COM Corner: COM+ Queued Components

by Steve Teixeira

Delphi developers normally do not have to be lectured on the benefits of briefcase model applications. When MIDAS was introduced in Delphi 3, the barrier of entry was forever lowered for creating applications with the ability to operate even when the client is disconnected from the server. Delphi developers quickly realised the power of enabling their users to work with their data in a disconnected, briefcase, model, embracing MIDAS and other technologies that provide this capability. Rather than having to write complicated code to, for example, enable a salesman to edit his customer database on his laptop while on the road and synchronize when he gets back into the office, this functionality is now easily accessible by dropping a few components and writing a few lines of code.

This is all really neat if you happen to be data, but what to do if you're an object? As object remoting technologies such as DCOM, MTS/COM+ and CORBA become easier to implement in our tools, our reliance on such technologies increases. Consequently, this reliance increases as we employ object remoting technologies to build ever more complex distributed apps. As a result, distributed component applications, like data applications, also have the need to function when disconnected from servers.

Queued Components: The Object Briefcase

COM+ queued components, provided in Windows 2000 Server, answer this need. Based on Microsoft Message Queue technology, queued components provide a means for COM+ clients to asynchronously invoke methods of COM+ server components. In essence, this means that clients can create instances of server objects and invoke their methods without regard to whether the server can be accessed by the client. COM+ manages this by storing the method invocations in a queue and executing them at a later time, when the server is accessible. What's more, the server objects likewise have little reason to know or care whether their methods are being invoked directly or via a COM+ queue. In this article I will cover the essential elements of working with COM+ queued components.

Figure 1 illustrates how queued components are implemented internally. When the *client* makes a method call on a queued component, that method call is captured by the *recorder*, which packages up the call and parameters and places them into a *queue*. Since the client has no knowledge that it is not actually communicating with the server, you can see that the recorder serves as a sort of a proxy for the server. The recorder knows how to behave because it obtains information on the server from its type library and its configuration and/or registration information. The *listener* removes the message, which contains the call information, from the queue and passes it on to the *player*. Finally, the player unpackages the call information (along with related information, such as the client's security context) and executes the method call on the server.

'All this sounds cool,' you might be saying to yourself, 'but I'll bet implementing it requires a degree in a new variety of non-Newtonian physics.' If you did say that to yourself, then you're only half right: it is cool, but it's also very easy to do, as you will soon see.

Why Queue Components?

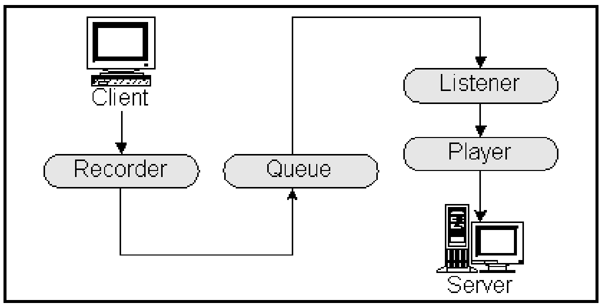
Before jumping into implementation, however, I'd like to address some of the specific reasons for using queued components.

System Scalability

In a non-queued system, there will be a finite number of server objects capable of handling requests from clients at any given time. When all of these objects become tied up handling client calls, other incoming client calls will be blocked until an object finishes and again becomes available. In a system having a large number of simultaneous transactions, this can seriously limit the number of concurrent clients that can be serviced. Using queues, the call always returns immediately to the client after being queued and played back to servers in the servers' own time. This enables the system to handle more concurrent transactions.

Scalability is also increased on the back-end, because the client doesn't manage the lifetime of the server. Rather than being active while the client carries on with its processing, a queued server only needs to be active while calls are being played back by the recorder. Reducing the time a server needs to remain in memory means that a greater number of servers can be activated over a given time with a given amount of RAM.

➤ Figure 1



```
unit TestImpl;
interface
uses Windows, ComObj, ActiveX, Srv_TLB, StdVcl;
type
  TQTest = class(TAutoObject, IQTest)
  protected
    procedure SendText(const Value: WideString; Time: TDateTime); safecall;
  end;
implementation
uses ComServ, SysUtils;
procedure TQTest.SendText(const Value: WideString; Time: TDateTime);
const
  SFileName = 'c:\queue.txt';
  SEntryFormat =
    'Send time:  %s'#13#10'Write time: %s'#13#10'Message: %s'#13#10#13#10;
var
  F: THandle;
  WriteStr: string;
begin
  F := CreateFile(SFileName, GENERIC_WRITE, FILE_SHARE_READ, nil, OPEN_ALWAYS,
    FILE_ATTRIBUTE_NORMAL, 0);
  if F = INVALID_HANDLE_VALUE then RaiseLastWin32Error;
  try
    FileSeek(F, 0, 2);  // go to EOF
    WriteStr :=
      Format(SEntryFormat, [DateTimeToStr(Time), DateTimeToStr(Now), Value]);
    FileWrite(F, WriteStr[1], Length(WriteStr));
  finally
    CloseHandle(F);
  end;
end;
initialization
  TAutoObjectFactory.Create(ComServer, TQTest, Class_QTest,
    ciMultiInstance, tmApartment);
end.
```

➤ *Listing 1*

Briefcase Model

As I mentioned, COM+ enables queued components to behave in a disconnected manner in much the same way MIDAS does for data. This enables clients to work without being connected to their network, and method calls to be played back to the server when the client connects up again.

Fail-Safety

If you are creating a mission-critical application that requires a high degree of availability, such as an e-commerce storefront, the last thing you want to happen is for the system to go down because your front end is having trouble communicating with server objects. Queued components provide an ideal safety net to prevent this problem, because they will queue method calls intended for servers if the servers become unavailable and play them back when the server again comes online.

Load Scheduling

Rather than having your servers work like rented mules in their peak hours of activity and sit nearly dormant during the rest of the day, using queued components you can spread processing throughout the day to even the workflow and place less demand on your servers at any one time.

Creating A Server

There's little difference between creating a queued component and creating a normal COM/COM+ component. The biggest adjustment you will need to make is that all methods on queued interfaces must accept only in parameters and must not make use of return values. Of course, these limitations make sense when you consider that the client won't be sitting around waiting for the server to return any values or out parameters. Also, you will need to perform a few extra steps in component configuration at install-time.

To illustrate, I'll create a Delphi server that contains one COM+ class with one interface with one method. To make life easier, I'll get started using the Automation Object Wizard accessible via the `File | New...` main menu item. I call this object `QTest`, and the wizard automatically names the primary interface `IQTest` (don't worry, it's easier than it sounds). To the `IQTest` interface I add one method, which is defined in the type library editor as follows:

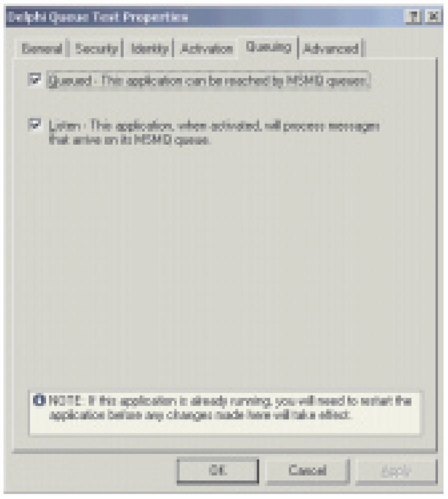
```
procedure SendText(Value:
  WideString; Time: TDateTime)
  [dispid $00000001]; safecall;
```

The idea is that this method takes two parameters, the first a string message and the second the time on the client the method was called. My implementation of this method simply writes this information, in addition to the time the message was processed by the server, to a log file I create called c:\queue.txt. The implementation file for this Automation object is shown in Listing 1.

After the server has been created, it needs to be installed into a new COM+ application using either the Component Services management tool or the COM+ Administration Library API. Using the Component Services tool, the first step is to create a new empty application by selecting that option from the local menu of the *COM+ Applications* node in the tree and following the prompts. Once the application has been created, the next step is to edit the application's properties to mark the application as queued, as shown in Figure 2. I also chose to enable queue listening on this application so that it would immediately play any incoming messages on its queue when it is active.

To install the server into the COM+ application, I select `New | Component` from the local menu of the *Component* node of the application in the tree. This invokes the COM component install wizard, using which I install a new component using the defaults and select the name of the COM+ server DLL created earlier. After installation

➤ *Figure 2*


Delphi Queue Test Properties

[✓] Queued - This application can be reached by MSMQ queues.

[✓] Listen - This application, when activated, will process messages that arrive on its MSMQ queue.

NOTE: If this application is already running, you will need to restart the application before any changes made here will take effect.

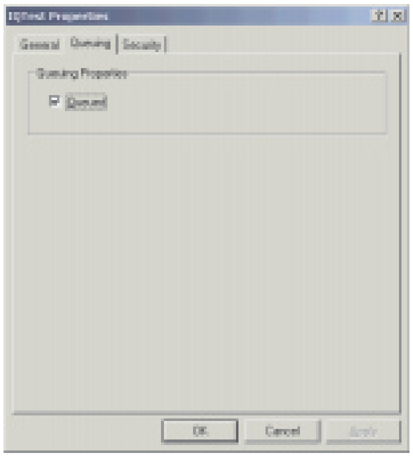
into the application, I edit the properties of the IQTest interface on this object to support queuing as shown in Figure 3.

Note that COM+ requires that queuing be enabled on both the COM+ application and at the interface-level.

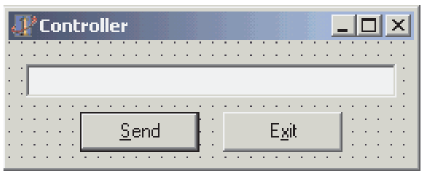
Creating A Client

The workflow for creating a queued component client is identical to creating a client of any old Automation client. In this case, I created an application with a main form as shown in Figure 4.

When the Send button is pressed, the contents of the edit are sent to the server via its SendText method. Listing 2 shows the code for this form's unit. The only element in this unit that sets it apart from a standard Automation controller is the means by which it creates the server object instance. Rather than using, for example, the CoCreateInstance COM API, this client uses the CoGetObject API. CoGetObject enables an object to be created via a moniker, and COM+ allows a special string moniker syntax that can be used to invoke components in a queued manner. The syntax of this moniker is queue:/new: followed by the CLSID or program ID of the server object. Listing 3 shows



➤ *Figure 3*



➤ *Figure 4*

```
unit Ctrl;
interface
uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs,
  StdCtrls, ColorGrd, ExtCtrls, Srv_TLB, Buttons;
type
  TControlForm = class(TForm)
    BtnExit: TButton;
    Edit: TEdit;
    BtnSend: TButton;
    procedure BtnExitClick(Sender: TObject);
    procedure BtnSendClick(Sender: TObject);
    procedure FormCreate(Sender: TObject);
  private
    FIntf: IQTest;
  end;
var ControlForm: TControlForm;
implementation
{$R *.DFM}
uses ComObj, ActiveX;
// Need to import CoGetObject because import in ActiveX unit is incorrect
function MyCoGetObject(pszName: PWideChar; pBindOptions: PBindOpts; const iid:
  TIID; out ppv): HResult; stdcall; external 'ole32.dll' name 'CoGetObject';
procedure TControlForm.BtnExitClick(Sender: TObject);
begin
  Close;
end;
procedure TControlForm.BtnSendClick(Sender: TObject);
begin
  FIntf.SendText(Edit.Text, Now);
  Edit.Clear;
end;
procedure TControlForm.FormCreate(Sender: TObject);
const
  SMoniker: PWideChar = 'queue:/new:{64C576F0-C9A7-420A-9EAB-0BE98264BC9D}';
begin
  // Create object using a moniker that specifies queued creation
  OleCheck(MyCoGetObject(SMoniker, nil, IQTest, FIntf));
end;
end.
```

➤ *Above: Listing 2*

➤ *Below: Listing 3*

```
queue:/new:Srv.IQTest
queue:/new:{64C576F0-C9A7-420A-9EAB-0BE98264BC9D}
queue:/new:64C576F0-C9A7-420A-9EAB-0BE98264BC9D
```

```
queue:Priority=6,ComputerName=foo/new:{64C576F0-C9A7-420A-9EAB-0BE98264BC9D}
queue:PathName=drevil\myqueue/new:{64C576F0-C9A7-420A-9EAB-0BE98264BC9D}
```

➤ *Above: Listing 4*

➤ *Below: Listing 5*

```
dim cat
set cat = CreateObject(
  "COMAdmin.COMAdminCatalog");
cat.StartApplication(
  "YourApplication");
```

examples of properly formatted queue monikers.

There are also a number of queue moniker parameters you can incorporate into the string to modify the destination queue or queue behavior. These are listed in Table 1. Using some of these options, other valid queue monikers might be as shown in Listing 4.

Running The Server

After invoking the client and typing a few strings into the edit, you can check for yourself on your hard disk, and you will see that the file c:\queue.txt isn't present on your hard disk. That is because the server application needs to be started running before queued messages will be played back.

There are three ways to start the server. First, manually, using the Component Services tool. This can be done simply by selecting Start from the local menu of the application node in the tree.

Second, you can start it programatically, using the COM+ Administration Library API.

Lastly, you can schedule the server start, using scripting. This can be done using a script similar to that shown in Listing 5 in the task scheduler.

After starting the application, you will see the c:\queue.txt file present on your hard disk. Its contents will look something like the following:

```
Send time:  2/6/2000 7:15:08 AM
Write time: 2/6/2000 7:15:18 AM
Message: this is a test
```

Parameter	Description	Acceptable Values
ComputerName	Specifies the computer name portion of a queue path name. If not specified, the `ComputerName` associated with the configured application is used.	String name of computer containing queue.
QueueName	Specifies the queue name. If not specified, the queue name associated with the configured application is used.	String name of queue on target server machine.
PathName	Specifies the complete queue pathname. If not specified, the queue path name associated with the configured application is used.	The queue path name must be formatted as ComputerName\QueueName.
FormatName	Specifies the queue format name.	Format name of queue, eg, DIRECT=9CA3600F-7E8F-11D2-88C5-00A0C90AB40E
AppSpecific	An unsigned integer design for application-specific use.	eg, AppSpecific=8675309
AuthLevel	Specifies the message authentication level. An authenticated message is digitally signed and requires a certificate for the user sending the message.	MQMSG_AUTH_LEVEL_NONE (0) or MQMSG_AUTH_LEVEL_ALWAYS (1)
Delivery	Specifies the message delivery option. Ignored for transacted queues.	MQMSG_DELIVERY_EXPRESS (0) or MQMSG_DELIVERY_RECOVERABLE (1)
EncryptAlgorithm	Specifies the encryption algorithm to be used by COM+ for the message.	CALG_RC2, CALG_RC4, or other integer value recognized by COM+ as an acceptable EncryptAlgorithm.
HashAlgorithm	Specifies a cryptographic hash function.	CALG_MD2, CALG_MD4, CALG_MD5, CALG_SHA, CALG_SHA1, CALG_MAC, CALG_SSL3_SHAMD5, CALG_HMAC, CALG_TLS1PRF, or other integer value recognized by COM+ as an acceptable HashAlgorithm.
Journal	Specifies the COM+ queue message journal option.	MQMSG_JOURNAL_NONE (0), MQMSG_DEADLETTER (1), MQMSG_JOURNAL (2)
Label	Specifies a message label string up to MQ_MAX_MSG_LABEL_LEN characters	Any string.
MaxTimeToReachQueue	Specifies a maximum time, in seconds, for the message to reach the queue.	INFINITE, LONG_LIVED, or an integer value indicating a specific number of seconds.
MaxTimeToReceive	Specifies a maximum time, in seconds, for the message to be received by the target application.	INFINITE, LONG_LIVED, or an integer value indicating a specific number of seconds.
Priority	Specifies a message priority level, within the MSMQ values permitted.	MQ_MIN_PRIORITY (0), Q_MAX_PRIORITY (7), MQ_DEFAULT_PRIORITY (3), or any integer between 0 and 7.
PrivLevel	Specifies the privacy level that is used to encrypt messages.	MQMSG_PRIV_LEVEL_NONE, NONE, MQMSG_PRIV_LEVEL_BODY, BODY, MQMSG_PRIV_LEVEL_BODY_BASE, BODY_BASE, MQMSG_PRIV_LEVEL_BODY_ENHANCED, BODY_ENHANCED
Trace	Specifies trace options, used in tracing COM+ queue routing.	MQMSG_TRACE_NONE (0), MQMSG_SEND_ROUTE_TO_REPORT_QUEUE (1)

➤ *Table1*

```
Send time:  2/6/2000 7:15:10 AM
Write time: 2/6/2000 7:15:18 AM
Message: this is another
```

Summary

That about sums it up for the fundamentals of COM+ queued component programming. Remember that you need to specifically install Message Queuing Services when you install Windows 2000 Server in order to create a queue server.

Using the skills you have learned in this article, you should now be able to create your own queued component applications and use them as an aid to scaling your environment, providing briefcase access, providing fail-safety, or load-levelling your server. Components no longer need to be jealous of data.

Steve Teixeira is the CTO of DeVries Data Systems (www.dvdata.com), a Silicon Valley internet professional services firm, and co-author of *Delphi 5 Developer's Guide*. Email him at steve@dvdata.com