CHAPTER 18

Using COM+ Transactions

In This Chapter

Reviewing the COM+ transaction architecture

Meeting the ACID test

Constructing COM+ transactions

Building a test program

In this chapter, I'll show you how to create a COM+ transaction. These transactions can be used to implement an *n*-tier application system, which can offer significant performance improvements when compared with more traditional client/server applications.

A Brief Overview of COM+

Writing a COM+ program isn't difficult. If you understand how to create and use a Class module, then writing a COM+ transaction isn't much more difficult than that. However, to write an effective COM+ application means that you need to know what COM+ is and how it works.

Multi-tier applications

In a traditional client/server application, the client computer communicates directly with the database server (see Figure 18-1). This is also known as a 2-tier application program.

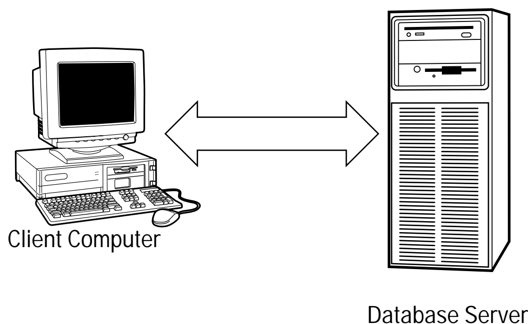

Figure 18-1: A 2-tier application

By using COM+ transactions, you can implement a 3-tier solution. In practice, you have a computer for the client, a second one for the COM+ transaction server, and a third one for the database server (see Figure 18-2). Note that the transaction server generally sits between the application client and the database server.

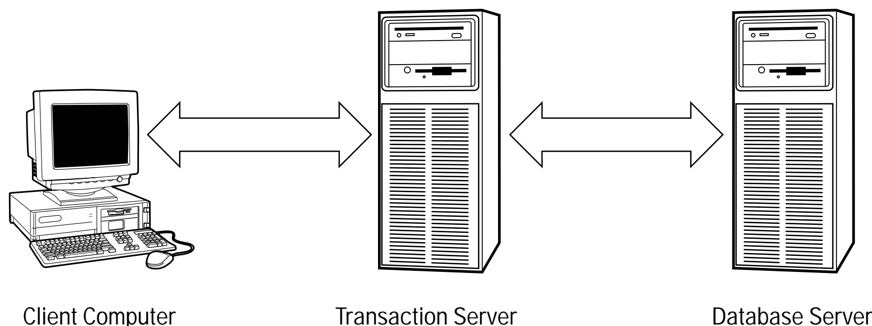

Figure 18-2: A 3-tier application

The transaction server is a place where you can move your application logic away from the client program yet keep it independent of the database server. This allows more flexibility when designing your application, because you can provide an object-oriented view of your data, while making available facilities that allow you to share resources for better performance and permit transaction support for better reliability.

A more interesting solution is shown in Figure 18-3. While I've labeled it a 4-tier processing solution, most people consider it just another variation of a 3-tier solution, because they feel that the client computer running a Web browser is not a true tier. However, given how easy it is to add VBScript or JavaScript to the Web page to perform basic functions like data validation, why not consider it a true tier?

In general, you can refer to multi-tier solutions as *n*-tier, where *n* refers to the number of levels of computers used in the application design. Note that I don't simply count the number of computers, because it is quite possible to have more than one Web server, more than one transaction server or more than one database server that provides the same basic services, but are implemented on multiple computers to better handle the workload. With *n*-tier solutions, each level of computers provides a new type of service. In practice, however, each new level of computers introduces additional overhead and beyond the four computers shown in Figure 18-3, it is doubtful that you would gain any additional benefit from adding a new tier.

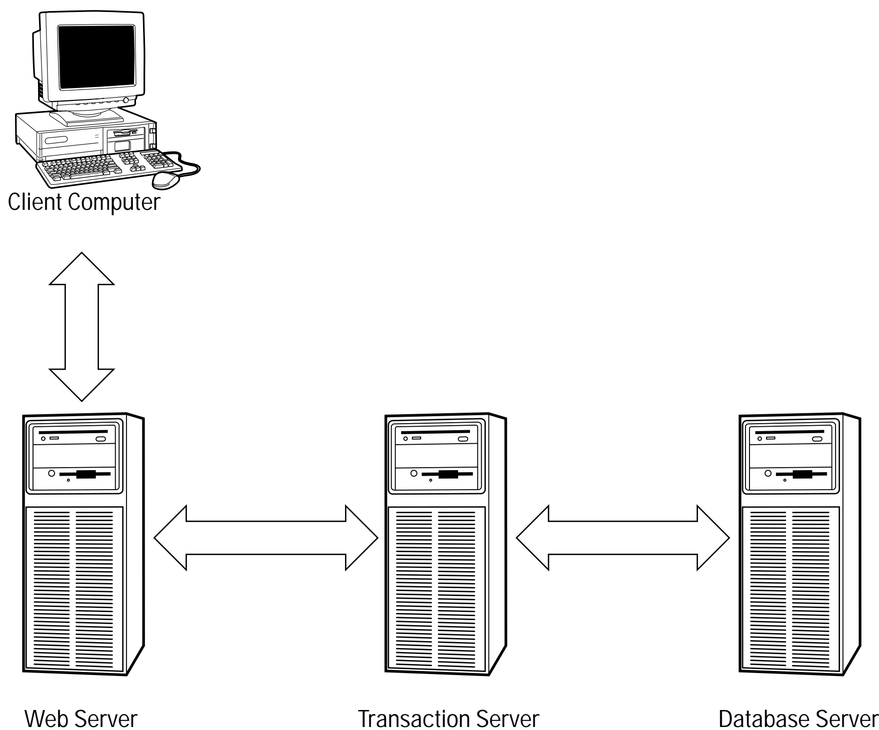

Figure 18-3: A 4-tier application

Transaction Servers

The transaction server receives requests from the program running on the client and starts a transaction to process it. If information is needed from the database server, the transaction will communicate with the database server. When the transaction has finished processing, it will return back to the calling program.

Adding the third tier, you offload some of the work from the database server, which allows the database server to perform additional work on the same hardware configuration. It also simplifies updating the application, since it is possible to make changes to a transaction on the transaction server without changing the client program that calls the transaction.

In a 4-tier solution, using a transaction server makes even more sense. It allows you to offload work from both the Web server and the database server onto a separate computer system, which serves to improve response time on both machines. This allows you to increase the available memory for Web page caching and database record caching.

Typically, the database server, transaction server, and Web server (if present) are all connected via a very high-speed connection. At least a 100 MHz Ethernet, if not a gigahertz Ethernet or other specialized networking technology, is used to connect these computers together. Often this network is isolated from the other networks to improve security and performance. This means that there are very few delays caused by the network. Yet by spreading the work around, you can ensure that no one system is overloaded. You can even add additional Web servers and transaction servers transparently if you really need them without requiring changes to your application.

COM+ applications

A COM+ transaction is basically a COM object that has been coded and compiled to run under the COM+ transaction server. In Visual Basic, you create a COM+ transaction by building an ActiveX DLL application. You need to include a few special objects and add a little code to perform some handshaking with the transaction server and use a special tool to define your transaction to the transaction server. But other than that, you're free to take advantage of the Visual Basic language.

A COM+ application is just a collection of one or more COM+ transactions that are grouped together into a single ActiveX DLL file. There are four basic types of COM+ applications:

- **Server applications:** A server application is the more common form of a COM+ application. It runs under control of the transaction server. The application can interact with the transaction server through the objects associated with the transaction's context.
- **Library applications:** A library application is a specialized form of a COM+ application that runs locally on the client computer. It runs in the same address space as the client program that created it. This means less overhead to the client program, because several COM+ applications can use the same components. However, because this object isn't running under the control of the transaction server, it can't take advantage of some of the features of the transaction server.
- **Application proxies:** An application proxy contains the registration information necessary for an application to remotely access a server application. If you run it on the client computer, all of the information necessary to access a specific remote server application will be installed on the client computer.
- **COM+ preinstalled applications:** COM+ comes with a set of applications that help you configure and administer your COM+ system. These applications include COM+ System Administration, COM+ Utilities, and IIS System Applications.

Of these types of COM+ applications, I'm going to focus on server applications, since they are the most useful of all.

The COM+ transaction server

The COM+ Transaction Server provides a framework to execute transactions on an *n*-tier application system. Included with the Server are facilities that allow you to share resources, such as ASO `Connection` objects, and provide additional security on your transaction.

You can only run the COM+ Transaction Server under Windows 2000 Server. You can't run it under Windows 9x, Windows NT, or even Windows 2000 Professional. It relies on facilities, such as the Active Directory, to manage much of the information it needs to run your transactions. Because the COM+ Transaction Server represents only one tier in an application, you can use any database server you choose, even if it runs under a different operating system such as Unix. The only restriction is that the database server must be supported by ADO.

Note

Serving transactions under Windows NT: If you want to run transactions under Windows NT Server, you need to use a tool called Microsoft Transaction Server, usually called MTS. Many of the basic facilities in MTS were upgraded to create COM+, but COM+ is not upwards compatible with MTS. COM+ requires less code to interact with the transaction server and is easier to administer than MTS.

The object context

Every COM+ transaction that is run under control of the COM+ Transaction Server is associated with a set of objects known as a *context*. A context represents the smallest possible unit of work for a component. Each instance of a COM object is assigned to a context. Multiple objects can use the same context depending on how they were defined and created.

The object context contains information about how the object was created and the status of the work that is currently active. You can access these facilities through the `ObjectContext` object. All of the other objects in the context can be referenced from this object either directly or indirectly.

The Component Services utility

The Component Services utility is used to manage and configure COM+ transactions (see Figure 18-4). You can start it in Windows 2000 by choosing Programs ➪ Administrative Tools ➪ Component Services from the Start button. This tool will be used frequently while testing your application to change the characteristics of the transaction, as well as to install updated copies of the COM component that makes up your COM+ application.

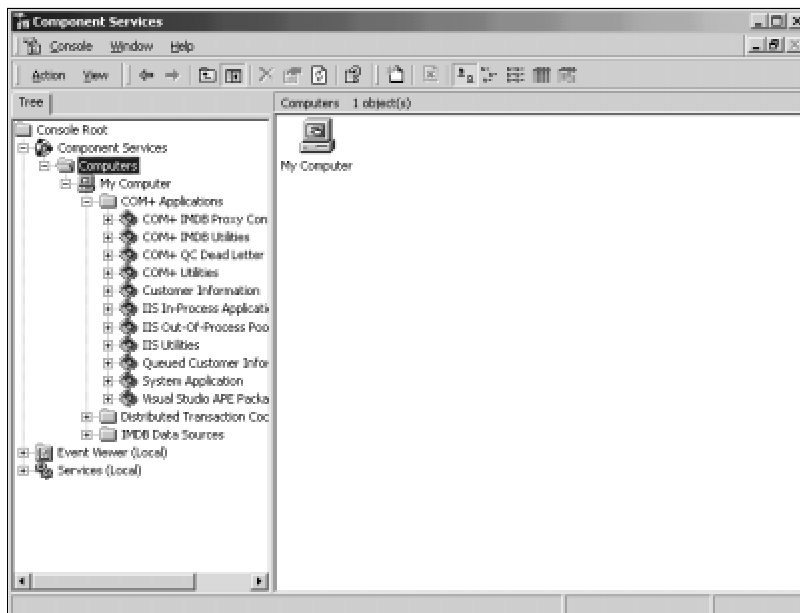

Figure 18-4: Running the Component Services utility

Introducing COM+ Transactions

COM+ transactions are a feature of Windows 2000 that you can use when building scalable, *n*-tier applications. COM+ is built on top of COM and integrates features such as the Microsoft Transaction Server and Microsoft Message Queues into a single technology.

COM+ is built using C++ and some of its features can only be accessed from C++. However, this doesn't mean that you can't use COM+ from Visual Basic 6. In fact, many of the features are actually very easy to use, once you understand how to work with them. Because this book is aimed at Visual Basic programmers, I'll focus on the features you can exploit today and leave the rest for the C++ programmers.

Note

A view of things to come: As I write this, there are many rumors surrounding the features that will be present in Visual Basic 7. While it isn't clear exactly what features will be present, you can bet that support for COM+ will be at the top of the list.

In Chapter 16, I talked about how to use the `BeginTransaction`, `CommitTransaction`, and `RollbackTransaction` methods to define a transaction over a database connection. These transactions represented a fundamental unit of work for the application.

They could contain the database calls necessary to make a withdrawal from your savings account, register you for a college course, or even order a book over the Internet.

Simply looking at a transaction from a database perspective may be a bit too limiting for many applications. In addition to manipulating data in the database, transactions often perform validation along with the database activities. For example, before making a withdrawal from your checking account, you must first verify that the account number is valid, and second, that there are sufficient funds in the checking account before subtracting the money from that account.

There are a several ways to implement a transaction. The most common way is to write complex stored procedures that run on the database server. This approach imposes extra work on the database server, which may detract from the database server's overall performance. In Windows 2000, you have the option of building COM+ transactions that run under control of a transaction server, which you can easily run on a separate machine.

The ACID test

Each COM+ transaction must meet the ACID test. The letters in ACID stand for Atomicity, Consistency, Isolation, and Durability, and serve to identify the different criteria that a transaction must meet.

Atomicity

Atomicity means that either all or none of a transaction is completed. In other words, the work a transaction performs can't be subdivided. If the transaction completes successfully, all of the changes it makes will remain. If the transaction fails, then all of the changes it made must be undone.

Consider an airline reservations system. When you make a reservation, you specify the date, flight number, number of seats you want, plus a number of parameters such as first class or economy. You want the reservation to succeed only if all of these parameters are met. Otherwise you want the reservation to fail, so you can try again. Either all of the seats you requested are available or none of them.

Consistency

Another aspect of a transaction is that it must always leave the system in a consistent state. Consider what happens if you move money from your savings account to your checking account. You have to read the current balance from the savings account, subtract the amount of money you want to move, and update the account with the new balance. Then you have to read the current balance from the checking account, add the money to it, and update the account with the new balance.

Consider what happens if the process fails after the first update is finished, but before the second update is made. The database will automatically ensure that the first update is saved, but it will not save an incomplete update. This means that the money would be removed from your saving account but not added to your checking account. For all practical purposes, the money would be lost and the database left in an inconsistent state.

By placing the two updates into a transaction, you can ensure that the database is left in a consistent state. Either both updates are completed or both updates are not completed. In this example, whether the transaction succeeds or fails, the money won't be lost.

Isolation

While not as obvious as the previous characteristics of a transaction, isolation is also very important. Each transaction lives in its own world, where it can't see any of the processing performed by another active transaction. The concept of isolation is important because it allows the system to recover from failures. While the system is actively processing transactions, you may have dozens or even hundreds of active transactions at any point in time. However, with transaction isolation, these transactions can be viewed as a single stream of transactions, where one transaction is completed before the next one begins. This process allows recovery programs to work, and makes it possible to implement distributed database systems.

Durability

The last characteristic of a transaction is its ability to survive system failures. Once a transaction has been completed, it is critical that the changes it made aren't lost. For this reason, it's important that the system keep logs and backups of all the transactions that were processed. These files make it possible for work to be recovered in case of a catastrophic system failure.

Class module properties for transactions

In order to identify a Visual Basic class module as a COM+ transaction, you need to adjust some of its properties. These properties control how Visual Basic will compile your program.

In order to run a transaction under control of COM+ you need to specify the value of the `MTSTransactionMode` property in the class module. While this value can be overridden by the Component Services utility, you should choose the proper value before you try to install it under COM+. Table 18-1 lists the Visual Basic constant that you can set using the Properties window for the class module.

Table 18-1
MTSTransactionMode Values

Visual Basic Constant	*COM+ Transaction Support*	*Numeric Value*
`NotAnMTSObject`	Disabled	0
`NoTransactions`	Not Supported	1
`RequiresTransaction`	Required	2
`UsesTransaction`	Supported	3
`RequiresNewTransaction`	Requires New	4

- **NotAnMTSObject** implies that the object isn't supported under COM+ Transaction Server. You should use this value when you do not expect to run your component under control of the Transaction Server.
- **NoTransactions** means that the object doesn't support transactions. When a new instance of the object is created, its object context is created without a transaction.
- **RequiresTransaction** means that the component's objects must execute without the scope of a transaction. When a new instance of the object is created, the object context is inherited from the object context of the client. If the client doesn't have a transaction, a new transaction is created for the object. If the client is a COM+ transaction, then a new transaction will not be started because one is already active.
- **UsesTransaction** means that the component's objects can execute within the scope of a transaction. When a new instance of the object is created, its object context is inherited from the object context of the client. If the client doesn't have a transaction, the new object will run without a transaction.
- **RequiresNewTransaction** means that the component's objects must execute within their own transactions. When a new instance of the object is created, a new transaction will automatically be created for the object, even if the calling object is already executing within a transaction.

The ObjectContext object

The `ObjectContext` object is the most important object when implementing a COM+ transaction. Through this object, you will control how the transaction behaves. You can use this object to communicate with the transaction server and find out information about the environment in which it is running.

You can get access to this object by selecting the COM+ Services Type Library from the References window by choosing Project ÿ References from the Visual Basic main menu (see Figure 18-5) and then declaring and calling the `GetObjectContext` function, as shown below:

```
Dim MyContext As ObjectContext
Set MyContext = GetObjectContext()
```

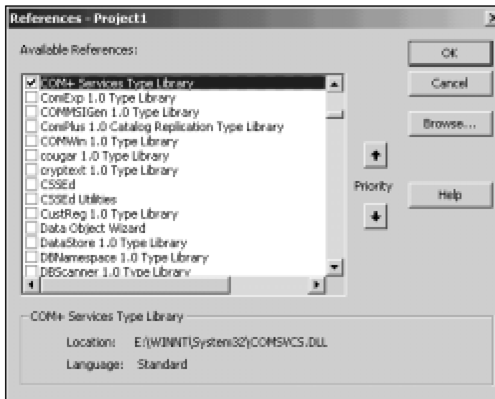

Figure 18-5: Selecting the COM+ Services Type Library

ObjectContext object properties

Table 18-2 lists the properties of the `ObjectContext` object.

Table 18-2
Properties of the ObjectContext Object

Property	*Description*
`ContextInfo`	An object reference to the `ContextInfo` object.
`Count`	A `Long` value containing the number of property objects.
`Item`	An object reference to a specific property object.
`Security`	An object reference to the `SecurityProperty` object.

ObjectContext object methods

Of the methods available for the `ObjectContext` object, the most important are `SetAbort` and `SetComplete`. These instruct the transaction server whether to allow the transaction's activities to be saved or undone.

Function CreateInstance (bstrProgID As String) As Variant

The `CreateInstance` method returns an object reference to a new instance of the specified object using the current object's context. This function should be used in place of the `CreateObject` function, because it implements the current context for its execution. If the object isn't registered with COM+, the object is merely created and no context will be assigned. If the object is registered with COM+, it will be created according to the COM+ Transaction Support value and the `MTSTransactionMode` property. `bstrProgID` is a `String` value containing the name of the object to be created.

Sub DisableCommit()

The `DisableCommit` method tells the transaction server that the object's work has left the system in an inconsistent state.

Sub EnableCommit()

The `EnableCommit` method tells the transaction server that the object's work may not be complete, but that the system is now in a consistent state.

Function IsCallerInRole(bstrRole As String) As Boolean

The `IsCallerInRole` method returns `True` when the object's direct caller is in the specified role, either individually or as part of a group. `bstrRole` is a `String` value containing the security role.

Function IsInTransaction() As Boolean

The `IsInTransaction` method is `True` when the object is executing inside a transaction.

Function IsSecurityEnabled () As Boolean

The `IsSecurityEnabled` method is `True` when security is enabled.

Sub SetAbort()

The `SetAbort` method instructs the transaction server to undo any of the transaction's actions. This may be because an unrecoverable error has occurred or the system is in an inconsistent state.

Sub SetComplete()

The `SetComplete` method instructs the transaction server to commit all updates to the system because the transaction has completed its work successfully.

Constructing a COM+ Transaction

Constructing a COM+ transaction is fairly easy. You start by creating an ActiveX DLL project to hold the transaction. If you want to pass any objects between the transaction and the program, you will need a second ActiveX DLL to hold the type information. Finally, to test your transaction, you'll need a simple application that calls the transaction.

For this example, I'm going to use the Customers table and create COM+ transactions to retrieve a single customer and to update a single customer. The information will be passed back and forth between the client and the transaction server using the Customer object. To demonstrate the transactions, I'm also going to build a simple IIS Application and run it through the IIS Web server.

Holding type information

One problem you will encounter when you begin using COM+ transactions is that you can't rely on global variables to pass information back and forth to the routines you may call. Everything must be passed as a parameter. Depending on the data you want to share with a routine, this can be a problem.

Using Visual Basic's class modules, you can create an object that contains a lot of information in a single entity. When using some of the more advanced features in Windows, such as COM+ transactions and message queuing, this is my favorite way to pass information around.

The downside to using class modules to pass information around in a COM+ transaction is that you really have to deal with two different programs. The COM+ transaction operates independently of the application program using the transaction. Therefore, both programs need a copy of the definitions. The only reasonable way to handle this situation is to introduce a third code module that holds the type definitions for your transactions. This third module exists on both the client machine and the transaction server machine.

Creating storage for property values

Like any object, the Customer object consists of a series of `Public Property Get` and `Property Let` routines, which are used to access a bunch of `Private` variables defined at the module level. Listing 18-1 contains the list of `Private` variables. Note that each of the variable names begins with an `X`. This is because I wanted the object's user to see the field names I used in the database rather than use a cryptic abbreviation. Since I'm the only person who expects to see inside the Customer object, I don't consider this a real hardship.

Listing 18-1: The module level declarations in Customer

```
Option Explicit

Private XCustomerID As Long
Private XName As String
Private XStreet As String
Private XCity As String
Private XState As String
Private XZip As Long
Private XPhone As String
Private XEMailAddress As String
Private XDateAdded As Date
Private XDateUpdated As Date
Private XMailingList As Boolean
Private XComments As String
Private XIsDirty As Boolean
```

You should note that in addition to the various fields from the database, I also added one more value, called `XIsDirty`. The `XIsDirty` property is used to indicate when the data in the object has been changed. It's a quick and easy way to determine if the information was changed and the database should be updated.

Managing property values

The `Property Get` routine just returns the value of the corresponding `X` variable (see Listing 18-2), while the `Property Let` routine is a little more complicated (see Listing 18-3).

The `Property Let` routine determines if the value of the property is different than the value already in private storage. If the value is different, then I save the new value, set the `XIsDirty` property, and mark the property as changed. Otherwise, I ignore the assignment. While checking the assignment is a little extra work, it allows someone to reassign the current value to the property without resetting the `XIsDirty` flag.

Listing 18-2: The CustomerId Property Get routine in Customer

```
Public Property Get CustomerId() As Long

CustomerId = XCustomerID

End Property
```

Listing 18-3: The CustomerId Property Let routine in Customer

```
Public Property Let CustomerId(c As Long)

If c <> XCustomerID Then
   XCustomerID = c
   XIsDirty = True
   PropertyChanged "CustomerId"

End If

End Property
```

Property May I: The `CanPropertyChange` method isn't required when dealing with properties in the `Customer` object, since the `Customer` object can't act as a data consumer.

Initializing property values

I chose to make the Customers object a persistable object. This will have a big benefit later when I talk about message queuing in Chapter 19. However, for normal COM+ transactions, you don't really need persistent objects. Once the object is instantiated, it will remain instantiated until you destroy it.

Listing 18-4 contains the statements necessary to initialize the variables in local storage. Note that I just use reasonable default values for all of these properties except for `XDateAdded` and `XDateUpdated`. For these values, I use the function `Now` to save the current date and time into these variables. This makes it easier for someone to create a new Customer object and not worry about assigning values to these two fields. It also won't cause a problem when the COM+ transaction returns information from the database, since these initial values will be overlaid with the live values from the database.

Listing 18-4: The Class_InitProperties event in Customer

```
Private Sub Class_InitProperties()

XCustomerID = -1
XName = ""
XStreet = ""
XCity = ""
XState = ""
```

```
XZip = 0
XPhone = ""
XEMailAddress = ""
XDateAdded = Now
XDateUpdated = Now
XMailingList = False
XComments = ""
XIsDirty = False

End Sub
```

As you would expect, the `ReadProperties` and `WriteProperties` events are pretty simple, just a series of statement that use the `ReadProperty` and `WriteProperty` methods to restore and save these values. Listing 18-5 shows the `ReadProperties` event.

Listing 18-5: The Class_ReadProperties event in Customer

```
Private Sub Class_ReadProperties(PropBag As PropertyBag)

XCustomerID = PropBag.ReadProperty("CustomerId", 0)
XName = PropBag.ReadProperty("Name", "")
XStreet = PropBag.ReadProperty("Street", "")
XCity = PropBag.ReadProperty("City", "")
XState = PropBag.ReadProperty("State", "")
XZip = PropBag.ReadProperty("Zip", 0)
XPhone = PropBag.ReadProperty("Phone", "")
XEMailAddress = PropBag.ReadProperty("EMailAddress", "")
XDateAdded = PropBag.ReadProperty("DateAdded", 0)
XDateUpdated = PropBag.ReadProperty("DateUpdated", 0)
XMailingList = PropBag.ReadProperty("MailingList", False)
XComments = PropBag.ReadProperty("Comments", "")
XIsDirty = False

End Sub
```

Note that even though I assign the current date and time in the `InitProperties` event to the `DateAdded` and `DateUpdated` properties, I use a default value of zero. This just means that these properties will always be stored in the property bag, which is likely to happen anyway, given the nature of the data stored in the properties.

Installing the DLL

Once you build your program, you need to compile it to an ActiveX DLL (Dynamic Linking Library) and then register it in the Windows Registry. When you register your DLL, selected information about the DLL will be loaded into the Registry, including the location of the DLL file, the name of the objects available, and the Globally Unique Identifiers (GUID) that are used to locate them.

When your program references an object, it presents the object's GUID to Windows, which in turn uses it as a key to locate information about the object in the Registry. This means that the actual location of the file can be independent of the application.

To register your file, you need to use the RegSvr32 utility program. To run it, choose Start ➪ Run from the Windows taskbar. Then enter the `RegSvr32` followed by the fully qualified path name to the DLL file, as shown below:

```
RegSvr32 d:\VB6DB\Chapter18\CustomerObjects\cust.dll
```

and press Enter. The registration program will run for a split second and display a message box saying that RegisterDLL Server succeeded.

If you need to make a change to the DLL after you have registered it, you must run the following command:

```
RegSvr32 /u d:\VB6DB\Chapter18\CustomerObjects\cust.dll
```

It will acknowledge the request by saying DLLUnregister Server succeeded.

DOS ain't dead: After using DOS and many other character operating systems over the years, I actually like typing my commands and seeing the results. I often use DOS to display directory information and to copy files, and as a result, I usually have a DOS window open. So rather than choosing Start ➪ Run, I usually just toggle to my DOS window to run the RegSvr32 command. Besides, if the DLL is in the current directory, I need only type the file name rather than the fully qualified path name.

Accessing the database with transactions

Now that I have an object I can pass back and forth to COM+ transactions, I want to build some transactions. For all practical purposes, building a COM+ transaction is just like building any other COM component that accesses a database — with two exceptions. First, you need to use the `ObjectContext` object to inform the transaction server of the transaction's status. Second, you need to grab and release database resources, such as the Connection object, quickly.

Getting customer information

The `GetCustomer` method (see Listing 18-6) retrieves the customer information for the customer specified in the `CustomerId` parameter and returns it to the calling program using the Customer object. While a method that strictly reads information from the database doesn't benefit from the COM+ transaction server's ability to manage complex transactions, it does allow this routine to operate more efficiently than it would if you embedded the code directly in your application program.

Listing 18-6: The GetCustomer method of CustomerInfo

```
Public Function GetCustomer(CustomerId As Long) As Customer

Dim c As Customer
Dim cmd As ADODB.Command
Dim db As ADODB.Connection
Dim o As ObjectContext
Dim parm As ADODB.Parameter
Dim rs As ADODB.Recordset

Set o = GetObjectContext()

Err.Clear
Set db = New ADODB.Connection

db.Provider = "sqloledb"
db.ConnectionString = "Athena"
db.CursorLocation = adUseNone
db.Open , "sa", ""
db.DefaultDatabase = "VB6DB"

Set cmd = New ADODB.Command
Set cmd.ActiveConnection = db
cmd.CommandText = "Select * from Customers " & _
   "Where CustomerId = ?"
Set parm = cmd.CreateParameter("CustomerId", adInteger, _
      adParamInput, 4)
cmd.Parameters.Append parm

Err.Clear
cmd.Parameters("CustomerId").Value = CustomerId
Set rs = cmd.Execute
If Err.Number <> 0 Then
   App.LogEvent "CustomerInfo(GetCustomer): " & _
      "Can't retrieve the " & _
      "record for CustomerId " & _
      FormatNumber(CustomerId, 0) & ". Error: " & _
      Err.Description & "-" & _
      Hex(Err.Number)
```

Continued

Listing 18-6 *(continued)*

```
   Set c = Nothing

Else
   Set c = New Customer
   c.CustomerId = rs.Fields("CustomerId")
   c.Name = rs.Fields("Name")
   c.Street = rs.Fields("Street")
   c.City = rs.Fields("City")
   c.State = rs.Fields("State")
   c.Zip = rs.Fields("Zip")
   c.Phone = rs.Fields("Phone")
   c.EMailAddress = rs.Fields("EMailAddress")
   c.DateAdded = rs.Fields("DateAdded")
   c.DateUpdated = rs.Fields("DateUpdated")
   c.MailingList = rs.Fields("MailingList")
   c.Comments = rs.Fields("Comments")

End If

Set GetCustomer = c
rs.Close
db.Close
Set rs = Nothing
Set db = Nothing
Set c = Nothing
Set o = Nothing

End Function
```

The `GetCustomer` method begins by declaring a number of objects that I'll use in this program. Next, I'll get an object reference to the `ObjectContext` for this transaction. While I don't really need it, it isn't a bad idea to get in the habit of starting every method by getting the context.

Next, I establish a connection to the database. Rather than creating a connection once and using it for all transactions that this object might execute, it is more efficient to get a connection, use it, and then release it as quickly as possible. The COM+ transaction server intercepts your call and satisfies your request with an existing connection from a pool of connections it maintains. When you release the connection, the COM+ transaction server adds it back to the pool and makes it available for someone else to use. By sharing connections in this fashion, both your application and the database server have a lot less work to do.

After the connection is opened, I create a Command object with a parameterized **Select** statement and its associated Parameter object. I chose to use the Command object rather than a Recordset because the Command object will be more efficient in the long run. SQL Server 7 will parse the query the first time it sees it, and the next time I use the query, it will be able to use the already parsed version. This makes it nearly as fast as a stored procedure, without the headaches of creating a stored procedure.

When I execute the Command, it will return a Recordset object containing the value I requested, in which case I'll save the values into a Customer object and return it as the value of the function. If executing the Command generated an error, I'll write the error to the application's log file and return Nothing as the value of the function.

At the end of the function, I close the Recordset and Connection objects and destroy all of the objects I used in the routine. Note that I destroy the ObjectContext object, since it is no longer necessary. A call to SetComplete or SetAbort isn't necessary since I didn't modify any data.

Saving customer information

The PutCustomer method shown in Listing 18-7 follows the same basic logic flow as the GetCustomer method you saw in 18-6. I begin by acquiring a connection to the database. Then I create a Command object that executes an SQL **Update** statement. Next I assign the appropriate values from the Customer object to the parameters in the command and then execute the **Update** statement. Finally, I return any error information as the value of the function, signal the ObjectContext that the transaction is complete and destroy the objects I used.

Listing 18-7: The PutCustomer method of CustomerInfo

```
Public Function PutCustomer(c As Customer) As Long

On Error Resume Next

Dim cmd As ADODB.Command
Dim db As ADODB.Connection
Dim o As ObjectContext
Dim parm As ADODB.Parameter

Set o = GetObjectContext()

PutCustomer = 0
Err.Clear
```

Continued

Listing 18-7 *(continued)*

```
Set db = New ADODB.Connection
db.Provider = "sqloledb"
db.ConnectionString = "Athena"
db.CursorLocation = adUseNone
db.Open , "sa", ""
db.DefaultDatabase = "VB6DB"

Set cmd = New ADODB.Command
Set cmd.ActiveConnection = db
cmd.CommandText = "Update Customers Set Name=?, Street=?, " _
      "City=?, State=?, " & _
      "Zip=?, Phone=?, EMailAddress=?, DateAdded=?, " & _
      "DateUpdated=?, MailingList=?, Comments=? " & _
      "Where CustomerId=?"

Set parm = cmd.CreateParameter("Name", adVarChar, _
      adParamInput, 64)
cmd.Parameters.Append parm

Set parm = cmd.CreateParameter("Street", adVarChar, _
      adParamInput, 64)
cmd.Parameters.Append parm

Set parm = cmd.CreateParameter("City", adVarChar, _
      adParamInput, 64)
cmd.Parameters.Append parm

Set parm = cmd.CreateParameter("State", adChar, _
      adParamInput, 2)
cmd.Parameters.Append parm

Set parm = cmd.CreateParameter("Zip", adInteger, _
      adParamInput, 4)
cmd.Parameters.Append parm

Set parm = cmd.CreateParameter("Phone", adVarChar, _
      adParamInput, 32)
cmd.Parameters.Append parm

Set parm = cmd.CreateParameter("EMailAddress", adVarChar, _
      adParamInput, 128)
cmd.Parameters.Append parm

Set parm = cmd.CreateParameter("DateAdded", adDBDate, _
      adParamInput)
cmd.Parameters.Append parm
```

```
Set parm = cmd.CreateParameter("DateUpdated", adDBDate, _
      adParamInput)
cmd.Parameters.Append parm

Set parm = cmd.CreateParameter("MailingList", adBoolean, _
      adParamInput)
cmd.Parameters.Append parm

Set parm = cmd.CreateParameter("Comments", adVarChar, _
      adParamInput, 256)
cmd.Parameters.Append parm

Set parm = cmd.CreateParameter("CustomerId", adInteger, _
      adParamInput, 4)
cmd.Parameters.Append parm

cmd.Parameters("CustomerId").Value = c.CustomerId
cmd.Parameters("Name").Value = c.Name
cmd.Parameters("Street").Value = c.Street
cmd.Parameters("City").Value = c.City
cmd.Parameters("State").Value = c.State
cmd.Parameters("Zip").Value = c.Zip
cmd.Parameters("Phone").Value = c.Phone
cmd.Parameters("EMailAddress").Value = c.EMailAddress
cmd.Parameters("DateAdded").Value = c.DateAdded
cmd.Parameters("DateUpdated").Value = c.DateUpdated
cmd.Parameters("MailingList").Value = c.MailingList
cmd.Parameters("Comments").Value = c.Comments
cmd.Execute

If Err.Number <> 0 Then
   App.LogEvent _
      "Customer(PutCustomer): Can't update the record " & _
      "for CustomerId " & _
      FormatNumber(c.CustomerId, 0) & ". Error: " & _
      Err.Description & "-" & _
      Hex(Err.Number)
   PutCustomer = Err.Number
End If

db.Close
o.SetComplete
Set cmd = Nothing
Set db = Nothing
Set o = Nothing

End Function
```

You may be wondering why I chose to use the **Update** statement, rather than update the database using a `Recordset` object. The answer is really simple: the **Update** statement is more efficient in this situation. In order to use a `Recordset`, you would have to retrieve the records a second time. Then you would have to change the values in the current record and use the `Update` method to send the changes back to the database. In this case, simply send the update directly to the server.

Installing the transaction into COM+

The COM+ transaction server is managed by using the Component Services utility. You can access this utility by choosing Start ➪ Programs ➪ Administrative Tools ➪ Component Services (see Figure 18-4).

To add your application to the COM+ transaction server, you need to perform these steps:

1. Choose Start ➪ Programs ➪ Administrative Tools ➪ Component Services from the Windows taskbar.
2. Under Console Root, open COM+ Applications by selecting Component Services, then Computers and the name of the computer on which you want to create service.
3. Right click on Component Services and choose New ➪ Application from the pop-up menu. This starts the COM Application Install Wizard.
4. Press Next from the welcome screen and then click on the Create an empty application button.
5. On the Create Empty Application window, enter the name for your new application and choose Server Application (see Figure 18-6).
6. Click Next to display the Set Application Identity window shown in Figure 18-7. Choose the user account under which the application will run. You can choose to use the security of the user name who is logged onto the server's console (the interactive user) when the application is run. You may also enter the name and password of a specific user. Click Next to go to the last step of the wizard.
7. Click Finish to install the empty application.

Security can be difficult: For testing purposes, I suggest that you use the user name of the interactive user logged onto the server's console. Then you should sign on as Administrator (or another user name with the same privileges as Administrator). This will remove most of the security restrictions on your transactions. Once the transactions have been debugged, you can go back and create a specific user name with only the privileges needed to run the transaction.

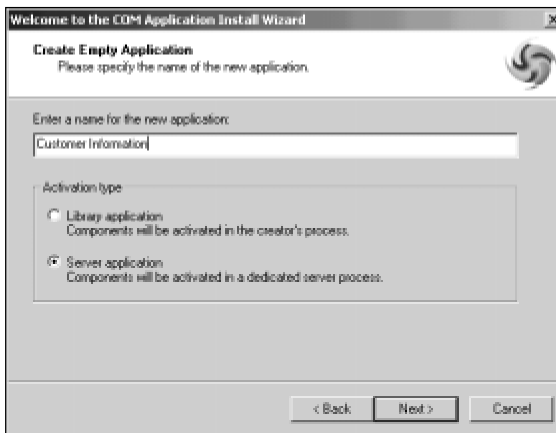

Figure 18-6: Entering the name of the application

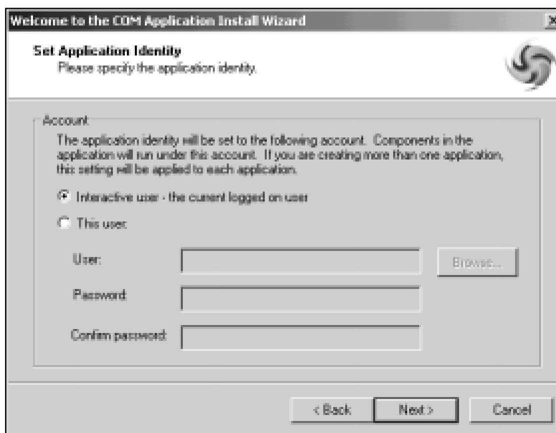

Figure 18-7: Associating the application with a user name

Once you have an empty application, you need to import your ActiveX DLL using these steps in the Component Services utility (choose Start ➪ Programs ➪ Administrative Tools ➪ Component Services in Windows 2000).

1. Expand the application you just created to display the Components and Roles folders beneath it.
2. Right click on the Components folder and select New ➪ Component from the pop-up menu. This will start the COM Component Install Wizard.

3. Click Next to move to the Import or Install a Component step and click on Install New Component(s) button.

4. A File Open dialog box will be displayed. Choose the name of the DLL file you just created and click on the Open button to display the Install New Components dialog box with the file you just selected (see Figure 18-8). If you need to install more than one file, press the Add button.

5. After you have selected the files to be installed, click Next to go to the last step of the wizard and press Finish to add the component to the application.

Figure 18-8: Selecting the name of your ActiveX DLL

Building a simple test program

In order to test the COM+ transactions, I created simple IIS Application. This program responds to the URL `http://Athena/VB6DB18/VB6DB18.ASP` with a Web page, as shown in Figure 18-9.

Tip

IIS Applications are neat: My favorite feature in Visual Basic 6 is the ability to create a compiled, Web server-based application using IIS Applications. IIS Applications are generally more efficient than ASP applications and they are more secure because the source code is separate from the executable program.

When someone enters the URL to run the IIS Application, IIS will trigger the `WebClass_Start` event shown in Listing 18-8. Depending on the request, one of three things will happen. If no form information is included, a blank Web form will be displayed, with default values for the `Customer` object. If a `GetCustomer` request is made, then the `CustomerId` value from the form will be passed to the `GetCustomer` transaction to retrieve the specified customer. If a `PutCustomer` request is made, then the information on the form will be passed to the `PutCustomer` method to update the database. A form with the information about the current customer is returned in both the `PutCustomer` and `GetCustomer` requests.

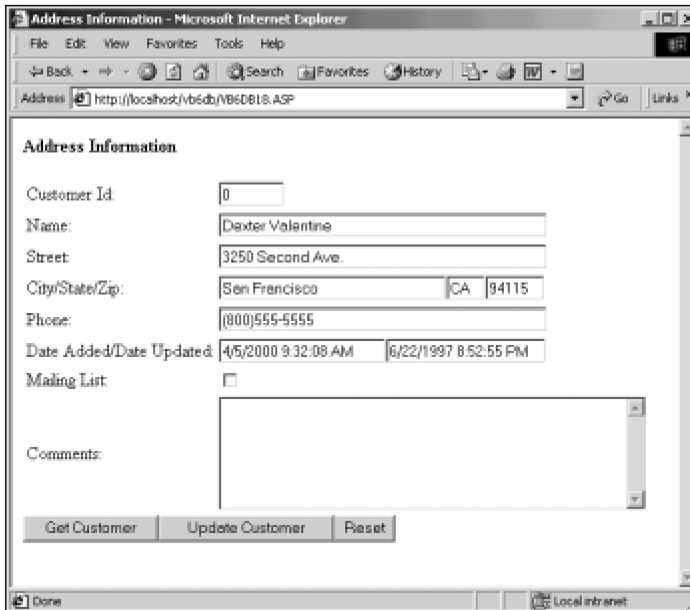

Figure 18-9: Viewing address information returned by the COM+ transaction

Listing 18-8: **The WebClass_Start event in Address Information**

```
Private Sub WebClass_Start()

Dim c As Cust.Customer
Dim ci As Object

If Len(Request.Form("GetCustomer")) > 0 Then
   Set ci = CreateObject("CustInfo.CustomerInfo")
   Set c = ci.GetCustomer(CLng(Request.Form("CustomerId")))
   Set ci = Nothing

ElseIf Len(Request.Form("PutCustomer")) > 0 Then
   Set ci = CreateObject("CustInfo.CustomerInfo")
   Set c = New Cust.Customer
   c.CustomerId = Request.Form("CustomerId")
   c.Name = Request.Form("Name")
   c.Street = Request.Form("Street")
   c.City = Request.Form("City")
   c.State = Request.Form("State")
   c.Zip = Request.Form("Zip")
```

Continued

Listing 18-8 *(continued)*

```
   c.Phone = Request.Form("Phone")
   c.EMailAddress = Request.Form("DateAdded")
   c.DateUpdated = Request.Form("DateUpdated")
   If Request.Form("MailingList") = "ON" Then
      c.MailingList = True
   Else
      c.MailingList = False
   End If
   c.Comments = Request.Form("Comments")
   ci.PutCustomer c
   Set ci = Nothing

Else
   Set c = New Cust.Customer
   c.CustomerId = 9999

End If

With Response
   .Write "<html>"
   .Write "<head>"
   .Write "<title>Address Information</title>"
   .Write "</head>"
   .Write "<body>"
   .Write "<strong>Address Information</strong>"
   .Write "<form align=""left"" name=""AddressInfo"" "
   .Write "action=""VB6DB18.ASP"" method=""post"">"
   .Write "<table border = ""0"">"
   .Write "<tr>"
   .Write "<td>Customer Id:</td>"
   .Write "<td>"
   .Write "<input type=""text"" name=""CustomerId"""
   .Write " size=""6"" value="
   .Write FormatNumber(c.CustomerId, 0, , , vbFalse) & ">"
   .Write "</td>"
   .Write "</tr>"
   .Write "<tr>"
   .Write "<td>Name:</td>"
   .Write "<td>"
   .Write "<input type=""text"" name=""Name"""
   .Write " size=""45"" value=""" & c.Name & """>"
   .Write "</td>"
   .Write "</tr>"
   .Write "<tr>"
   .Write "<td>Street:</td>"
   .Write "<td>"
   .Write "<input type=""text"" name=""Street"""
   .Write " size=""45"" value=""" & c.Street & """>"
   .Write "</td>"
```

```
.Write "</tr>"
.Write "<tr>"
.Write "<td>City/State/Zip:</td>"
.Write "<td>"
.Write "<input type=""text"" name=""City"""
.Write " size=""30"" value=""" & c.City & """>"
.Write "<input type=""text"" name=""State"""
.Write " size=""2"" value=""" & c.State & """>"
.Write "<input type=""text"" name=""Zip"""
.Write " size=""5"" value="
.Write FormatNumber(c.Zip, 0, , , vbFalse) & ">"
.Write "</td>"
.Write "</tr>"
.Write "<tr>"
.Write "<td>Phone:</td>"
.Write "<td>"
.Write "<input type=""text"" name=""Phone"""
.Write " size=""45"" value=""" & c.Phone & """>"
.Write "</td>"
.Write "</tr>"
.Write "<tr>"
.Write "<td>Date Added/Date Updated:</td>"
.Write "<td>"
.Write "<input type=""text"" name=""DateAdded"""
.Write " size=""21"" value=""" & c.DateAdded & """>"
.Write "<input type=""text"" name=""DateUpdated"""
.Write " size=""20"" value=""" & c.DateUpdated & """>"
.Write "</td>"
.Write "</tr>"
.Write "<tr>"
.Write "<td>"
.Write "Mailing List:"
.Write "</td>"
.Write "<td>"
If c.MailingList Then
   .Write "<input type=""checkbox"" "
   .Write "name=""MailingList"" value=""ON"" checked>"
Else
   .Write "<input type=""checkbox"" "
   .Write "name=""MailingList"" value=""ON"">"
End If
.Write "</td>"
.Write "</tr>"
.Write "<tr>"
.Write "<td>Comments:</td>"
.Write "<td>"
.Write "<textarea rows=""6"" name=""Comments"""
.Write " cols=""45"">"
.Write c.Comments
.Write "</textarea>"
.Write "</td>"
.Write "</tr>"
```

Continued

Listing 18-8 *(continued)*

```
    .Write "</table>"
    .Write "<input type=""submit"" value=""Get Customer"" "
    .Write "name=""GetCustomer"">"
    .Write "<input type=""submit"" value=""Update Customer""  "
    .Write "name=""PutCustomer"">"
    .Write "<input type=""reset"" value=""Reset""
    .Write " name=""Reset"">"
    .Write "</form>"
    .Write "</body>"
    .Write "</html>"
End With

Set c = Nothing

End Sub
```

This program is a little tricky in the way it works. It relies on the fact that the `Request.Form` method doesn't return an error when I try to access a particular field that isn't present, plus the fact that the value of a button is present in the form data only when it is pressed. So I can check `Request.Form("GetCustomer")` to see if someone pressed the Get Customer button and know that the value will only be there when someone pressed that button.

This means that I can use a compound `If` statement to determine which of the buttons on the form were pressed, if any. If the user pressed the Get Customer button, I can create an instance of the `CustomerInfo` object and call the `GetCustomer` method to return the specified customer value. Likewise, I can create a `CustomerInfo` object to call the `PutCustomer` method if I find the Put Customer button in the `Form.Request` method.

If neither button on the form was pressed, I can simply create an empty `Customer` object. This means that no matter how I start the form, I'll have a valid `Customer` object to use when I generate the form.

The second half of the routine is devoted to creating the HTML tags necessary to display the form. I embedded the references to the `Customer` object where necessary to display information on the form.

Developing IIS Application forms the lazy way: I used Microsoft FrontPage 2000 to create the basic form and then I copied the HTML tags in WordPad and added the appropriate calls to `Write`. Finally, I copied the statements into Visual Basic and added the code to reference the various objects I created.

Thoughts on COM+The COM+ Transaction Server isn't appropriate for all applications. While COM+ transactions help large applications, they may actually hurt small ones. First, it adds another level of complexity that isn't necessary in a small application. Second, it may actually perform slower, depending on the hardware configuration.

However, there is no substitute for COM+ if you are building a high-volume application. This is especially true if you support multiple user interfaces to the data, such as a traditional Visual Basic form-based client/server program and a Web based application using an IIS Application.

There is a flaw in the update logic of the sample application. You may run into a conflict when multiple users try to update the data. Consider the case where Christopher gets a copy of the data, then Samantha gets a copy of the data and updates it, and then Christopher performs his update. Samantha's update would be lost and Christopher would not have seen her changes. For many applications this approach is fine, but it may cause a problem with others.

This problem is similar to the problem with optimistic locks, and you can use a similar approach to correct it. You could modify the `Customer` object to store both the original values for each field and the current values for each field. Then, when you update the database, you can compare the original values to the values in the database and abort the transaction if there is a difference.

Summary

In this chapter you learned the following:

- You can build applications by using multiple tiers of processing. This is known as *n*-tier processing.
- You can use transaction servers to improve the performance of your application.
- You can use the Component Services utility to add transactions to a COM+ Transaction Server.
- You can use the ACID test to determine if you have a valid transaction.
- You can use the `ObjectContext` object to communicate from your transaction to the transaction server.
- You can control security on a transaction so that it can perform tasks with the authority of someone other than the user name that called the transaction.