Get
   End Property
End Class
```

The server process is not very interesting. It registers the *ServerProcess.Database* class with the runtime and then sits there. The code is really no different from the server project in the Simple Remoting example. The client, on the other hand, is quite different. The client requests an object from the remoting channel that is based solely on the *INorthwindDatabase* interface. It has no reference to or knowledge of the underlying *Database* class and thus cannot call any methods that are not exposed by the *INorthwindDatabase* interface. Here is what the code looks like:

```
Imports System.Runtime.Remoting
Imports System.Runtime.Remoting.Channels
Imports System.Runtime.Remoting.Channels.Tcp

Public Class Form1
   Inherits System.Windows.Forms.Form

#Region " Windows Form Designer generated code "

   Dim northwindDB As MyInterfaces.INorthwindDatabase
   Public Function Initialize() As Boolean
      If northwindDB Is Nothing Then
         Dim chan As New TcpChannel()
         ChannelServices.RegisterChannel(chan)
         Dim t As Type = Type.GetType( _
            "MyInterfaces.INorthwindDatabase,MyInterfaces")
         northwindDB = Activator.GetObject(t, _
            "tcp://localhost:8086/NorthwindDB")

         If northwindDB Is Nothing Then
            MsgBox("Could not initialize the remoting client." & _
               "Check your configuration.")
            Return False
         End If
      End If

      Return True
   End Function

   Private Sub CustomersButton_Click(ByVal sender As System.Object, _
         ByVal e As System.EventArgs) Handles CustomersButton.Click
      If Not Initialize() Then Return
      QueryDataGrid.DataSource = northwindDB.Customers
   End Sub

   Private Sub EmployeesButton_Click(ByVal sender As System.Object, _
         ByVal e As System.EventArgs) Handles EmployeesButton.Click
      If Not Initialize() Then Return
      QueryDataGrid.DataSource = northwindDB.Employees
   End Sub
   Private Sub CategoriesButton_Click(ByVal sender As System.Object, _
         ByVal e As System.EventArgs) Handles CategoriesButton.Click
      If Not Initialize() Then Return
      QueryDataGrid.DataSource = northwindDB.Categories
   End Sub

   Private Sub OrderDetailsButton_Click _
         (ByVal sender As System.Object, _
         ByVal e As System.EventArgs) Handles OrderDetailsButton.Click
      If Not Initialize() Then Return
      QueryDataGrid.DataSource = northwindDB.OrderDetails
   End Sub

End Class
```

Distributed COM+ Applications

Probably one of the most important uses for COM+ is in distributed applications. Creating application proxies was always one of the easiest ways to implement a distributed application. The number of choices in .NET might seem somewhat daunting by comparison. Not too worry, though. The increased choices are all designed to address certain deficiencies of the WinDNA platform and technologies such as DCOM and RDS and to make it easier to create distributed COM+ applications.

COM+ and Remoting

The process of creating server applications that run through remoting is fairly straightforward, if not obvious. Remember that remoting requires a host process to handle registration and configuration. This is still true for COM+ applications. Granted, you are already halfway there because any component derived from *ServicedComponent* has *MarshalByRefObject* as a base class—a prerequisite for all objects to be made available through remoting. The Architecting for Remoting example demonstrates how to register a *ServicedComponent* with the common language runtime. Support for the COM+ context and other services is implicit through remoting, and there really isn't anything extra that you need to do for it to work.

Using SOAP Services

Publishing a *ServicedComponent* through Remoting is powerful, but it requires you to do your own work to register the individual components that are intended to be available. There is an alternative: SOAP. The self-registration process in .NET for *ServicedComponent*s provides the ability to mark your entire COM+ application as publishable through SOAP Services (essentially an XML Web service interface). Your COM+ application is then published through a virtual root of the IIS server the application is registered on. This technique is powerful, but it has greater implicit calling overhead than a binary format such as Remoting. On the other hand, using SOAP Services allows you to publish your COM+ application in a simple and straightforward manner. Recall from Chapter 16 how important attributes are to COM+ applications. The SOAP Service is made accessible through the *ApplicationActivation* assembly attribute. There is a default property that you can set in this attribute that instructs the self-registration process to create a SOAP proxy for your application (and any *ServicedComponent* contained within the Assembly). This property allows these *ServicedComponent*s to be activated through a Web Service.

Limitations of SOAP Services

While using SOAP Services is a painless way to publish your COM+ application as an XML Web service, it does lose some of its abilities. For example, the COM+ transaction context ends at the XML Web service boundary. This means that you cannot have a transactional client call the XML Web service and have the transaction context propagate from the client to the server. COM+ essentially begins and ends at the XML Web service boundary. Although your COM+ application can take advantage of all of the COM+ Services available, those services are not available to any calling clients. As far as any client is concerned, there is nothing COM+ about the XML Web service.

Included on the CD is the Soap Services example. It contains a simple class, SoapServicesTest, that defines the COM+ application.

```
Imports System.EnterpriseServices
Imports System.Reflection

<Assembly: AssemblyKeyFile("SoapServicesTest.snk")>
<Assembly: ApplicationName("COM+ SOAP Services Test")>
<Assembly: ApplicationActivation(ActivationOption.Server,
SoapVRoot:="SOAPServicesTest")>

Public Class SoapServicesTest
   Inherits ServicedComponent
   Public Sub New()
      MyBase.New()
      Debug.WriteLine("SOAPServicesTest::New()")
   End Sub

   Public Function HelloWorld() As String
      Return "Hello World"
   End Function
End Class
```

When this class is registered (by using RegSvcs.exe or loading it via another process), the COM+ self-registration process will create the IIS virtual directory and build all the necessary support files for an XML Web service, including the inspection page (accessible through a Web browser).

Note Windows XP will require Service Pack 1 before the included sample application will function as expected. At the time of this writing, the SOAP Services attribute is not supported for Windows 2000. Support exists only for Windows XP and the Windows .NET server product family.

COM+ Application Proxies in .NET

Using COM+ application proxies has always been an easy way to implement DCOM in your applications. Such proxies enable you to create remote components that you can deploy and program against as though the components resided on the local machine. COM+ allows you to register your component on a server in a COM+ application and generate an export package that can be registered on virtually any client (domain security issues aside). As a result, you can avoid the whole issue of creating COM proxy stub packages and implementing custom object marshaling—COM+ does it all for you.

You will be relieved to discover that nothing has changed in Visual Basic .NET. After a *ServicedComponent* is registered on the server, it is possible to create an export package, deploy it to the client, and use it as you always have. You will need the assembly containing the class and interface definitions on the client (unless you want to everything to be late bound), but COM+ will still marshal the objects through DCOM for you. There really isn't a lot more that needs to be said here. It just works. Create your export package, register it on the client, and you're done.

Note There is a caveat to the "just works" mantra for COM+ proxy applications. It turns out that there are some platform requirements for this to work correctly. Because of problems in Windows 2000, support for .NET-based application proxies will not be available until Service Pack 3 has been released. At the time of this writing, Service Pack 2 is the most recent available. Application proxies will work just fine on Windows XP and the Windows .NET server product family.

Conclusion

At this point you should have at least a basic understanding of how to implement the core distributed technologies and where the relative strengths of remoting and XML Web services lie. Traditionally, developing distributed applications has required a lot of work on the part of the developer. Microsoft started to make things a great deal easier with the introduction of Microsoft Transaction Services (MTS) and RDS. The .NET platform takes the process to a whole new level. With objects that support the innate capability of serialization and protocol abstraction, you are free to concentrate more on the architecture and design of your applications rather than spend time on network and marshaling issues.

Moving forward, you need to think carefully about what type of solution will fit well with your application. It is possible, and sometimes desirable, to maintain some components in Visual Basic 6 rather than moving the entire application to Visual Basic .NET. Don't feel pressured to make unnecessary changes just to be "pure" .NET. Visual Basic .NET provides the groundwork for developing whole new classes of distributed applications, but this need not be at the expense of your existing applications.