

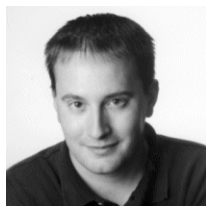
COM+ Queued Components for the ASP Developer:

Why and How

Christopher S. Blexrud, Blex Consulting, Inc.

Introduction

With the introduction of Windows 2000 and COM+ comes an exciting new suite of services. This presentation will concentrate on COM+ Queued Components, why we have them and how they can be integrated into today's ASP applications to incorporate asynchronous processing.



Christopher S. Blexrud

Christopher Blexrud, a Microsoft Certified Solution Developer, is a senior consultant from Minneapolis Minnesota. His main area of expertise resides in the Windows DNA architecture and the components within it. Chris has worked with Wrox on several projects including coauthoring Professional ASP 3.0 along with writing numerous articles on ASPToday.com. When Chris is not developing business solutions, he enjoys snowmobiling and hunting in northern Minnesota and spending time with family and friends. Chris can be reached at Chris@BlexOnline.net.

COM+ Queued Components for the ASP Developer

Queued Components (QC) form an exciting new feature with COM+ 1.0, which gives developers the ability to write code that interact with components asynchronously.

To fully understand the architecture behind QC and the reason we have it, let's first step back and look at normal synchronous processing and the limitation that surround it. Next we'll look into Message Queuing (MSMQ) and what it brings. Once we have an understanding of these areas we will look into QC and see what it have to offer.

Synchronous Processing

Synchronous processing is the communication style that most of us program in on a day-to-day process. For example we build an ASP page, which allows for the processing of an order or the editing of customer data. Most of the time we wait for the user to submit the form from the browser, which kicks off the ASP page and our code starts carrying out the necessary tasks. Once the processing is completed the HTML is sent down to the browser where the user can see what happened and receive any necessary information.

Normally this type of processing is acceptable and the preferred way to carry out the task at hand. In fact this type of processing is commonly done in Component Object Model (COM) and Distributed COM (DCOM) within the Windows platform.

COM is binary standard (not a language) built around interfaces, used to provide a foundation for components to be built upon.

DCOM expands COM and provides the functionality for COM calls to be made across process and across machine boundaries. DCOM does this my using Remote Procedure Calls (RPC) and marshalling parameters and return value between the consumer and the server.

Microsoft has developed a wide variety of tools and resources that aid in the development of applications that use COM/DCOM. MTS and now COM+, provide an environment in which COM components can reside and run in.

When a COM object that resides in a COM+ application is called upon from an ASP page, there are a series of steps that are executed in a synchronous fashion before the web sever can continue processing the page.

Depending on the setting within the COM+ application and the components within it, the actual timing of the following steps may vary. However, they still happen in the same order.

First, the request is sent from the web server to the server requesting an instance of the desired class. The server name for the desired component is looked up within the web server's COM+ catalog or Service Control Manager (SCM), depending on the version of Windows.

Once the web server receives a reference to the new instance of the class – the object – it can now perform the desired method call(s). As each method call is made on the remote object two important things happen: firstly, the parameters are marshaled across the network or process boundaries, then the client is blocked until the method call completes. If the COM consumer is a script client, like VBScript in the following example, there is also additional overhead due to the IDispatch interface and late binding.

The following code is a basic example of creating a COM object from an ASP page. It starts out by declaring a variable and then setting it to the return value of the Server.CreateObject methods call.

```
<%  
    'Create a new object  
    Dim objSample  
    Set objSample = Server.CreateObject("MyDll.MyClass")  
  
    'work on this object  
    objSample.FirstName = "John"  
    objSample.LastName = "Doe"  
    objSample.AddUser  
  
    'clean up  
    Set objSample = Nothing  
%>
```

Most of the time the duration of this process is fairly short and completes within an acceptable amount of time. However, if the tasks within the method call are very process intensive and require a large processing time, the users of the site could experience a timeout, hit refresh ten more times, or worst of all lose faith in the site.

Please keep in mind that this just touched on COM/DCOM and did not fully explain the inner works of it. The above information is merely an introduction to set the stage for asynchronous processing.

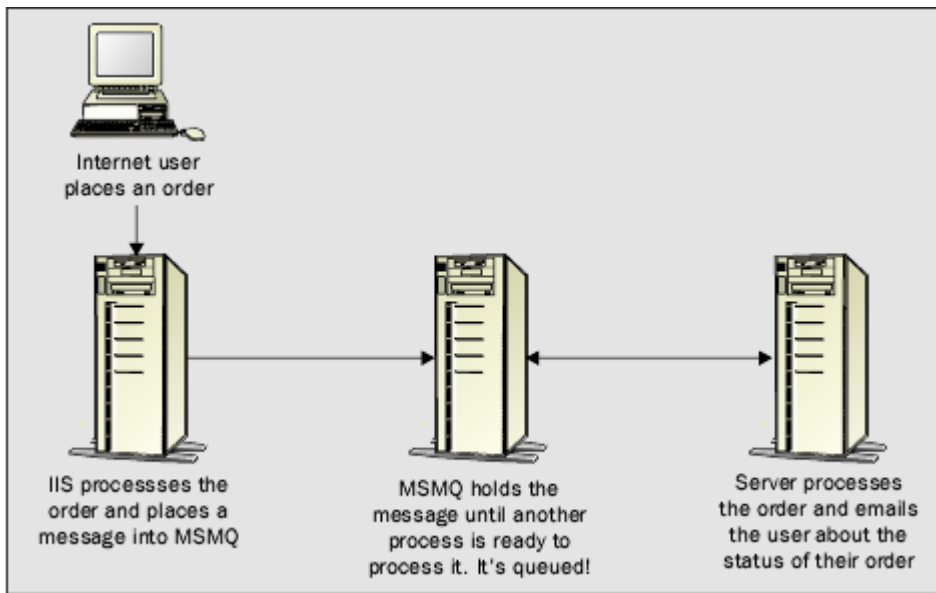
Message Queuing

Microsoft Message Queue is a service that runs on Windows NT 4.0 and Windows 2000. MSMQ provides an architecture to enable asynchronous communication between applications. MSMQ 1.0 is available for Windows NT 4.0 on the NT 4.0 Option Pack and MSMQ 2.0 is available in the Windows 2000 installation.

The basic concept of MSMQ is very simple to understand – it's email for applications. Just like an email server has inboxes, outboxes, etc., MSMQ contains a series of queues. These queues hold the messages that are sent from one application or process to another.

In a web application the web server usually sends a message through server-side code that interacts with MSMQ's API (COM dll) or through a component that interacts with it. Just like an email application, there is a receiver of the message who will know what to do with it. This is typically a server process and/or service that is watching the queue for messages to process.

Since the processing of a request or message through MSMQ is completed in two distinct phases (sending and receiving the message), the process becomes asynchronous. This can be a tremendous benefit when the interaction with the user must kick off a set of large processing tasks. However, with asynchronous processing, immediate feedback becomes fairly hard to implement, especially within the web environment.



MSMQ is also great in situations when required servers may not always be online. If this happens, MSMQ can queue the messages locally until the desired server becomes available. MSMQ also provides a logical break in the process. Since a message is sent to a queue where it sits until another process reads it, MSMQ provides a very nice location to break up the processing and distribute it across multiple servers.

The following section of code displays how easy it is to interact with MSMQ's API (COM API). It connects to a queue on a remote machine and sends a simple message containing a string. The code is written in VBScript for an ASP page, but can easily be converted to run within a COM component or a normal VB executable.

```
Dim objQueueInfo
Dim objQueue
Dim objMessage

'open the queue with send access
Set objQueueInfo = Server.CreateObject("MSMQ.MSMQQueueInfo")
objQueueInfo.FormatName = "DIRECT=OS:MyServer\MyQueue"
Set objQueue = objQueueInfo.Open(MQACCESS.MQ_SEND_ACCESS, _
    MQSHARE.MQ_DENY_NONE)

'set the message
Set objMessage = Server.CreateObject("MSMQ.MSMQMessage")
With objMessage
    .Label = "Message label"
    .Body = "Message Body"
    .Send objQueue
End With

'Clean up
Set objMessage = Nothing
objQueue.Close
Set objQueue = Nothing
Set objQueueInfo = Nothing
```

Notice the use of the `FormatName` property with `Direct:OS` when connecting to the queue. This allows the code to send the message even though the server and/or queue may not be available, which is great for disconnected users and dealing with server availability issues. The

alternative way to connect to a queue is by specifying the server and queue in the `PathName` property. However, the server must be available if the `PathName` property is specified.

```
objQueueInfo.PathName = "MyServer\MyQueue"
```

The next section of code displays how to connect to a queue on a remote server and look for the messages. The processing of message within a queue is usually done by a server-side process and/or service. This code simply opens the queue and waits 1000 milliseconds for a message to arrive. Basically, it is just checking to see if one exists. Remember this is usually server-side process/service, therefore this example is written in VB instead of VBScript like the example above.

```
Dim objQueueInfo As MSMQ.MSMQQueueInfo
Dim objQueue As MSMQ.MSMQQueue

'open the queue with receive access
Set objQueueInfo = New MSMQ.MSMQQueueInfo
objQueueInfo.FormatName = "DIRECT=OS:MyServer\MyQueue"
Set objQueue = objQueueInfo.Open(MQACCESS.MQ_RECEIVE_ACCESS, _
    MQSHARE.MQ_DENY_NONE)

'get the message
Set objMessage = objQueue.ReceiveCurrent(, , , 1000)

'process the message
... objMessage.Label ...
... objMessage.Body ...

'Clean up
Set objMessage = Nothing
objQueue.Close
Set objQueue = Nothing
Set objQueueInfo = Nothing
```

Although this is an effective way to look for a message, MSMQ's API also provides a mechanism to listen for messages asynchronously. Don't get too confused on this part; the processing of the messages is asynchronous from the client no matter how the messages are read from the queue - MSMQ provides a class that raises events to notify the presence of a new message, as opposed to checking the queue every so often.

One of the best parts about MSMQ is its flexibility. Any number of clients with the proper security settings can send to any number of queues. At the same time, these messages can be pulled out from the various queues by a single process or by multiple processes at the same time. Another excellent flexibility that MSMQ provides is the ability to hold a wide variety of information within the messages. The body of a MSMQ message can hold a simple string like above, numeric values, arrays, and even COM objects that support the `IPersistStream` or `IPersistStorage` interfaces. COM components that support one of these interfaces, like ADO Recordsets, have the ability to serialize themselves. This serialized state is what is placed into the MSMQ message and is also what is used to rebuild the COM object in the receiver's process space.

Queued Components

Now it's time to have some fun and look at what Microsoft has done with Queued Components!

Microsoft has taken the idea of passing the state of the COM object through an MSMQ message, and put a little spin on it. Instead of serializing the objects and sending the state in

the MSMQ message, they have designed architecture to record the interaction between a COM consumer and the component, which is then sent to a queue within MSMQ. From there, a listener picks up the message and passes it off to a player that invokes the methods on the COM object that was logged by the recorder.

With that said, let's step back and dissect the paragraph above since it holds a lot of information. First, start with the client and the recorder. When the client creates an instance of the desired class, it is actually talking to the recorder who impersonates the object requested by the client. The following code displays how this is accomplished.

```
<%
  Dim objSample
  Set objSample = GetObject("queue:/new:MyServer.MyClass")

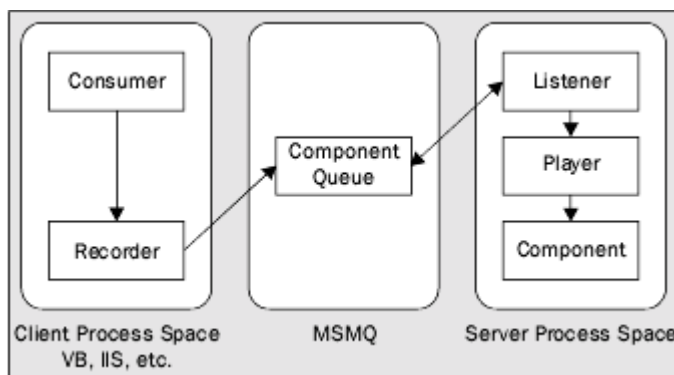
  objSample.MyMethod CLng(Request.Form("txtAge"))

  Set objSample = Nothing
%>
```

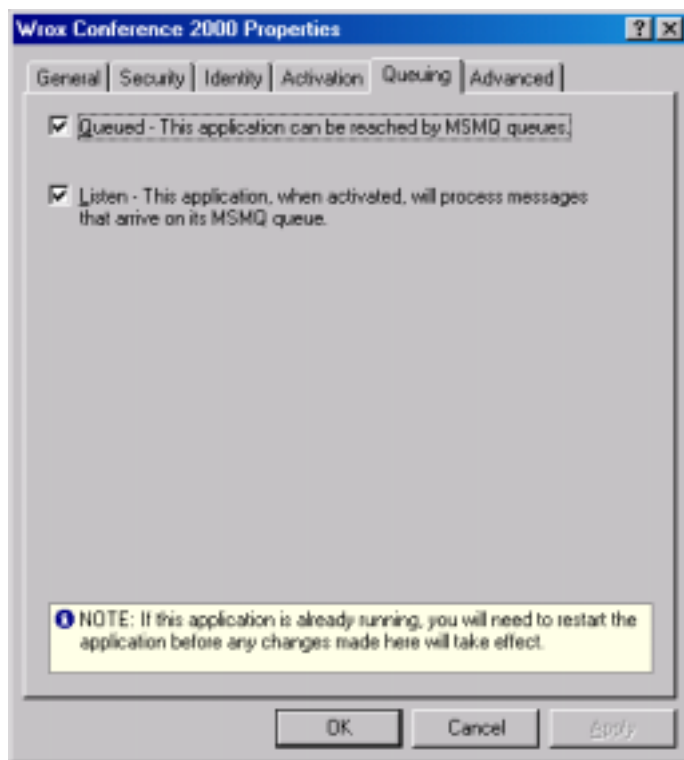
Notice the use of the `GetObject` function instead of the `CreateObject` from the server object. When a call is made to the `GetObject`, two monikers (another way to create or reference objects) must be used: the `queue` and `new`. The `queue` moniker informs COM+ that the client wishes to interact with the component via COM+'s Queued Components. The `new` moniker informs COM+ that the consumer wants to interact with a new object.

Besides the line of code where the component is instantiated, the rest of the code is identical to that when interacting with an object over COM and DCOM. However, the method calls are not actually invoked on the server object. Instead, since the consumer is actually invoking calls on the recorder, the recorder is building a message logging all of the method calls from the client and the parameters associated with each of them. In the sample code above, the recorder is logging the fact that the consumer invoked the `MyMethod` routine and passed the value that is currently held in `Request.Form("txtAge")`.

Once the client releases its reference by setting the object to nothing, or when the object reference falls out of scope, the recorder sends off the message containing the log information from the consumer's interaction to a queue. The diagram below helps to sum up the flow described in the paragraphs and code above.



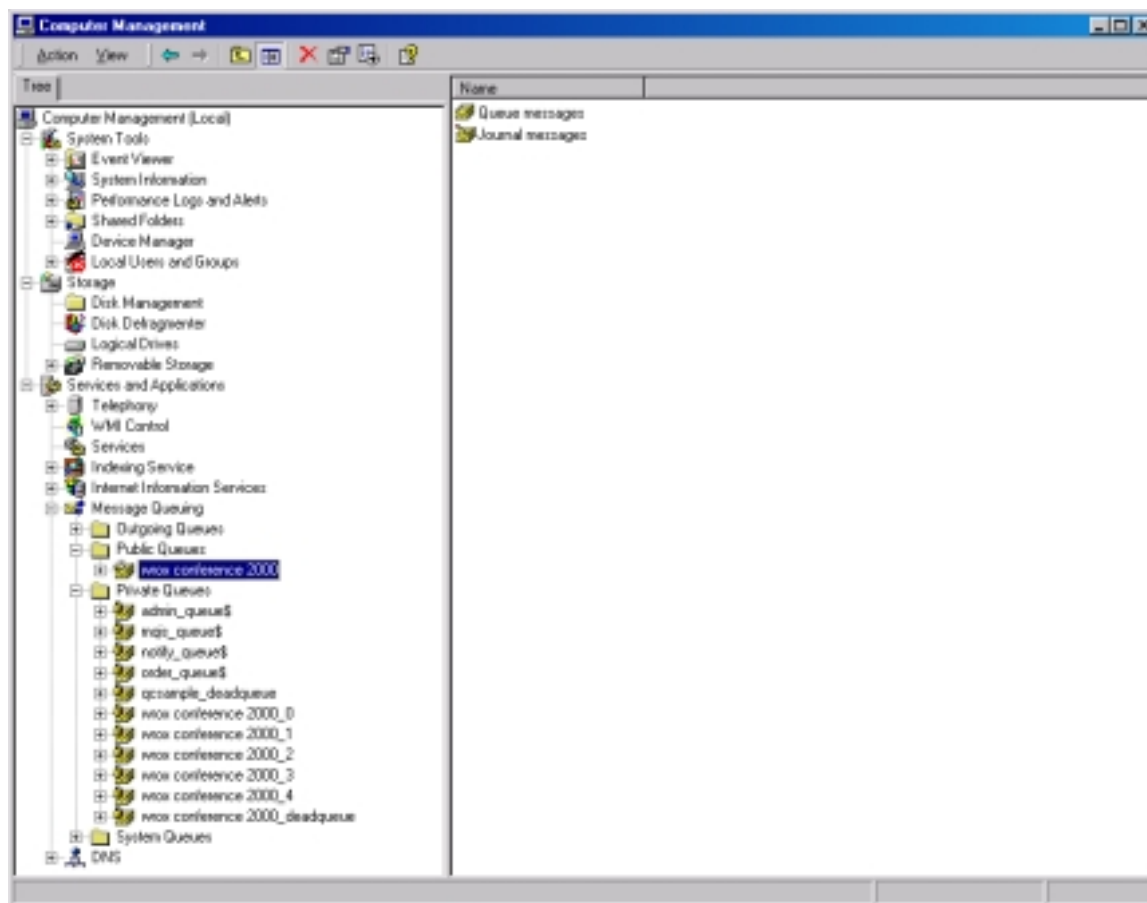
In order for components to be available through QC, there are a couple of server side settings that must be configured. First of all, a COM+ application must be created and set as queueable. This is done by adding a normal COM+ application under the Component Services MMC snap-in and then checking the `Queued` option on the `Queuing` tab in the `Properties` of the COM+ application.



Once a COM+ application is configured as queuable in COM+, a series of queues are created for it within MSMQ – one public and six private. Although these queues are created by COM+, it is helpful to understand how they are used and what purpose they serve. The one public queue provides a single repository in which all client requests are stored. The six private queues are used by COM+ when it encounters errors while processing messages on the server site. There are five queues that have names ending in _0 through to _4. Each time a message fails to process correctly it is moved to the next queue. The name of sixth queue ends with _deadqueue and is used if a message fails five times.

Once a message fails five times QC will create an instance of an exception class and query it for the IPlaybackControl interface. For more information on creating a exception class reference Microsoft's MSDN site, http://msdn.microsoft.com/library/psdk/cossdk/pgservices_queuedcomponents_8q9f.htm

The following screenshot displays the various queues created by COM+ for a COM+ application called "wrox conference 2000" after it was marked as queuable:



As messages appear in the public queue for a COM+ queued application, the listener section of the QC architecture responds to it and passes it off to the player. The player is the section of QC which actually interacts with the real component. The player reads the information from the message it received from the listener, and plays (invokes) the routines in the exact same order that the recorder logged them in on the client.

Queued Components Demo

- Start by developing an ActiveX dll in Visual Basic.
- Rename the project to "WroxConf2000" and class1 to "QCSample".
- Next, add two public subroutines to the QCSample class:
 - one called `WaitExample`, which takes one double parameter called `dblWaitTime` which is passed by value
 - one called `SQLExample`, which takes two parameters, `strConnectionString` and `strSQL`, both strings passed by value.

The following code displays the `WaitExample` method in its completed form. The routine really doesn't do anything except waste CPU cycles. However, it will allow for an easy way to simulate a long procedure call.

```
Public Sub WaitExample(ByVal dblWaitTime As Double)
    Dim dtUntil As Date
    On Error GoTo Error_Handler

    'get the time to wait until
    dtUntil = DateAdd("s", dblWaitTime, Now)
```

```

'sit and wait
Do Until Now >= dtUntil
    DoEvents
Loop

Exit_Handler:
    Exit Sub
Error_Handler:
    App.LogEvent Err.Description & " (" & Err.Number & ")",
vbLogEventTypeError
End Sub

```

The other method, `SQLExample`, is a little more useful than the routine above – it does, however, take a little more to setup since it needs a value connection string and SQL statement.

```

Public Sub SQLExample(ByVal strConnectionString As String, ByVal strSQL As String)
    Dim objConn As ADODB.Connection
    On Error GoTo Error_Handler

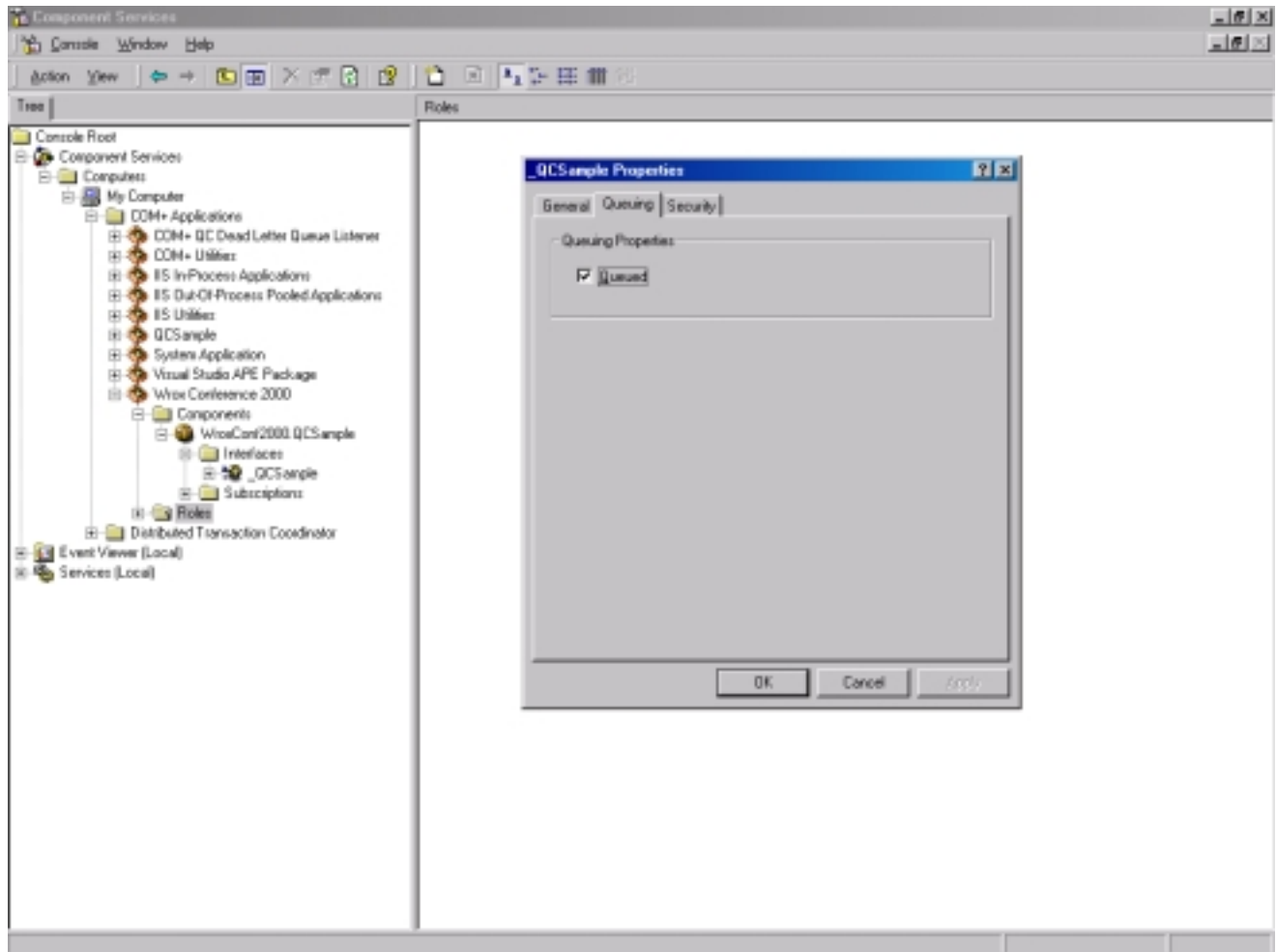
    'connect to the DB and execute the SQL
    Set objConn = New ADODB.Connection
    objConn.Open strConnectionString
    objConn.Execute strSQL, , adExecuteNoRecords

Exit_Handler:
    'clean up
    Set objConn = Nothing
    Exit Sub
Error_Handler:
    App.LogEvent Err.Description & " (" & Err.Number & ")",
vbLogEventTypeError
End Sub

```

- Now it is time to compile the ActiveX dll and install it into a queued COM+ application. This is done by selecting the `Make WroxConf2000.dll` item in the File menu.
- Once the DLL is built, open the Component Services MMC snap-in and create a new COM+ application called "Wrox Conference 2000".
- Next, mark the application as queueable by checking the `Queued` and `Listen` setting under the `Queuing` tab on the `Properties` of the new COM+ application.
- Now, add the newly created DLL, `WroxConf2000.dll`, to the COM+ application by right clicking the `Components` folder under the `Wrox Conference 2000` application and selecting `New`, then `Component`.
- Now import a component that is already registered and select the class that was built above, `WroxConf2000.QCSample`.
- The last configuration on the server-side is to mark the interface on the `WroxConf2000.dll` as queueable. Since this example was built in VB and it did not implement any other interfaces, the only one on it is `_QCSample`. Drill down to the `_QCSample` interface through the `WroxConf2000` component under the COM+ application called `Wrox Conference 2000`. Right-click on the interface and check the `Queued` setting on the `Queuing` tab.

The following screen shot displays how the COM+ application should be structured along with the queued setting at the interfaced level.



Now that the ActiveX dll has been built and all of the required settings have been made in COM+, it is time to build a user interface (UI) in ASP to interact with the WroxConf2000.QCSample component.

The UI is broken into three simple ASP pages. The first is just used to gather information from the user and pass it on the one of two other pages. The following code is from the default.asp page. Notice that it contains two forms; one which posts to ProcessWait.asp and the other which posts to ProcessSQL.asp.

```
<%@ Language=VBScript %>
<html>
<head>
  <title>
    COM+ Queued Components
  </title>
</head>
<body>
  <blockquote>
    <table BORDER="0" WIDTH="90%">
      <tr>
        <td ALIGN="LEFT"><h3>COM+ Queued Components</h3></td>
```

```

        <td ALIGN="RIGHT"><strong>Christopher S.
        Blexrud<br>Born</strong></td>
    </tr>
</table>

<hr>
<br>

<!-- Wait Form -->
<form ID="frmWait" Name="frmWait" Method="POST"
    Action="ProcessWait.asp">
    <strong>Call Wait Routine</strong>
    <br><br>Seconds:<br>
    <input Type="INPUT" Name="txtSeconds" ID="txtSeconds"
        Style="Width:200px" Value="10"><br><br>
    <input TYPE="Submit" Name="btnWait" ID="btnWait"
        VALUE="Submit">
</form>

<hr>

<!-- SQL Request Form -->
<form ID="frmSQL" Name="frmSQL" Method="POST"
    Action="ProcessSQL.asp">
    <strong>SQL Example</strong>
    <br><br>Connection String:<br>
    <input Type="INPUT" Name="txtConnectionString"
        ID="txtConnectionString" Style="Width:600px"
        Value="Provider=SQLOLEDB;Server=blex05;" & _
        "DATABASE=QCSample;UID=sa;PWD="><br><br>
        SQL Statement:<br>
    <input Type="INPUT" Name="txtSQL" ID="txtSQL"
        Style="Width:600px"
        Value="INSERT INTO Sample VALUES(GetDate())">
    <br><br>
    <input TYPE="Submit" Name="btnSQL" ID="btnSQL"
        VALUE="Submit">
</form>

<hr>

</blockquote>
</body>
</html>

```

As stated above, creating components through QC is the only part of the code that differs from utilizing the traditional COM/DCOM components. The following code displays both the traditional way to create the object and the way to create it through QC. The traditional method which, uses the CreateObject function, is commented out with GetObject method below it.

```

<%@ Language=VBScript %>

<%
    Dim objSample
    'Set objSample = Server.CreateObject("WroxConf2000.QCSample")
    Set objSample = GetObject("queue:/new:WroxConf2000.QCSample")

    objSample.WaitExample CLng(Request.Form("txtSeconds"))

```

```

Set objSample = Nothing
%>

<HTML>
<HEAD>
  <TITLE>Processing</TITLE>
</HEAD>
<BODY>
  <BLOCKQUOTE>Processing completed!</BLOCKQUOTE>
</BODY>
</HTML>

```

As you can see above, other than the line creating the object, everything is fairly straightforward. Notice the `CLng` around the reading of the request object. Without correctly typecasting all of the parameters to their expected value, COM+ will generate an error. Actually, just by removing the `CLng` conversion, COM+ will generate a "(0x8000FFFF) Catastrophic failure", because it is actually receiving a reference to the object passed back from the item in the form collection.

The following code displays what the `ProcessSQL.asp` page is comprised of. Just like the code above, it too contains a commented out line displaying the traditional method for creating objects with the `QC` line below it. Notice that use of the `CStr` functions. This is required just like the `CLng` was required above.

```

<%@ Language=VBScript %>

<%
  Dim objSample

  'Set objSample = Server.CreateObject("WroxConf2000.QCSample")
  Set objSample = GetObject("queue:/new:WroxConf2000.QCSample")

  objSample.SQLExample CStr(Request.Form("txtConnectionString")), _
    CStr(Request.Form("txtSQL"))

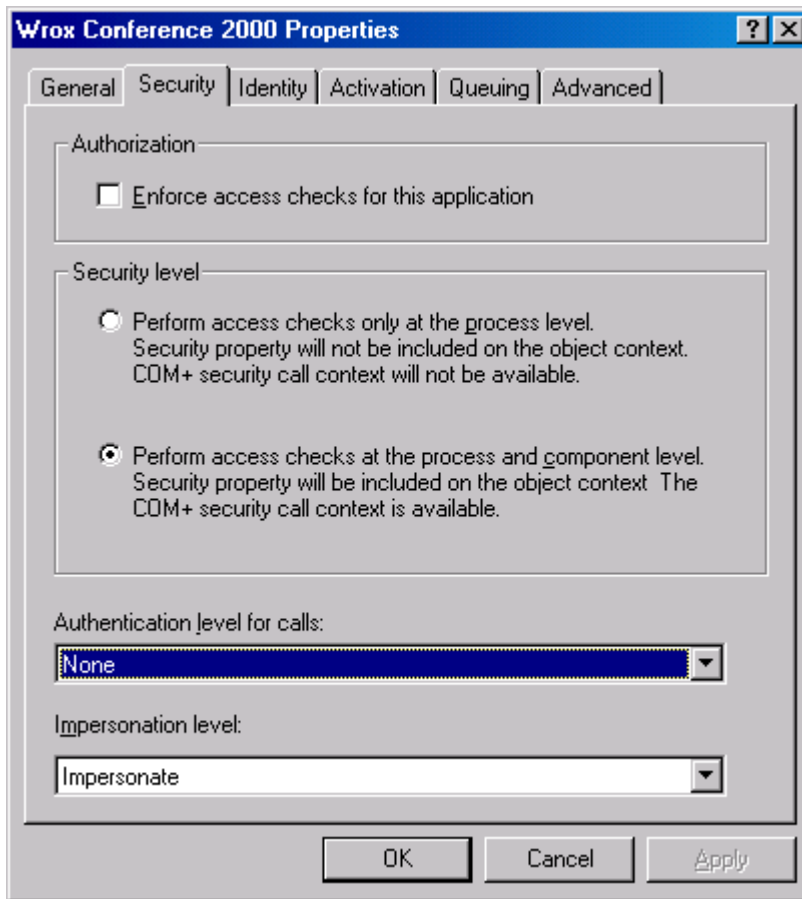
  Set objSample = Nothing
%>

<HTML>
<HEAD>
  <TITLE>Processing</TITLE>
</HEAD>
<BODY>
  <BLOCKQUOTE>Processing completed!</BLOCKQUOTE>
</BODY>
</HTML>

```

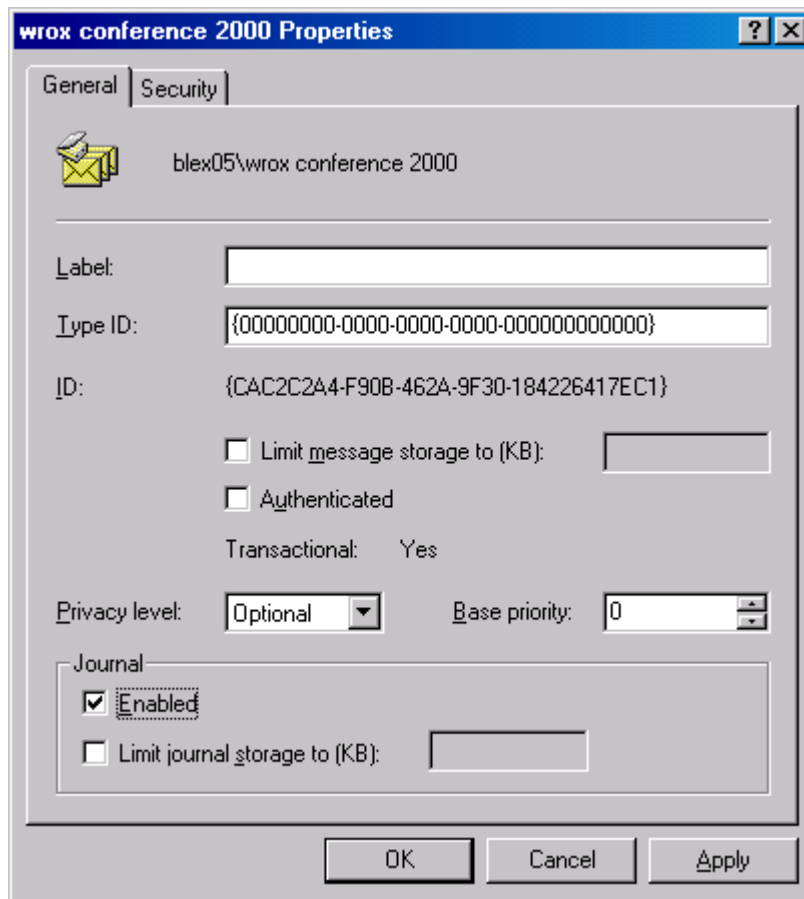
Depending on the configuration of your web server, there may be the need for a certain security setting to be adjusted - if a site is configured to allow anonymous access, the authentication level on the COM+ application must be set to none. This is because all security checks will resolve back to the `IUSER_<ComputerName>` account. If the site utilizes Windows authentication it can identify the actual user and therefore the authentication level can remain at packet.

To adjust the authentication level to allow anonymous access, open the Properties dialog box on the desired COM+ application and set the Authentication level to none.

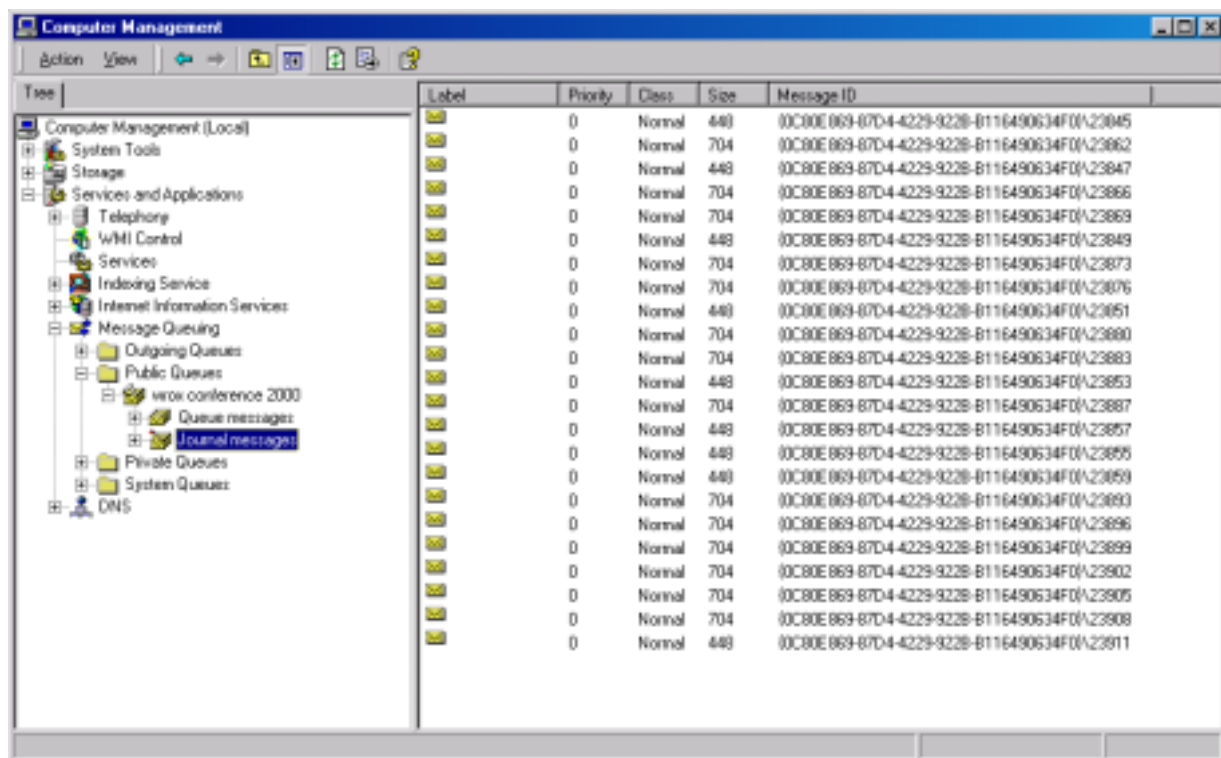


That's it! As you can see there is actually more work configuring components to run through QC than to actually code the component. Try everything out! If you know a valid connection string and SQL statement, a request can be issued to the `SQLExample` request. A request can also be submitted to the `WaitExample` method.

So did it work? Of course the easiest way is to query the data source and see if your SQL statement was ran. Another way to see if things are running is if enough requests are submitted the messages can actually be seen in queue within MSMQ. Another way to view the actually message submitted by COM+ through MSMQ is to enable journaling on the public queue. This is done by selecting the Properties for the desired queue (wrox conference 2000, in this case) and checking the Enabled setting in the Journal section under the General tab.



Now as COM calls are invoked through QC, the messages sent by COM+ will be queued in the Journal Messages node under the wrox conference 2000 queue. The following screen shot displays the journaled messages:



Limitations of Queued Components

In the routines from the demo above, there are no parameters passed by reference and function. This is a very important restriction of QC to keep in mind. Since all of the processing is queued and processed on the server at a different time there is no way to pass references to objects or any other data types. This limitation is really not a limitation of QC, but rather a limitation of message based programming in general. If a response is needed, a response queue and/or object can be built to listen for notification from the server. However, this really only works in a traditional client/server architecture and doesn't usually make sense in the disconnected environment of the Internet, in which emails are more often used to send notification to users.

Summary

Queued Components is an excellent architecture to provide asynchronous communication among various components throughout a solution. Although asynchronous processing cannot be and should not be used everywhere, it does provides the ability to distribute solutions and enable some of the processing to continue while the user is free to accomplish other tasks.

More Information

- Professional Active Server Pages 3.0 (Wrox Press, ISBN: 1861002610).
- Professional Windows DNA: Building Distributed Web Applications with COM+ (Wrox Press, ISBN: 1861004451).
- www.asptoday.com
- MSDN.Microsoft.com
- www.microsoft.com/com