.NET Enterprise Services Performance

Richard Turner, Program Manager, XML Enterprise Services
Larry Buerk, Program Manager, XML Enterprise Services
Dave Driver, Software Design Engineer, XML Enterprise Services

Microsoft Corporation

March 2004

Applies to:
COM+ components
Microsoft .NET Enterprise Services

Summary: See the performance of native COM+ and .NET Enterprise Services components when applied to different activation and calling patterns. Get guidelines to make .NET Enterprise Services components execute just as quickly as C++ COM+ components, and get key recommendations to help you create high-performance .NET Enterprise Service components. (45 printed pages)

Download the associated EnterpriseServicesPerf.exe code sample.

Contents

Introduction

Developers who consider moving their COM+ code from "native" Visual C++® or Visual Basic® 6 to managed .NET Enterprise Services components sometimes raise concerns such as:

- Why should I switch to managed code?
- How much change will be required to my code?
- How will my Enterprise Services components perform?
- What is the future roadmap for COM+ and .NET Enterprise Services?

This paper discusses the points above, and particularly focuses on the performance question. Resources listed in Appendix 4: Further Reading discuss these subjects in more detail.

This document is targeted at developers and architects who have developed COM+ components and are considering migrating their code to .NET Enterprise Services.

Why Migrate to Managed Code?

There are many reasons for developers to develop their code in .NET. Some of the benefits are:

- **Increased Developer Productivity**. Developers often find that they have to write considerably less "plumbing code" when developing with .NET, which enables them to focus more on writing application logic. Also, most developers find that .NET provides a rich library of resources, organized in a clear and consistent manner, which results in an easier learning

curve compared to that of other technologies.

- **Increased code reliability and security**. A developer can more easily write reliable and secure code with .NET than with native code. This is due to features such as code access security and the common language runtime (CLR), which help to prevent .NET code from inadvertently affecting other running code and reduce the opportunity for hackers to use .NET code to disrupt or control an environment.
- **Enhanced performance and scalability**. Developers might notice an improvement in the performance and scalability of their code when it is migrated to .NET because all .NET languages support and are able to take advantage of features such as multithreading.
- **XCOPY deployment**. For most .NET applications, deployment is simply a matter of copying the necessary files to a folder on the hard drive and, optionally, registering shared components with the operating system. This is a much cleaner deployment strategy than other applications typically use.

For a discussion of the top 10 reasons why developers might want to migrate to .NET, see the Top 10 Reasons for Developers to Use the .NET Framework 1.1.

.NET also benefits systems administrators. For a list of the top 10 reasons why administrators might want to move to .NET, see the Top 10 Reasons for Systems Administrators to Use the .NET Framework 1.1.

For an introduction to methods for migrating C++ code to Managed C++, see the Introduction to Wrapping C++ Classes in the Managed Extensions for C++ Migration Guide.

How Much Change to My Code Will Be Required?

In most cases, there will be some manual effort required to port COM+ code to .NET Enterprise Services, depending on a number of factors, including which language your COM+ components are developed in. Visual Basic 6 developers, for example, will have some support from tools in the upcoming Visual Studio 2005 that convert class definitions and method signatures without modifying the body of subroutines and functions (except for calls to other converted methods).

C++ developers will have to do most of their COM+ code to .NET code conversion manually, although they might find that the converted code might be more concise, because most of the plumbing is implemented in the CLR instead of class libraries, such as the Active Template Library (ATL). C++ developers can also select which language they want to port their applications to, such as Managed C++, for the highest fidelity with their existing code base, or C#, which could result in even more concise code.

Porting COM+ Components to .NET Enterprise Services

Developers who migrate existing COM+ code from native programming languages and tools, such as Visual Basic 6 and Visual C++, will need to modify some of their existing code in order to convert it fully to .NET code. The amount of work that this entails depends upon the existing code base and the tools that are available. The following table summarizes the options to consider when migrating code to .NET.

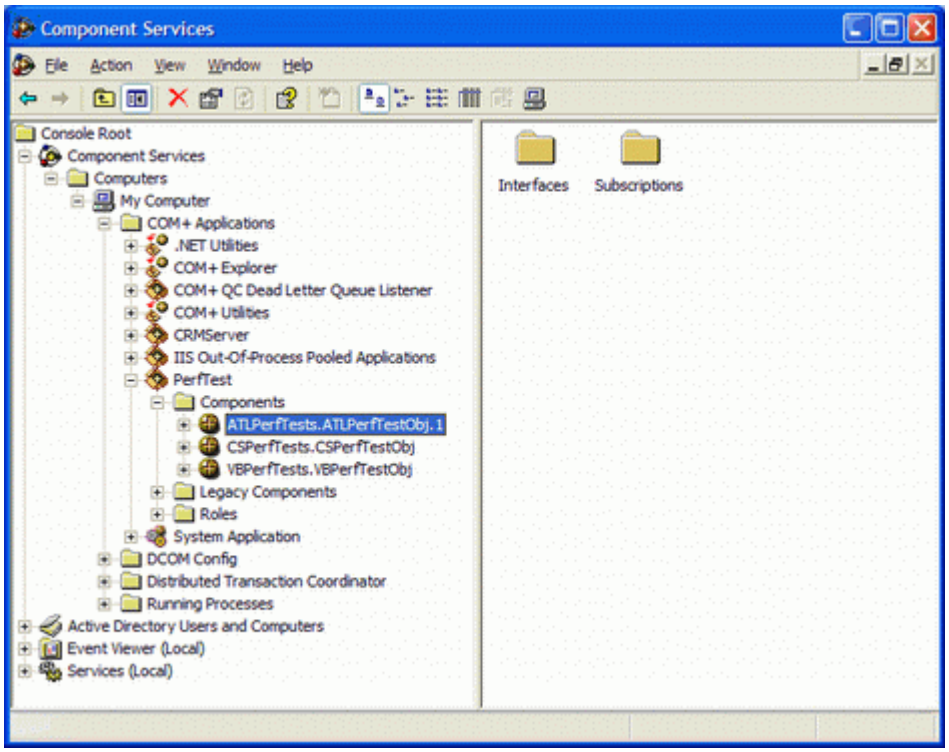
Convert from	Convert to	Code conversion tools available	Code conversion effort required
Visual Basic 6	Visual Basic .NET	Yes (in Visual Studio 2005)	Much of the Visual Basic 6 code ports directly to Visual Basic .NET. The Visual Basic 6 code migration tool in Visual Studio 2005 will convert most of the class and method declarations and types into Visual Basic .NET syntax.
Visual C++	Visual C++ .NET	No	C++ .NET is particularly useful for writing code that interoperates between native code and .NET.
Visual C++	Visual C#	No	C# syntax is similar to C++ in many ways. Developer effort is required in order to perform the conversion.

.NET Attributes

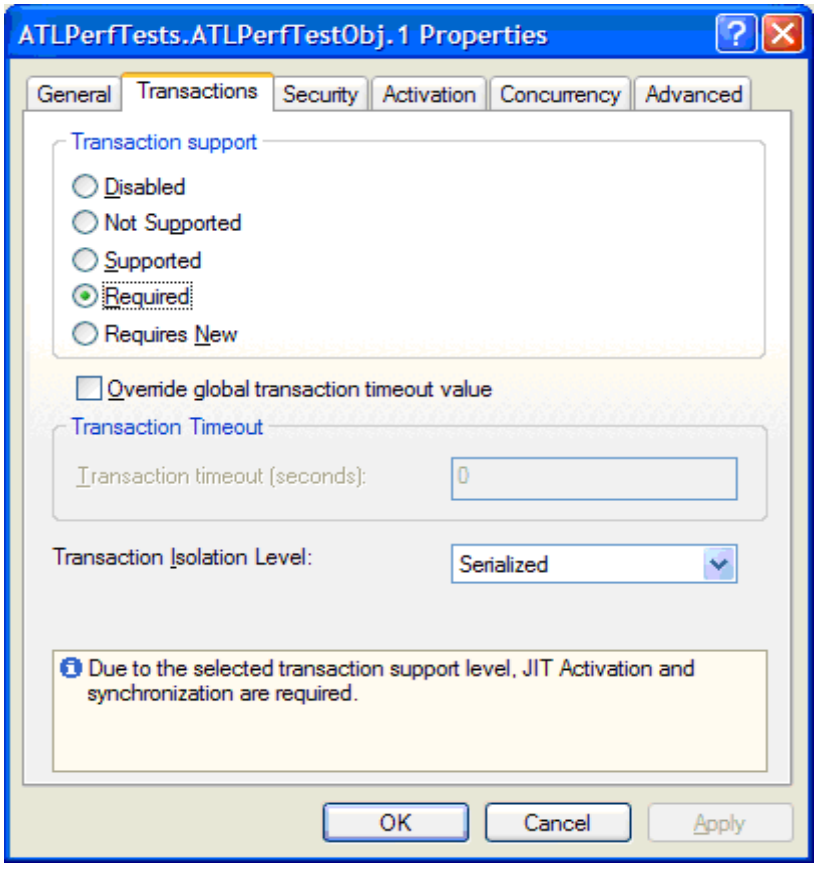
When COM+ components are installed, they must either be configured manually using the Component Services snap-in tool, or configured through script or code.

For example, suppose you want to mark your components as requiring transaction support, and you specify that each component's **AddSale()** method should automatically commit the transaction if it completes without error. To do this, manually configure the properties of the components and required methods in the COM+ Component Services management console:

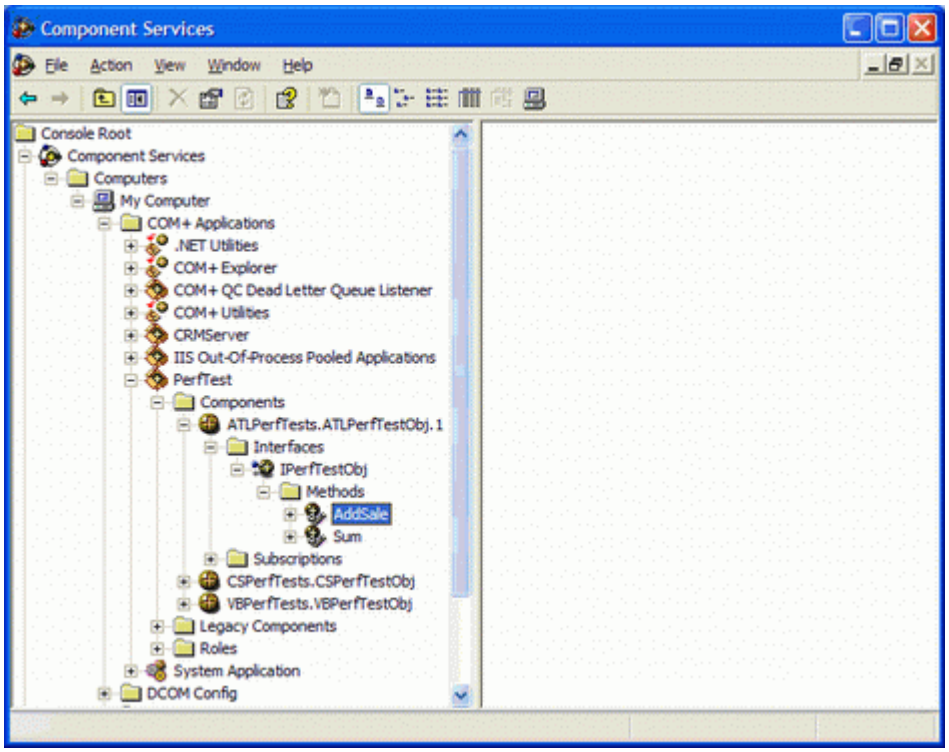
1. Open the Component Services Management Console and navigate to the correct application and component.



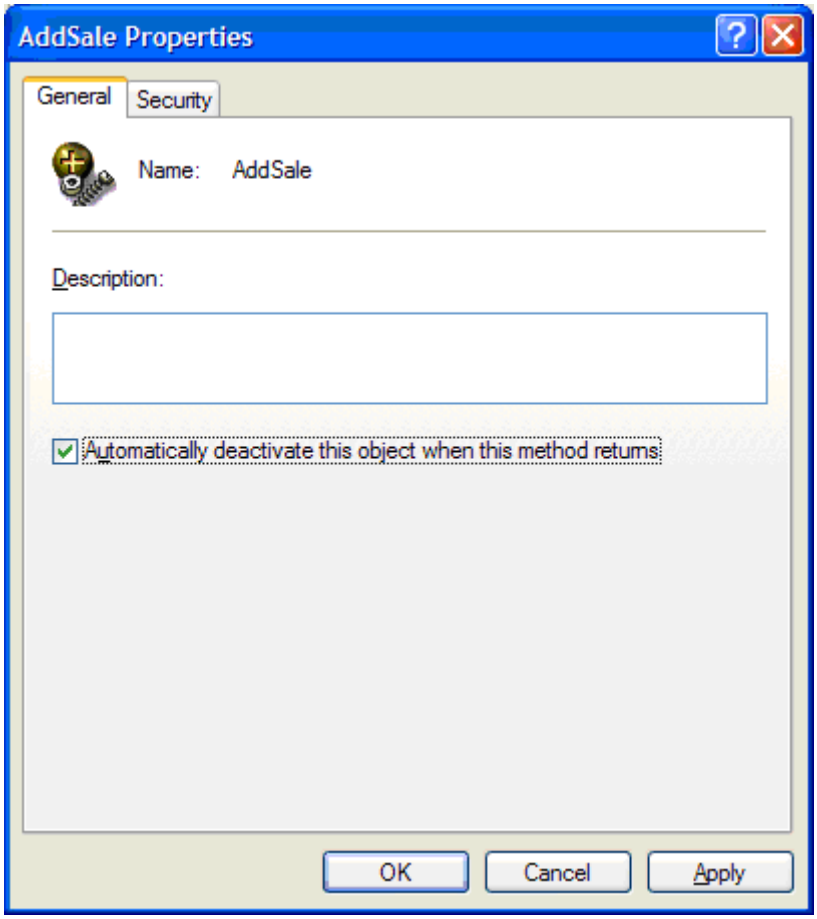
2. Open the properties of this component, click the **Transactions** tab, and then click **Required**.



3. Click **OK**, and navigate through the component explorer to find the ATLPerfTests object's **AddSale()** method.



4. Right-click **AddSale()**, open the method's properties, and then select the **Automatically deactivate this object when this method returns** check box.



Administrators typically configure deployment-oriented settings, such as security and identity settings, or runtime-oriented settings, such as role members and recycling. Developers configure development-oriented features for their components, such as transaction support. In COM+, however, it is difficult for developers to specify from within their code how their components should be configured. Visual Basic Developers can specify to some extent what transaction support the components need, but C++ developers cannot. In order to reliably and repeatedly install COM+ components, scripts, installer applications or installation instructions need to be written.

.NET simplifies component configuration by enabling developers to specify, within the component's code, what services their components need and how they should be configured. When the component is installed, the platform sets the configuration settings automatically, although these configuration settings can be changed after installation.

Because COM+ allows its configuration settings to be changed after installation, great care should be taken when changing many of these properties. For example, changing security settings might affect who can instantiate objects and call their methods. Removing transaction support, on the other hand, might cause a component to become unreliable, unpredictable, or even to lose or corrupt data.

Developers use attributes (enclosed in square brackets such as **[attribute]** for C# and C++, and angled brackets like **<attribute>** for Visual Basic .NET) to configure elements of the assembly, application, component, or method. For example, in the code below, we declare that the **SimpleTest** component requires transactions and that the transaction will complete automatically if the **AddSale()** method completes without error.

C#

```
    [Transaction(TransactionOption.RequiresNew)]
    public class SimpleTest: ServicedComponent
    {
       ...
       [AutoComplete]
public void AddSale(int orderNumber, int storeID, int
titleID, int qty)
       {
          ...
       }
       ...
    }
```

Visual Basic .NET

```
<Transaction(TransactionOption.RequiresNew)> _
Public Class VBTestObject : Inherits ServicedComponent
       ...
<AutoComplete> _
Public Function Sum(ByVal number1 As Integer, ByVal number2
As Integer) As Integer
          ...
       End Function
       ...
    End Class
```

The following table contains the commonly used attributes that can be applied to an Enterprise Services component and indicates whether they are safe to change after installation.

Attribute	Scope	Primary Owner	Safe to modify at runtime?
ApplicationAccessControl	Assembly	Developer	No. Developer might have written code with a hard or implied dependency on how the application is secured. Reducing access controls can compromise security of your system.
ApplicationActivation	Assembly	Administrator	Yes (with care). Changing to or from Library or Service can affect performance and can break queued components.
ApplicationID	Assembly	Developer	Yes (with care). Changing this can affect hard-coded component registration tools.

ApplicationName	Assembly	Developer	Yes (with care). Changing this can affect hard-coded component registration tools.
ApplicationQueuing	Assembly	Developer	No.
AutoComplete	Method	Developer	No. (See "Transactions" below.)
ComponentAccessControl	Class	Developer	No. Developer might have written code with a hard or implied dependency on how the component is secured. Reducing access controls can compromise security of your system.
ConstructionEnabled	Class	Administrator	Yes. The constructor string can be changed, but do not toggle construction on/off.
Description	Assembly Class Method Interface	Administrator	Yes.
EventTrackingEnabled	Class	Administrator	Yes.
InterfaceQueuing	Class Interface	Developer	No.
JustInTimeActivation	Class	Developer	No.
MustRunInClientContext	Class	Developer	No. The component might not be compatible with the client's context.
ObjectPooling	Class	Developer	Yes—Pool settings can be changed. Do NOT toggle object pooling on/off because this might break the component.
PrivateComponent	Class	Developer	No. The developer specifically wanted this component to not be publicly callable. The component might not have been tested to support unknown usage patterns. This could expose serious security threats if made public.
SecurityRole - Role Names	Assembly Class Interface	Developer	No—do NOT remove roles. The developer might have written code with an explicit dependency on specified roles, or implicit expectations on the presence of a role.
SecurityRole - Role Members	Assembly Class Interface	Administrator	Yes (with care). Opening access controls too widely can compromise the security of your system, and tightening them might restrict access too much.
Synchronization	Class	Developer	No.
Transaction	Class	Developer	No. Modifying a component's transaction support can compromise the system's reliability and integrity.

The table above also shows which configuration elements the component developer primarily owns and which the administrator primarily owns. It is not recommended to change any of the developer's settings unless you have detailed knowledge of how it will affect the code. Attributes related to security that were specified by the developer may be changed by the administrator, but great care must be taken to ensure that the security is configured so that access to the component or application is sufficiently restricted, but not overly so.

Attributes are a simple and effective way for developers to specify the configuration requirements of their components while enabling administrators to change configuration settings after installation. For details of the attributes that Enterprise Services provides, see the .NET Framework Class Library.

.NET Enterprise Services vs. COM+ Performance

To measure the performance of Enterprise Services compared to COM+, we created components in the following languages:

- Visual C++ .NET and ATL COM+
- Visual Basic 6 COM+
- C# and .NET Framework 1.1 Enterprise Services
- Visual Basic .NET and .NET Framework 1.1 Enterprise Services

Each component contains two public methods:

- **Sum()**: This trivial method adds two numbers together to simulate a lightweight operation that performs no disk or database access operations.
- **AddSale()**: This typical method is transacted and calls the private method **InsertSale()** that inserts a record into a table and completes the transaction before returning. This method illustrates the performance characteristics of a "real-world" method doing typical business application work.

We then created a test program that performed the following tests on each component:

- Repeatedly create/call/release. This test creates an object, calls it, and releases it inside a loop.
- Create/call repeatedly/release. This test instantiates an object outside of the loop, calls it several thousand times inside the loop, and releases the object once at the end.

The test program executed both of these tests against each component and measured the time taken to perform each test using a high-resolution timer that wrote the results to a comma-separated file. The results in this file were imported into Microsoft® Excel and analyzed. For code listings for each of these components, see Appendix 5: Performance Test Source Code.

The tests were run on machines with the configurations shown in the following table.

	Machine 1: Server Machine	**Machine 2: Client Machine / Single Machine**
CPU	Dual Pentium 4 Xeon 3.06 GHz	Dual Pentium 4 Xeon 2.8 GHz
RAM	1GB	1GB
Disk	Local SCSI	Local SCSI
Network	Gigabit Ethernet	Gigabit Ethernet
OS and .NET	Windows Server™ 2003 .NET Framework 1.1	Windows Server 2003 .NET Framework 1.1

The specific results you might see from running the test application on other hardware might differ by varying amounts from the results we report below. However, your results should be proportional to the results we present here.

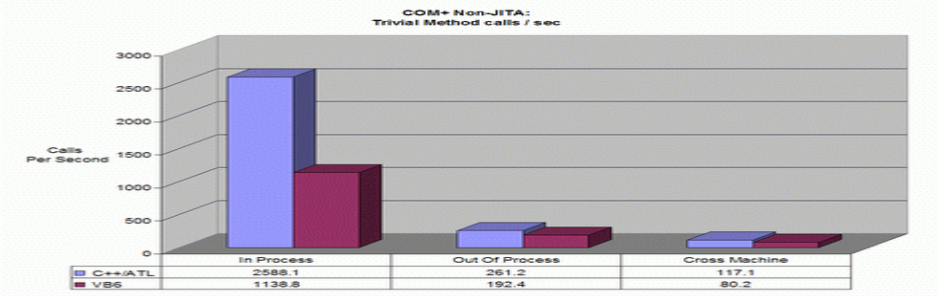
Test Results and Analysis

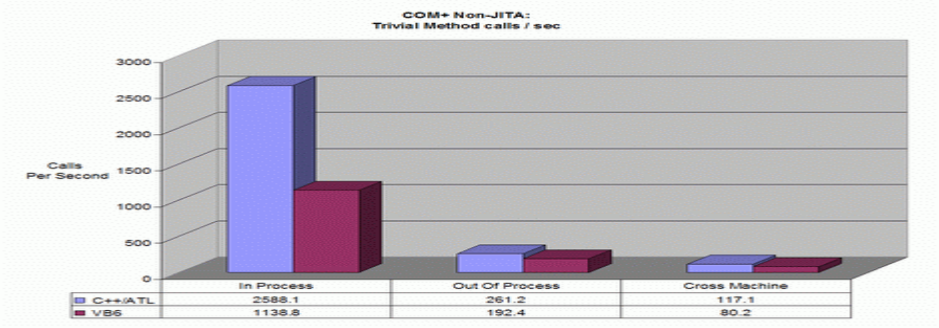
In the following sections we analyze the results of the performance tests run against the code discussed previously. These results were obtained from running the tests on the hardware and software listed above. The list of all results is included in Appendix 6: Test Results.

In the following charts that show these results, note that the taller the chart column or the larger the numbers, the better the performance.

Object Activation and Disposal Performance

First, let's look at the performance of native COM+ components developed using C++ and Visual Basic 6 to gain an understanding of how the COM+ infrastructure performs. The chart below shows the number of calls per second achieved by repeatedly creating an object, calling its trivial method, and releasing it.

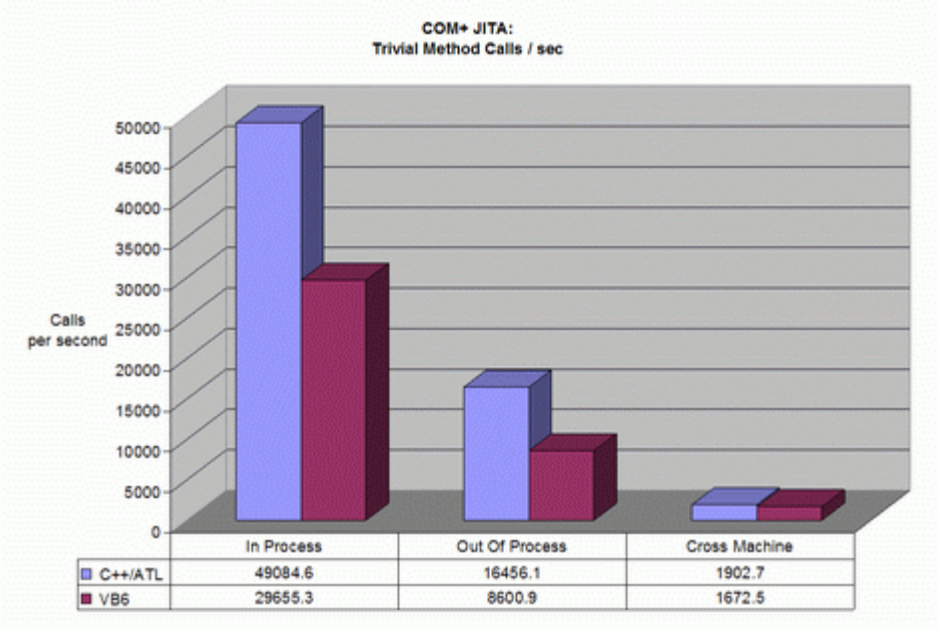




The original designers of Microsoft Transaction Server (MTS) 1.0 looked at similar data and found that the cross-process and cross-machine activation times are dominated by establishing the infrastructure necessary to deliver the call—the proxy, the DCOM (or cross-process) channel, the stub, and the context. This was a primary motivation for designing Just In Time (JIT) activation, which does the following:

- Allows the server component to control its own lifetime by calling **SetComplete()** or **SetAbort()** before returning
- Amortizes the cost of establishing the DCOM plumbing over multiple method calls

The following chart illustrates what happens if we run a modified test that takes advantage of JIT-activation by creating a single object, and repeatedly calling a trivial method that adds two numbers together and then releases the object at the end of the test loop.

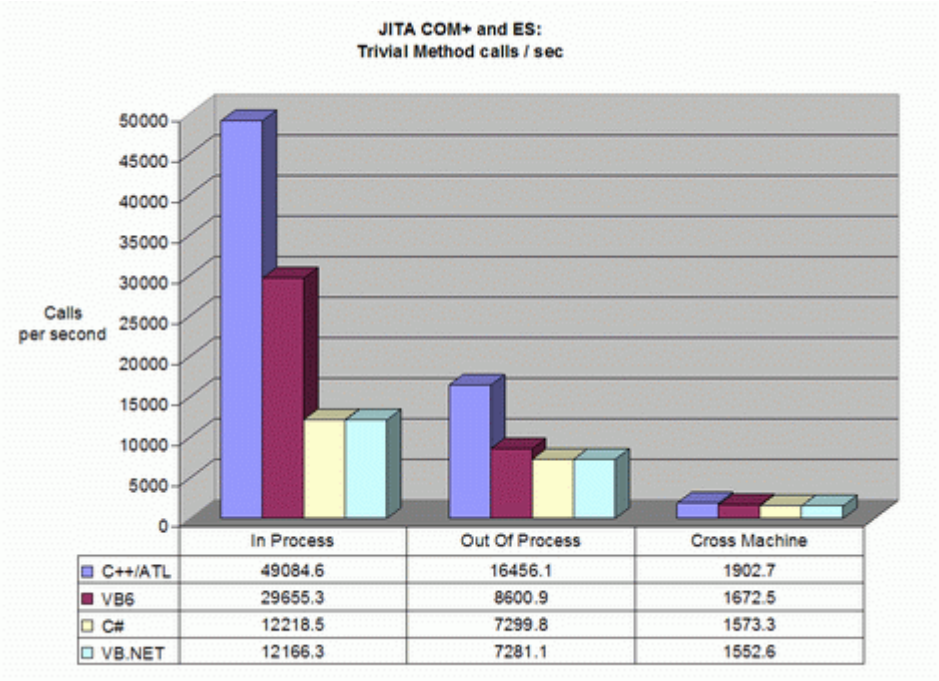


These results show a significant performance improvement in the number of calls per second when using JIT-activation. Using JIT-activation and Visual Basic 6 produces results that are almost 33 times faster than C++ without using JIT-activation (approximately 8600 Visual Basic 6 JIT-activated calls per second compared to approximately 261 Visual C++ non-JIT-activated calls per second).

Once you've established the plumbing necessary to do work, and when making cross machine calls, the network begins to dominate the performance. In this case, Visual Basic 6 and C++ are very close performance-wise with Visual Basic 6, performing 88% as fast as C++.

In Enterprise Services, extra work is done to establish the necessary plumbing. In particular, extra calls are required to construct the object and to release it. This means that if you were to compare the cost of simply creating and destroying an object without doing any work in the object, the cost of the extra round trips would dominate the performance comparison. In Visual Studio 2005, Enterprise Services will be enhanced to eliminate one of the activation round trips, yielding a 20-30% improvement in performance (compared to the .NET Framework 1.1) when using the "activate/single call/release" pattern. However, you should avoid this pattern if at all possible.

Given that C++ and Visual Basic 6 are so close in performance when using JIT-activation, it would be expected that Enterprise Services using C# and Visual Basic .NET would have approximately the same performance. The figure below shows this by running the above test that calls the trivial method.

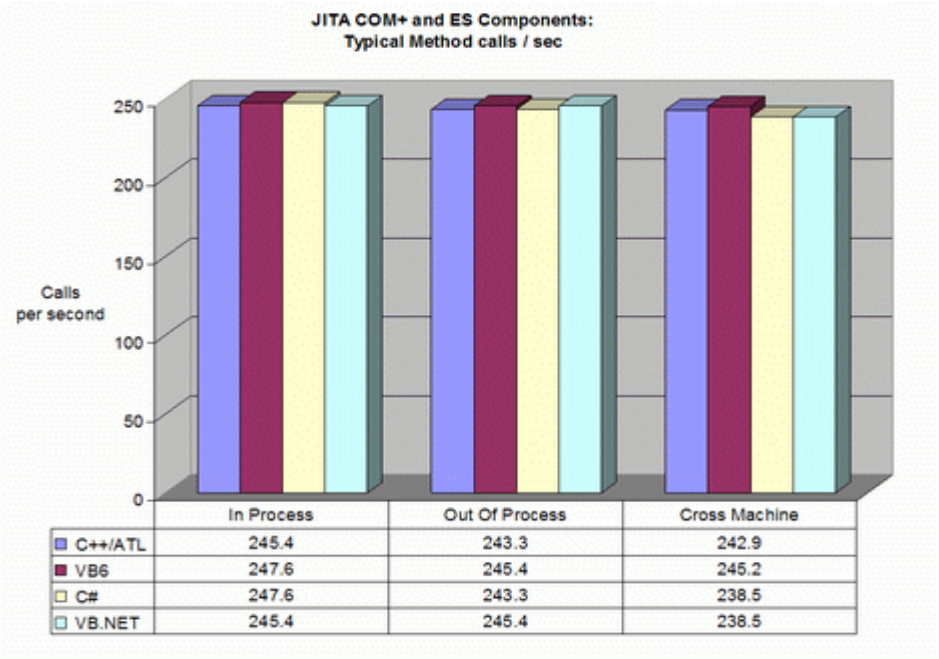


	In Process	Out Of Process	Cross Machine
C++/ATL	49084.6	16456.1	1902.7
VB6	29655.3	8600.9	1672.5
C#	12218.5	7299.8	1573.3
VB.NET	12166.3	7281.1	1552.6

This data illustrates the number of calls per second made against a trivial method that simply adds two integers, calls **SetComplete()** on the object's context, and returns the result. Most of the cost of activating and releasing the object is gone, but the cost of delivering the call is still there, due to operations such as marshaling the buffers and converting to a call stack.

Even with this very simple method, Enterprise Services is very close to the performance of Visual Basic 6 when going cross process. When calling across machines, all the languages perform very closely to each other.

However, a typical business application will have more complex work being done in the methods. The following chart shows the relative performance of the same application written in four different languages that repeatedly calls a typical method to open a database connection and execute a simple SQL statement while inside a distributed transaction.



	In Process	Out Of Process	Cross Machine
C++/ATL	245.4	243.3	242.9
VB6	247.6	245.4	245.2
C#	247.6	243.3	238.5
VB.NET	245.4	245.4	238.5

The preceding results show that, within the experimental error, all languages give equivalent results when doing significant work inside the method. COM+ native applications written using C++ and Visual Basic 6 using ADO perform at the same speed as C# or Visual Basic .NET applications using Enterprise Services. Note that it matters very little from a performance perspective if you are running cross-process or cross-machine.

Results Summary

The results above illustrate how important JIT-activation and the "create/repeat call/release" calling pattern are as an aid to ensuring that your components perform as well as possible.

Conclusion

We have shown some important reasons why migrating your code to .NET can be beneficial. We also discussed the performance characteristics of native COM+ and .NET Enterprise Services components when applied to different activation and calling patterns. By following guidelines, we showed that .NET Enterprise Services components execute just as quickly as C++ COM+ components. Appendix 1: Performance Recommendations, provides key recommendations that will help you create high performance .NET Enterprise Service components.

By consistently applying the techniques described here, you can convert your existing COM+ code to .NET Enterprise Service components today and benefit from the usability, security, and developer productivity advantages of the .NET Framework without sacrificing performance.

It is also important to note that converting your COM+ components to Enterprise Services components today will make your code easier to migrate to "Indigo" in the future. Appendix 2: "Indigo" and the Future of .NET provides a brief discussion of this topic.

Appendix 1: Performance Recommendations

The following sections provide tips and guidance for crafting COM+ and Enterprise Services components that are agile and offer high levels of performance. It is important to note that most of these suggestions apply equally to .NET Enterprise Services components as to native COM+ components.

Use Object Pooling & JIT-Activation where appropriate

As the test results above show, method calls are faster than component activations, and activations of unmanaged components are faster than activations of Enterprise Services components. Therefore, in order to make component-based applications as fast as possible, it is important to minimize the number of component activations and disposals in your code.

Two services offered by COM+ that provide ways to minimize object activations are:

1. Just-in-time (JIT) activation, which, as explained previously, is a COM+ service that enables an object to be seamlessly deactivated while the caller holds an active reference to that object. The client simply calls methods on the object and COM+ dynamically manages the allocation of objects to serve the request.
2. Object pooling, which enables objects to be kept active in a pool, ready to be used by any client that requests an instance of that type of component. COM+ manages the pool for you, handling the details of object activation and reuse according to the criteria you have specified—for example, the size of the pool.

By holding and reusing references to pooled and JIT-activated components, you can minimize component activations and disposals and achieve high levels of performance.

For more details about COM+ JIT-activation and object pooling, see the Platform SDK: COM+ (Component Services) documentation.

Avoid round trips

In order to optimize the performance of COM+ components, it is important to minimize the number of cross-process or cross-machine calls made between caller and component. Every method call made on a COM+ component results in the call transitioning across processes (and even machines), and every transition takes time. It is therefore essential to ensure that any method calls made to COM+ objects are kept to a minimum. A good way to achieve this is to design COM+ components with methods that perform as much work as possible in a single call, even if that means designing components that deviate from architectural purity.

Optimize your use of COM+ services

While COM+ provides a broad and valuable range of services, it is important to use these services wisely. If COM+ provides a service that your component needs, then you should readily adopt that service, as it will most likely be the highest performance implementation. However, if you do not need a service, you might not want to use it, since your components might be performing unnecessary work and will not run as fast as possible.

Use COM-marshalable parameters

If the methods of your Enterprise Services component accept parameters through which the caller passes data, we very strongly suggest that you try to use types that are easy to marshal between COM and .NET, such as:

- Boolean
- Byte, SByte
- Char
- DateTime
- Decimal
- Single, Double
- Guid
- Int16, UInt16, Int32, UInt32, Int64, UInt64
- IntPtr, UIntPtr
- String

If you use only these types and avoid passing other complex types (such as structures or arrays), the .NET serializer can optimize the call-processing stack and serialize your call straight onto the wire (for RPC) or onto the virtual wire (for LRPC). This lets your call execute much faster. However, if your methods require complex types, your code will call through the normal DCOM call stack, which incurs extra processing.

Avoid using Finalizers

Avoid implementing Finalizers (the **~Classname()** destructor in C# and C++) in your Enterprise Services components. Finalization is a single-threaded operation of the garbage collector. Finalizing Enterprise components can take considerable time to complete, hindering garbage collector performance. Your application remains blocked while the garbage collector engine executes, so if the garbage collector takes a long time to complete, the performance of your whole application will be impeded.

Instead, consider overriding **Dispose(bool)** in your objects and performing finalization type actions when **Dispose(true)** is called. Also, try to keep such termination code as clean, safe, and simple as possible.

Avoid creating single-threaded COM+ Components

Objects that do not support concurrent access by more than one thread are marked as supporting single-threaded apartment (STA) semantics. Components that do support multiple concurrent threads accessing the same instance are marked as being multithreaded-apartment(MTA)-aware.

Because .NET Enterprise Services components are always marked as supporting both STA and MTA, they are not included in the remainder of this discussion.

All Visual Basic 6 COM+ components are STA only. C++ COM+ developers can choose whether to mark their components as STA, MTA, or both.

The potential problem with STA COM+ components is that an object can only execute on a single thread, and that thread is the only one that will execute the object's methods. This serialization enables developers to more easily write STA components, but at the cost of performance (because cross-domain marshaling will often be required) and scalability (because only one thread will ever execute in an STA).

We suggest that you avoid creating or using STA components wherever possible, particularly where scalability is important. It is particularly important to avoid STA threading if a component calls other COM+ components. Such calls often require a thread switch, which blocks all other COM+ components in that apartment.

To compound things further, the garbage collector's Finalizer blocks when it makes calls on the STA thread that owns an

object. This serializes the finalization process onto a single thread, which, as explained in the previous topic, can significantly reduce a system's performance.

Appendix 2: "Indigo" and the Future of .NET

You might have heard about a new platform for connected applications, code-named "Indigo," which Microsoft is currently developing. So, what is "Indigo"?

"Indigo" is Microsoft's strategic technology platform for developing Service Oriented Connected Applications and was introduced at the 2003 Professional Developers Conference (PDC) in Los Angeles.

"Indigo" consolidates into one technology stack the concepts, features, and functionality of:

- COM
- DCOM
- COM+/Enterprise Services
- ASMX/Web Services
- .NET Remoting
- Web Service Enhancements
- Elements of MSMQ

"Indigo" is a multi-layered platform that abstracts the notion of a service from the protocols and transports necessary to expose that service to callers. In order to interoperate with as many systems as possible, "Indigo" fully supports advanced Web services (WS-*) protocols through HTTP, TCP, and IPC. Microsoft currently plans to ship "Indigo" in the Microsoft® Windows® code-named "Longhorn" timeframe, along with "Indigo" support for Windows XP and Windows Server 2003.

With the future introduction of "Indigo", you might be wondering whether existing technologies such as .NET Enterprise Services (or ASMX and Remoting, for that matter) are still valid for developing connected applications today. ASMX & WSE, Enterprise Services, Remoting and MSMQ are absolutely the technologies of choice for today's enterprise-class solutions. When used appropriately, they provide a sound platform on which to develop applications until "Indigo" is released and broadly available.

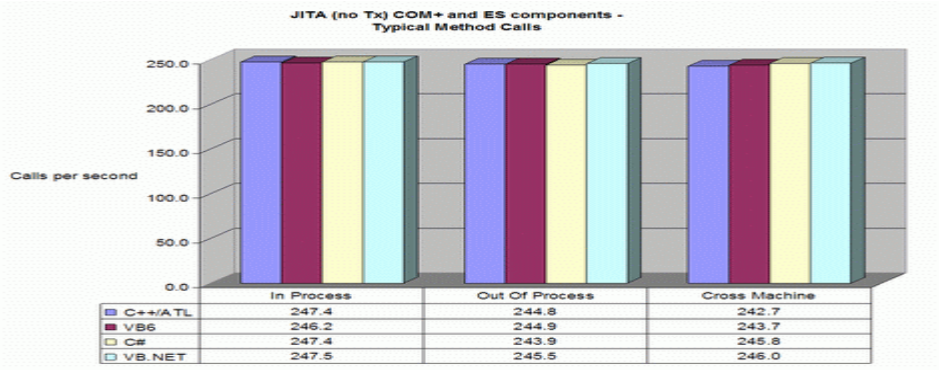
If you write new applications using .NET and migrate existing applications to .NET, you will benefit from improved security, reliability, management, and scalability. Also, your code will then be much easier to upgrade to "Indigo" than native code. Watch for detailed guidance on how to prepare your applications for migration to "Indigo", and how to interoperate with "Indigo" from existing technologies on MSDN in the future.

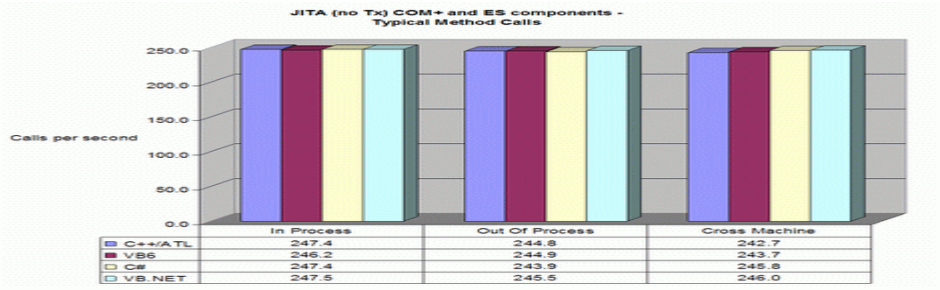
For a high-level introduction to "Indigo", refer to the MSDN Magazine article, Code Name Indigo: A Guide to Developing and Running Connected Systems with Indigo. In this article, the author provides an architectural overview of what the future Microsoft application platform will look like.

For more general information about "Indigo", see Microsoft "Indigo" Frequently Asked Questions.

Appendix 3: Effect of Distributed Transactions on Performance

A question that might arise from the tests above is, "How much impact do COM+ distributed transactions have on the performance of these components?" In order to answer this question, we re-ran our tests after turning off the "requires transaction" setting in COM+ for each of the components; the results were as shown in the following chart.





As can be seen from the chart above, the performance of the components without COM+ transaction support is practically identical to the performance of the components with transactions turned on. This clearly illustrates that the impact of COM+ transactions is negligible in these tests.

Appendix 4: Further Reading

Upgrading Microsoft Visual Basic 6.0 to Microsoft Visual Basic .NET

> Covers upgrading your Visual Basic 6.0 applications to Visual Basic .NET with programming tips, tricks, and side-by-side code comparisons.

Programming with Managed Extensions for Microsoft Visual C++ .NET

> Updated for Visual C++ .NET 2003, this book offers developers in-depth and expert coverage of the new features in the compiler and linker extensions to the language.

.NET Enterprise Services and COM+ 1.5 Architecture

> Discusses how Microsoft .NET and Enterprise Services fit together and how to build, control, manage, and secure COM+/Enterprise Services components.

Performance page on the .NET Framework Developer Center

> A great collection of links and resources for further investigation into writing high-performance code and diagnosing problems when they arise.

Performance Tips and Tricks in .NET Applications

> A collection of tips and hints on how to make your .NET applications perform well.

Writing Faster Managed Code: Know What Things Cost

> A detailed breakdown of what various actions in .NET code cost.

Garbage Collector Basics and Performance Hints

> Describes how the garbage collector works, how the garbage collector affects your code, and how to write your code to minimize the effects of garbage collection.

Performance Considerations for Run-Time Technologies in the .NET Framework

> Discusses topics such as garbage collection and memory usage, JIT, threading, .NET Remoting, and security.

Appendix 5: Performance Test Source Code

C++\ATL Component

Header File

```
// ATLPerfTestObj.h : Declaration of the CATLPerfTestObj

#pragma once
#include "ATLPerfTests.h"
#include "resource.h"       // main symbols
#include <comsvcs.h>
#include <mtxattr.h>


// CATLPerfTestObj
```

```
class ATL_NO_VTABLE CATLPerfTestObj :
   public CComObjectRootEx<CComSingleThreadModel>,
   public CComCoClass<CATLPerfTestObj, &CLSID_ATLPerfTestObj>,
   public IDispatchImpl<IPerfTestObj, &IID_IPerfTestObj, &LIBID_ATLPerfTestsLib,
     /*wMajor =*/ 1, /*wMinor =*/ 0>
{
public:
   CATLPerfTestObj()
   {
   }

   DECLARE_PROTECT_FINAL_CONSTRUCT()

   HRESULT FinalConstruct()
   {
      return S_OK;
   }

   void FinalRelease()
   {
   }

DECLARE_REGISTRY_RESOURCEID(IDR_ATLPERFTESTOBJ)

DECLARE_NOT_AGGREGATABLE(CATLPerfTestObj)

BEGIN_COM_MAP(CATLPerfTestObj)
   COM_INTERFACE_ENTRY(IPerfTestObj)
   COM_INTERFACE_ENTRY(IDispatch)
END_COM_MAP()


// IPerfTestObj
public:
   STDMETHOD(Sum)(LONG number1, LONG number2, LONG* result);
   STDMETHOD(AddSale)(LONG orderNumber, LONG storeID, LONG titleID,
LONG qty);

private:
   HRESULT InsertSaleRecord(LONG orderNumber, LONG storeID, LONG
titleID, LONG qty);

};

OBJECT_ENTRY_AUTO(__uuidof(ATLPerfTestObj), CATLPerfTestObj)
```

Source File

```
// ATLPerfTestObj.cpp : Implementation of CATLPerfTestObj

#include "stdafx.h"
#include "ATLPerfTestObj.h"
#include ".\atlperftestobj.h"

#include "atlstr.h"

#import "c:\Program Files\Common Files\System\ADO\msado15.dll" rename_namespace("ADO")
         rename("EOF", "EndOfFile")
using namespace ADO;

// CATLPerfTestObj

//   Here we perform a simple operation to simulate a trivial method.
STDMETHODIMP CATLPerfTestObj::Sum(LONG number1, LONG number2, LONG* result)
{
   //   Obtain the object context.
   IObjectContext* ctx = NULL;
HRESULT hr = CoGetObjectContext(IID_IObjectContext,
(LPVOID*)&ctx);
   if(SUCCEEDED(hr))
   {
      //   Perform the calculation.
      *result = number1 + number2;

      //   Commit the transaction.
      ctx->SetComplete();
      ctx->Release();
   }

   return hr;
}

STDMETHODIMP CATLPerfTestObj::AddSale(LONG orderNumber, LONG storeID, LONG titleID, LONG qty)
{
   //   Get the COM+ object context.
   IObjectContext* ctx = NULL;
    HRESULT hr = CoGetObjectContext(IID_IObjectContext, (LPVOID*)&ctx);

   //   Check if we have a COM+ context.
   if(SUCCEEDED(hr))
   {
      //   The default action is to abort.
      ctx->SetAbort();

      //   Insert the record into the database.
```

```
      hr = InsertSaleRecord(orderNumber, storeID, titleID, qty);

      //   Check the result of the insert operation.
      if(SUCCEEDED(hr))
      {
         //   If all went okay, mark the transaction as
//   complete.
         ctx->SetComplete();
      }

      //   Clean up the context pointer.
      ctx->Release();
      ctx = NULL;
   }

//   Return the overall result. Note that the result will only
//   be S_OK if the insert operation executed without error.
   return hr;
}

// Inserts the record of a sale into the database.
HRESULT CATLPerfTestObj::InsertSaleRecord(LONG orderNumber, LONG storeID,
         LONG titleID, LONG qty)
{
   //   The default result is to return a failure.
   HRESULT hr = E_FAIL;

   try
   {
      //   Format the SQL that you want the server to execute.
      CString str;
      str.Format("insert into sales (order_no, store_id, \
title_id, order_date, qty) values (%i, %i, %i, \
GetDate(), %i)",
         orderNumber, storeID, titleID, qty);

      //   Open up a connection to the database.
      _ConnectionPtr   cn("ADODB.Connection");
      cn->Open("Provider=SQLOLEDB;SERVER=localhost;Integrated \
Security=SSPI;DATABASE=ESPERFTESTDB",
         "", "", adConnectUnspecified);

      //   Execute the command.
      _variant_t rs;
      rs = cn->Execute(_bstr_t(str), &rs, adCmdText);
      hr = S_OK;

      //   Explicitly close the connection.
      cn->Close();
   }
   catch(_com_error e)
   {
      //   Any problems, return an error.
      hr = e.Error();
   }

   return hr;
}
```

Visual Basic 6 Component

```
Private Function IPerfTestObj_Sum(ByVal nA As Long, ByVal nB As Long) As Long
    IPerfTestObj_Sum = nA + nB

    GetObjectContext.SetComplete
End Function

Sub IPerfTestObj_AddSale(ByVal OrderNumber As Long, ByVal StoreID As Long,
         ByVal TitleID As Long, ByVal Qty As Long)
    ' The default is to abort if an error is thrown.
    GetObjectContext.SetAbort

    Call InsertSaleRecord(OrderNumber, StoreID, TitleID, Qty)

    ' This did not fail, so you can complete the transaction.
    GetObjectContext.SetComplete
End Sub

Sub InsertSaleRecord(ByVal OrderNumber As Long, ByVal StoreID As Long,
         ByVal TitleID As Long, ByVal Qty As Long)
    Dim command As String

    connDB.Open ("Provider=SQLOLEDB;SERVER=localhost;Integrated" + _
"Security=SSPI;DATABASE=ESPERFTESTDB")

    command = "insert into sales (store_id, order_no, order_date, qty, title_id)
         values (" & StoreID & ", " & OrderNumber & ", GetDate(),
         " & Qty & ", " & TitleID & ")"

    connDB.Execute (command)
    connDB.Close
End Sub
```

C# Component

```
using System;
using System.EnterpriseServices;
using System.Data.SqlClient;
using System.Reflection;
using System.Runtime.InteropServices;

using perftestsinterop;

[assembly: AssemblyVersion("1.0.*")]
[assembly: AssemblyKeyFile("..\\..\\sign.key")]
[assembly: ApplicationName("PerfTest")]
[assembly: ApplicationActivation(ActivationOption.Server)]
[assembly: ApplicationAccessControl(false)]

namespace CSPerfTests
{
   [Guid("0BA5534E-8544-42e2-A909-3265105BEA09")]
   [Transaction(TransactionOption.Required)]
   public class CSPerfTestObj : ServicedComponent, IPerfTestObj
   {
      public CSPerfTestObj()
      {
      }

      //   Here you perform a simple operation to simulate a
//   trivial method.
      public int Sum(int number1, int number2)
      {
         int result = 0;

         //   Perform the calculation.
         result = number1 + number2;

         //   Commit the transaction.
         ContextUtil.SetComplete();

         return result;
      }

      //   Add a new sale item in a transaction.
      public void AddSale(int orderNumber, int storeID, int
titleID, int qty)
      {
         try
         {
            //   Insert the record into the database.
            InsertSaleRecord(orderNumber, storeID, titleID,
qty);

            //   If all went okay, mark the transaction as
//   complete.
            ContextUtil.SetComplete();
         }
         catch(Exception)
         {
            ContextUtil.SetAbort();
            throw;
         }
      }

      // Insert the record of a sale into the database.
      private void InsertSaleRecord(int orderNumber, int storeID,
int titleID, int qty)
      {
         //   Format the SQL that you want the server to
//   execute.
         string commandStr = string.Format("insert into sales\
    (order_no, store_id, title_id, order_date, qty) \
    values ({0}, {1}, {2}, GetDate(), {3})",
             orderNumber, storeID, titleID, qty);

         //   Open up a connection to the database.
         SqlConnection connection = new
SqlConnection("SERVER=localhost;Integrated
Security=SSPI;DATABASE=ESPERFTESTDB");
         connection.Open();

         //   Execute the command.
         SqlCommand cmd = new SqlCommand(commandStr,
connection);
         cmd.ExecuteNonQuery();

         //   Explicitly close the connection.
         connection.Close();
      }
   }
}
```

Visual Basic .NET Component

```
Imports System
Imports System.EnterpriseServices
Imports System.Data.SqlClient
Imports System.Reflection
```

```
Imports System.Runtime.InteropServices

Imports perftestsinterop

<Assembly: AssemblyVersion("1.0.0")>
<Assembly: AssemblyKeyFile("..\..\sign.key")>
<Assembly: ApplicationName("PerfTest")>
<Assembly: ApplicationActivation(ActivationOption.Server)>
<Assembly: ApplicationAccessControl(False)>

<Guid("53400EEB-E8C1-4D15-81D4-AFAC896B34FA"), _
Transaction(TransactionOption.Required)> _
    Public Class VBPerfTestObj
    Inherits ServicedComponent
    Implements IPerfTestObj

    '--   Here you perform a simple operation to simulate a
    '--   trivial method.
    Public Function Sum(ByVal number1 As Integer, ByVal number2 As
Integer) As Integer _
    Implements IPerfTestObj.Sum
        Dim result As Integer = 0

        '--   Perform the calculation.
        result = number1 + number2

        '--   Commit the transaction.
        ContextUtil.SetComplete()

        Return result
    End Function

    '--   Add a new sale item in a transaction.
    Public Sub AddSale(ByVal orderNumber As Integer, ByVal storeID As
Integer, ByVal titleID As Integer, ByVal qty As Integer) _
    Implements IPerfTestObj.AddSale
        Try
            '--   Insert the record into the database.
            InsertSaleRecord(orderNumber, storeID, titleID, qty)

            '--   If all went okay, mark the transaction as complete.
            ContextUtil.SetComplete()

        Catch
            ContextUtil.SetAbort()
            Throw
        End Try

    End Sub

    '-- Insert the record of a sale into the database.
    Private Sub InsertSaleRecord(ByVal orderNumber As Integer, ByVal
storeID As Integer, ByVal titleID As Integer, ByVal qty As
Integer)
        '--   Format the SQL that you want the server to execute.
        Dim commandStr As String
        commandStr = String.Format("insert into sales (order_no," + _
"store_id, title_id, order_date, qty) values ({0}, {1}, _
{2}, GetDate(), {3})", _
            orderNumber, storeID, titleID, qty)

        '--   Open up a connection to the database.
        Dim connection As SqlConnection = New
        SqlConnection("SERVER=localhost;Integrated
            Security=SSPI;DATABASE=ESPERFTESTDB")
            connection.Open()

        '--   Execute the command.
        Dim cmd As SqlCommand = New SqlCommand(commandStr, connection)
        cmd.ExecuteNonQuery()

        '--   Explicitly close the connection.
        connection.Close()
    End Sub

End Class
```

Test Application

```
using System;
using System.Reflection;
using System.Diagnostics;
using System.Threading;
using System.Runtime.InteropServices;
using System.EnterpriseServices;
using System.Text;
using System.IO;
using COMAdmin;

using ATLPerfTestsLib;
using CSPerfTests;
using VBPerfTests;

[assembly: AssemblyVersion("1.0.*")]
```

```
namespace TestApp
{
   [Flags]
   enum TestType
   {
      NoOp        = 0x0,
      LightMethod = 0x1,
      HeavyMethod = 0x2,
      Activations = 0x40000000,

      ActivateLight = Activations | LightMethod,
      ActivateHeavy = Activations | HeavyMethod,

   }

   public class App
   {
      public static HighResolutionTimer Timer;
      public static int RunID = 0;
      public static TextWriter outputFile;

      [STAThread]
      public static void Main(string[] args)
      {
         Timer = new HighResolutionTimer();
         RunID = (int)Timer.TickCount;

         Console.WriteLine("\a");

         outputFile = new StreamWriter(new
FileStream("results.csv", FileMode.Create,
FileAccess.Write), Encoding.UTF8);

         Type t;
         long iterations = 5000;

         //   Test empty activations and default disposals.
         outputFile.WriteLine("Test Type, Number of
Iterations, Object Type, Elapsed Time (sec),
Operations/sec");

         t = typeof(ATLPerfTestObjClass);
         TestActivations(t, iterations, TestType.Activations,
ApartmentState.MTA);

         t = Type.GetTypeFromProgID("VB6Perf.HPerf");
         TestActivations(t, iterations, TestType.Activations,
ApartmentState.STA);

         t = typeof(CSPerfTestObj);
         TestActivations(t, iterations, TestType.Activations,
ApartmentState.MTA);

         t = typeof(VBPerfTestObj);
         TestActivations(t, iterations, TestType.Activations,
ApartmentState.MTA);


         t = typeof(ATLPerfTestObjClass);
         TestActivations(t, iterations,
TestType.ActivateLight, ApartmentState.MTA);

         t = Type.GetTypeFromProgID("VB6Perf.HPerf");
         TestActivations(t, iterations,
TestType.ActivateLight, ApartmentState.STA);

         t = typeof(CSPerfTestObj);
         TestActivations(t, iterations,
TestType.ActivateLight, ApartmentState.MTA);

         t = typeof(VBPerfTestObj);
         TestActivations(t, iterations,
TestType.ActivateLight, ApartmentState.MTA);

         t = typeof(ATLPerfTestObjClass);
         TestActivations(t, iterations,
TestType.ActivateHeavy, ApartmentState.MTA);

         t = Type.GetTypeFromProgID("VB6Perf.HPerf");
         TestActivations(t, iterations,
TestType.ActivateHeavy, ApartmentState.STA);

         t = typeof(CSPerfTestObj);
         TestActivations(t, iterations,
TestType.ActivateHeavy, ApartmentState.MTA);

         t = typeof(VBPerfTestObj);
         TestActivations(t, iterations,
TestType.ActivateHeavy, ApartmentState.MTA);

         // Increase the number of iterations by a
// couple of orders of magnitude in order to get good
// Trivial Method Test numbers:
         t = typeof(ATLPerfTestObjClass);
         TestActivations(t, iterations*100,
TestType.LightMethod, ApartmentState.MTA);

         t = Type.GetTypeFromProgID("VB6Perf.HPerf");
```

```
         TestActivations(t, iterations*100,
TestType.LightMethod, ApartmentState.STA);

         t = typeof(CSPerfTestObj);
         TestActivations(t, iterations*100,
TestType.LightMethod, ApartmentState.MTA);

         t = typeof(VBPerfTestObj);
         TestActivations(t, iterations*100,
TestType.LightMethod, ApartmentState.MTA);

         t = typeof(ATLPerfTestObjClass);
         TestActivations(t, iterations, TestType.HeavyMethod,
ApartmentState.MTA);

         t = Type.GetTypeFromProgID("VB6Perf.HPerf");
         TestActivations(t, iterations, TestType.HeavyMethod,
ApartmentState.STA);

         t = typeof(CSPerfTestObj);
         TestActivations(t, iterations, TestType.HeavyMethod,
ApartmentState.MTA);

         t = typeof(VBPerfTestObj);
         TestActivations(t, iterations, TestType.HeavyMethod,
ApartmentState.MTA);

         outputFile.Close();

         Console.WriteLine("\a");
      }

      private static void TestActivations(Type type, long
iterations, TestType test, ApartmentState aptype)
      {
         // Create a new thread to control the
// apartment type of running thread.
         TestRunner runner = new TestRunner(type, iterations,
test);
         Thread thread = new Thread(new
ThreadStart(runner.Run));
         thread.ApartmentState = aptype;
         thread.Start();
         thread.Join();
         COMAdminCatalog catalog = new COMAdminCatalogClass();
         catalog.ShutdownApplication("PerfTest");
      }

      class TestRunner
      {
         Type type;
         long iterations;
         TestType test;

         string testTypeName;
         string componentTypeName;

         public TestRunner(Type type, long iterations,
TestType test)
         {
            this.type = type;
            this.iterations = iterations;
            this.test = test;

            //   Form the test type and component type
//   name strings for later use.
            switch(test)
            {
               case TestType.Activations:
                  testTypeName = "Activations, ";
                  break;

               case TestType.ActivateLight:
                  testTypeName = "Activations with \
Light Method Call, ";
                  break;

               case TestType.ActivateHeavy:
                  testTypeName = "Activations with \
Heavy Method Call, ";
                  break;

               case TestType.LightMethod:
                  testTypeName = "Light Method \
Calls, ";
                  break;

               case TestType.HeavyMethod:
                  testTypeName = "Heavy Method \
Calls, ";
                  break;
            };

            if (type == typeof(ATLPerfTestObjClass))
               componentTypeName = "C++/ATL, ";
            else if(type == typeof(CSPerfTestObj))
               componentTypeName = "C#, ";
            else if(type == typeof(VBPerfTestObj))
```

```
               componentTypeName = "VB.NET, ";
            else
               componentTypeName = "VB6, ";
         }

         void TestLoop(long iterations)
         {
            bool timingActivations = (this.test &
TestType.Activations) != 0;
            perftestsinterop.IPerfTestObj o = null;

            // If you are not timing activations, do this
// outside the loop.
            if(!timingActivations)
            {
               o = (perftestsinterop.IPerfTestObj)
Activator.CreateInstance(type);
            }

            // Perform the test.
            for (int i = 0; i < iterations; i++)
            {
               if(timingActivations)
               {
                  o = (perftestsinterop.IPerfTestObj)
    Activator.CreateInstance(type);
               }

               //   Work out which test you need to
//   perform and do it. Pull out the
//   activation bit so you are left
//   with the kind of method you want to
//   run.
               switch(test & ~TestType.Activations)
               {
                  case TestType.NoOp:
                     break;

                  case TestType.LightMethod:
                     o.Sum(100, 200);
                     break;

                  case TestType.HeavyMethod:
                     int count =
(int)Timer.TickCount;
                     o.AddSale(RunID, count, count
% 10000, count % 10);
                     break;
               };

               if(timingActivations)
               {
                  //   Dispose of the object now.
//   You are done.
                  if (type.IsCOMObject)
                  {
                     Marshal.ReleaseComObject(o);
                  }
                  else
                  {
                     ServicedComponent.DisposeObject((ServicedComponent)o);
                  }
               }
            }
         }

         public void Run()
         {
            float elapsedTime = 0.0f;

            // Prepare for the run.
            TestLoop(10);

            //   Stop and reset the timer.
            Timer.Stop();
            Timer.Reset();
            Timer.Start();

            TestLoop(iterations);

            // When the loop has ended, stop the timer and
// return the elapsed time.
            elapsedTime = Timer.ElapsedTime;
            Timer.Stop();

            //   Output the result.
            StringBuilder sb = new StringBuilder();
            sb.Append(testTypeName);
            sb.Append(iterations.ToString());
            sb.Append(", ");
            sb.Append(componentTypeName);
            sb.Append(elapsedTime.ToString());
            sb.Append(", ");
            sb.Append((iterations/elapsedTime).ToString());
            outputFile.WriteLine(sb.ToString());

            GC.WaitForPendingFinalizers();
         }
```

```
        }
    }
}
```

Appendix 6: Test Results

TX	Scope	JITA	Method Type	Object Type	Operations/second
Tx	In Process	No JITA	No Method	C++/ATL	2846.9
Tx	In Process	No JITA	No Method	Visual Basic 6	1186.2
Tx	In Process	No JITA	No Method	C#	1581.7
Tx	In Process	No JITA	No Method	Visual Basic .NET	1581.6
Tx	In Process	No JITA	Trivial Method	C++/ATL	2588.1
Tx	In Process	No JITA	Trivial Method	Visual Basic 6	1138.8
Tx	In Process	No JITA	Trivial Method	C#	1423.8
Tx	In Process	No JITA	Trivial Method	Visual Basic .NET	1423.5
Tx	In Process	No JITA	Typical Method	C++/ATL	243.3
Tx	In Process	No JITA	Typical Method	Visual Basic 6	243.3
Tx	In Process	No JITA	Typical Method	Visual Basic .NET	245.4
Tx	In Process	No JITA	Typical Method	C#	245.4
Tx	In Process	JITA	Trivial Method	C++/ATL	49084.6
Tx	In Process	JITA	Trivial Method	Visual Basic 6	29655.3
Tx	In Process	JITA	Trivial Method	C#	12218.5
Tx	In Process	JITA	Trivial Method	Visual Basic .NET	12166.3
Tx	In Process	JITA	Typical Method	C++/ATL	245.4
Tx	In Process	JITA	Typical Method	Visual Basic 6	247.6
Tx	In Process	JITA	Typical Method	C#	247.6
Tx	In Process	JITA	Typical Method	Visual Basic .NET	245.4
Tx	Out Of Process	No JITA	No Method	C++/ATL	351.5
Tx	Out Of Process	No JITA	No Method	Visual Basic 6	237.2
Tx	Out Of Process	No JITA	No Method	C#	176.8
Tx	Out Of Process	No JITA	No Method	Visual Basic .NET	176.8
Tx	Out Of Process	No JITA	Trivial Method	C++/ATL	261.2
Tx	Out Of Process	No JITA	Trivial Method	Visual Basic 6	192.4
Tx	Out Of Process	No JITA	Trivial Method	C#	124.3
Tx	Out Of Process	No JITA	Trivial Method	Visual Basic .NET	123.8
Tx	Out Of Process	No JITA	Typical Method	C++/ATL	123.2
Tx	Out Of Process	No JITA	Typical Method	Visual Basic 6	121.7
Tx	Out Of Process	No JITA	Typical Method	Visual Basic .NET	80.0
Tx	Out Of Process	No JITA	Typical Method	C#	79.3
Tx	Out Of Process	JITA	Trivial Method	C++/ATL	16456.1
Tx	Out Of Process	JITA	Trivial Method	Visual Basic 6	8600.9
Tx	Out Of Process	JITA	Trivial Method	C#	7299.8
Tx	Out Of Process	JITA	Trivial Method	Visual Basic .NET	7281.1
Tx	Out Of Process	JITA	Typical Method	C++/ATL	243.3
Tx	Out Of Process	JITA	Typical Method	Visual Basic 6	245.4
Tx	Out Of Process	JITA	Typical Method	C#	243.3
Tx	Out Of Process	JITA	Typical Method	Visual Basic .NET	245.4
Tx	Cross Machine	No JITA	No Method	C++/ATL	142.8
Tx	Cross Machine	No JITA	No Method	Visual Basic 6	93.5
Tx	Cross Machine	No JITA	No Method	C#	100.7
Tx	Cross Machine	No JITA	No Method	Visual Basic .NET	100.7
Tx	Cross Machine	No JITA	Trivial Method	C++/ATL	117.1
Tx	Cross Machine	No JITA	Trivial Method	Visual Basic 6	80.2
Tx	Cross Machine	No JITA	Trivial Method	C#	72.8
Tx	Cross Machine	No JITA	Trivial Method	Visual Basic .NET	73.0
Tx	Cross Machine	No JITA	Typical Method	C++/ATL	81.7
Tx	Cross Machine	No JITA	Typical Method	Visual Basic 6	61.7
Tx	Cross Machine	No JITA	Typical Method	C#	48.0

Tx	Cross Machine	No JITA	Typical Method	Visual Basic .NET	48.9
Tx	Cross Machine	JITA	Trivial Method	C++/ATL	1902.7
Tx	Cross Machine	JITA	Trivial Method	Visual Basic 6	1672.5
Tx	Cross Machine	JITA	Trivial Method	C#	1573.3
Tx	Cross Machine	JITA	Trivial Method	Visual Basic .NET	1552.6
Tx	Cross Machine	JITA	Typical Method	C++/ATL	242.9
Tx	Cross Machine	JITA	Typical Method	Visual Basic 6	245.2
Tx	Cross Machine	JITA	Typical Method	C#	238.5
Tx	Cross Machine	JITA	Typical Method	Visual Basic .NET	238.5
No Tx	In Process	No JITA	No Method	C++/ATL	2695.3
No Tx	In Process	No JITA	No Method	Visual Basic 6	1207.0
No Tx	In Process	No JITA	No Method	C#	1664.7
No Tx	In Process	No JITA	No Method	Visual Basic .NET	1654.1
No Tx	In Process	No JITA	Trivial Method	C++/ATL	2586.3
No Tx	In Process	No JITA	Trivial Method	Visual Basic 6	1154.6
No Tx	In Process	No JITA	Trivial Method	C#	1517.1
No Tx	In Process	No JITA	Trivial Method	Visual Basic .NET	1507.0
No Tx	In Process	No JITA	Typical Method	C++/ATL	245.1
No Tx	In Process	No JITA	Typical Method	Visual Basic 6	245.4
No Tx	In Process	No JITA	Typical Method	C#	247.0
No Tx	In Process	No JITA	Typical Method	Visual Basic .NET	246.2
No Tx	In Process	JITA	Trivial Method	C++/ATL	50660.9
No Tx	In Process	JITA	Trivial Method	Visual Basic 6	31121.7
No Tx	In Process	JITA	Trivial Method	C#	12928.1
No Tx	In Process	JITA	Trivial Method	Visual Basic .NET	12808.6
No Tx	In Process	JITA	Typical Method	C++/ATL	247.4
No Tx	In Process	JITA	Typical Method	Visual Basic 6	246.2
No Tx	In Process	JITA	Typical Method	C#	247.4
No Tx	In Process	JITA	Typical Method	Visual Basic .NET	247.5
No Tx	Out Of Process	No JITA	No Method	C++/ATL	327.0
No Tx	Out Of Process	No JITA	No Method	Visual Basic 6	239.7
No Tx	Out Of Process	No JITA	No Method	C#	177.4
No Tx	Out Of Process	No JITA	No Method	Visual Basic .NET	177.9
No Tx	Out Of Process	No JITA	Trivial Method	C++/ATL	259.3
No Tx	Out Of Process	No JITA	Trivial Method	Visual Basic 6	192.2
No Tx	Out Of Process	No JITA	Trivial Method	C#	125.1
No Tx	Out Of Process	No JITA	Trivial Method	Visual Basic .NET	124.7
No Tx	Out Of Process	No JITA	Typical Method	C++/ATL	123.3
No Tx	Out Of Process	No JITA	Typical Method	Visual Basic 6	123.6
No Tx	Out Of Process	No JITA	Typical Method	C#	82.6
No Tx	Out Of Process	No JITA	Typical Method	Visual Basic .NET	82.6
No Tx	Out Of Process	JITA	Trivial Method	C++/ATL	16637.9
No Tx	Out Of Process	JITA	Trivial Method	Visual Basic 6	8847.9
No Tx	Out Of Process	JITA	Trivial Method	C#	7345.1
No Tx	Out Of Process	JITA	Trivial Method	Visual Basic .NET	7392.8
No Tx	Out Of Process	JITA	Typical Method	C++/ATL	244.8
No Tx	Out Of Process	JITA	Typical Method	Visual Basic 6	244.9
No Tx	Out Of Process	JITA	Typical Method	C#	243.9
No Tx	Out Of Process	JITA	Typical Method	Visual Basic .NET	245.5
No Tx	Out Of Process	No JITA	No Method	C++/ATL	147.5
No Tx	Cross Machine	No JITA	No Method	Visual Basic 6	95.1
No Tx	Cross Machine	No JITA	No Method	C#	102.3
No Tx	Cross Machine	No JITA	No Method	Visual Basic .NET	101.7
No Tx	Cross Machine	No JITA	Trivial Method	C++/ATL	119.6
No Tx	Cross Machine	No JITA	Trivial Method	Visual Basic 6	81.6
No Tx	Cross Machine	No JITA	Trivial Method	C#	74.1
No Tx	Cross Machine	No JITA	Trivial Method	Visual Basic .NET	73.5
No Tx	Cross Machine	No JITA	Typical Method	C++/ATL	82.2

No Tx	Cross Machine	No JITA	Typical Method	Visual Basic 6	61.9
No Tx	Cross Machine	No JITA	Typical Method	C#	60.3
No Tx	Cross Machine	No JITA	Typical Method	Visual Basic .NET	61.2
No Tx	Cross Machine	JITA	Trivial Method	C++/ATL	1896.7
No Tx	Cross Machine	JITA	Trivial Method	Visual Basic 6	1620.5
No Tx	Cross Machine	JITA	Trivial Method	C#	1576.8
No Tx	Cross Machine	JITA	Trivial Method	Visual Basic .NET	1591.5
No Tx	Cross Machine	JITA	Typical Method	C++/ATL	242.7
No Tx	Cross Machine	JITA	Typical Method	Visual Basic 6	243.7
No Tx	Cross Machine	JITA	Typical Method	C#	245.8
No Tx	Cross Machine	JITA	Typical Method	Visual Basic .NET	246.0