COM Corner: Microsoft Transaction Server, Part 1

by Steve Teixeira

The COM development community has been making a lot of noise of late about Microsoft Transaction Server (MTS), and not without reason. MTS represents a new concept for COM developers, who have long enjoyed the advantages of language-independent interfaces, location transparency, and automatic activation and deactivation. However, thanks to MTS, COM developers can now take advantage of powerful runtime services, like lifetime management, security, resource pooling, and transaction management. While MTS brings a lot of useful features to the table, it also requires some changes in system design that in some cases contradict the ideas COM has pounded into our skulls over the years.

In this article, I will discuss MTS technology, and in part 2 of this article next month, we will talk more specifically about MTS and Delphi 4, Delphi's MTS framework and IDE support, and walk through some sample MTS components and applications.

Before we leap into the technical details, I want to tell you up front that transaction handling is only a small part of the MTS picture, and the fact that 'transaction' appears in the name is quite unfortunate. It's like calling your new home entertainment system a soap opera viewer. Sure, it does that, but so much more. To their credit, when I've spoken with folks at Microsoft close to the technology, they generally hate the name. Fortunately, the name won't be with us much longer, as MTS will be folded into the operating system as a part of the upcoming enhancements to COM known as COM+.

Why MTS?

The magic word of system design these days is *scalability*. With the hyper-growth of internet and intranets, the consolidation of corporate data into centrally-located data stores, and the need for everyone and their cousin to get at the data, it's absolutely crucial that a system be able to scale to ever larger numbers of concurrent users. It's definitely a challenge, especially considering the rather unforgiving limitations we must deal with, such as finite database connections, network bandwidth, server load, and so on. In the good old days of the early 90s, client/server computing was all the rage and considered *The Way* to write scalable applications. However, as databases were bogged down with triggers and stored procedures and clients were complicated with various bits of code here and there in an effort to implement business rules, it shortly became obvious that such systems would never scale to large numbers of users. The multi-tier architecture soon became popular as a way to scale a system to a greater number of users. By placing application logic and shared database connections in the middle tier, database and client logic could be simplified and resource usage optimized for an overall higher-bandwidth system.

A sidenote to this, that I'll mention just because it is interesting, is that the added infrastructure introduced in a multi-tier environment tends to increase latency as it increases bandwidth. In other words, you may very well need to sacrifice the performance of the system in order to improve scalability.

Microsoft extended to COM developers the ability to build applications that are distributed across multiple machines with the introduction of DCOM several years ago. DCOM was a step in the right direction. It provided the means by which things COM may communicate with one another over the wire, but it did not make many significant steps toward solving the real-world problems encountered by developers of distributed applications. Issues such as lifetime optimization, thread management, flexible security, and transaction support were still left to individual developers. Enter MTS.

What Is MTS?

MTS is a COM-based programming model and collection of runtime services for developing scalable and/or transactional COM-based applications. The programming model part of MTS isn't much different than what you are familiar with already as a COM developer. There are a few wrinkles that you will learn about shortly, but for the most part, any in-process (DLL) COM object with a type library can be an MTS object. However, it's not recommended that you run non-MTS-aware COM components within MTS. MTS runtime services mean that MTS serves as the caregiver for your COM components. MTS can host them, manage their lifetime, provide security for them, and so on. This means that, rather than running within the context of your application, MTS COM objects run within the context of the MTS runtime. All this adds up to a bunch of new features that you can take advantage of with little or no coding changes in your client or COM object code.

It's interesting to note that because MTS objects do not run directly within the context of a client like other COM objects, clients never really obtain interface pointers directly to an object instance. Instead, MTS inserts a proxy between the client and the MTS object such that the proxy is

identical to the object from the client's point of view. However, because MTS has complete control over the proxy, it can control access to interface methods of the object for purposes such as lifetime management and security, as you will soon learn.

Stateful Versus Stateless

The number one topic of conversation amongst folks looking at, playing with, and working on MTS technology seems to be the discussion of stateful versus stateless objects. While COM itself doesn't give a whit as to the state of an object, in practice most traditional COM objects are stateful. That is, they continuously maintain state information from the time that they're created, while they're being used, and up until the time that they're destroyed. The problem with stateful objects is that they aren't particularly scalable, since state information would have to be maintained for every object being accessed by every client. A stateless object is one that generally does not maintain state information between method calls. MTS prefers stateless objects because they enable MTS to play some optimization tricks. If an object doesn't maintain any state between method calls, then MTS could theoretically make the object go away between calls without causing any harm. Furthermore, since the client maintains pointers only to MTS's internal proxy for the object, MTS could do so without the client being any the wiser. It's more than a theory, this is actually how MTS works. MTS will destroy the instances of the object between calls in order to free up resources associated with the object. When the client makes another call to that object, the MTS proxy will intercept it and a new instance of the object will be created automatically. This helps the system scale to a larger number of users, since there will be comparatively few active instances of a class at any given time.

Writing interfaces to behave in a stateless manner will probably require a slight departure from your usual way of thinking for interface design. For example, consider the classic COM-style interface in Listing 1.

As you might imagine, you would use the Listing 1 interface in a manner something like Listing 2. The problem with this style is that the object is not stateless between method calls, because state information regarding the account number must be maintained across the call. A better approach to this interface for use in MTS would be to pass all of the necessary information to the `AddActivity` method so that the object could behave in a stateless manner:

```
procedure AddActivity(
  AccountNum: WideString;
  Amount: Integer); safecall;
```

The particular state of an active object is also referred to as a *context*. MTS maintains a context for each active object that tracks things such as security and transaction information for the object. An object can at any time call `GetObjectContext` to obtain an `IObjectContext` interface pointer for the object's context. `IObjectContext` is defined in the `Mtx` unit as shown in Listing 3.

The two most important methods in this interface are `SetComplete` and `SetAbort`. If either of these methods are called, then the object is telling MTS that it no longer has any state to maintain. MTS will therefore destroy the object (unknown to the client, of course), thereby freeing up resources for other instances. If the object is participating in a transaction, `SetComplete` and `SetAbort` also have effect of a commit or rollback for the transaction, respectively.

Lifetime Management

From the time we were itty bitty COM programmers, we were taught to only hold on to interface pointers for as long as necessary, and to release them as soon as they are unneeded. In traditional COM this makes a lot of sense because we don't want to occupy the system with maintaining resources that aren't being used. However, since MTS will automatically free up stateless objects after they call `SetComplete` or `SetAbort`, there is no expense associated with holding a reference to such an object indefinitely. Furthermore, since the client never knows that the object instance may have been deleted under the sheets, clients do not have to be rewritten to take advantage of this feature.

Packages

As if the word 'package' weren't already overloaded enough, with Delphi packages, C++Builder packages, and Oracle packages all coming to mind as examples of the overuse of this word. MTS also has a notion of packages that no doubt

➤ *Listing 1*

```
ICheckbook = interface['{2CCF0409-EE29-11D2-AF31-0000861EF0BB}']
  procedure SetAccount(AccountNum: WideString); safecall;
  procedure AddActivity(Amount: Integer); safecall;
end;
```

➤ *Listing 2*

```
var
  CB: ICheckbook;
begin
  CB := SomehowGetInstance;
  // open my checking account
  CB.SetAccount('12345ABCDE');
  // add a debit for $100
  CB.AddActivity(-100);
  ...
end;
```

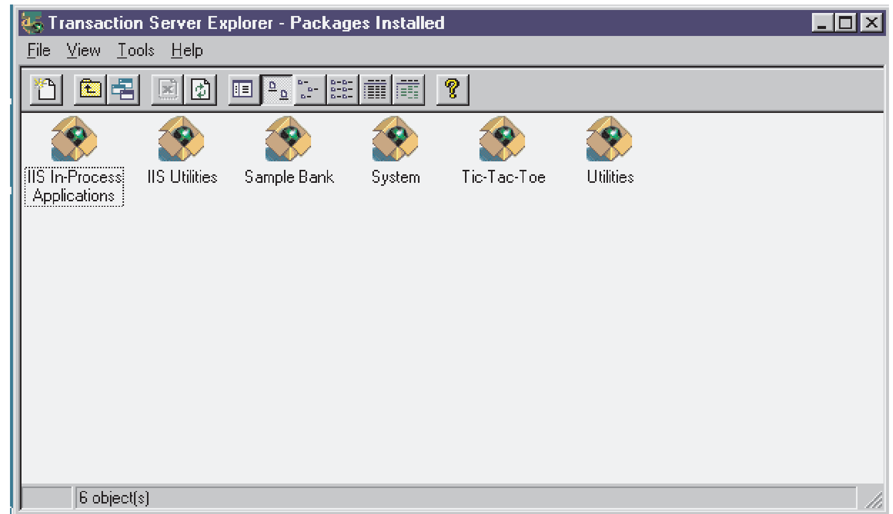
➤ *Listing 3*

```
IObjectContext = interface(IUnknown)
  ['{51372AE0-CAE7-11CF-BE81-00AA00A2FA25}']
  function CreateInstance(const cid, rid: TGUID; out pv): HResult; stdcall;
  procedure SetComplete; safecall;
  procedure SetAbort; safecall;
  procedure EnableCommit; safecall;
  procedure DisableCommit; safecall;
  function IsInTransaction: Bool; stdcall;
  function IsSecurityEnabled: Bool; stdcall;
  function IsCallerInRole(const bstrRole: WideString): Bool; safecall;
end;
```

differs from those other varieties. An MTS package is more logical than physical, as it represents a programmer-defined collection of MTS objects with like activation, security, and transaction attributes. The physical part of a package is a file that contains references to the COM server DLLs and MTS objects within those servers that make up the package. The package file also contains information on the attributes of the MTS objects within.

MTS will run all components within a package in the same process. This enables you to configure your well behaved and error-free packages insulated from the potential problems that could be caused by faults or errors in other packages. It is also interesting to note that the physical location of components has no bearing on eligibility for package inclusion; a single COM server can contain several COM objects, each in a separate package.

Packages are created and manipulated using either the `Run | Install MTS Objects...` menu item in Delphi or the Transaction Server Explorer that is installed with MTS and shown in Figure 1.



➤ Figure 1

Security

MTS provides a roll-based security system that is much more flexible than the standard Windows NT security normally used with DCOM. A *roll* is a category of user, for example in a banking system typical rolls might be teller, supervisor, and manager. MTS allows you to specify the degree to which any particular roll can manipulate an object on a per-interface basis. For example, you can specify that the manager roll has access to the `ICreateHomeLoan` interface, but the teller roll does not. If you need to get more granular than access to entire interfaces, you can determine the roll of the user in the current context by calling the `IsCallerInRole` method of `IObjectContext`. Using this, for example, you could enforce a business rule that stipulates that tellers can approve normal account closures, but only supervisors can approve an account closure when the account balance is over $100,000. Security rolls can be configured in the Transaction Server Explorer.

Oh, And It Also Does Transactions

And of course, as the name implies, MTS also does transactions. You might be thinking to yourself, ‘big deal, my database server already supports transactions. Why do I need my components to support them as well?’ A fair question, and luckily I’m equipped with good answers. Transaction support in MTS can enable you to perform transactions across multiple databases or can even make a single atomic action out of some set of operations having nothing to do with databases. In order to support transactions on your MTS objects, you must either set the correct transaction flag on your object’s `coclass` in the type library during development (this is what the Delphi MTS wizard does) or after deployment in the Transaction Server Explorer.

When should you use transactions in your objects? That’s easy: you should use transactions whenever you have a process involving multiple steps that you wish to make into a single, atomic transaction. In doing so, the entire process can be either committed or rolled back, but you will never leave your logic or data in an incorrect or indeterminate state somewhere in between. For example, if I am writing software for a bank and I wish to handle the case where a client bounces a check, there would likely be several steps involved in handling that, including: debiting the account for the amount of the check, debiting the account for the bounced check service charge and sending a letter to the client.

In order to properly process the bounced check, each of these things must happen. Therefore, wrapping them in a single transaction would ensure that all will occur, if no errors are encountered, or all will roll back to their original pre-transaction state if an error occurs.

Resources

With objects being created and destroyed all the time and transactions happening everywhere, it’s important for MTS to provide a means for sharing certain finite or expensive resources (such as database connections) across multiple objects. MTS does this using *resource managers* and *resource dispensers*.

A resource manager is a service that manages some type of durable data, such as account balance or inventory. Microsoft provides a resource manager in MS SQL Server. A resource dispenser manages non-durable resources, such as database connections. Microsoft provides a resource dispenser for ODBC database connections, and Borland provides a resource dispenser for BDE database connections.

When a transaction makes use of some type of resource, it *enlists* the resource to become a part of the transaction so that all changes made to the resource during the transaction will participate in the commit or rollback of the transaction.

Summary

MTS is a powerful addition to the COM family of technologies. By adding services such as lifetime management, transaction support, security, and transactions to COM objects without requiring significant changes to existing source code, Microsoft has leveraged COM into a more scalable technology, suitable for large-scale distributed development. This article took you through a tour of the basics of MTS, and next month we will go deeper into the specifics of MTS development with Delphi. Until then, it's time to go back and inspect those old COM objects on your hard disk and work on making them function in a stateless environment!

Steve Teixeira is the Vice President of Software Development at DeVries Data Systems, a Silicon Valley consulting and training firm. Send your questions, comments, and thoughts to steve@dvdata.com.
Steve wishes to thank Lino 'MTS' Tadros for his assistance with this article.