

Chapter 5

Message Queues II

In the previous Chapter I covered the basic concepts of what message queues are and how to send both simple and complex messages. In this Chapter I will continue the discussion with the more advanced topics of transactions including integrating COM+ for distributed transaction support. The next important issue that will be covered is what happens after you send the message? Like in the analogy of sending a letter, you can never be sure if the recipient has read the letter or not. The same issues are present with electronic messaging and I'll discuss the techniques available to ensure you do know the message was read.

As this Chapter is really an extension of the previous discussion, you'll find that the examples continue from where the previous Chapter left off. This also means that technically there is no sample code for this Chapter as all the samples you see in this Chapter will be part of the samples for the previous Chapter.

Using Transactions

The concept of a transaction is where you execute a multi-step process such that either all steps succeed or none of them succeed. Most database texts will describe a transaction as an atomic operation where it meets the ACID test (Atomicity, Consistency, Isolation and Durability). Should a process fail, then that process and any previous processes within the transaction should revert to their original state prior to the transaction (rolled back). For a more technical discussion on the ACID test search for *ACID* in the online help.

Now that we have an idea of what a transaction is, how do they relate to MQ's? MSMQ supports transactions in two ways:

- * **Internal Transaction:** by allowing multiple messages to be sent or received as part of a transaction
- * **External Transactions:** by participating in Distributed Transaction Coordinator (DTC) transactions.

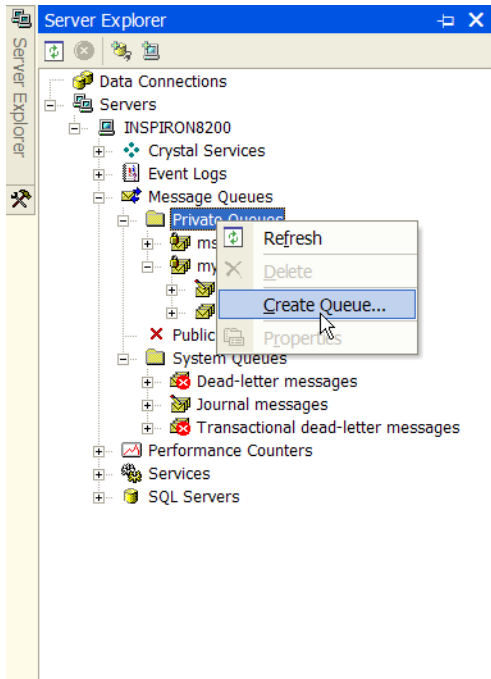


Figure 5-1. – Creating a Queue

The difference between the two types will be covered later in the Chapter, but before we can use a transactional MQ, we need to set the queue up to be transactional. To create a transactional queue you simply repeat the steps that you did to create a non-transactional queue but this time tick the transactional check box. To recap, using the Server Explorer window click on Private Queues, then right click and choose create queue as in Figure 5-1. Type in the name of the queue you want, which in this case is Transactional, and then tick the transactional check box, then click OK (see Figure 5-2).

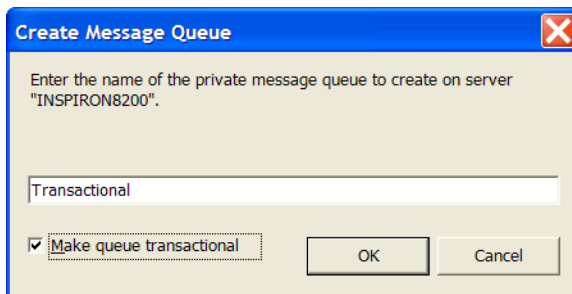


Figure 5-2. - Creating a Transactional Queue

Now that we have created our new transactional MQ, it's important to know if the queue we are using supports transactions otherwise we may try sending a message in a transaction when the queue doesn't support it, which will result in a MessageQueueException. The reverse is also true, in that if you try to receive or send messages to a transactional queue without using transactions you will also get an exception. So it is important to check the queue first if you don't know if the queue is

transactional. To do this check is very easy, and only requires that you have an *instance* of the MessageQueue object and test the following Boolean property.

```
objMessageQueue.Transactional()
```

When you send or receive a message you have the option of specifying a transaction type for the send or receive; Table 5-1 details the options available to you.

Table 5-1 Message Queue Transaction Types

Enumeration	Description
Automatic	A transaction type used for Microsoft Transaction Server (MTS) or COM+ 1.0 Services. If there is already an MTS transaction context, it will be used when sending or receiving the message. This is the type used when doing heterogeneous transactions.
None	Operation will not be transactional.
Single	A transaction type used for single internal transactions.

If you are going to use the simple version of the Send method for example in a transaction (i.e. not using a Message object), then you must tell the Send method that you want to use transactions. To do this you use the following line of code.

```
objMessageQueue.Send(<message txt>, <label>,
MessageQueueTransactionType.Single)
```

Internal Transactions

As mentioned MSMQ supports two types of transactions, the easiest to use is the internal type. This allows you to send and receive several messages and have them rollback should an error occur during the receipt of any of the messages. This allows you to execute multiple messages (both sending a receiving) against one or more queues, wrapped up in a transaction so that you can guarantee that either all or none of the messages will be received. The transactional support in MSMQ is limited to MSMQ actions however, if you want to include external resources such as databases, then you'll have to use the external support of MSMQ.

Example 5 – Sending a Transactional Message

To send a message using a transaction you must first create a transaction object using the System.Messaging.MessageQueueTransaction class. After you have created the transaction object you then need to supply it when ever you send a message. Example5a is exactly the same as Example2 with the following code added.

```
Dim objTrans As MessageQueueTransaction
objTrans = New MessageQueueTransaction()

' Send message within a transaction
objTrans.Begin()
objMessageQueue.Send(objMessage, objTrans)
objTrans.Commit()
```

This code uses the `MessageQueueTransaction` class which has several methods which are of interest. Table 5-2 describes the methods and their meaning which can be compared to their use in the code samples.

Table 5-2 MessageQueueTransaction Methods

Member Name	Description
Abort	Allows you to programmatically abort a transaction which is currently pending. This results in the transaction being rolled back to its original state and the transaction stopped.
Begin	As the name implies, begins a new internal transaction.
Commit	Used to signify that the transaction has completed successfully and that the data should be saved. Note that if the transaction is nested within another then it will only be committed if all transactions are committed.

The example below is taken from Example5b, which shows the sending of two messages. The second message will cause an exception to occur due to the `Path` property being set to an invalid path. You'll notice that you won't see the first message appear on the queue due to the first one being automatically rolled back. However, if you remove the highlighted lines from this example you get Example5a, which only sends one message which should succeed. You should then see the message appear in the Transactional MQ.

Try

```
objMessageQueue = New MessageQueue(Constants.MQ.LocalTransQueue)
' Create a transaction object to be used on all sends
objTrans = New MessageQueueTransaction()

' Send message within a transaction
objTrans.Begin()
objMessageQueue.Send(objMessage, objTrans)
' Cause an error here, within a transaction
objMessageQueue.Path = "xxxx"
objMessageQueue.Send(objMessage, objTrans)
objTrans.Commit()
Catch ex As Exception
' An error occurred so we need to rollback any trans if we started one
If objTrans.Status = MessageQueueTransactionStatus.Pending Then
    objTrans.Abort()
End If
Debug.WriteLine(ex.Message)
Debug.WriteLine(ex.StackTrace)
Finally
objMessageQueue.Close()
End Try
```

As you can see, you need to include good error handling when dealing with transactions so you can abort any that fail. When we do our error handling, we should always check that a transaction is currently active, otherwise it will throw an exception.

To do this we make use of the Status property of the MessageQueueTransaction class which returns a MessageQueueTransactionStatus enumeration type; see Table 5-3.

Table 5-3 MessageQueueTransactionStatus Enum

Member name	Description
Aborted	The transaction has been aborted which may have been the result of calling the <u>Abort</u> method.
Committed	The transaction has been committed which may have been the result of calling the <u>Commit</u> method.
Initialized	The transaction has been initialized; although it hasn't been started as yet. As the transaction hasn't started yet, you can't pass it to a method that uses it as its not ready.
Pending	The transaction has been started and is currently active. At this stage you can use either the <u>Abort</u> or <u>Commit</u> methods to complete the transaction.

Finally you should always ensure that the MQ is closed at the end of the procedure. This is best done in the Finally block as this is guaranteed to always be called regardless of whether an error occurred or not.

External Transactions

We have learned how to use transactions by combining several messages within the same transaction. This can be useful, however it is somewhat limiting, as the 'real' work is usually done by an external process such as a class updating a database. Take an example of an order that is received from a queue. Then for auditing purposes it is logged into a database. Also assume that another message that is part of the transaction arrives but this time it fails when attempting to update the audit log. If MSMQ transactions were in use, the MSMQ activities would be rolled back, but the database work would not. Such a situation is little better than using no transaction processing at all, because the result could still be unreliable data in your system. The solution is to use a form of transaction processing that is capable of handling the database and MSMQ work as part of the same transaction. MSMQ provides this capability via a technology known as MS DTC (Microsoft Distributed Transaction Coordinator). Using DTC allows you to initiate a transaction and then proceed to use resources from multiple sources that support DTC, such as SQL Server and Oracle, and then have those resources participate in the transaction, committing or rolling back as appropriate when the transaction ends.

Using COM+ for DTC Support

As mentioned earlier the database must support DTC, but you must also make your class support it too. Before .NET, this was done by creating a COM component and then registering it in a COM+ application (or if you've been in the game awhile an MTS package). Well Microsoft hasn't done away with COM+ yet, so we need to make our classes suitable for running in COM+. This is achieved by inheriting from the System.EnterpriseServices.ServicedComponent class and then setting the class up with

attributes indicating that it requires transactions. For the following example we will write a component which will add an Order to the Northwind database. We will create a new class file for the new component called OrdersUpdateCOM+. As the name suggests it will be a COM+ component. We will call this component from the original Orders class via its Update method. For now let's concentrate on what is required to make a class support COM+.

```
Imports System.EnterpriseServices
Imports System.Reflection

' Supply the COM+ application name.
<Assembly: ApplicationName("SalesOrders")>
' Supply a COM+ Type.
<Assembly: ApplicationActivation(ActivationOption.Library)>
' Define where the strong name Key pair is
<Assembly: AssemblyKeyFileAttribute("KeyPair.snk")>

Namespace Sales.COMPlus
  <Transaction(TransactionOption.Required)> Public Class OrderUpdate
    Inherits ServicedComponent

    Sub Update(ByVal order As Sales.Order)
      Try
        ' Do your database and other work here
        ' Now tell COM+ the transaction completed successfully
        ContextUtil.SetComplete
      Catch ex As Exception
        ' Do error handling here if required

        ' Now tell COM+ the transaction failed
        ContextUtil.SetAbort
        Throw New Exception("Unable to save Order:" & order.CustomerID)
      Finally
        'Check if the connection object is open and if so close it!
        If conDatabase.State = ConnectionState.Open Then
          conDatabase.Close()
        End If
      End Try
    End Sub
  End Class
End Namespace
```

Looking at the code presented, you'll notice that we are using two namespaces System.EnterpriseServices, which contains the ContextUtil class, ApplicationName and ApplicationActivation attributes. The other System.Reflection contains the AssemblyKeyFileAttribute attribute. The attributes tell COM+ details of how it should be registered. An example of which can be seen if you look in Component Services, which

you can find by going to Control Panel ➤ Administrative Tools ➤ Component Services. This then displays the components which have been registered with COM+. At the conclusion of this Chapter you'll see the sample we are about to create there too as shown in Figure 5-3.

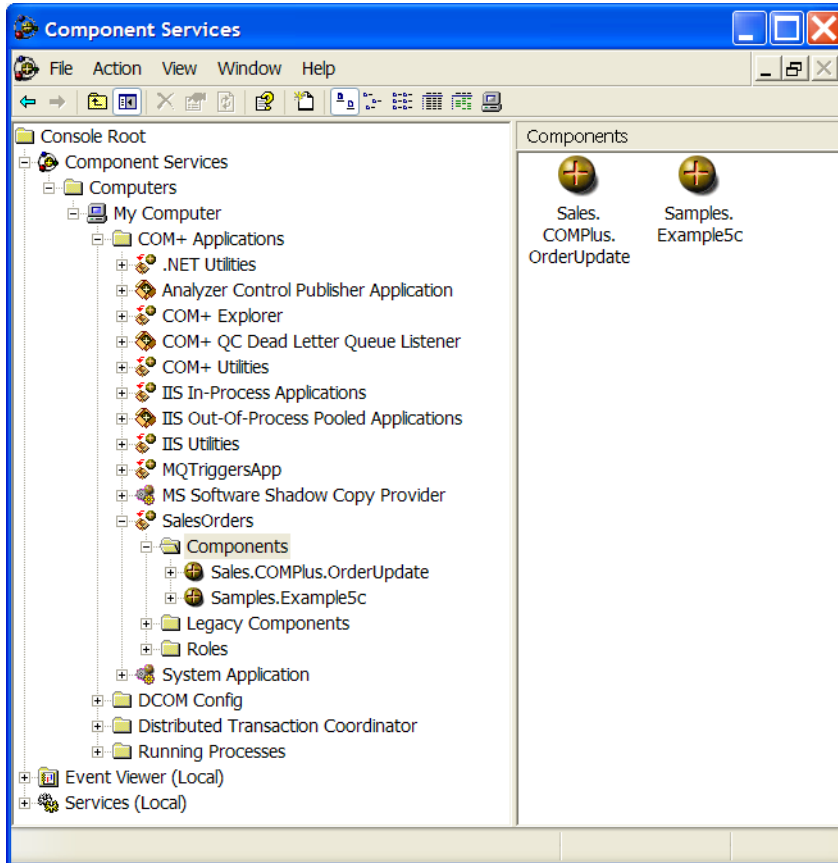


Figure 5-3. Component Services with a registered COM+ application

When you register a COM+ component you need to determine whether the component will run in its own process (ActivationOption.Server) or in the same process as the caller (ActivationOption.Library). I have opted for the latter as it means we can take advantage of dynamic registration which doesn't require us to register the assembly in the GAC (Global Assembly Cache), which is fine for our purposes. Another thing that is important when using COM+ is that each assembly that contains a COM+ class must have a strong name. You can ensure that a name is globally unique by signing an assembly with a strong name. To do this you need a public/private key pair, which is a file that is easily generated with the following command line utility:

```
sn -k KeyPair.snk
```

If you're using VS.NET as we are in our examples then this file must be in the same directory as the VS.NET solution file otherwise the file won't be found. You assign this file to your assembly by using the AssemblyKeyFileAttribute attribute. Also all COM+ classes must be inherited from the ServiceComponent class which has all the required code to be a COM+ component. The only other thing left to do is tell COM+ what type of

transaction support we want our class to have. This is done with the Transaction class attribute, which can be either of the following.

- * Disabled
- * NotSupported
- * Required
- * RequiresNew
- * Supported

For our class we have set the Required option as it will ensure that we always work in a transaction even if the caller doesn't have an existing transaction.

Once we have set the class to use COM+, we have done enough to indicate that our class will participate in a transaction. It is therefore enough that we are running our code within a COM+ hosted component to enlist them into the current transaction. However, there is one extra piece of code that you will require when using transactions within a COM+ component. Those of you who are already familiar with COM+ will recognize, the SetComplete and SetAbort methods found in the System.EnterpriseServices.ContextUtil class. Simply put you should call one of these methods at the end of your routine to either commit the transaction (SetComplete) or to rollback the transaction (SetAbort).

The ContextUtil class can do more than is indicated here, but this is out of the scope of the Chapter, if you wish to learn more about this then I encourage you to search for 'Writing Serviced Components' in the .NET documentation for more information.

Note: You can also use the <AutoComplete()> attribute in replace of SetComplete and SetAbort. Using this attribute will automatically call SetComplete at the end of the routine if no exceptions occurred, otherwise it will call SetAbort if an exception is thrown. Personally, I prefer to use the more explicit version in my code.

Using COM+ to update a Database

This is where the fun really starts as we are now going to create an update method in our new OrdersCOM+ component. We will then call this class from our standard Orders class, which will allow us to use our existing Orders class but have it also be part of a transaction if we decide to store the data. The following code is all that is required for the COM+ update order class:

```
Imports System.EnterpriseServices
Imports System.Reflection

' Supply the COM+ application name.
<Assembly: ApplicationName("SalesOrders")>

' Supply a COM+ Type.
<Assembly: ApplicationActivation(ActivationOption.Library)>

' Define where the strong name Key pair is
```



```
<Assembly: AssemblyKeyFileAttribute("KeyPair.snk")>
```

```
Namespace Sales.COMPlus
```

```
<Transaction(TransactionOption.Required)> Public Class OrderUpdate
```

```
    Inherits ServicedComponent
```

```
    ' This method will take an existing order and use the data to update
```

```
    ' the database
```

```
    Sub Update(ByVal order As Sales.Order)
```

```
        Const conConnectionString As String = _
```

```
            "server=(local);database=NorthWind;uid=sa;pwd="
```

```
        Dim conDatabase As SqlClient.SqlConnection
```

```
        Dim cmdCommand As SqlClient.SqlCommand
```

```
        Dim intOrderID As Integer
```

```
    Try
```

```
        ' Do your database and other work here
```

```
        ' This example will only deal with New orders
```

```
        Debug.WriteLine(ContextUtil.TransactionId())
```

```
        If order.OrderID = 0 Then
```

```
            ' We have a new order
```

```
            conDatabase = New _
```

```
                SqlClient.SqlConnection(conConnectionString)
```

```
            cmdCommand = New SqlClient.SqlCommand()
```

```
            ' First we create the Order Entry
```

```
            With cmdCommand
```

```
                .CommandType = CommandType.StoredProcedure
```

```
                .CommandText = "usp_Order_Insert"
```

```
                .Parameters.Add("@CustomerID", SqlDbType.NChar).Value _
```

```
                    = order.CustomerID
```

```
                .Connection = conDatabase
```

```
                .Connection.Open()
```

```
                intOrderID = CType(.ExecuteScalar, Integer)
```

```
            End With
```

```
            ' Next we create the OrderDetail Entries
```

```
            Dim objLineItem As Sales.LineItem
```

```
            For Each objLineItem In order.OrderDetails
```

```
                ' Ensure we have a fresh Command Object each time
```

```
                cmdCommand = New SqlClient.SqlCommand()
```

```
                With cmdCommand
```

```
                    .Connection = conDatabase ' Still open from before
```

```
                    .CommandType = CommandType.StoredProcedure
```

```
                    .CommandText = "usp_OrderDetails_Insert"
```

```
                    .Parameters.Add("@OrderID", SqlDbType.Int).Value _
```

```
                        = intOrderID
```

```

.Parameters.Add("@ProductID", SqlDbType.Int).Value _
    = objLineItem.ProductID
.Parameters.Add("@UnitPrice", _
    SqlDbType.Money).Value = objLineItem.UnitPrice
.Parameters.Add("@Quantity", _
    SqlDbType.SmallInt).Value = objLineItem.Quantity
.Parameters.Add("@Discount", SqlDbType.Real).Value _
    = objLineItem.Discount
If .ExecuteNonQuery() = 0 Then
    ' No records added, so its an ERROR!
    Throw New Exception("Unable to Insert Record")
End If
End With
Next
Else
    Throw New Exception("Existing Orders not supported yet.")
End If
' Now tell COM+ the transaction completed successfully
ContextUtil.SetComplete()
Catch ex As Exception
    ' Do error handling here if required

    ' Now tell COM+ the transaction failed
    ContextUtil.SetAbort()
    Throw New Exception("Unable to save Order:" & order.CustomerID)
Finally
    'Check if the connection object is open and if so close it!
    If conDatabase.State = ConnectionState.Open Then
        conDatabase.Close()
    End If
End Try
End Sub
End Class
End Namespace

```

This code is relatively straightforward in what it's doing. Firstly, we setup the class to work under COM+ as discussed previously. Next we go on to the real work of the class, which is to take an existing Order object and add it to the database. We do this by adding the Order to the Orders table and receiving back the new key for use in the next section which adds the order details. We then add a new order detail record for each item in the OrderDetails collection of the Orders object. The code could have been made a bit more efficient, but for simplicity I've not tried to make it too complicated. The end result is if everything goes to plan we have a new entry in the Orders table and *n* records in the Order Details table. Something you may find interesting is that in the Catch block after aborting the transaction I throw an exception. This is to alert the calling application that an error has occurred. This can save the calling application doing any unnecessary processing given

the transaction has already failed. You will also notice I've made use of namespaces as well, this is to make it easier to see that a particular object is a COM+ one by placing it in a namespace that highlights this. I've also used SQL Server stored procedures, and for those interested in what they look like I've included them here. You can create these stored procedures easily by using the techniques discussed in SQL Server section of Chapter 1.

Create a new Order.

```
CREATE PROCEDURE usp_Order_Insert
(
    @Customerid nchar(5)
)
AS
INSERT INTO Orders
    (CustomerID)
VALUES (@CustomerID)
/* Return the ID of the record we just created */
SELECT SCOPE_IDENTITY() As OrderID /*This is more accurate than @@IDENTITY */
```

Add an Order Detail for a given Order

```
CREATE PROCEDURE usp_OrderDetails_Insert
(
    @OrderID int,
    @ProductID int,
    @Quantity smallint,
    @UnitPrice money,
    @Discount real
)
AS
INSERT INTO [Order Details]
    (OrderID, ProductID, UnitPrice, Quantity, Discount)
VALUES (@OrderID, @ProductID, @UnitPrice, @Quantity, @Discount)
RETURN
```

Now we have all the code required to add the Order object to the database. We only need to wire up this code to the original Order object. To do this we simply add the following code to the existing Save method.

```
Sub Save()
    ' This method will save the record
    Dim objOrdersCOMPlus As New Sales.COMPlus.OrderUpdate()
    objOrdersCOMPlus.Update(Me)
End Sub
```

This is all the code needed to allow our Orders object to participate in a DTC transaction, which means if anything goes wrong it will automatically rollback any changes made.

Example 5c – Creating a DTC enabled Message receiver

So now that we have COM+ enabled our Orders object, we can create our message receiver which will process incoming messages and then have them update the database:

```
' Supply the COM+ application name.
<Assembly: ApplicationName("SalesOrders")>
' Supply a COM+ Type.
<Assembly: ApplicationActivation(ActivationOption.Library)>
' Define where the strong name Key pair is
<Assembly: AssemblyKeyFileAttribute("KeyPair.snk")>
... Other code omitted for brevity ...
Sub ProcessMessage()
    Dim objMessageQueue As MessageQueue
    Try
        ' Retrieving an object from a message queue
        objMessageQueue = New MessageQueue(Constants.MQ.LocalTransQueue)
        objMessageQueue.Formatter = New XmlMessageFormatter(New _
            Type() {GetType(Order)})

        Do While objMessageQueue.GetAllMessages.Length <> 0
            Dim objMessage As Message = objMessageQueue.Receive(New _
                TimeSpan(0, 0, 10), MessageQueueTransactionType.Automatic)
            Dim objOrder As Order = CType(objMessage.Body, Order)
            ' Process message
            objOrder.Save()
            MessageBox.Show("Processed Order for Customer: " & _
                objOrder.CustomerID)
        Loop
        ContextUtil.SetComplete()
    Catch ex As Exception
        MessageBox.Show(ex.Message, "Error")
        ContextUtil.SetAbort()
    Finally
        objMessageQueue.Close()
    End Try
End Sub
```

The code to send the messages is not much different to that of Example5a, however we have included a separate example in the download which will send a good message and a bad one to demonstrate the rollback features. However, the code above is new and deserves some discussion. The most important thing to understand is that in order for COM+ and DTC to do its work, *all* parts of the transaction *must* share the same transaction id. This means you have to create another COM+ object which will set the transaction context. This is exactly the same procedure we used earlier for our Orders object. As you can see the attributes are identical to those we used earlier. Although not shown here the above class also inherits from ServicedComponent, as all COM+ classes

must do. Apart from the COM+ changes there is one very important difference regarding how we receive our messages. Unlike the internal transactional support of MQ's, external support doesn't use a transaction object. Transactions are handled completely by COM+; all you need to do is specify that MSMQ uses any existing transaction context. You do this by simply specifying `MessageQueueTransactionType.Automatic` as the transaction type when using the `Receive` method. Our example here will process all the messages on the queue until either there's an error or there's no more messages to process. If you run Example5c – Send DTC Message (No Error) followed by Example5c – Receive/Process DTC Message then you'll notice you have two new records in your NorthWind database (see Figure 5-4). However, if you run Example5c – Send DTC Message (Error). You'll notice that although the first message is processed correctly, the second message errors. This has the effect of rolling back both messages and also leaves the original messages in the queue ready to be processed again. The example here however doesn't account for this and so you won't be able to process any good messages unless you purge the bad ones; you can use Server Explorer to do this.

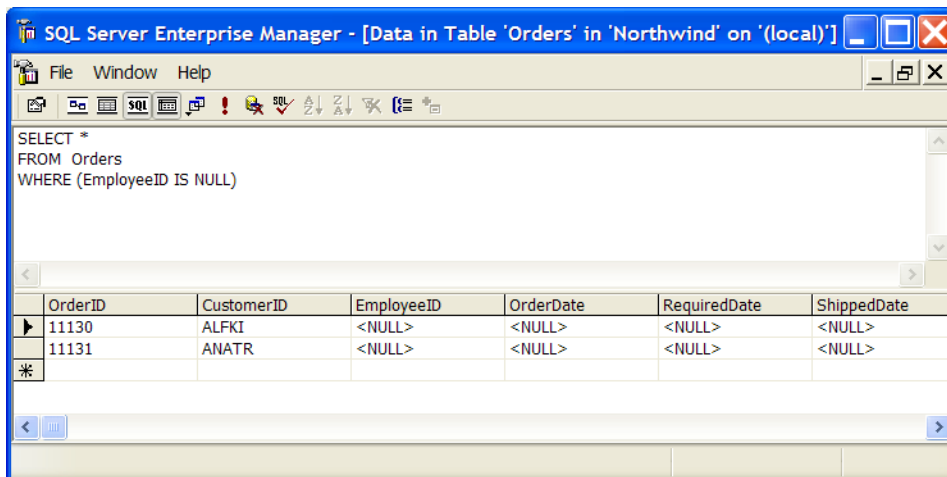


Figure 5-4. Enterprise Manager

This is easily the hardest section in this Chapter, and I really encourage you to play with these examples and get them working on your machine. COM+ can be tricky if you're not familiar with it, or even if you are it's a bit different in .NET. I would say the dynamic registration that .NET provides does make using COM+ easier than what it was previously, however COM+ is again a large section which I unfortunately can't cover in the depth it deserves. There are resources at the end of the Chapter where you can get more information on COM+.

Acknowledgements, Time outs and Journaling

As we have seen so far it's fairly easy to send a message, and then retrieve the message. We can even create transactions when we send or receive a message. However, we cannot wrap the entire process within a transaction. That is, we can create a transaction that handles the sending of the message, and another transaction that handles the receiving of

the message, but we can't have a transaction that handles both the sending and receiving of the message.

This can cause an issue as the application that sent the message may not know if the application receiving the message processed it correctly. This limitation is just another aspect of asynchronous processing and it means we have to use different methods to determine if a message has made it all the way through the system correctly. Some of the methods include the use of acknowledgement messages, time outs, and journal queues. This is what we will be covering in this section.

Acknowledgements

When we call a method on an object, we can get immediate feedback on the success of the operation. Asynchronous processing by its nature doesn't do this, so a mechanism needs to be provided that will give us this feedback. In MSMQ, this mechanism is provided by *acknowledgements*, which are special messages generated by MSMQ which are sent to inform the sender that a message was successfully delivered and/or retrieved at the destination.

All the examples that have been presented thus far have used a local message queue, which doesn't require notification of the messages delivery. However should our application be working in an offline (disconnected) mode, then this acknowledgement becomes more important, so we know the message has reached its destination. MSMQ provides the following acknowledgement masks when you send a message:

- * **FullReachQueue:** Success or failure of the message to be delivered to the queue.
- * **FullReceive:** Success or failure of the message to be received.
- * **NegativeReceive:** Failure of the message to be received.
- * **NotAcknowledgeReachQueue:** Failure of the message to be delivered.
- * **PositiveArrival:** Successful delivery of the message.
- * **PositiveReceive:** Successful receive of the message.

Making use of these settings is again quite straightforward and only requires that you setup a non-transactional queue where you want the messages to be delivered, and to determine what settings you want to use. You will need to use this administration queue to receive any of the messages that are generated by the system.

Example 6a – Setting Acknowledgements

Building on Example2, we only need to modify the message properties section, so that it looks like the following to start supporting acknowledgements:

```
Dim objMessage As New Message()  
With objMessage  
    .Body = objOrder
```

```
.Priority = MessagePriority.Highest
' We want them to hang around between computer restarts
.Recoverable = True
.AdministrationQueue = New MessageQueue(Constants.MQ.LocalAckQueue)
.AcknowledgeType = AcknowledgeTypes.FullReachQueue Or _
    AcknowledgeTypes.FullReceive
End With
```

The AcknowledgeType uses a bit mask, so you can define one or all of the possible settings using an Or between your options. This section of code sets the AdministrationQueue to the queue where the messages are to be sent, which in this case is Constants.MQ.LocalAckQueue. Then we ask that we be notified when the message is delivered and also when it has been received (i.e. read off the queue). We also made the messages recoverable; this will ensure that we don't lose any notifications. Also, ensure that you have created the queue first otherwise you'll receive an exception.

Example 6b – Receiving Acknowledgements

We have just learned how to send a message that will notify us when it's arrived and when it's been read. However, the notifications are of little use unless we know how to read them. The messages that are sent are slightly different to normal messages, in that they usually don't contain any body information, just header information. However, the good news is that they are received in exactly the same way as regular messages from the administrative queue you specified when sending the message. Once we have the message however, we need to make use of other properties of the Message object, to compensate for there being no body information.

Table 5-4. Default values for MessageReadPropertyFilter.

Property	Default Value	.NET Docs
AcknowledgeType	True	False
Acknowledgment	True	False
AdministrationQueue	True	True
AppSpecific	False	False
ArrivedTime	False	True
AttachSenderId	False	False
Authenticated	False	False
AuthenticationProviderName	False	False
AuthenticationProviderType	False	False
Body	True	True
ConnectorType	False	False
CorrelationId	False	True

DefaultBodySize	1024	1024 bytes
DefaultExtensionSize	255	255 bytes
DefaultLabelSize	255	255 bytes
DestinationQueue	False	False
DestinationSymmetricKey	False	False
DigitalSignature	False	False
EncryptionAlgorithm	False	False
Extension	False	False
HashAlgorithm	False	False
Id	True	True
IsFirstInTransaction	False	False
IsLastInTransaction	False	False
Label	True	True
MessageType	True	False
Priority	False	False
Recoverable	False	False
ResponseQueue	True	True
SenderCertificate	False	False
SenderId	False	False
SenderVersion	False	False
SentTime	False	True
SourceMachine	False	False
TimeToBeReceived	False	False
TimeToReachQueue	False	False
TransactionId	False	False
TransactionStatusQueue	False	False
UseAuthentication	False	False
UseDeadLetterQueue	True	False
UseEncryption	False	False
UseJournalQueue	True	False
UseTracing	False	False

Earlier we learned how to read messages off the queue using the default properties. However, sometimes we need to ask for extra properties to be read as well. A list of the properties available can be seen in Table 5-4 along with their default behavior. The last column is what the .NET documentation has, but as you can see this is actually fairly

inaccurate; Table 5-4 contains the correct version and the version in the .NET documentation. In our case we need to retrieve the CorrelationID which isn't returned by default; contrary to what the documentation tell us. To ask that a property be retrieved, we need to set the respective property on the MessageReadPropertyFilter object.

```
objMessageQueue = New MessageQueue(Constants.MQ.LocalAckQueue)
Dim objMessage As Message
' This is the property which identifies the message
objMessageQueue.MessageReadPropertyFilter.CorrelationId = True

' We will wait only 5 seconds for the Ack to arrive.
objMessage = objMessageQueue.Receive(New TimeSpan(0, 0, 5))
If objMessage.MessageType = MessageType.Acknowledgment Then
    MessageBox.Show("The following Ack was received: " & _
        objMessage.Acknowledgment.ToString & " by " & _
        objMessage.CorrelationId.ToString)
End If
```

Tip: A timeout acknowledgement does contain the body of the original message allowing for easy resending of the message if required.

Example 6c – Receiving Messages

This example is really only here to allow you to experiment with sending and receiving messages. To fully test how the previous examples work, try sending a message using Example1a, then try Example6b. You should get an error message telling you that there's a timeout. This is because the earlier example didn't use Acknowledgements, but if you run Example6a then Example6b you should get a confirmation that the message arrived at the destination queue. If you run Example6b again however you'll get another timeout as there are no more messages. Now, try running Example6c, this will receive the message sent (Example1a had a default priority of 3 and Example6a a priority of 7, so Example6a will be received first), and if it was a message that requested acknowledgements, then when you run Example6b again you should get confirmation that the message was read see Figure 5-5.

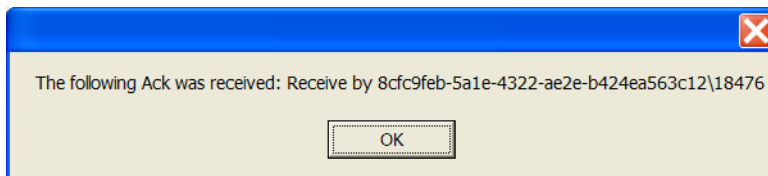


Figure 5-5. – Acknowledgement of message Received

```
' Retrieving an object from a message queue
objMessageQueue = New MessageQueue(Constants.MQ.LocalNonTransQueue)
```

```
objMessageQueue.Formatter = New XmlMessageFormatter(New Type() {GetType(Order)})
Dim objMessage As Message = objMessageQueue.Receive(New TimeSpan(0, 0, 10))
Dim objOrder As Order = DirectCast(objMessage.Body, Order)
MessageBox.Show("(Message: " & objMessage.Label & ") The customer (" & _
    objOrder.CustomerID & "): has ordered " & objOrder.OrderDetails.Count & " _
    products.")
```

Timeouts

As we learned we can ask that the system notify us of when a message arrives at the destination queue, and when it is received amongst other things. However, what happens if a message is never received, that is some type of error occurs which prevents it from being received. As there hasn't been any failure as it was delivered, there wouldn't be any message sent back to the sender telling them that the message can't be received. The solution is to tell MSMQ that unless the message has been received within a specified timeframe it should be considered as a failed transmission. This functionality is provided by way of timeouts in MSMQ, which allows you to specify the time that a message has to arrive at the destination and also how long it has to be received once it gets there. Should either timeout (i.e. fail) then they will both have the same effect, MSMQ will treat it as an error, remove the message from the queue and send any acknowledgements that were requested.

Example 7 – Timeouts

The two properties that need to be set are TimeToReachQueue and TimeToBeReceived, modifying Example6a, to include the following makes the message time sensitive:

```
With objMessage
    .Body = objOrder
    .Priority = MessagePriority.Highest
    .AdministrationQueue = New MessageQueue(Constants.MQ.LocalAckQueue)
    .AcknowledgeType = AcknowledgeTypes.FullReachQueue Or _
        AcknowledgeTypes.FullReceive
    ' We want them to hang around between computer restarts
    .Recoverable = True
    ' This message must arrive within 10 seconds
    .TimeToReachQueue = New TimeSpan(0, 0, 10)
    ' This message must be received within 15 seconds
    .TimeToBeReceived = New TimeSpan(0, 0, 15)
End With
```

This example should never cause a timeout on the delivery as its all on the local queue, however this does become important when sending messages across the Enterprise. We can simulate the timeout of receiving the message by running Example7, then waiting 15 seconds. Running Example6b should then notify us that the message was delivered correctly. Running it again will then notify us that the message has failed to be received in the allotted time.

Journaling & Dead-letters

Journaling and Dead-letter queues, serve a similar purpose to that of acknowledgements, in that they provide a way to track a message. When you send a message it is possible to have a copy of that message stored in a journal queue to act like a form of auditing. Each queue has its own associated journal queue to hold copies of messages as they are received from the queue. Each MSMQ machine also has a single journal queue that can be used to keep copies of each message that is sent from that machine. Dead-letter queues on the other hand deal with messages that generate errors such as timeouts. When such an event happens the message is sent to the Dead-letter queue of the sender's computer. Both of these queues are system queues and therefore shouldn't be created.

Example 8a – Keeping copies of received messages

The property that needs to be set is UseJournalQueue on the MessageQueue object, modifying Example6c, to include the following creates a copy of any message read:

```
objMessageQueue.UseJournalQueue = True
```

You can test this by running Example6a which sends a message, then running Example8a. Using the Server Explorer in VS.NET to look in the JournalQueue (remember to refresh) you will see a copy of the messages that were received. Assuming your queues were empty prior to running the above then you should see a similar result to that in Figure 5-6; where the entries in the Ack queue represent the successful send and receive of the message.

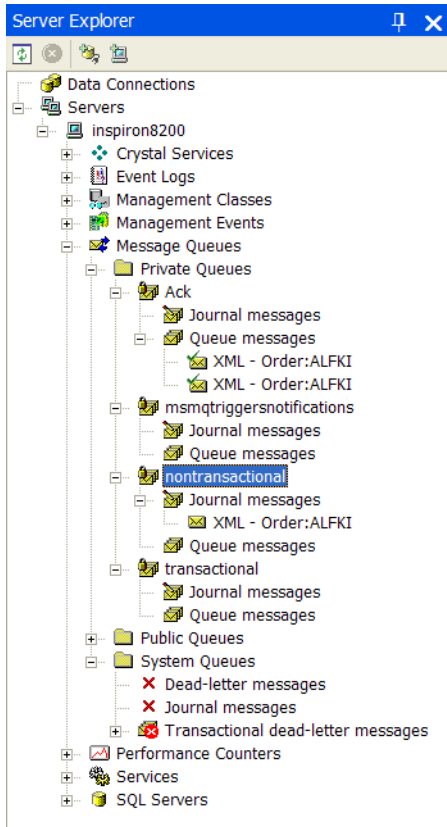


Figure 5-6. – Journaling

Example 8b – Keeping copies of sent messages

The property that needs to be set is UseJournalQueue on the Message object or the DefaultPropertiesToSend.UseJournalQueue property of the MessageQueue object if you're not using a message object. Setting these properties will ensure that all messages that are sent are copied into the journal queue. There is however a catch, in that it will only work for public queues. Which means you need to have a computer running the Server version of NT 4.0, Windows 2000 or Windows .NET (Windows XP doesn't have a Server version), and have the queue setup as public. From a coding standpoint however, there really is no difference except in the path used for the queue. To get Example8b to work in your environment, you'll need to modify the queue constant to match one that's on your network. The required code is again only one line which can be added easily to the code of Example7.

```
objMessageQueue = New MessageQueue(Constants.MQ.PublicNonTransQueue)
```

```
With objMessage
```

```
  .Body = objOrder
```

```
  .Priority = MessagePriority.Highest
```

```
  .AdministrationQueue = New MessageQueue(Constants.MQ.LocalAckQueue)
```

```
  .AcknowledgeType = AcknowledgeTypes.FullReachQueue Or _
```

```
    AcknowledgeTypes.FullReceive
```

```
' We want them to hang around between computer restarts
.Recoverable = True
' This message must arrive within 10 seconds
.TimeToReachQueue = New TimeSpan(0, 0, 10)
' This message must be received within 15 seconds
.TimeToBeReceived = New TimeSpan(0, 0, 15)
' Make a copy of any messages we send
.UseJournalQueue = True
End With
```

Once a message arrives in a journal queue it won't be removed unless you receive it, like you would a normal message. Possible uses for journal queues are handling failed messages or negative acknowledgements. For example, if you receive an acknowledgement indicating that a message timed out, then you may wish to resend the message again. Using the CorrelationID from the acknowledgement, you can locate the message within the journal queue, and then resend it again.

Example 8c Receiving messages from Public Queues

The code required to receive a message from a public queue as mentioned previously is exactly the same as it is for receiving one from a local queue. The only difference is that you use a different queue name. The following shows the difference between a local queue name and a public queue name:

```
Public Const LocalAckQueue As String = ".\private$\Ack"
Public Const PublicNonTransQueue As String = "webserver\NonTransactional"
```

The important thing to notice here is that the public queue doesn't have the word `public` as part of the address. The private queue however does that this. This example is primarily to allow you to receive message that were posted using Example8b.

Example 8d – Resending a Timed out message

As was mentioned earlier, it is possible to take action based on a negative acknowledgement. Here I'll demonstrate how to take read a ReceivedTimeout message and then resend the message again if the user elects to do so. The first part is simple enough and has been covered already. You simply need to read the acknowledgements from the local queue where they are sent. Then based on the type of acknowledgement we can determine if the message has timed out or not:

```
Shared Sub Main()
    Dim objAckQueue As MessageQueue
    Dim objMQ As MessageQueue
    Try
        ' Define the queue from which to receive the Acks
        objAckQueue = New MessageQueue(Constants.MQ.LocalAckQueue)
        Dim objMessage As Message
```

```

objAckQueue.UseJournalQueue = True
' If we want to resend we need this information
objAckQueue.MessageReadPropertyFilter.DestinationQueue = True
' We will wait only 30 seconds for the Ack to arrive.
objMessage = objAckQueue.Receive(New TimeSpan(0, 0, 30))
Select Case objMessage.Acknowledgment
    Case Acknowledgment.ReceiveTimeout
        ' We will ask if we want to resend the message
        If MessageBox.Show("The message: " & objMessage.Label & _
            " just timed out, do you want to resend?", "Time out", _
            MessageBoxButtons.YesNo) = DialogResult.Yes Then

            ' Set the destination of the new message to that of the old.
            objMQ = New _
                MessageQueue(objMessage.DestinationQueue.FormatName)
            ' Now send the duplicate message
            objMQ.Send(copyMessage(objMessage))
        End If
    Case Else
        MessageBox.Show("Ack: " & objMessage.Acknowledgment.ToString & _
            " arrived")
    End Select
Catch ex As Exception
    MessageBox.Show(ex.Message, "Error")
Finally
    objAckQueue.Close()
    objMQ.Close()
End Try
End Sub

```

There is one statement here which I haven't covered yet, which is the DestinationQueue property of the MessageReadPropertyFilter. If you don't set this property then the destination queues details are not retrieved. One we have this information we are ready to copy the message and then resend it.

When you resend a message you need to create a new message object, and then transfer the data from the old message to the new one. However, some of this data can't be recovered, such as timeouts, administration queues etc. You'll therefore need to recode these again when you recreate the message. Another thing to note about the following code is how we transfer the data of the message body. In the examples I've seen they all try to read the data back from the Body property back into the original object. This is fine if you know what the original object is, however if you want to go generic (always good for reuse), then you shouldn't care what the original object was. To achieve this we will transfer the BodyStream from the original message to the new message. This will allow us to copy either simple XML or binary messages, without knowing which it is. To do this we read from the BodyStream of the original message and write the same data back to the new message BodyStream. As can be seen from the code, this only takes two lines of

code. You may have remembered that I mentioned this technique in the previous Chapter when discussing the Formatter property and this is a good example of its use.

Note: A Stream object is the abstract base class of all streams. A stream is a sequence of bytes, which can be in memory, a file, or some other device. The Stream class and its derived classes provide a generic view of these different types of input and output, isolating the programmer from the specific details of the operating system and the underlying devices. It is critical to master the Stream class given its widespread use within the framework.

```

Shared Function copyMessage(ByVal originalMessage As Message) As Message
    Dim objNewMessage As New Message()
    ' Use the BinaryReader to easily read the BodyStream object
    Dim objReader As New System.IO.BinaryReader(originalMessage.BodyStream)
    Dim intBytes As Long

    Try
        ' We need to convert the Long to an Integer as the Reader only
        ' supports Integers. This is enough for us as an Integer can now
        ' handle numbers greater than 4 Million (Message limit is 4Mb)
        Dim intLength As Integer = _
            CType(originalMessage.BodyStream.Length, Integer)
        ' Copy one stream to another
        objNewMessage.BodyStream.Write(objReader.ReadBytes(intLength), 0, _
            intLength)
        With objNewMessage
            .AdministrationQueue = New _
                MessageQueue(Constants.MQ.LocalAckQueue)
            .AcknowledgeType = AcknowledgeTypes.FullReceive Or _
                AcknowledgeTypes.FullReachQueue
            .TimeToBeReceived = New TimeSpan(0, 0, 40)
            .TimeToReachQueue = New TimeSpan(0, 0, 10)
            .UseJournalQueue = True
            .Label = originalMessage.Label & "(Copy)"
        End With
        Return objNewMessage
    Finally
        ' Ensure we close any resources used
        objReader.Close()
    End Try
End Function

```

Once we have copied the message object to another as returned by the copyMessage function, we are able to resend it as if it were created for the first time.

WMI

This will be the shortest section of WMI in the book as there is actually no support for MSMQ. Well that's not completely true as there are a few performance counters that relate to MSMQ which are:

- * Win32_PerfRawData_MSMQ_MSMQQueue
- * Win32_PerfRawData_MSMQ_MSMQService
- * Win32_PerfFormattedData_MSMQ_MSMQQueue
- * Win32_PerfFormattedData_MSMQ_MSMQService

You will have to make use of the managed classes within the framework which has been the topic of the last two chapters.

Summary

The past two Chapters have introduced a lot of material which if mastered will produce more reliable and responsive systems. This Chapter covered the more advanced topics relating to MQ's where we learned that MSMQ offers many methods to monitor the progress of our messages throughout the system, and even allows us to know when we should resend a message because of a failure. We also went into some detail on how transactions are handled both internally (simple method) and externally using COM+ and DTC. As was mentioned earlier, the topic of Message Queues and MSMQ is large and I have not covered every aspect which deserves attention as this is not really a book on MSMQ. However, the knowledge you have now will allow you to start writing quite comprehensive MSMQ applications and will hopefully help you understand the rest of the documentation that comes with .NET.

Resources:

MSMQ Home: http://msdn.microsoft.com/library/default.asp?url=/library/en-us/msmq/msmq_overview_4ilh.asp

MSMQ Triggers: http://msdn.microsoft.com/library/default.asp?url=/library/en-us/msmq/msmq_about_triggers_114j.asp.