Chapter 4

Message Queues

In today's environment we are increasingly subject to a greater degree of disconnection from our data sources. This may be for a number of different reasons, such as not being physically connected in the case of mobile users. Alternatively we may be using a less reliable networking infrastructure such as the internet. Although we live in this environment, we need a solution to account for missing or dropped connections to our data source. One possible solution to this dilemma is the use of Message Queues which allow for guaranteed delivery of messages or data, enabling your applications to work in a connected or disconnected environment. The technology provides many other important features as well and I will cover those throughout the discussion.

I will be covering this topic over the next two Chapters as it deserves the extra attention. Although I'll be using two Chapters, there is still much that could have been written, however the aim is to provide you with the knowledge you need to implement the majority of the tasks related to Message Queues.

Message Queues 101

Given that MSMQ (Microsoft Message Queue) is probably fairly new to a lot of you, I'd like to begin by defining a number of terms that you should understand to ensure proper use of this technology.

* **MSMQ:** According to Microsoft it is a development tool that includes a store and forward protocol model to be used for developing messaging applications. MSMQ is also part of Microsoft's Distributed interNet Application (DNA) methodology and is designed to be used as part of a distributed application. Specifically, MSMQ is designed to help developers add a communications infrastructure to distributed applications. MSMQ also helps solve two problems:

    1. Unavailability of communications with servers or clients, and
    2. Communications between disparate systems and components.

* **Asynchronous:** A process within a multitasking system whose execution can proceed independently, "in the background". Other processes may be started before the asynchronous process has finished, thereby not having to wait for a result. The actual processing may be performed on a separate machine at a time long after it was initiated (similar to setting up a batch).

* **Synchronous:** A synchronous process requires that each end of an exchange of communication respond in turn without initiating a new process. This means that when a call is made synchronously, the process will wait or stop until the process has completed before going onto the next process (or step). This is the exact opposite of asynchronous where the first process doesn't wait for the calls completion. Therefore each call requires a response to the previous call before a new one can be initiated. Most of the programs we write call functions to perform certain tasks. When you call a function in your program your call is said to be "synchronous" as the execution of the calling code only resumes after the called function has exited.
* **Message Queue:** An area where you can add messages representing a task (such as a sales order) where it will remain until an application receives it, which does not necessarily need to be the one that put it there. The application may also be on a different server or even a different operating system on the other side of the world.
* **MSMQ:** This is Microsoft's version of Message Queuing, and the one we will be using throughout this Chapter. However, IBM also has a popular product known as MQSeries which does basically the same job. Although it is possible to 'bridge' the two products, it is outside the scope of this Book.
* **Private Queues:** These are queues that reside on your local computer, which most of the examples will use.
* **Public Queues:** These queues can be either on your local computer or normally on a central server. You must be running Windows NT Server 4.0 or higher to create these however.

Given that we will only be using Microsoft's Message Queuing, I'll refer to it as MSMQ and Message Queues simply as MQ from this point on.

A Simple Analogy

Now that we have covered the more academic definition let's consider how they relate to the real world, by looking at a simple analogy making use of our new terminology to reinforce the ideas. Assume we are in the situation where we need to inform a customer that their bill is overdue. You have several ways in which you can communicate this to them.

Your first option might be to make a phone call, which is a synchronous process that has several steps. These steps are; Pickup the phone, Dial the phone number, Wait for caller to respond by answering the phone, Begin speaking, Hang up the phone when call is finished. This process is synchronous because you can't start speaking until they have answered the phone, in other words each step requires the previous step to have been completed first. If you are unable to complete any one of the steps, such as they never answer the phone (they know you're calling using caller ID) then you can't complete the process. The advantage of this synchronous method is that you are immediately aware of the success or failure of the process. The disadvantage is that you are unable to perform any other tasks during this time, which if they answer may be lengthy.

Your second option is to send them a letter via the postal service (I hear people still use this method). This would be an asynchronous process which also has several steps. These steps are; Write letter, Address the letter, Send the letter, Postie picks the letter up, Postal Service sorts the letter, Postie delivers the letter, Receiver opens and reads the letter, Receiver rips the letter up while mumbling 'I'm not paying this!'. This process is synchronous until we send it (ie you can't send a letter you haven't written or addressed), but it then becomes asynchronous from this point. Once we send the letter (Post Box = Public or Private Message Queue) we can start doing other things while another process (postal service = MSMQ) continues the processing, which is then delivered to the intended recipient (customer opens mail box = custom application reads destination queue) who then reads it. The advantage here is that we are able to continue doing other work even though the process hasn't finished yet. The disadvantage however is that we don't know if the process was successful, that is if the letter was delivered correctly or if the receiver actually read the letter. We can overcome some of these disadvantages by using registered mail which will at least tell us if the letter was delivered correctly, but it can't tell us if it was read. MSMQ has several ways of overcoming most of the limitations of the asynchronous model as well which we discuss later in the Chapter.

Using another option such as email is very similar to the traditional postal service, in that from the moment you send your message it becomes an asynchronous process relying on the receiver to regularly check their mail. Again some email systems have features that allow acknowledgement of receipt and acknowledgement of the message being read.

The Mechanics

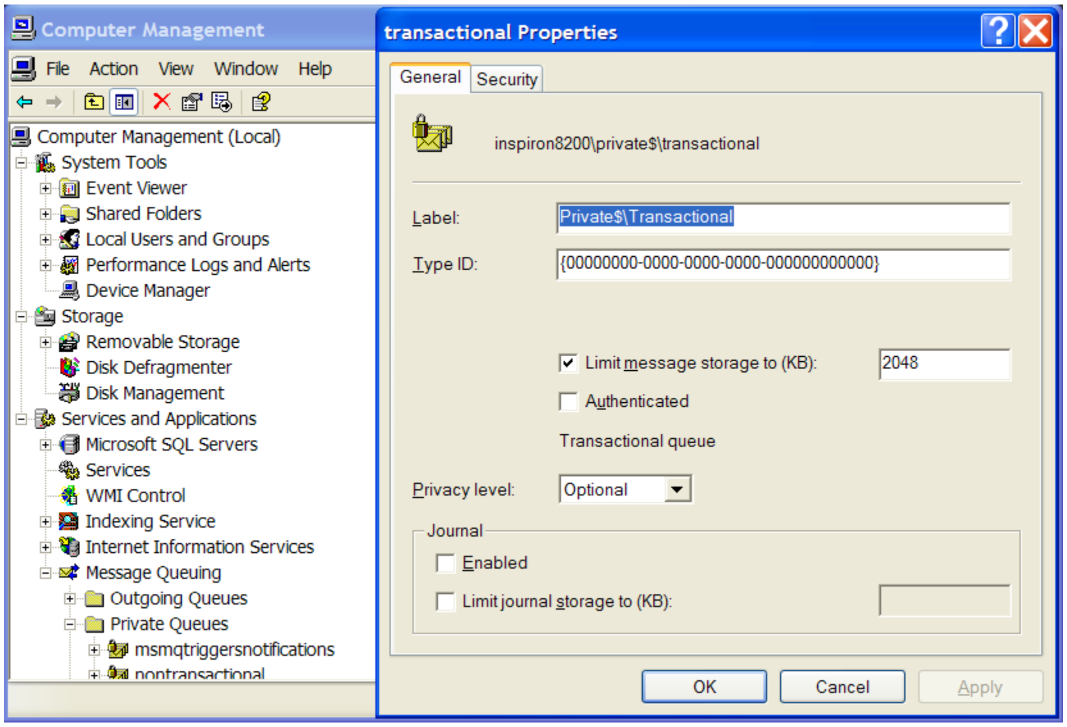
MQ applications also communicate by using messages, much like email and the postal service. A message can be almost any type of data, including simple text or more complex binary data. Messages can be used to communicate between different computers or on the same computer. MSMQ also supports transactional messages which like their database equivalents can be used to ensure that a unit of work is carried out as an atomic operation or put another way; everything succeeds or fails as a whole. Transactional messages can also be used to ensure that a message is only delivered once, and to ensure that all messages are delivered in order. You can even have positive and negative acknowledgment messages sent to confirm that messages have reached or were retrieved from the destination queue.

MSMQ supports two types of delivery method: express and recoverable. The choice between the two comes down to a matter of trading performance and resource use for reliability and failure recovery. Express messages use fewer resources and are faster than recoverable messages. However, express messages cannot be recovered if the computer which is storing the message fails, because with the express method the messages are only stored in memory and are not written to disk. Recoverable messages on the other hand use more resources and are slower than express messages, but they can be recovered should something go wrong with any of the computers involved, such as a failure; because the messages are stored on disk not just in memory. For this reason, if you are building an application that requires that a message be delivered (which would be most of the time) then it's highly recommended that recoverable messages be used.

Types of Queues

MSMQ supports both public and private queues which can be used to send and retrieve messages. These queues can be manipulated in the same way, with only the connection string being different. Below is a summary of the other types of queues supported.

* **Private Queues:** Used to store user defined queues which are only available on the local machine. This is where you would normally do your testing.
* **Public Queues:** Used to store user defined queues which are available to the enterprise. You can only create these queues with one of the Server operating systems such as Windows NT 4.0 Server, Windows 2000 Server or .NET Server.
* **Outgoing Queues:** You can monitor sent messages here on the local machine for both public and privately queued messages.
* **System Queues:** – These queues are used by the operating system and include Journal messages, Dead-letter messages and Transactional Dead-letter messages.



inspiron8200\private$\transactional

Label: Private$\Transactional

Type ID: {00000000-0000-0000-0000-000000000000}

[✓] Limit message storage to (KB): 2048

[ ] Authenticated

Transactional queue

Privacy level: Optional

Journal

[ ] Enabled

[ ] Limit journal storage to (KB):

Figure 4-1. Message Queue Properties

MSMQ also supports the ability to limit the number of messages by the space that is used to store them. You can access this quota on the general tab of a given queues properties, under the option of 'limit message storage to (KB)' in Computer Management; see Figure 4-1. Alternatively you can use VS.NET, by using the properties window of a given queue. When a queue's quota is reached, messages can no longer be sent to the queue until one or more messages are removed from the queue. MSMQ routes messages to queues based on the message priority.

Caution: When a computer's quota is reached, messages can no longer be sent to any queues on the computer until one or more messages are removed from one of the queues, or more space is made available to the computer. This however does not apply to System Queues.

Why use Message Queues?

Message Queuing allows developers to write applications that can communicate with each other quickly and reliably by the use of sending and receiving messages. This allows your application to send off a message for another application to process while the current application returns back to the user. With the correct properties set, messaging also provides you with guaranteed message delivery and a robust, fail-safe way to carry out many of your business processes. An example of which might be an order system that must run 24x7. Assume that the database or the connection between the client and server fails. With a traditional client/server application, you would not be able to collect any more sales. However, if the client application made use of messages when a connection to the server was unavailable, then orders that couldn't be processed would stay in a message queue, and when the connection was restored, the messages would automatically be sent off to the backend system for processing. This would therefore minimize the possible effect of any system failure. The Microsoft documentation identifies the following key points about MSMQ and messaging in general.

* **Robustness:** Messages are considerably less affected by component failures than direct calls between components, because messages are stored in queues and remain there until processed appropriately. Messaging is similar to transaction processing, because message processing is guaranteed.
* **Availability:** Queuing provides an availability design benefit: by increasing the number of routes for successful message delivery, an application can increase the chances for successful and immediate message completion. A subtle benefit of queuing is that its store and forward, guaranteed delivery, and dynamic routing features appear (to customers) to increase the availability of your application.
* **Message prioritization:** More urgent or important messages can be received before less important messages, so you can guarantee adequate response time for critical applications.

* **Offline capabilities:** Messages can be sent to temporary queues when they are sent and remain that way until they are successfully delivered. Users can continue to perform operations when access to the necessary queue is unavailable for whatever reason. In the meantime, additional operations can proceed as if the message had already been processed, because the message delivery is guaranteed when the network connection is restored.
* **Transactional messaging:** You can couple several related messages into a single transaction, ensuring that the messages are delivered in order, delivered only once, and are successfully retrieved from their destination queue. If any errors occur, the entire transaction is cancelled.
* **Security:** The Message Queuing technology on which the MessageQueue component is based uses Windows security to secure access control, provide auditing, which encrypt and authenticate the messages your component sends and receives.

Taking this into account then, if you are writing a web application and you need to run a long process in response to a user action, you probably don’t want to make the user wait until the process is complete. In these situations you would prefer to return back to the user immediately by running the process asynchronously. Although there are numerous ways to run code in an asynchronous manner, few give the level of robustness that MQ’s do.

As good as MQ’s are they are not always the correct choice for all business situations, due to their business model needs. Some processes require synchronous processing, such as getting an account balance. The user will wait until the process has completed as they need to see the result. However, if the same user asks for a new statement to be sent out, then the actual processing of the request doesn’t have to be done immediately and so can be queued for a batch later that day or night. This ability to process requests asynchronously allows our applications to scale much better as we can return to the user very quickly while the time consuming processing is being done either at a later time or in the background.

There are times when you can use asynchronous processing, but make it look like everything is synchronous. Let’s say you have an application that allows you to buy tickets for sporting events online. You may have a series of questions or steps to fulfil the order, one of which is to check the availability which can take up to 30 seconds. You could ask for the type of ticket they want then wait for the availability check to complete. This would mean the user is waiting while the check is being done. Alternatively you could fire off a message to a queue and return immediately, and then ask the next question which might be the postal address. Meanwhile the first request is being processed in the background asynchronously. Then by the time the user has completed the second step the availability check has completed and the results are waiting in a database or some other accessible place. There might of course be times when the data hasn’t been retrieved yet, in which case you might notify the user and ask them to try again or just wait for the result to return. The end result is that you can achieve better response times by designing your application to work with asynchronous processing.

What are my Options?

Before we get into the details of how to use Message Queues it's worth finding out what the options are, and how best to chose an option for a particular application. Here I'll present the different options available, and later we will examine how to implement these options.

MQ's can be used in a very simple way or they can be used in fairly complex ways. Which way you use them really depends on the needs of your application, how much time you have for coding and your knowledge of MQ's. Its unfortunate that two Chapters on MQ's can't really do them justice, however by the end of the Chapter you will know enough about MQ's to handle most of the situations in which you would want to use them. I do however encourage you to check out the SDK and other resources at the end of the Chapter. Our focus for this Chapter and the next will be on the following items which comprise the majority of what MQ's have to offer.

* **Simple:** This is where you basically use the MessageQueue objects Send method, and use the default options that control how a message is sent.
* **Complex:** Here you have more control on how a message is sent and can even determine how you want the message formatted i.e. XML or binary. This is achieved by creating an instance of the Message object and then sending this object which contains the user's message.
* **Recoverable:** By setting a property on the message object you can ensure that the message isn't lost should a failure occur such as a computer crash.
* **Transactional:** This allows you to send several messages as a unit of work or participate in a DTC (Distributed Transaction Coordinator) transaction.
* **Acknowledgements, Time Outs, Journaling and Dead-letters:** As MQ's are asynchronous it would be nice if we could tell if the message was received and also whether it was processed. These system messages allow us to do this and much more.
* **Server Explorer:** The VS.NET IDE as mentioned before has some RAD features, one of which is the ability to drag and drop MQ's into your application. This can reduce the amount of code you need to write.
* **Hand Coding:** As with everything in .NET you can get down and dirty and code everything from the ground up.
* **Queued Components:** Built upon MSMQ v2,0, Queued Components allow the asynchronous execution of COM component methods using MSMQ and COM+. This is outside the scope of the book however.

* **MSMQ Triggers:** Built upon MSMQ 3.0 MSMQ Triggers are associated with specific queues on a computer and are invoked every time a Message Queuing message arrives at such queues. A trigger is comprised of one or more rules. These rules are defined by actions that will be invoked when all conditions associated with a rule are true. This technology is also specific to COM components and doesn't have any direct .NET support. Again, you can use this technology using COM InterOp, but this won't be covered as this falls outside the Books scope. Example 4 demonstrates how you can build a managed code alternative to both Queued Components and Triggers.

Knowing how to make use of these different options will allow you to make the correct decision when it comes time to implement them in your own applications. Next I will provide some guidance on determining which option to use.

Choosing the correct option

We have detailed the different options available, however how do we know which options to use in any given situation? This section will discuss each of the available options, and when you might like to make use of them. They are of course only suggestions, and you will ultimately need to consider your applications overall architecture when making a decision.

Simple

As the name suggests, the simple method is just that, although you do have access to many of the features of the more complex method, by using the DefaultPropertiesToSend object, which can give you a good level of control. However, each message will use these settings, so if you need to be more specific on a message by message basis, then it's not a good option for your application. In general it's best not to use the simple method, unless you only need to send simple text messages through the system. When you start dealing with the more advanced features such as acknowledgements etc, you will need to become familiar with the Message object anyway.

Complex

The complex method is most suited to situations where you need to have more control over the sending of the message, including its format, acknowledgments, and journaling etc. Although you can still achieve this using the simple method, you don't have the same level of control over each message sent. The only thing that separates the simple from the complex is that you create an instance of the Message object and then apply the messages contents and any other required properties. Given the subtle difference, and that you need to use the Message object when receiving a message, there is no real reason to make use of the simpler version.

Recoverable

If you need the robustness and fail-safe features that MQ provides, then you must take advantage of recoverable messages. Although there is a slight overhead with using them, this is far outweighed by their usefulness. The only occasion where you may not wish to use recoverable messages is when it is not critical that a message be received, or speed is paramount and you use other methods to ensure delivery if required.

Transactional

Transactional messages are very good if you need a group of messages to be processed in order and as a single unit of work. You may for example need to update several systems for a given users action. Each update could be its own message, all of which are contained within a single transaction. Assuming that the programs that do the updates support MSDTC (Microsoft Distributed Transaction Coordinator) then should any of the message updates fail, all of the work will be rolled back. There is however some overhead in using transactions of any type, so if you don't require this functionality it's best not to use it.

Acknowledgements, Time Outs, Journaling and Dead-letters

Given the nature of asynchronous processing, using methods to identify the status of a message is critical for most applications. Although you can get away without making use of these features, you may find that it becomes difficult to monitor your system if they are not used. Implementing these features can take a bit more effort, but the rewards are that you have a lot of extra control over how failed messages are handled and you can use notifications to ensure your messages have reached their intended targets.

Server Explorer

The Server Explorer as detailed later allows you to create message queue applications quickly and easily. However, the draw back to using the GUI tool is that, it's not really doing very much behind the scenes for you. Also it hard codes the details into the designer area of your code behind, which can make debugging more difficult. However, the biggest drawback is if you want to create a class that uses messages queues. The GUI tool will only work with WinForm's and Component Classes, which unfortunately is not where you should be putting this type of code in the first place.

Hand Coding

I have a general bias towards hand coding for things that don't require a lot of code to begin with. I feel that MQ falls into this category, and although it is nice to make use of the GUI tools when you're starting out (which I would recommend), when you start doing more advanced work, you'll find that the GUI really doesn't offer us much except to maybe highlight visually that there's a MessageQueue on our form. Therefore, unless you're just starting to get to grips with message queues or you want to knock something up fairly fast, I'd be inclined to hand code it.

Queued Components

Although we won't be covering the details of how to use and program Queued Components, it is probably necessary to understand why you might want to use them. If you need to work with COM components then Queued Components can be a good way to go. If you have to code in VB6 then QC's become an even better choice. There is a bit of work to get QC's working correctly and they will require Windows 2000 or better on all machines involved.

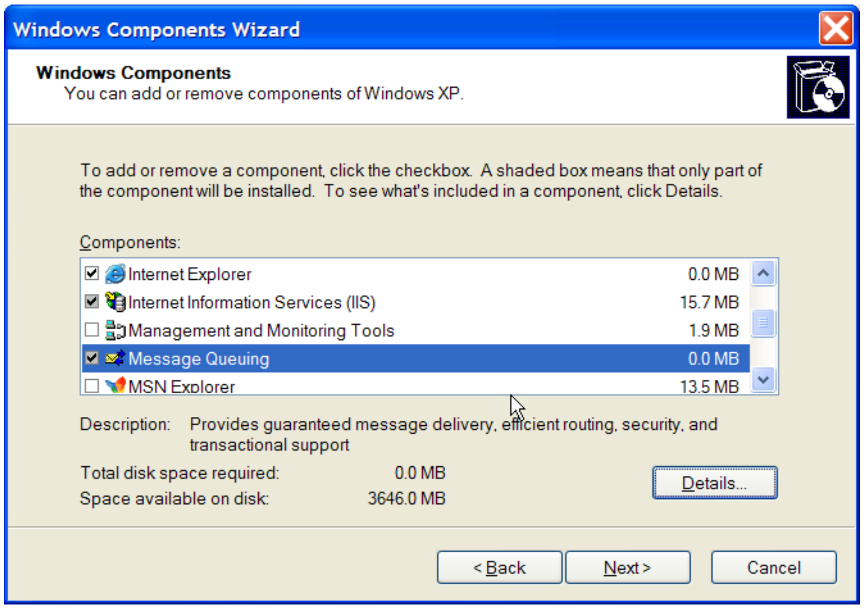
MSMQ Triggers

Although we won't be covering the details of how to use and program MSMQ Triggers either, it is probably necessary to understand why you might want to use them. Again, if you need to work with COM components and want to execute them based on rules then Triggers can be a good way to go. If you have to code in VB6 then Triggers again become an even better choice.

Installing MSMQ

Now that we have an idea of what MSMQ is, before we can start to use it we need to ensure we have it installed on our computer. MSMQ is not installed by default and you will need to perform the following installation procedure if you haven't already done so.

MSMQ is part of the Microsoft Windows NT 4.0 Option Pack, Microsoft Windows 2000, Microsoft Windows XP and .NET Server. Note again that if you are not running a server version of these operating systems then you will only have the ability to create private (local) queues.


Windows Components Wizard
Windows Components
You can add or remove components of Windows XP.

To add or remove a component, click the checkbox. A shaded box means that only part of the component will be installed. To see what's included in a component, click Details.

Components:

✓	Internet Explorer	0.0 MB
✓	Internet Information Services (IIS)	15.7 MB
☐	Management and Monitoring Tools	1.9 MB
✓	Message Queuing	0.0 MB
☐	MSN Explorer	13.5 MB

Description: Provides guaranteed message delivery, efficient routing, security, and transactional support

Total disk space required: 0.0 MB
Space available on disk: 3646.0 MB

Figure 4-2. Windows Component Wizard

To install on Windows 2000 or Windows XP you need to go to Control Panel➢Add or Remove Programs➢Add/Remove Windows Components where you'll see Figure 4-2. To install you simply need to select Message Queuing, click Next, then Finish. Once you have installed MSMQ you will have access to it through the Server Explorer in Visual Studio.NET.

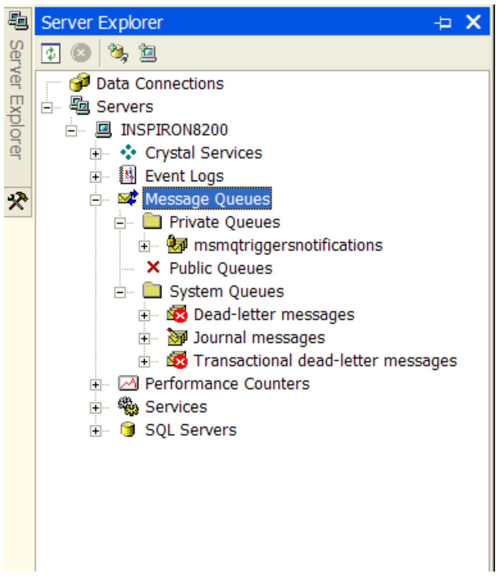


Figure 4-3. Server Explorer with MSMQ Installed

You'll notice that in Figure 4-3, my public queues appear to be disabled, this is because my local machine is running Windows XP Professional which isn't a server product and so public queues are not available for my local machine. As most of you will also be using desktop operating systems as your local machine you will also see this.

MessageQueue Object

Here we start to get our hands dirty and start looking at some code! We will present each of the different options explained previously using both the IDE and hand coding where appropriate. By the end of this section you should have a clear idea of how to implement a message queue for you particular situation.

Working with a MQ is somewhat similar to working with a file in that you need to take the following steps:

* Figure out which queue to use for our messages (identifier/name)?
* Connect to the required queue (open).
* Send/Receive messages (write/read)
* Disconnect from the queue (close)

The MessageQueue object controls access to all MQ's throughout a network. Using different connection strings as discussed next we can attach to any queue that is visible to our computer. We will be discussing in-depth how to send and receive messages of different types and also some very useful administration features that MQ's offer.

Queue to connect

Before you can start using a MQ you must first connect to it. In order to be able to connect to it you must be able to uniquely identify that queue from anywhere in the enterprise. Once you have identified the queue you must ensure that it is also consistently referenced. .NET provides us with three different ways of doing this:

* **Path:** By using the queues path <server name>\<queue type>\<queue name>. An example of which would be inspiron8200\private$\NonTransactional. Note that if you are using your local machine you can use a "." as the server name as this will default to the local computer. I've elected to do this for the samples to make installation easier.
* **Format Name:** This identifies the queue by using the connection details combined with the queues path, or you could use the GUID that is generated after the queue was created. The format is DIRECT=OS:<server name>\<queue type>\<queue name>. An example of which would be DIRECT=OS:inspiron8200\private$\NonTransactional.
* **Label:** This allows you to specify a descriptive text which you can then use to identify the queue, however as this text may not be unique it will cause an error if there are duplicates. The prime reason for referencing in this way is if you think you are about to move the queue from its current server. However, given the possibilities of failure and the more robust options available to solve this issue (discussed in Chapter 9), it's not a course of action I'd recommended.

Tip: If you intend to send messages to a disconnected queue, you must refer to the queue by *format name* rather than path, as you will need access to your domain controller to have the path resolved, which will be unavailable offline. Also both Label and Path options are resolved at runtime to a Format Name so there is a little overhead in using these options as well.

Once we know the specified queue name we can use it to create an instance of the MessageQueue class object that represents any queue. You will find this class in the System.Messaging namespace which needs to be added both as a reference, and as an alias at the top of your class, using the Imports keyword. If you make use of the RAD features in VS.NET (see Example 1b) then you won't need to worry as the IDE will add these references for you.

Generally you would make use of constants defined at the top of the class or read them from a configuration file when referencing your queue paths. I've decided to go a slightly different route, which allows us to do either. I've created a Constants namespace which has a class MQ with public constants that define the constants that I'll be using throughout the Chapter. This technique makes the code much easier to understand and also guarantees that I use my constants consistently as I only define them once. So to refer to the LocalNonTransQueue, we simply put in our code Constants.MQ.LocalNonTransQueue. Now when you read the code it's very obvious what type of queue you're using. When you start using this in your own code, you can make the constants more meaningful based on how you intend to use them. You can also extend this approach to make them more intelligent by converting the constants into shared functions and referencing databases or configuration files to get the path details if necessary; this approach doesn't affect the consumer of the constant:

```
Namespace Constants
    Class MQ
        Public Const LocalNonTransQueue As String = _
            ".\private$\NonTransactional"
        Public Const LocalTransQueue As String = ".\private$\Transactional"
        Public Const LocalAckQueue As String = ".\private$\Ack"
        Public Const PublicNonTransQueue As String = _
            "webserver\NonTransactional"
    End Class
End Namespace
```

When you go to run the samples yourself, you need only change the PublicNonTransQueue value as this is the only one which is currently hardcoded to an external server. This constant is only used in Example8b when we discuss Journaling. So to open a connection to a given queue, we code the following:

```
Imports System.Messaging
Sub Main()
     Dim objMessageQueue As New MessageQueue(Constants.MQ.LocalNonTransQueue)
End Sub
```

MessageQueue.Send

Now that we know how to connect to a MQ, we are now able to send a message using the Send method. However, before we can do this, we have to ensure that the destination queue exists, otherwise an exception will be thrown when we try to send something to it. We will cover later how to do this programmatically, but for now lets do it manually so we know how it's done.

Firstly using the Server Explorer window click on Private Queues, then right click and choose create queue as in Figure 4-4. Type in the name of the queue you want, which in this case is NonTransactional then click OK. You have now created a queue which can now be used to send and receive messages.

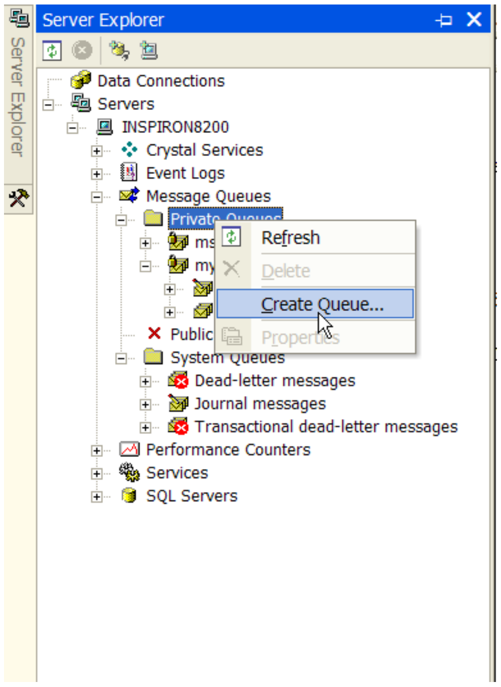


Figure 4-4. Creating a Message Queue

Now that we have our queue ready to use, let’s examine the Send method, which as we can see has several overloaded parameters.

Table 4-1. Overloaded Send Method

Method	Description
Send(object)	Sends an object to a non-transactional queue referenced by this MessageQueue.
Send(object, label)	Sends an object to the non-transactional queue referenced by this MessageQueue and specifies a label for the message.
Send(object, transaction)	Sends an object to the transactional queue referenced by this MessageQueue.
Send(object, label, transaction)	Sends an object to the transactional queue referenced by this MessageQueue and specifies a label for the message.
Send(object, transactionType)	Sends an object to the queue referenced by this MessageQueue.
Send(object, label, transactionType)	Sends an object to the queue referenced by this MessageQueue and specifies a label for the message.

Simple Method

As can be seen the Send method allows us to send messages in many ways, the simplest of which is to supply an object which could be any serializable object such as a string. This forms the basis for a simple send, however we can also send a specific Message object, which forms the basis of the complex send. The following examples will examine each of these methods in detail.

Having identified how to connect to a queue we can now apply the Send method. The easiest of the methods is identified in Example1, which shows how to use it from both the IDE and hand coding perspectives.

To try and make using the examples from the download easier I have wrapped them all into the one project and attached a simple main Form from where you can test the samples (see Figure 4-5). I won't be going into detail on how to create this form as I would expect you all know how to create a simple WinForm, so I won't insult your intelligence by explaining how to do it.

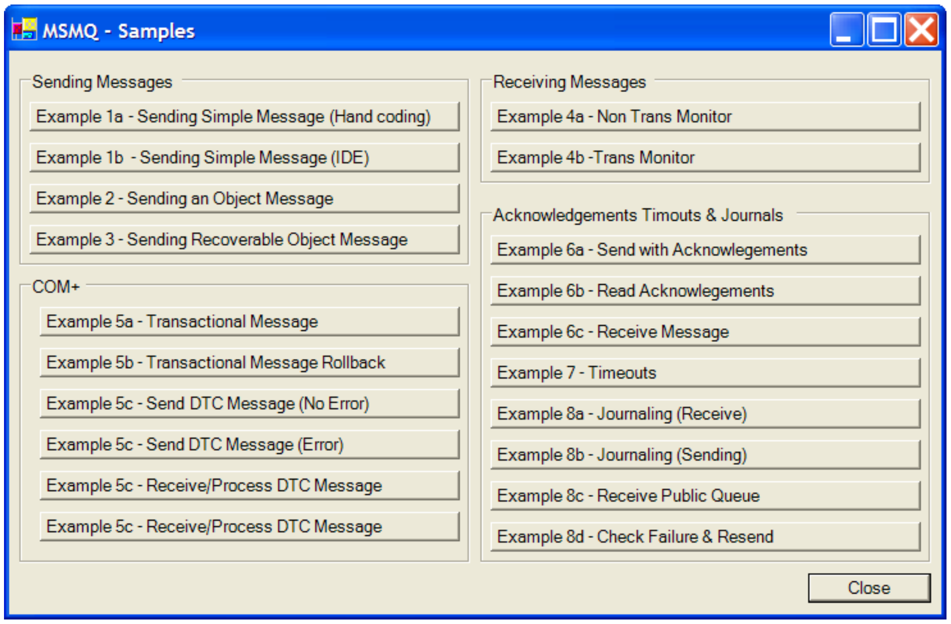

MSMQ - Samples
Sending Messages
Example 1a - Sending Simple Message (Hand coding)
Example 1b - Sending Simple Message (IDE)
Example 2 - Sending an Object Message
Example 3 - Sending Recoverable Object Message
COM+
Example 5a - Transactional Message
Example 5b - Transactional Message Rollback
Example 5c - Send DTC Message (No Error)
Example 5c - Send DTC Message (Error)
Example 5c - Receive/Process DTC Message
Example 5c - Receive/Process DTC Message
Receiving Messages
Example 4a - Non Trans Monitor
Example 4b -Trans Monitor
Acknowledgements Timouts & Journals
Example 6a - Send with Acknowlegements
Example 6b - Read Acknowlegements
Example 6c - Receive Message
Example 7 - Timeouts
Example 8a - Journaling (Receive)
Example 8b - Journaling (Sending)
Example 8c - Receive Public Queue
Example 8d - Check Failure & Resend
Close

Figure 4-5. Samples Main Form

Example 1a – Sending a Text Message (Hand coding)

The full code listing is presented, and as you can see it doesn't take much to send a message in .NET. We are using a class here to make the code listings easier to follow. One thing to note is the use of the shared member. I've done this to make executing the class' code easier by not having to instantiate the object first, although you don't have to do this:

```
Imports System.Messaging
Imports System.Diagnostics


Public Class Example1a
    Shared Sub Main()
        Dim objMessageQueue As MessageQueue
        Try
          objMessageQueue = New MessageQueue(Constants.MQ.LocalNonTransQueue)
```

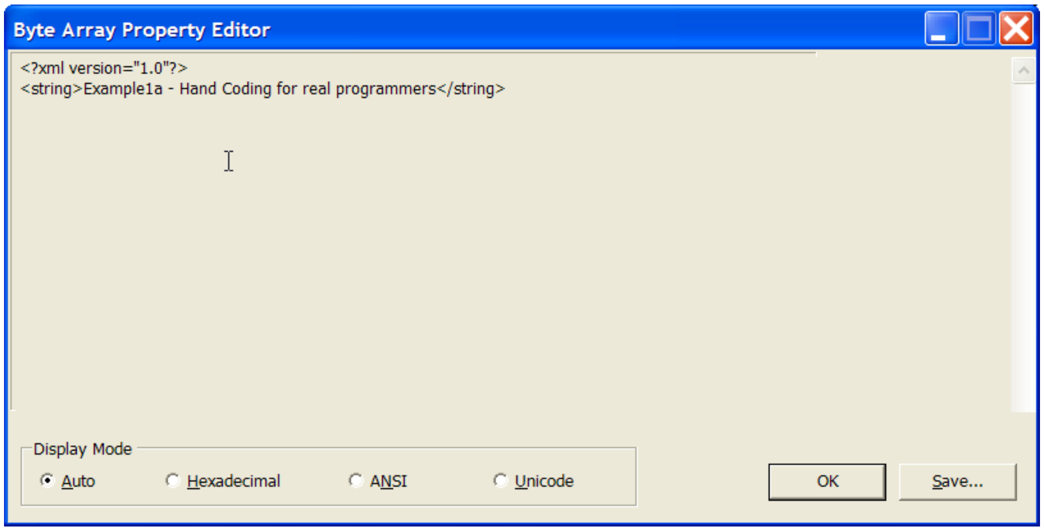
```
      objMessageQueue.Send("Example1a - Hand Coding for real programmers", _
        "Example1a")
      MessageBox.Show("Message Sent")
    Catch ex As Exception
      MessageBox.Show(ex.Message)
    Finally
      objMessageQueue.Close()
    End Try
  End Sub
End Class
```

The only difference here is that we have added the line that executes the Send and also put a bit of error trapping in which is always a good idea. We use the Send method that takes the object (*message*) and the message *label* as arguments. As you can see there's pretty much nothing to sending a simple message in .NET.

Note: In order to keep the examples as specific to the tasks at hand I'll be omitting the error handling and other non-critical duplicate code from the printed listings, as it will make understanding the new code easier. All the code and some error handling will be contained in the downloadable version for each sample however.

If you run this sample you should see that a message has appeared under Message Queues➢NonTransactional➢Queue messages in the Server Explorer in VS.NET. Now it might be less than obvious how you see the contents of the message. You can see what the content of the message is by selecting the message and then clicking Properties➢BodyStream➢… and it should look like Figure 4-6. You'll notice that the output isn't straight text as you might have thought, but rather an XML document. Later you'll learn how to change this behavior.

Tip: When you send a message the VS.NET IDE doesn't automatically update the message list. You need to right click on the queue and select Refresh. This will repopulate the list of messages on that queue.



```
<?xml version="1.0"?>
<string>Example1a - Hand Coding for real programmers</string>
```

Display Mode: [✓ Auto] Hexadecimal ANSI Unicode

Figure 4-6. Contents of a sent message

Example 1b – Sending a Text Message (IDE)

Now in Example1a we had to write a bit of code, however in this example we will be using a WinForm application and make use of the IDE. Using the IDE gives us two options when dragging and dropping onto the form. We can either drag the queue we have already created from the Server Explorer straight onto our form or we can drag a MessageQueue component from the components section of our toolbox onto our form.

If we use the Server Explorer then we don't have to set all the properties for the Path etc, as they will be automatically set for us. Therefore, in this example we will simply drag the NonTransactional MQ we created earlier straight onto the form. As the default name isn't very good we will also rename it to IDESimple, which isn't a name I'd recommend in a production environment, however I'll make my names a bit better as we go along.

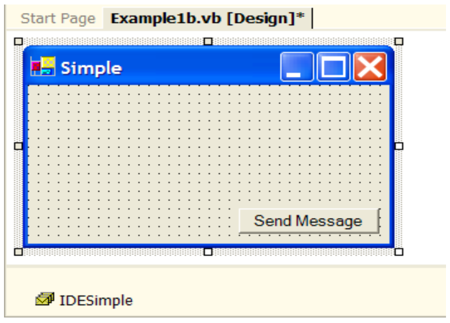


Figure 4-7. Simple Message

Now we have our component on our form nicely named and ready to code against. To send the message we will need to put a button on the form which I've named btnSendMessage, doubling clicking on the button we can add this line of code.

```
Me.IDESimple.Send("Example 1b - IDE Coding's faster!", "Example1b")
```

Now when we run the code we will see that another message has appeared with the contents we just sent. All this and it only took us one line of code!

Complex Method

Earlier we looked at the easiest way to create a MQ and send a message. Although this may be suitable in some cases, most of the time you need to have more control over how a message is sent. The alternative is to create an instance of the System.Messaging.Message class, and send that instead of a simple string. This class represents a single message which has many properties which you can alter, Table 4-2 shows some of the properties which you can change.

Table 4-2. Extract of the Message object Properties (*source: .NET Framework Class Library documentation*)

Property	Description
AcknowledgeType	Gets or sets the type of acknowledgment message to be returned to the sending application.
AdministrationQueue	Gets or sets the queue that receives the acknowledgement messages that Message Queuing generates.
Body	Gets or sets the content of the message.
BodyStream	Gets or sets the information in the body of the message.
BodyType	Gets or sets the type of data that the message body contains.
EncryptionAlgorithm	Gets or sets the encryption algorithm used to encrypt the body of a private message.
Formatter	Gets or sets the formatter used to serialize an object into or deserialize an object from the message body.
Label	Gets or sets an application-defined Unicode string that describes the message.
Priority	Gets or sets the message priority, which determines where in the queue the message is placed.
Recoverable	Gets or sets a value indicating whether the message is guaranteed to be delivered in the event of a computer failure or network problem.
ResponseQueue	Gets or sets the queue that receives application-generated response messages.
TimeToBeReceived	Gets or sets the maximum amount of time for the message to be received from the destination queue.
TimeToReachQueue	Gets or sets the maximum amount of time for the message to reach the queue.
UseAuthentication	Gets or sets a value indicating whether the message was (or must be) authenticated before being sent.

UseDeadLetterQueue	Gets or sets a value indicating whether a copy of the message that could not be delivered should be sent to a dead-letter queue.
UseEncryption	Gets or sets a value indicating whether to make the message private.
UseJournalQueue	Gets or sets a value indicating whether a copy of the message should be kept in a machine journal on the originating computer.
UseTracing	Gets or sets a value indicating whether to trace a message as it moves toward its destination queue.

As we saw earlier when a message is sent it is by default encoded into XML which may not be what we require. The MessageQueue object supports three types of formatters via the Formatter property:

* **XMLMessageFormatter:** (default) Serializes the object into an XML representation suitable for most purposes. Has the advantage of being human readable, which can aid in debugging. It is also good for transmitting over HTTP.
* **BinaryMessageFormatter:** Serializes data into a binary non-human readable format. This will generally result in a smaller packet and is also generally faster than the XML formatter, but it can be harder to debug.
* **ActiveXMessageFormatter:** This is used when you are passing primitive or COM objects to a non-.NET system, such as to a Microsoft Visual Basic 6 application, and shouldn't be used unless this functionality is required.

Note: Regardless of which formatter you decide to use, you *must* use the same formatter when you receive the message, otherwise you won't be able to parse it.

Tip: If you want to pass data in a message which isn't 'formatted', then you can use a technique whereby you stream your message directly into the BodyStream property. This bypasses the need for a formatter as the other formatters only translate the Body data into the BodyStream anyway. However, note that you'll have to read the stream back out at the other end if you use this technique as MSMQ won't know how to deserialize the BodyStream. An example of how this is used is presented in the next Chapter.

Example 2 – Sending an Object

In this example we will be creating a simple Orders object (which would normally have many more properties and methods) to demonstrate that we can send not only simple text messages but full objects. When we send the object, we have three options on the format we use. This example will show how to use both the XMLMessageFormatter and the BinaryMessageFormatter, which are the two options you're most likely to use. In order to serialize an object using the BinaryMessageFormatter or the XMLMessageFormatter you must either implement the ISerializable interface or simply set the Serializable attribute of the class. This example uses the Serializable attribute as it's the easiest. For more information on serializing objects consult the online documentation.

Caution: MSMQ supports messages of up to 4Mb per message, so it is important to understand how large your objects might be when they are serialized, using binary serialization may help if the XML version is too large.

Firstly we will create a new project for our Orders object, so that we can leverage the code in further examples. Create a new project called Orders, after inserting the following code, we have a relatively complex business object to send around. Note you'll need to add a reference to this project in the main project otherwise you won't be able to use it.

```
<Serializable()> Public Class Order
    ' This is a simple object to demonstrate what can be sent
    Private m_strCustomerID As String
    Private m_intOrderID As Integer
    Private m_objOrderDetails As New OrderDetails()

    Property CustomerID() As String
        Get
            Return m_strCustomerID
        End Get
        Set(ByVal Value As String)
            m_strCustomerID = Value
        End Set
    End Property
    Property OrderID() As Integer
        Get
            Return m_intOrderID
        End Get
        Set(ByVal Value As Integer)
            m_intOrderID = Value
        End Set
    End Property

    Property OrderDetails() As OrderDetails
        Get
            Return m_objOrderDetails
        End Get
        Set(ByVal Value As OrderDetails)
            m_objOrderDetails = Value
        End Set
    End Property
    Sub Save()
        ' This method will save the record
    End Sub
End Class

<Serializable()> Public Class OrderDetails
    Inherits System.Collections.CollectionBase

    ' This class implements a strongly typed custom collection
    Public Function Add(ByVal lineItem As LineItem) As Integer
        Return MyBase.List.Add(lineItem)
    End Function
    Public Function Add() As LineItem
        ' No item supplied so will create a new one
        Return Me.Item(Me.Add(New LineItem()))
    End Function
    Public Sub Remove(ByVal index As Integer)
        ' Check to see if there is a widget at the supplied index.
        If index > Count - 1 Or index < 0 Then
            Throw New Exception("Index not valid!")
        Else
            ' Invokes the RemoveAt method of the List object.
            List.RemoveAt(index)
        End If
    End Sub

    Default Public ReadOnly Property Item(ByVal index As Integer) As LineItem
        Get
            Return CType(List.Item(index), LineItem)
        End Get
    End Property
End Class

<Serializable()> Public Class LineItem
    Private m_intProductID As Integer
    Private m_dblUnitPrice As Double
    Private m_intQuantity As Integer
    Private m_dblDiscount As Double

    Property ProductID() As Integer
        Get
            Return m_intProductID
        End Get
        Set(ByVal Value As Integer)
            m_intProductID = Value
        End Set
    End Property
    Property UnitPrice() As Double
        Get
```

```
      Return m_dblUnitPrice
    End Get
    Set(ByVal Value As Double)
      m_dblUnitPrice = Value
    End Set
  End Property
  Property Quantity() As Integer
    Get
      Return m_intQuantity
    End Get
    Set(ByVal Value As Integer)
      m_intQuantity = Value
    End Set
  End Property
  Property Discount() As Double
    Get
      Return m_dblDiscount
    End Get
    Set(ByVal Value As Double)
      m_dblDiscount = Value
    End Set
  End Property
End Class
```

To keep the example as real as possible, I've created an OrderDetails class which uses a custom early bound collection to store multiple LineItem classes. If you are not currently using this technique for your own class collections, I strongly suggest that you start doing so, as it will make your code more robust. You can learn more about this by searching for *Walkthrough: Creating Your Own Collection Class* in the online help or read Dan Applemans book "Moving to VB.NET: Strategies, Concepts, and Code" from Apress Press, which also covers this topic in depth.

Update: If you're now using .NET 2.0 then you should be using one of the generic collection classes which are both faster and type safe.

So now we have an object which we can send around, we now need to know how to actually send the object. The following listing creates an instance of our Orders object, and then populates it with some dummy order data. It then creates a Message object, sets the Body property to be the instance of the Order, sets a few other associated properties, then sends it to the NonTransactional queue. Note that we change how the message is formatted twice to highlight how the message will look using the two different options:

```
Imports System.Messaging

Public Class Example2
  Shared Sub Main()
    Dim objMessageQueue As MessageQueue
```

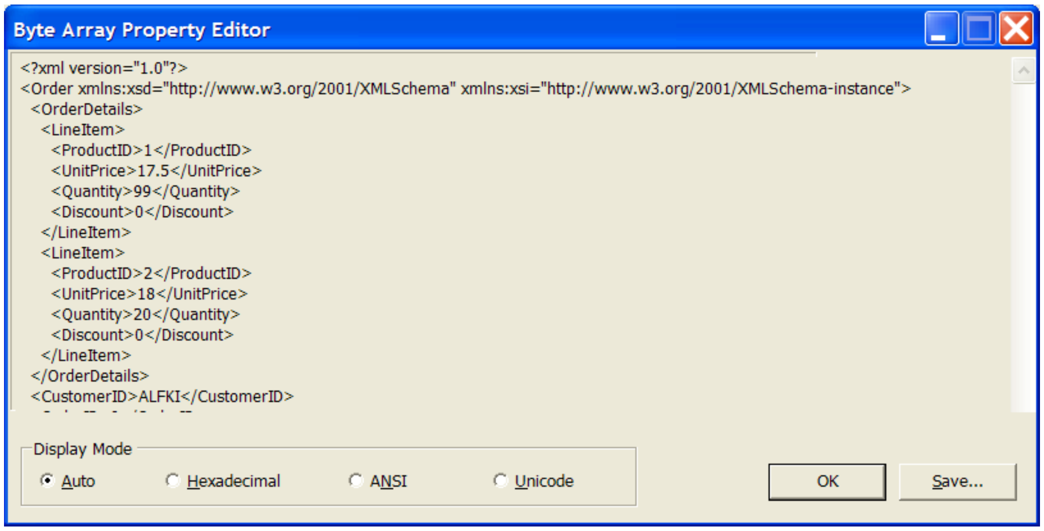
```
        Try
            objMessageQueue = New MessageQueue(Constants.MQ.LocalNonTransQueue)
            ' Create an instance of the simple Order Class and populate
            Dim objOrder As New Order()
            objOrder.CustomerID = "ALFKI"
            With objOrder.OrderDetails.Add()
                .ProductID = 1
                .Quantity = 99
                .UnitPrice = 17.5
                .Discount = 0
            End With
            With objOrder.OrderDetails.Add()
                .ProductID = 2
                .Quantity = 20
                .UnitPrice = 18
                .Discount = 0
            End With

            ' Prepare the message for sending
            Dim objMessage As New Message()
            With objMessage
                .Body = objOrder
                .Priority = MessagePriority.Highest
            End With

            ' Send Message as Binary
            objMessage.Formatter = New BinaryMessageFormatter()
            ' Prefix the Label with its datatype so we can see it clearly
            objMessage.Label = "Binary - Order:" & objOrder.CustomerID
            objMessageQueue.Send(objMessage)

            ' Now Send Message as XML
            objMessage.Formatter = New XmlMessageFormatter()
            objMessage.Label = "XML - Order:" & objOrder.CustomerID
            objMessageQueue.Send(objMessage)
            MessageBox.Show("Message Sent")
        Catch ex As Exception
            MessageBox.Show(ex.Message)
        Finally
            objMessageQueue.Close()
        End Try
    End Sub
End Class
```

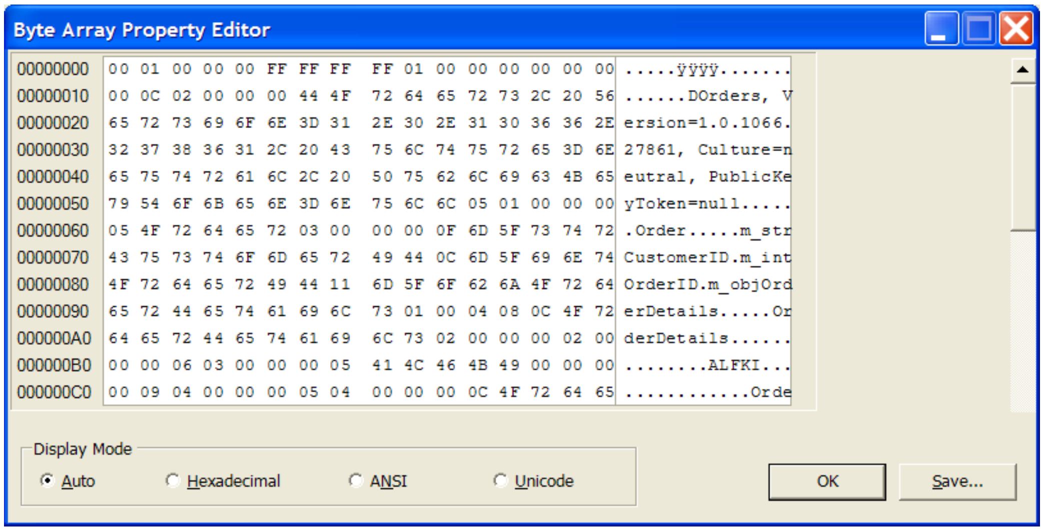
If we now run this sample you will notice that we have two new messages in our NonTransactional MQ. If you take a look at the output of the message as we did before you'll notice that the object has been converted to XML with each property getting its own element see Figure 3-8a. Figure 3-8b however shows the same object when it was formatted as binary, which is for the most part unreadable.



```
<?xml version="1.0"?>
<Order xmlns:xsd="http://www.w3.org/2001/XMLSchema" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
 <OrderDetails>
  <LineItem>
   <ProductID>1</ProductID>
   <UnitPrice>17.5</UnitPrice>
   <Quantity>99</Quantity>
   <Discount>0</Discount>
  </LineItem>
  <LineItem>
   <ProductID>2</ProductID>
   <UnitPrice>18</UnitPrice>
   <Quantity>20</Quantity>
   <Discount>0</Discount>
  </LineItem>
 </OrderDetails>
 <CustomerID>ALFKI</CustomerID>
```

Display Mode: [✓ Auto] Hexadecimal ANSI Unicode

Figure 3-8a. Contents of Message sent as XML



```
00000000  00 01 00 00 00 FF FF FF  FF 01 00 00 00 00 00 00  .....ÿÿÿÿ.......
00000010  00 0C 02 00 00 00 44 4F  72 64 65 72 73 2C 20 56  ......DOrders, V
00000020  65 72 73 69 6F 6E 3D 31  2E 30 2E 31 30 36 36 2E  ersion=1.0.1066.
00000030  32 37 38 36 31 2C 20 43  75 6C 74 75 72 65 3D 6E  27861, Culture=n
00000040  65 75 74 72 61 6C 2C 20  50 75 62 6C 69 63 4B 65  eutral, PublicKe
00000050  79 54 6F 6B 65 6E 3D 6E  75 6C 6C 05 01 00 00 00  yToken=null.....
00000060  05 4F 72 64 65 72 03 00  00 00 0F 6D 5F 73 74 72  .Order.....m_str
00000070  43 75 73 74 6F 6D 65 72  49 44 0C 6D 5F 69 6E 74  CustomerID.m_int
00000080  4F 72 64 65 72 49 44 11  6D 5F 6F 62 6A 4F 72 64  OrderID.m_objOrd
00000090  65 72 44 65 74 61 69 6C  73 01 00 04 08 0C 4F 72  erDetails.....Or
000000A0  64 65 72 44 65 74 61 69  6C 73 02 00 00 00 02 00  derDetails......
000000B0  00 00 06 03 00 00 00 05  41 4C 46 4B 49 00 00 00  ........ALFKI...
000000C0  00 09 04 00 00 00 05 04  00 00 00 0C 4F 72 64 65  ............Orde
```

Display Mode: [✓ Auto] Hexadecimal ANSI Unicode

Figure 3-8b. Contents of Message sent as Binary

Example 3 – Recoverable Messages

By default messages that are sent are only stored in memory of the computer that holds the message. This allows for rapid delivery of the messages, but should one of the computers crash between the time the message is sent and the time it is processed due to a hardware, software or other external event then the message will be lost forever! In production environments, I'd doubt you'd be able to get away with loosing a sales order. So to combat this issue, you can mark a message as *recoverable*, which will write the message to disk as well as keep it in memory. Should a failure happen the system will automatically recover the lost message from disk. This is done by simply setting the Recoverable property of the Message object to True, or if you are not using a message object then you can set the DefaultPropertiesToSend.Recoverable property of the MessageQueue object, which will apply to any messages not sent using a Message object:

```
' Message object
objMessage.Recoverable = True

' MessageQueue object
objMessageQueue.DefaultPropertiesToSend.Recoverable = True
```

Tip: Although the preferred method of sending messages is by using the Message object, you can still set a number of the same properties used by the Message object via the DefaultPropertiesToSend property of the MessageQueue object. Each message sent will then use these properties.

I've not gone with a full example here as there was only one line of code difference between Example2 and Example3. However, I have included a full version in the download under Example3. Try running Example2 and then Example3 then restart your computer to see how many messages you have left in the queue (*hint*: you should have only one).

MessageQueue.Receive

We have seen how easy it is to send a message, however it's not much good to send one if you don't know how to receive it. To do this requires a few important steps.

* You will need to obtain a reference to the queue that the message was sent to.
* Select the correct formatter, that being the same as the one which was used to send the original message.
* If you're using the XMLMessageFormatter (default), you'll need to setup the Message object with a list of possible data types that might be contained in the message. Usually this is a simple data type when sending a string or number, or the class type if you sent a full object.

Below are samples of the way you would retrieve a message from the queue, we will discuss how to do so when a transaction is involved later, but for the most part this is all you need to do. Notice how you can very easily turn an object that was sent (possibly from one side of the world to the other) using either XML or binary back into a working .NET object in only a few lines of code, assuming you have the code for the class locally.

Tip: You may notice the use of the DirectCast keyword, which many will be unfamiliar with. Essentially it's a faster version of CType when dealing with reference objects of the same type.

DirectCast requires the run-time type of an object variable to be the same as the specified type. If the specified type and the run-time type of the expression are the same, however, the run-time performance of DirectCast is better than that of CType. You would however get the same result by using CType, it would just be less efficient.

```
' Using the XMLMessageFormatter
' Retreiving a simple data type from a message queue
objMessageQueue = New MessageQueue(Constants.MQ.LocalNonTransQueue)
objMessageQueue.Formatter = New XmlMessageFormatter(New Type() _
   {GetType(String)})
Dim objMessage As Message = objMessageQueue.Receive()
Dim strMessage As String = DirectCast(objMessage.Body, String)

' Retreiving 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()
Dim objOrder As Order = DirectCast(objMessage.Body, Order)

' Using the BinaryMessageFormatter
' Retreiving a simple data type from a message queue
objMessageQueue = New MessageQueue(Constants.MQ.LocalNonTransQueue)
objMessageQueue.Formatter = New BinaryMessageFormatter()
Dim objMessage As Message = objMessageQueue.Receive()
Dim strMessage As String = DirectCast(objMessage.Body, String)

' Retreiving an object from a message queue
objMessageQueue = New MessageQueue(Constants.MQ.LocalNonTransQueue)
objMessageQueue.Formatter = New BinaryMessageFormatter()
Dim objMessage As Message = objMessageQueue.Receive()
Dim objOrder As Order = DirectCast(objMessage.Body, Order)
```

Caution: When using the BinaryMessageFormatter you *must* use the same assembly that was used to create the message; this however is not the case for the XMLMessageFormatter. You cannot copy and paste the class code into the receiving application. To make it work correctly create the class(s) into a separate dll and have both projects reference the same dll. It is for this reason that we created the separate project for the Orders class.

Where the type is defined I have used braces ({ }), which if you are not familiar with what they mean in .NET you might be rather confused as to why they are there. In .Net if you want to identify an array of objects (remember everything's an object) you can place them in braces separated by commas. In our case we only needed to define one element in the array.

Example 4 – Message Monitor

As we have seen it's not very hard to receive a message, however most of the time we won't know that a message has arrived. This means we have to poll or listen for messages arriving on the queue we are interested in. Microsoft provided in MSMQ 3.0 (was a standalone product in MSMQ 2.0) a technology that allowed you to assign triggers to queues, which based on rules you can define allowed you to run code when a message was received. This technology was pre-.NET and as such .NET has no direct support, although using COM wrappers you can still use this if you wish. This approach is a little limited, but can be enough for most purposes. If you were using VB6, it was definitely your best option as it can process multiple messages in different threads, a task that was near impossible (Ok it was possible, with a number of hacks, but in reality it was more trouble than it was worth) using VB6 alone. If you want to learn more about MSMQ Triggers see the resources at the end of the Chapter.

Due to the new found power that .NET has given all programmers, I'm going to demonstrate how you can achieve a similar thing to MSMQ Triggers but because you have the source, you'll also be able to modify it to your own specific needs. There are a few approaches to solving this problem. I've seen many an article explain this technique using threading, whereby you spawn a new thread to monitor a queue. This technique while very effective can be complex and prone to many difficulties if you are not very familiar with threading issues. Now you may be asking yourself, well if you're not going to use threading, then what are you going to use? The answer is, we are going to use threading of sorts, however, we won't create the new thread ourselves. We will allow the .NET framework to handle the actual creation of the thread and also let it manage our thread pool. This makes for a much simpler solution and one that should require less debugging!

What allows us to do this is a technology called delegates. You may be familiar with delegates as they relate to events. Well they can also be used to create asynchronous calls to any class method.

We will be creating a Form which can be called from another application which will supply the path of the MQ to monitor. This allows us to monitor multiple queues using the same piece of code. Figure 4-9 shows how our monitor will look; I've included the full listing (less IDE generated code) as it warrants some discussion.

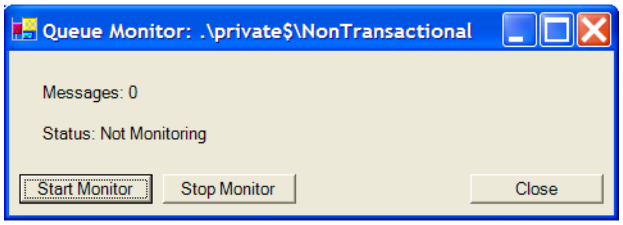


Figure 4-9. Queue Monitor Screen

```
Imports System.Messaging
```

```
Public Class frmQueueMonitor
    Inherits System.Windows.Forms.Form
    Public Delegate Function MonitorStart(ByVal queuePath As String) As Boolean
    Dim m_delMonitor As MonitorStart
    Public WithEvents m_objMonitor As New MonitorQueue()
    Private m_strMonitoredQueue As String

    Private Sub btnMonitor_Click(ByVal sender As System.Object, ByVal e As _
        System.EventArgs) Handles btnMonitor.Click
        m_delMonitor = AddressOf m_objMonitor.Start
        m_delMonitor.BeginInvoke _
            (m_strMonitoredQueue, AddressOf MonitorComplete, Nothing)
        Me.lblStatus.Text = "Status: Monitoring - " & m_strMonitoredQueue
    End Sub
    Private Sub MonitorComplete(ByVal ar As IAsyncResult)
        Dim blnError As Boolean = Not m_delMonitor.EndInvoke(ar)

        ' This is a very basic error reporting message, its not suggested you
        ' follow this!
        If Not blnError Then
            MessageBox.Show("MQ monitor has finished with no errors")
        Else
            MessageBox.Show("MQ monitor has finished with errors!")
        End If
        Me.lblStatus.Text = "Status: Not Monitoring"
        Me.Text = "Queue Monitor"
    End Sub

    Private Sub btnStop_Click(ByVal sender As System.Object, ByVal e As _
        System.EventArgs) Handles btnStop.Click
        m_objMonitor.Monitor = False
    End Sub

    Private Sub m_objMonitor_OnMessageArrive(ByVal messageCount As Integer) _
        Handles m_objMonitor.OnMessageArrive
        Me.lblStatus.Text = "Message Count: " & messageCount
        MessageBox.Show("You have mail.")
    End Sub

    Private Sub btnClose_Click(ByVal sender As System.Object, ByVal e As _
        System.EventArgs) Handles btnClose.Click
        Me.Close()
    End Sub

    Public Sub MonitorQueue(ByVal queuePath As String)
        m_strMonitoredQueue = queuePath
        Me.Text = "Queue Monitor: " & m_strMonitoredQueue
        Me.Show()
    End Sub
End Class

' This is the Class that will be run Async
Public Class MonitorQueue
    Public Event OnMessageArrive(ByVal messageCount As Integer)
    Private m_blnStop As Boolean = True

    Function Start(ByVal queuePath As String) As Boolean
        Dim intMessageCount As Integer

        ' Ensure that we set Monitor
        Me.Monitor = True
        Try
            Dim objMQ As New MessageQueue(queuePath)
            Do While Me.Monitor
                ' Check if we have a different message count ie New Message
                If intMessageCount < objMQ.GetAllMessages.Length Then
                    ' We have new messages
                    intMessageCount = objMQ.GetAllMessages.Length
                    RaiseEvent OnMessageArrive(intMessageCount)
                End If
                ' Reduce the stress on the computer and sleep for a bit
                System.Threading.Thread.Sleep(10)
            Loop
            Return True
        Catch ex As Exception
            ' An error happened so we will simply return false back to the
            ' application
            Return False
        End Try
    End Function

    Property Monitor() As Boolean
        Get
            Return m_blnStop
        End Get
        Set(ByVal Value As Boolean)
            m_blnStop = Value
        End Set
    End Property
End Class
```

If you are new to delegates then how this code works might be somewhat confusing, so I'll try to explain the steps necessary to create an asynchronously call using delegates, then apply the rules to our code. You will find most of the code mentioned below is in the frmQueueMonitor class.

. Create the class that will be called asynchronously. This class contains a method what we want to call via our delegate.

In our sample this is the MonitorQueue class

```
' Function to be called asynchronously by delegate
Function Start(ByVal queuePath As String) As Boolean
…
End Function
```

. Define a delegate which matches the signature of the method in the class. Place this at the top of the calling class. Note this doesn't need to have the same name as the actual function, just the signature must be the same but again parameter names need not be the same.

```
Delegate Function MonitorStart(ByVal queuePath As String) As Boolean
```

. Create an instance of the class which you want to call at the top of the calling class. Note in our example, it includes the WithEvents keyword because our object (MonitorQueue) has events which we will use later.

```
Public WithEvents m_objMonitor As New MonitorQueue()
```

. Create an object of the Delegate type you created.

```
Dim m_delMonitor As MonitorStart
```

. Set the new delegate object to equal the address of the actual method you wish to call using the AddressOf operator.

```
m_delMonitor = AddressOf m_objMonitor.Start
```

. Using the BeginInvoke method of the Delegate object pass any parameters (which will be dynamically generated by the .NET runtime to match the signature of the delegate's parameters – which is very cool), then using the AddressOf operator again pass in the address of the callback function you want to have called after the thread terminates. In our case the parameter is the queue to be monitored stored in the variable m_strMonitoredQueue, and we want to call the subroutine MonitorComplete when the process is finished. This code is called to start the task of monitoring in the btnMonitor_Click routine:

```
m_delMonitor.BeginInvoke _
  (m_strMonitoredQueue, AddressOf MonitorComplete, Nothing)
```

. That's it; the method will now be called in a separate thread that the Framework will take care of.

. Now you only need to decide what you want to do after the routine terminates. This is when the MonitorComplete routine will be called.

The other useful bits that might need explaining are how I've decided to monitor the queue. In this example, rather than actually retrieve the message, I've gone for a slightly different tact. The purpose of this monitor is to alert us to any new messages that arrive without affecting them. To do this we determine the current number of messages in the queue by calling the shared (static) function objMQ.GetAllMessages.Length, and then during each iteration of the loop check if this number has changed. If it has then we must have received a message, so we use the RaiseEvent keyword to alert the calling program of this fact (this is why we declared the instance with the WithEvents keyword earlier). To ensure that we don't bog the CPU down with our tight loop, we force the current thread to sleep (suspend operations) for 10 milliseconds. This seems to be long enough on my computer to keep the CPU usage at or close to zero. Depending on your computers configuration you may need to increase this value so you don't affect other programs.

To run our monitor we need to create a new instance of the class then pass it the path of the queue that we wish to monitor:

```
Dim objQueueMonitor As New frmQueueMonitor()
objQueueMonitor.MonitorQueue(Constants.MQ.LocalNonTransQueue)
```

The MonitorQueue method simply stores the path and then shows the form ready to use. The Form won't automatically start monitoring the queue; you need to first start the queue by pressing the appropriate button. When we run this example, and start monitoring the NonTransactional queue you'll notice that the program will tell you that you have mail, assuming you have run any of the other examples. To really test the code, run some of the earlier examples and you'll see the monitor will recognize their arrival immediately.

Another thing to note about the code is that because we are running in a different thread, we need a way to tell it to stop running should we wish to stop our monitoring. This is where the property Monitor of the MonitorQueue class comes in. As we have a reference to this object, we can easily set this property to False, which is checked during each iteration of the monitors loop. When it becomes False the monitor will drop out of the loop and finish the process, thereby finishing the thread. This is the beauty of using delegates as the framework does all the clean up code for us when dealing with threads.

Static Methods

Although you can create, delete and examine MQ's through creating an instance of the MessageQueue object, there are a number of static methods too (meaning you can call them without creating an actual instance of the class). The following will show you how to make use of these more maintenance style tasks.

I'm going to forgo a full example of each of the methods as they are only one line of code each. Instead I'll show you the line in question and explain any useful bits that may be necessary.

Note: Creating and deleting queues requires certain security permissions, and by default can only be done by the creator/owner of the queue or by an administrator. If you do not have sufficient permissions an exception will be raised with an "Access Denied" message.

Checking the existence of a MQ can be useful, however it does take time so is best done only when needed, such as at the beginning of an application. If you don't know if a queue exists however, you should check it otherwise an exception will be raised. Simply check this boolean condition to see if the queue already exists:

```
MessageQueue.Exists(<message queue path>)
```

After you've checked the existence of a MQ you may wish to create the queue if it doesn't already exist. To create a new queue you simply run the following static method:

```
MessageQueue.Create(<message queue path>,<is transactional?>)
```

There may also be occasions when you want to delete a queue. However, you should use extreme caution here as it will also remove any messages that are currently in the queue. Therefore it is best that you also check if there are any messages before you try deleting the queue; this was shown in the previous example.

```
MessageQueue.Delete(<message queue path>)
```

Summary

We learned that using asynchronous processing can lead to better real or perceived response times, better availability, increased robustness and an overall increase in scalability for our applications. We have also learned that using MQ's we can very easily take advantage of robust guaranteed delivery of messages. These messages can be queued up should a failure occur within part of the system and then processed when the failure has been rectified.

The next Chapter will continue the discussion on MQ's and introduce the concepts of transactions and acknowledgements, timeouts and journaling. These more advanced features are necessary to take full advantage of what MSMQ offers.

Resources:

MSMQ Home: http://msdn2.microsoft.com/en-us/library/ms711472.aspx

MSMQ Triggers: http://msdn2.microsoft.com/en-us/library/ms703197.aspx