

Introducing System.Transactions

Juval Lowy



This is a preliminary document and may be changed substantially prior to final commercial release of the software described herein.

The information contained in this document represents the current view of Microsoft Corporation on the issues discussed as of the date of publication. Because Microsoft must respond to changing market conditions, it should not be interpreted to be a commitment on the part of Microsoft, and Microsoft cannot guarantee the accuracy of any information presented after the date of publication.

This white paper is for informational purposes only. MICROSOFT MAKES NO WARRANTIES, EXPRESS OR IMPLIED, IN THIS DOCUMENT.

Complying with all applicable copyright laws is the responsibility of the user. Without limiting the rights under copyright, no part of this document may be reproduced, stored in, or introduced into a retrieval system, or transmitted in any form or by any means (electronic, mechanical, photocopying, recording, or otherwise), or for any purpose, without the express written permission of Microsoft Corporation.

Microsoft may have patents, patent applications, trademarks, copyrights, or other intellectual property rights covering subject matter in this document. Except as expressly provided in any written license agreement from Microsoft, the furnishing of this document does not give you any license to these patents, trademarks, copyrights, or other intellectual property.

© 2005 Microsoft Corporation. All rights reserved.

Microsoft and Windows are either registered trademarks or trademarks of Microsoft Corporation in the United States and/or other countries.

The names of actual companies and products mentioned herein may be the trademarks of their respective owners.

Contents

Introduction	1
.NET 1.x Transaction Programming Models	2
.NET Framework Version 2.0 Transaction Managers	8
Transaction Promotion	10
Triggering Promotion	13
Working with System.Transactions	14
Declarative Programming Model	14
Explicit Programming Model	14
Transaction Flow Management	18
More on TransactionScopeOption	24
Voting inside a nested scope	25
Setting the TransactionScope Timeout	25
Setting the TransactionScope isolation level	26
TransactionScope Benefits	28
Advanced Topics	29
Transaction Events	34
Code-Access Security	37
Concurrency Management and Cloning	38
Interoperability	41
Conclusion	46
About the Author	47

Introduction

Developers on the Microsoft® Windows® platform traditionally choose between two transactional programming models: explicit transaction management or declarative transaction flow and management. Both these programming models have their disadvantages and neither one is superior to the other in every respect. Version 2.0 of the .NET Framework introduces a new transactional programming model available in the `System.Transactions` namespace. The new model allows developers to easily write transactional code with the lowest overhead possible while minimizing the amount of hand-crafted code and separating it from the application hosting environment and instance management. This whitepaper starts by stating the problem with the traditional programming models and the motivation for the new model. The whitepaper then presents the new programming model, its features and its capabilities, and some advanced features such as asynchronous work, events, security, concurrency management and interoperability.

.NET 1.x Transaction Programming Models

ADO.NET 1.0 offers an explicit transaction management programming model. The developer is responsible for explicitly starting and managing the transaction, as shown in Example 1.

```
string connectionString = "...";
IDbConnection connection = new
    SqlConnection(connectionString);
connection.Open();
IDbCommand command = new SqlCommand();
command.Connection = connection;

IDbTransaction transaction;
transaction = connection.BeginTransaction(); //Enlisting
    database
command.Transaction = transaction;
try
{
    /* Interact with database here, then commit the
    transaction */
    transaction.Commit();
}
catch
{
    transaction.Rollback(); //Abort transaction
}
finally
{
    connection.Close();
}
```

Example 1: Explicit transaction management in ADO.NET

You obtain an object representing the underlying database transaction by calling `BeginTransaction()` on the connection object.

`BeginTransaction()` returns an implementation of the interface `IDbTransaction` used to manage the transaction. If all updates or other changes made to the database are consistent, simply call `Commit()` on the

transaction object. If any error occurred, you need to abort the transaction by calling `Rollback()`.

While the explicit programming model is straightforward, requiring nothing of the class performing the transaction, it is most-suitable for a single object interacting with a single database (or a single transactional resource), as shown in Figure 1.

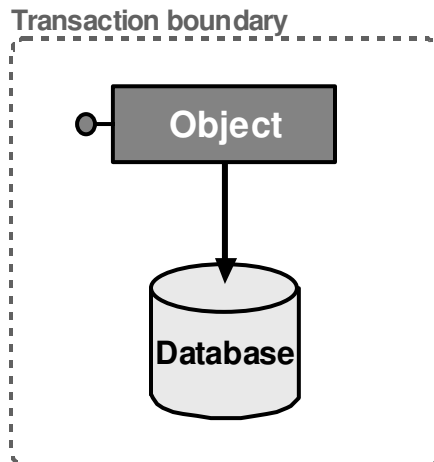


Figure 1: Single object / single resource transaction.

This is due to transaction coordination. Consider for example an application where multiple objects interact with each other and with a resource such as a database, as shown in Figure 2. The question now is which one of the participating objects is responsible for beginning the transaction and enlisting the resource? If all of them will do that you will end up with multiple transactions. Furthermore, which one of the objects is responsible for committing or rolling back the transactions? How would one object know what the rest of the objects feel about the transaction? How would the object managing the transaction inform other objects about the transaction's outcome? The objects could also be deployed in different processes or even across different machines, and issues such as network or machine crashes introduce yet additional complexity for managing the transaction. One possible solution is to couple the objects by adding logic for the transaction coordination, but such an approach is very fragile and would not withstand even minor changes to the business flow or the number of participating objects. In addition, the objects in Figure 2 could have been developed by different vendors, which would preclude any such co-ordination.

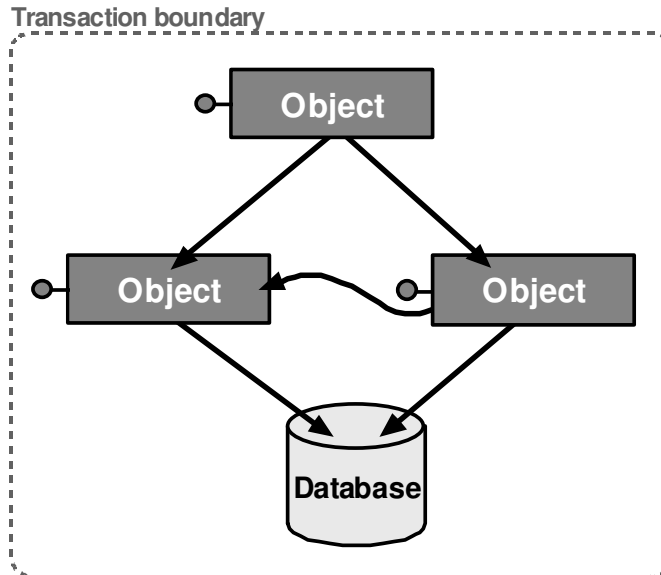


Figure 2: Multiple objects / single resource transaction.

The situation gets significantly more complex when multiple resources are involved (as shown in Figure 3).

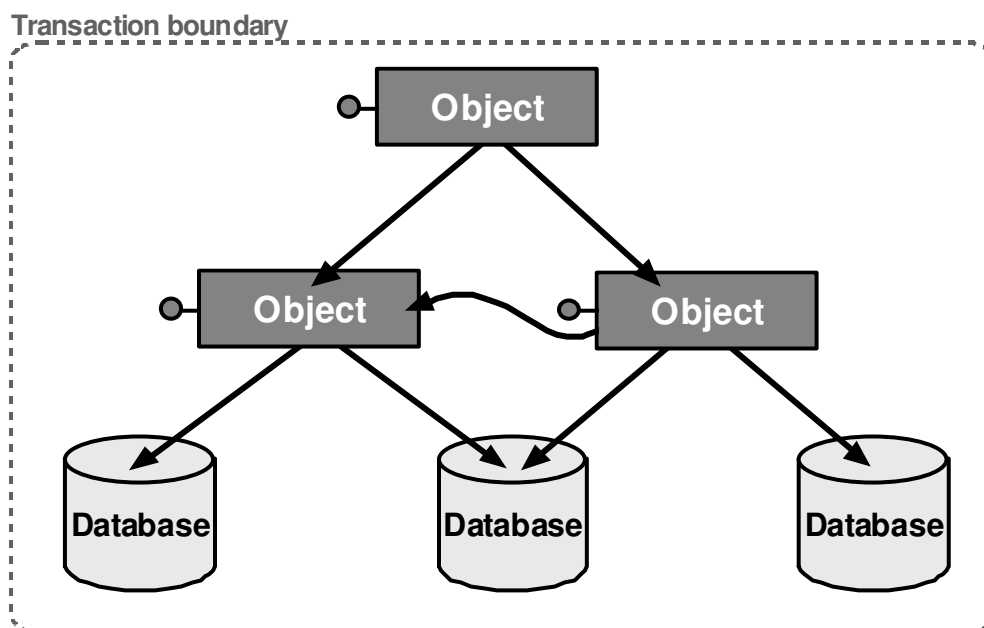


Figure 3: Multiple objects accessing multiple resources.

On top of the challenges involving multiple objects in a single transaction, the introduction of multiple resources actually introduces just as many additional points of failure. Each of the resources could fail committing its part of the transaction. The predicament just described is called a *distributed transaction*. A distributed transaction contains two or more independent parties or objects,

(often in different execution contexts), or two or more transactional resources, or both multiple objects and multiple resources. It is impractical to try to explicitly manage all the potential error cases of a distributed transaction. For a distributed transaction, you need to rely on the two-phase commit protocol and a dedicated transaction manager.

In Windows, the Distributed Transaction Coordinator (DTC) system service manages such transactions across components, processes and machines, using a protocol called *OLE Transactions (OleTx)*. While it is possible to program directly against the DTC, in .NET the most common and easy way to utilize DTC transactions is to use the Enterprise Services available in the `System.EnterpriseServices` namespace. Example 2 shows the use of Enterprise Services transaction.

```

using System.EnterpriseServices;

[Transaction]
public class MyComponent : ServicedComponent
{
    [AutoComplete]
    public void MyMethod()
    {
        /*Interact with other serviced components
        and resource managers */
    }
}

```

Example 2: Declarative transaction management via Enterprise Services.

.NET Enterprise Services offer a declarative programming model: any class that derives from the abstract class `ServicedComponent` can use the `Transaction` attribute. The attribute ensures that when any method of the class is called, that method executes inside a transactional context. A `context` is the inner-most execution scope of the serviced component. .NET intercepts calls coming into the context, and starts a transaction on behalf of the object. No explicit enlistment of resources is required. Obviously, this only works if the resource can automatically enlist in the transaction. Such resources are called *transactional resource managers*, and many of the popular commercial databases and durable resources (such as Microsoft Message Queue) are resource managers. All the object has to do is inform .NET whether it should commit or abort the transaction. While it is possible to do so using the methods of the `ContextUtil` helper class, you can also accomplish that declaratively via the `AutoComplete` method attribute. If no exception was thrown in the method, the `AutoComplete` attribute will commit the transaction and if an exception occurred, the transaction will be aborted.

While the declarative model offers significant productivity benefits, it is not without flaws:

- Forcing inheritance from `ServicedComponent` occupies the precious place of a base class normally reserved for internal application modeling.
- Use of an Enterprise Services transaction always implies the use of a distributed DTC transaction, even when a single resource and a single object are involved. The two-phase commit protocol has its cost, both at the transaction manager level and at the resource level since the

resource has to keep logging its operations. The overhead could cause degradation in performance compared with explicit transaction management.

- Implied with the use of Enterprise Services is the COM+ hosting model, which many developers find intimidating.
- Enterprise Services transactions are tightly-coupled with Enterprise Services instance management strategies. All transactional objects are also just-in-time activated, and there are some issues when it comes to combining transactions with object pooling. While this coupling is well-appreciated in a scalable application, for all other applications it forces a state-aware programming model that most developers have difficulty with.
- Enterprise Services transactions are always thread-safe – there is no way for multiple threads to participate in the same transaction. While this greatly simplifies transaction management especially in a multithreaded environment, in some edge cases it is a limitation.

In effect, .NET 1.0 and 1.1 equates the use of a non-distributed transaction with explicit transaction management, and equates the use of a distributed transaction with that of declarative transaction via Enterprise Services. There is no way of using a declarative transaction without using a DTC transaction, nor is there an easy way in managed code for explicit transaction management that utilizes the DTC. Choosing a programming model (explicit or declarative) invariably chooses a transaction manager as well, and vice-versa.

.NET Framework Version 2.0 Transaction Managers

To address the problems just described with both the explicit and the declarative transactional programming models, the .NET Framework Version 2.0 introduces two new types of transaction manager and a management namespace. The two new transaction managers are the *Lightweight Transaction Manager* (LTM), and the *OleTx Transaction Manager*.

The LTM is used to manage a transaction inside a single app domain that involves at most a single durable resource. The *OleTx Transaction Manager* manages transactions across the app domain boundaries (including across process and machine boundaries), or any transaction that involves more than one durable resource, even in the same app domain. The OleTx transaction manager relies on RPC for cross-machine calls, and it is equivalent to the DTC behavior with minor changes (discussed later in the whitepaper). The LTM uses only intra-app domain calls and so is of higher performance than the OleTx transaction manager. Developers need not interact with these transaction managers directly. Instead, common infrastructure that defines interfaces, transaction factories, common behaviors and helper classes is available in the `System.Transactions` namespace.

The resources managed by the two transaction managers are called *System.Transactions Resource Managers*. Similar to Enterprise Services, a `System.Transactions Resource Manager` is a resource that can automatically enlist in a transaction managed by the LTM or the OleTx transaction managers.

The main advantage of programming against a common transaction management namespace (`System.Transactions`) rather than a particular transaction manager is promotion. *Transaction manager promotion* is an innovative technique supported by `System.Transactions`. The idea behind promotion is simple: Developers should only decide on the desired programming model (either explicit or declarative transaction management), and `System.Transactions` will correctly assign the appropriate transaction manager.

For example, suppose a single object interacts with a single durable resource. This should only require the LTM and will obviously yield the best throughput and performance. If you were to provide the transaction to another object in another app domain on the same machine, or if you were to enlist another durable resource manager, the transaction will automatically be promoted for management by OleTx. Once promoted the transaction remains managed in its elevated state till its completion.

Both the LTM and the OleTx transaction manager represent their transaction using a class called `Transaction`, defined in the `System.Transactions` namespace. `Transaction` is used to enlist resources in the transaction, to abort the transaction, to set the isolation level, to obtain the transaction status and ID, and to clone the transaction. To commit the transaction, `System.Transactions` defines the `CommittableTransaction` class:

```

public interface ITransaction : IDisposable
{
    void Rollback();
}
[Serializable]
public class Transaction : ITransaction, ISerializable
{
    public void Rollback(); //Abort the transaction
    public static Transaction Current{get;set;}
    //Other members
}
[Serializable]
public sealed class CommittableTransaction :
    Transaction, IAsyncResult
{
    public void Commit();
    //Other members
}

```

The reason for two classes instead of one is discussed later on.

When using `System.Transactions` you cannot interact directly with the underlying resource manager, because that would bypass the promotion mechanism. Instead, always rely implicitly or explicitly on `Transaction` or its sub classes. `System.Transactions` defines a concept called an ambient transaction. The *ambient transaction* is the transaction your code executes in. To obtain a reference to the ambient transaction call the static `Current` property of `Transaction`:

```
Transaction ambientTransaction = Transaction.Current();
```

If there is no ambient transaction, `Current` will return `null`.

The ambient transaction object is stored in the thread local storage (TLS). As a result, when the thread winds its way across multiple objects and methods, all objects and methods can access their ambient transaction.

Transaction Promotion

Unless specified otherwise, every `System.Transactions` transaction in .NET 2.0 starts as a transaction managed by the LTM. As long as at most a single durable resource manager is involved, there is nothing wrong with

letting the underlying resource (such as Microsoft SQL Server™ 2005) manage the transaction. In such case, the LTM does not need to actually manage the transaction at all – its role should be reduced to monitoring the transaction for a need for promotion. This is exactly what the LTM does – it functions as an adapter for the underlying resource converting calls on its managed transaction to calls on the underlying resource. On the other hand, if promotion is required, the LTM must inform the resource manager to relinquish control over the LTM transaction, and provide the LTM instead with a transaction object that is capable of participating in an OleTx transaction. To support this interaction, the resource manager needs to implement the interface `IPromotableSinglePhaseNotification` defined as:

```

public interface IPromotableSinglePhaseNotification
{
    void Initialize();
    Transaction Promote();
    void Rollback(SinglePhaseEnlistment
singlePhaseEnlistment);
    void SinglePhaseCommit(SinglePhaseEnlistment ←
singlePhaseEnlistment);
}

```

When an LTM transaction accesses a resource manager, the resource manager needs to detect that. The resource manager then obtains a reference to the current LTM transaction object and enlists itself. This is called *auto-enlistment*. Once enlisted, the LTM queries the resource manager for its implementation of `IPromotableSinglePhaseNotification`. Calls to commit or abort the LTM's transaction are converted to calls on `IPromotableSinglePhaseNotification` (`SinglePhaseCommit()` and `Rollback()` respectively).

The LTM overhead is extremely lightweight. Performance benchmarking done by Microsoft with SQL Server 2005, comparing the use of an LTM transaction to using a native transaction directly found no statistical differences between using the two methods.

`System.Transactions` makes it easy for resource manager vendors to support the interaction just described. To obtain a reference to the transaction the resource manager calls the static `Current` property of `Transaction`. To enlist in a transaction, the resource manager calls the `Transaction` object's `EnlistDurable()` method for a durable resource or the `EnlistVolatile()` method for a *volatile resource* (a resource that stores state only in memory):

```

[Serializable]
public class Transaction : ITransaction, ISerializable
{
    public Enlistment EnlistDurable(...);
    public Enlistment EnlistVolatile(...);
    //Other members
}

```

When the LTM decides to promote the transaction to an `OleTx` transaction, it simply calls the `Promote()` method of

`IPromotableSinglePhaseNotification`. Internally the resource manager will convert the transaction from a local transaction (which does not require logging) to a distributed transaction. This will have the net effect as if the resource was enlisted from the very beginning in a DTC or OleTx transaction.

Since implementing `IPromotableSinglePhaseNotification` is key to participating in a promotable transaction, existing Enterprise Services resource managers such as SQL Server™ 2000, MSMQ, Oracle Database or IBM DB2 cannot participate in an LTM transaction. When such resources are accessed by an LTM transaction, the transaction is automatically promoted to an OleTx transaction, even if only one such resource is involved.

Triggering Promotion

There are two kinds of events that trigger promotion: the first is the enlistment into the transaction of a second durable resource manager. For example, whenever you open a second connection to a SQL Server 2005 database, this will cause the database to enlist. The LTM will detect it is the second durable resource in the transaction and will promote the transaction to an OleTx transaction.

The second promotion trigger is the serialization of the transaction object across an app domain boundary. `Transaction` is marshaled-by-value object, meaning, any attempt to pass it across an app domain boundary (even in the same process) will result in serialization of the transaction object.

You can pass the transaction objects yourself (by making a call on a remote method that takes a `Transaction` as a parameter), or you can try to access a remote transactional serviced component. This too will serialize the transaction object. Serializing the transaction amounts to promoting it because when serialized across an app domain, you are in fact distributing the transaction and the LTM (which functions merely as a pass-through) is no longer adequate. The LTM transaction class uses custom serialization. In its handling of the serialization request it will promote the transaction, and in its handling of the deserialization request it will enlist in the new app domain in a new OleTx transaction.

Working with System.Transactions

`System.Transactions` finally decouples the application programming model from transaction managers. You can develop applications that explicitly program against the transaction by accessing objects from `System.Transactions`. You can rely on the declarative use of transaction available in Enterprise Services.

Declarative Programming Model

Declarative transactional programming in .NET 2.0 relies on .NET Enterprise Services. Microsoft rewrote ADO.NET to take advantage of `System.Transactions`. There is absolutely nothing you have to do differently when using Enterprise Services compared to .NET 1.1. The same Enterprise Services code presented in Example 2 will automatically use the LTM if possible, and will be promoted to OleTx when required. This maintains the paramount productivity advantages of Enterprise Services while eliminating the performance penalty when dealing with single resources or local objects only. In addition, this change will not affect existing applications because it is decoupled from the application logic itself.

Explicit Programming Model

If you cannot use .NET Enterprise Services, you can explicitly program against `System.Transactions`. The most common and easy way of doing so is utilizing the class `TransactionScope`:

```
public class TransactionScope : IDisposable
{
    public void Complete();
    public void Dispose();
    public TransactionScope();
    public TransactionScope(Transaction
        transactionToUse);
    public TransactionScope(TransactionScopeOption
        scopeOption);
    public TransactionScope(TransactionScopeOption
        scopeOption, ↵ TransactionOptions
        transactionOptions);
    public TransactionScope(TransactionScopeOption
        scopeOption, ↵ TimeSpan scopeTimeout);
```

```
//Additional constructors  
}
```

As the name implies, the `TransactionScope` class is used to scope a code section with a transaction, as demonstrated in Example 3. Internally in its constructor, the `TransactionScope` object creates a transaction (LTM by default in .NET 2.0), and assigns it as the ambient transaction by setting the `Current` property of the `Transaction` class. `TransactionScope` is a disposable object—the transaction will end once the `Dispose()` method is called (the end of the `using` statement in Example 3).

```
using(TransactionScope scope = new TransactionScope())
{
    /* Perform transactional work here */
    //No errors - commit transaction
    scope.Complete();
}
```

Example 3: Using the TransactionScope class.

The `Dispose()` method also restores the ambient transaction to its original state, `null` in the case of Example 3.

If the code inside the transactional scope (typically inside the `using` statement) takes a long time to complete, it may be indicative of a transactional deadlock. To address that, the transaction will automatically abort if executed for more than a pre-configured timeout (60 seconds by default). The timeout is configurable both programmatically and administratively via a configuration file.

Finally, if the `TransactionScope` object is not used inside a `using` statement, it would become garbage once the transaction timeout is expired and the transaction is aborted.

The `TransactionScope` object has no way of knowing whether the transaction should commit or abort, yet the main objective of `TransactionScope` is to shield the developers from the need to interact with the transaction directly. To address this, every `TransactionScope` object has a consistency bit, which is by default set to `false`. You can set the consistency bit to `true` by calling the `Complete()` method. Note that you can only call `Complete()` once. Subsequent calls to `Complete()` will raise an `InvalidOperationException` exception. This is deliberate to encourage developers to have no transactional code after the call to `Complete()`.

If the transaction ends (due to calling `Dispose()` or garbage collection) and the consistency bit is set to `false`, the transaction will abort. For example, the following `scope` object will always rollback its transaction, because the consistency bit is never changed from its default value:

```
using(TransactionScope scope = new TransactionScope())
{
    // ...
}
```

On the other hand, if you do call `Complete()` and the transaction ends with the consistency bit set to `true` as in Example 3, the transaction will commit.

Note that after calling `Complete()` you cannot access the ambient transaction, and trying to do so will result with an invalid operation exception. You can access the ambient transaction (via `Transaction.Current`) again once the scope object (on whom you have called `Complete()`) is disposed of. The fact that the code in the scope called `Complete()` does not guarantee committing the transaction. As a result, `Dispose()` will throw `TransactionAbortedException` if it fails to commit the transaction. You can catch handle that exception, perhaps by alerting the user, as shown in Example 4.

```

try
{
    using(TransactionScope scope = new
    TransactionScope())
    {
        /* Perform transactional work here */
        //No errors - commit transaction
        scope.Complete();
    }
}
catch(TransactionAbortedException e)
{
    MessageBox.Show(e.Message);
}
catch //Any other exception took place
{
    Trace.WriteLine("Cannot complete transaction");
    throw;
}

```

Example 4: TransactionScope and Error handling.

Transaction Flow Management

Transaction scopes can nest both directly and indirectly. A *direct scope nesting* is simply one scope nested inside another, as shown in Example 5.

```

using(TransactionScope scope1 = new TransactionScope())
{
    using(TransactionScope scope2 = new
    TransactionScope())
    {
        scope2.Complete();
    }
    scope1.Complete();
}

```

Example 5: Direct scope nesting.

An *indirect scope nesting* occurs when calling a method that uses a `TransactionScope` from within a method that uses its own scope, as is the case with the `RootMethod()` in Example 6.

```

void RootMethod()
{
    using(TransactionScope scope = new
        TransactionScope())
    {
        /* Perform transactional work here */
        SomeMethod();
        scope.Complete();
    }
}

void SomeMethod()
{
    using(TransactionScope scope = new
        TransactionScope())
    {
        /* Perform transactional work here */
        scope.Complete();
    }
}

```

Example 6: Indirect scope nesting.

You can have multiple scope nesting, involving both direct and indirect nesting. The top-most scope is referred to as the *root scope*. The question is of course what is the relation between the root scope and all the nested scopes? How will nesting a scope affect the ambient transaction? Will all the scopes participate in the same transaction? Will voting on transaction in a nested scope affect its containing scope?

To address the previous questions, the `TransactionScope` class provides several overloaded constructors that accept an enumeration of the type `TransactionScopeOption`, defined as:

```

public enum TransactionScopeOption
{
    Required,
    RequiresNew,
    Suppress
}

```

The value of `TransactionScopeOption` lets you control whether the scope takes part in a transaction, and if so, will it join the ambient transaction or will it be the root scope of a new transaction.

For example, here is how you specify the value of the `TransactionScopeOption` in the scope's constructor:

```
using (TransactionScope scope = new  
    TransactionScope(TransactionScopeOption.Required) )  
{...}
```

The default value for the scope option is `TransactionScopeOption.Required`, meaning, this is the value used when you use one of the constructors which do not accept a `TransactionScopeOption` parameter.

The `TransactionScope` object determines which transaction to belong to when it is constructed. Once determined, the scope will always belong to that transaction. `TransactionScope` bases its decision on two factors: whether an ambient transaction is present and the value of the `TransactionScopeOption` parameter.

A `TransactionScope` object has three options:

- Join the ambient transaction.
- Be a new scope root, that is, start a new transaction and have that transaction be the new ambient transaction inside its own scope.
- Do not take part in a transaction at all.

If the scope is configured with `TransactionScopeOption.Required`, and an ambient transaction is present, the scope will join that transaction. If on the other hand there is no ambient transaction, then the scope will create a new transaction, and become the root scope.

If the scope is configured with `TransactionScopeOption.RequiresNew` then it will always be the root scope. It will start a new transaction, and its transaction will be the new ambient transaction inside the scope.

If the scope is configured with `TransactionScopeOption.Suppress` it will never be part of a transaction, regardless of whether an ambient transaction is present. A scope configured with `TransactionScopeOption.Suppress` will always have `null` as its ambient transaction.

The `TransactionScopeOption` allocation decision truth table is summarized in Table 1.

TransactionScopeOption	Ambient Transaction	The scope will take part in
Required	No	New Transaction (will be the root)
Requires New	No	New Transaction (will be the root)
Suppress	No	No Transaction
Required	Yes	Ambient Transaction
Requires New	Yes	New Transaction (will be the root)
Suppress	Yes	No Transaction

Table 1: TransactionScopeOption decision truth table.

When a `TransactionScope` object joins an existing ambient transaction, disposing of the scope object does not end the transaction. If the ambient transaction was created by a root scope, only when the root scope is disposed of will the transaction end. If the ambient transaction was created manually, it will end when it is committed or aborted by its creator.

Example 7 demonstrates a `TransactionScope` object that creates three other scope objects, each configured with a different `TransactionScopeOption` values, and Figure 4 shows graphically the resulting transactions.

```

using(TransactionScope scope1 = new TransactionScope())
//Default is Required
{
    using(TransactionScope scope2 = new ↵
        TransactionScope(TransactionScopeOption.Required))
    {...}

    using(TransactionScope scope3 = new ↵
        TransactionScope(TransactionScopeOption.RequiresNew))
    {...}

    using(TransactionScope scope4 = new ↵
        TransactionScope(TransactionScopeOption.Suppress))
    {...}

    ...
}

```

Example 7: Transaction flow across scopes.

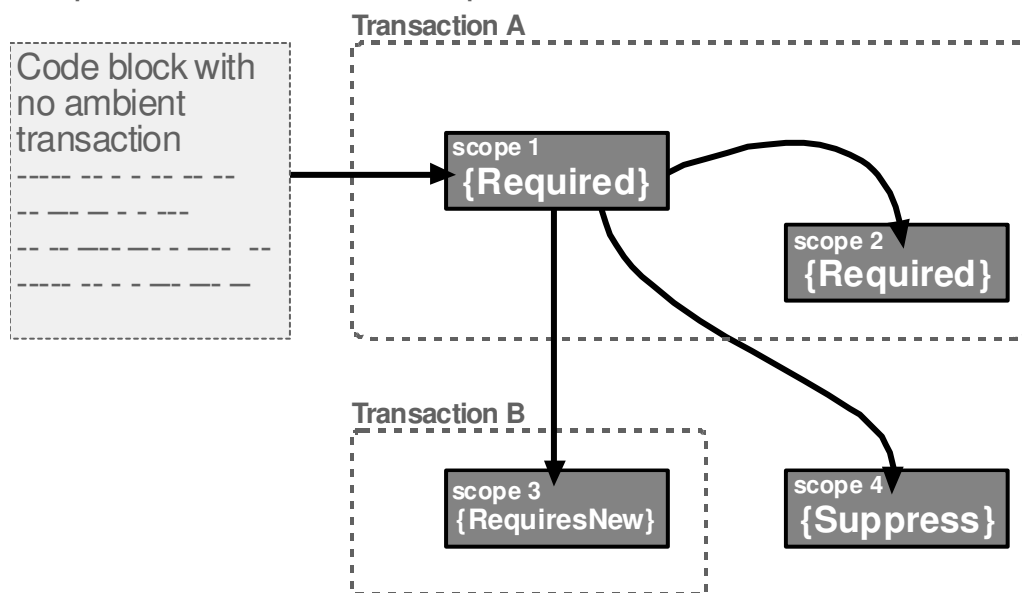


Figure 4: Transaction flow across transaction scopes.

Example 7 lists a code block without any ambient transaction creating a new `TransactionScope (scope1)` with `TransactionScopeOption.Required`. `scope1` will be a root scope: it

creates a new transaction (Transaction A) and will make Transaction A the ambient transaction. `scope1` then goes on to create three more objects, each with a different `TransactionScopeOption` value. For example, `scope2` is configured for required support, and since there is an ambient transaction, it will join Transaction A. Note that `scope3` is the root scope of a new transaction, Transaction B, and that `scope4` has no transaction.

More on TransactionScopeOption

Although the default and most commonly used value of `TransactionScopeOption` is `TransactionScopeOption.Required`, each of the other values has its use.

`TransactionScopeOption.Suppress` is useful when the operations performed by the code section are nice to have and should not abort the ambient transaction if the operations fail.

`TransactionScopeOption.Suppress` allows you to have a non-transactional code section inside a transactional scope, as shown in Example 8.

```
using(TransactionScope scope1 = new TransactionScope())
{
    try
    {
        //Start of non-transactional section
        using(TransactionScope scope2 = new ←
TransactionScope(TransactionScopeOption.Suppress))
        {
            //Do non-transactional work here
        }
        //Restores ambient transaction here
    }
    catch
    {}
    //Rest of scope1
}
```

Example 8: Using TransactionScopeOption.Suppress

Another example where `TransactionScopeOption.Suppress` is useful is when you want to provide some custom behavior, and you need to perform your own programmatic transaction support or manually enlist resources.

That said, you should be careful when mixing transactional scopes with non-transactional scopes, as it can jeopardize isolation and consistency. The non-transactional scope may have errors and could not affect any transaction outcome (threatening consistency) or it can act based on information not committed yet (threatening isolation).

When `TransactionScopeOption.Required` is used, the code inside the `TransactionScope` must not behave differently when it is the root or when it is just joining the ambient transaction. It should operate identically in both cases. There is no way your code can tell the difference anyway. Configuring the scope to `TransactionScopeOption.RequiresNew` is useful when you want to perform transactional work outside the scope of the ambient transaction: for example, when you want to perform some logging or audit operations, or when you want to publish events to subscribers, regardless of whether your ambient transaction commits or aborts.

You should be extremely careful when using the `TransactionScopeOption.RequiresNew` value, and verify that the two transactions (the ambient transaction and the one created for your scope) do not jeopardize consistency if one aborts and the other commits.

Voting inside a nested scope

It is important to realize that although a nested scope can join the ambient transaction of its parent scope, the two scope objects will have two distinct consistency bits. Calling

`Complete()` in the nested scope has no affect on the parent scope:

```
using(TransactionScope scope1 = new TransactionScope())
{
    using(TransactionScope scope2 = new
        TransactionScope())
    {
        scope2.Complete();
    }
    //Consistency bit of scope1 is still false
}
```

Only if all the scopes from the root scope down to the last nested scope vote to commit the transaction, will the transaction commit.

Setting the TransactionScope Timeout

Some of the overloaded constructors of `TransactionScope` accept a value of type `TimeSpan`, used to control the timeout of the transaction, for example:

```
public TransactionScope(TransactionScopeOption
    scopeOption, TimeSpan ← scopeTimeout);
```

To specify a timeout different from the default of 60 second, simply pass in the desired value:

```
TimeSpan timeout = TimeSpan.FromSeconds(30);
using(TransactionScope scope = new ←
    TransactionScope(TransactionScopeOption.Required,
        timeout))
{...}
```

A timeout set to zero means an infinite timeout. Infinite timeout is useful mostly for debugging, when you want to try to isolate a problem in your business logic, by stepping through your code, and you do not want the transaction you debug to timeout while you figure out the problem. Be extremely careful with infinite timeout in all other cases, because it means there are no safeguards against transaction deadlocks.

You typically set the `TransactionScope` timeout to values other than default in two cases. The first is during development, when you want to test the way your application handles aborted transactions. By setting the `TransactionScope` timeout to a small value (such as one millisecond), you cause your transaction to fail and can thus observe your error handling code. The second case in which you set the `TransactionScope` transaction timeout to be less than the default timeout is when you have reason to believe that the scope is involved (more than its fair share) in resource contention, resulting in deadlocks. In that case, you want to abort the transaction as soon as possible and not wait for the default timeout to expire. The question is what happens when a `TransactionScope` joins the ambient transactions yet specifies a different timeout than the one the ambient transaction is set to. If the nested scope's timeout is less than that of the ambient transaction, it has the affect of enforcing the new, shorter timeout on the ambient transaction, and the transaction must end within the nested time specified, or it is automatically aborted. If the nested scope's timeout is more than that of the ambient transaction, then it has no effect.

Setting the `TransactionScope` isolation level

Some of the overloaded constructors of `TransactionScope` accept a struct the type `TransactionOptions` defined as:

```
public struct TransactionOptions
{
    public IsolationLevel IsolationLevel{get;set;}
```

```

    public TimeSpan Timeout{get; set;}
    //Other members
}

```

While you can use the `TransactionOptions Timeout` property to specify a timeout, the main use for `TransactionOptions` is for specifying isolation level. By default, the transaction will execute with isolation level set to `Serializable`. However, you could assign into `TransactionOptions IsolationLevel` property a value of the enum type `IsolationLevel` defined as:

```

public enum IsolationLevel
{
    ReadUncommitted,
    ReadCommitted,
    RepeatableRead,
    Serializable,
    Unspecified,
    Chaos, //No isolation whatsoever
    Snapshot //Special form of read committed ↵
    supported by SQL Server 2005
}

```

Example 9 shows how to specify an isolation level.

```

TransactionOptions options = new TransactionOptions();
options.IsolationLevel = IsolationLevel.ReadCommitted;
options.Timeout = TransactionManager.DefaultTimeout;

using(TransactionScope scope = new ↵
    TransactionScope(TransactionScopeOption.Required,
        options))
{...}

```

Example 9: Specifying an isolation level.

Selecting an isolation level other than `Serialized` is commonly used for read-intensive systems, and it requires a solid understanding of transaction processing theory and the semantics of the transaction itself, the concurrency issues involved, and the consequences for system consistency. In addition, not all resource managers support all levels of isolation, and they may elect to take part in the transaction at a higher level than the one configured. Every isolation level besides `Serialized` is susceptible to some sort of inconsistency resulting from other transactions accessing the same information. The

difference between the four isolation levels is in the way the different levels use read and write locks. A lock can be held only when the transaction accesses the data in the resource manager, or it can be held until the transaction is committed or aborted. The former is better for throughput, the latter for consistency. The two kinds of locks and the two kinds of operations (read/write) give four basic isolation levels. See a transaction-processing textbook for a comprehensive description of isolation levels.

When using nested `TransactionScope` objects, all nested scopes must be configured to use exactly the same isolation level if they want to join the ambient transaction. If a nested `TransactionScope` object tries to join the ambient transaction yet it specifies a different isolation level, an `ArgumentException` is thrown.

TransactionScope Benefits

The `TransactionScope` object offers clear advantages and benefits compared with the programming model of Example 1:

- The code inside the transactional scope is not only transactional, it is also promotable. The transaction starts with the LTM and `System.Transactions` will promote it as required, according to the nature of its interaction with the resources or remote objects.
- The scope is independent of the application object model – any piece of code can use the `TransactionScope` and thus become transactional. There is no need for special base class or attributes.
- There is no need to enlist resources explicitly with the transaction. Any `System.Transactions` resource manager will detect the ambient transaction created by the scope and automatically enlist.
- Overall simple and intuitive programming model even for the more complex scenarios that involve transaction flow and nesting.

Advanced Topics

The `TransactionScope` class wraps the underlying `System.Transactions` transaction managers and the committable transaction object. Everything you have seen so far with using the `TransactionScope` class can be achieved directly as well. Example 10 demonstrates this point.

```
Transaction oldAmbient = Transaction.Current;
CommittableTransaction committableTransaction;
committableTransaction = oldAmbient as
    CommittableTransaction;
if(committableTransaction == null)
{
    committableTransaction = new
        CommittableTransaction();
    Transaction.Current = committableTransaction;
}

try
{
    /* Perform transactional work here */
    //No errors - commit transaction
    committableTransaction.Commit();
}
finally
{
    committableTransaction.Dispose();
    //Restore the ambient transaction
    Transaction.Current = oldAmbient;
}
```

Example 10: Requiring a transaction manually.

Example 10 is the equivalent of Example 3 of using a `TransactionScope` set to `TransactionScopeOption.Required`. Example 10 starts by obtaining a reference to the ambient transaction object via the static property `Current` of the `Transaction` class. `Current` is of the type `Transaction`,

and while `Transaction` has many essential methods, it does not have a `Commit()` method.

This is by design, so that you could pass the transaction object (or clones of it) to other parties (potentially on other threads), have them vote on the transaction, yet you reserve the act of committing the transaction to yourself. To commit a transaction, you need to use `CommittableTransaction`, mentioned already. `CommittableTransaction` is listed in Example 11.

```

[Serializable]
public sealed class CommittableTransaction : ↵
    Transaction, IAsyncResult
{
    public CommittableTransaction();
    public CommittableTransaction(TimeSpan timeout);
    public CommittableTransaction(TransactionOptions ↵
        transactionOptions);

    public void Commit();
    public IAsyncResult BeginCommit(AsyncCallback ↵
        AsyncCallback, object AsyncState);
    public void EndCommit(IAsyncResult AsyncResult);
}

```

Example 11: The CommittableTransaction class.

Note that while `CommittableTransaction` does not set the ambient transaction. Therefore, after creating a committable transaction, make it the ambient transaction by assigning it to `Transaction.Current`:

```
Transaction.Current = committableTransaction;
```

You should also save the old ambient transaction, and in the finally statement, restore it when you are done using the `CommittableTransaction` object.

While Example 10 helps to demystify how `TransactionScope` actually works, there is an interesting use for the `CommittableTransaction` - asynchronous commit. It is possible that committing the transaction will take a while, because it involves multiple database access, network latency and so on. While you do care how long a transaction executes (especially when resolving deadlocks), in high throughput applications, you may want to finish the transnational work as soon as possible, and have the actual commit itself carry out in the background. The `BeginCommit()` and `EndCommit()` methods of `CommittableTransaction` serve just such purpose. You use these methods like any other case of asynchronous invocation in .NET, as shown in Example 12. Calling `BeginCommit()` dispatches the commit holdup to a thread from the thread pool.

Note that you must call `EndCommit()` to find out if the transaction actually committed. If the transaction failed to commit (for whatever reason), `EndCommit()` will raise a transaction exception. If the transaction is not yet committed by the time `EndCommit()` is called, it will block its caller until the

transaction is committed (or aborted). The easiest way of calling commit asynchronously is by providing a callback method, to be called when committing is finished. You must call `EndCommit()` on the original committable transaction object used to invoke the call. Fortunately, you can do easily obtain that object by down-casting the `IAsyncResult` parameter of the callback method, since `CommittableTransaction` is derived `IAsyncResult`, and it is the same object. Also worth mentioning is that until the commit is done, the transaction maintains its locks in the resource managers (according to its isolation level).

```

public void DoTransactionalWork()
{
    Transaction oldAmbient = Transaction.Current;
    CommittableTransaction committableTransaction = new ←
    CommittableTransaction();
    Transaction.Current = committableTransaction;

    try
    {
        /* Perform transactional work here */
        //No errors - commit transaction asynchronously.

        committableTransaction.BeginCommit(OnCommitted, null
    );
    }
    finally
    {
        //Restore the ambient transaction
        Transaction.Current = oldAmbient;
    }
}

void OnCommitted(IAsyncResult asyncResult)
{
    CommittableTransaction committableTransaction;
    committableTransaction = asyncResult as
    CommittableTransaction;
    Debug.Assert(committableTransaction != null);
    try
    {
        using(committableTransaction)
        {
            committableTransaction.EndCommit(asyncResult);
        }
    }
    catch(TransactionException e)
    {

```

```
        //Handle the failure to commit
    }
}
```

Example 12: Asynchronously commit a transaction

Transaction Events

The Transaction class provides a public event called `TransactionCompleted`, defined as:

```

public delegate void
    TransactionCompletedEventHandler(object sender,
    TransactionEventArgs e);
public class TransactionEventArgs: EventArgs
{
    public TransactionEventArgs();
    public Transaction Transaction {get;}
}
[Serializable]
public class Transaction : ITransaction, ISerializable
{
    public event TransactionCompletedEventHandler
    TransactionCompleted;
    //Rest of the members
}

```

TransactionCompleted is raised after the transaction is completed (be it committed or aborted). The event is of the delegate type **TransactionCompletedEventHandler**, which takes two parameters: **sender** is the transaction which just completed, and **e** is of the type **TransactionEventArgs**, which also provides access to the same transaction.

You can subscribe to the **TransactionCompleted** event other parties that want to be notified when the transaction is completed, as shown in Example 13.

```

public void DoTransactionalWork()
{
    using(TransactionScope scope = new
    TransactionScope())
    {
        Transaction transaction = Transaction.Current;
        transaction.TransactionCompleted += OnCompleted;
        /* Perform transactional work here */
        scope.Complete();
    }
}
void OnCompleted(object sender, TransactionEventArgs e)

```

```

{
    Debug.Assert (sender.GetType() ==
        typeof(Transaction));
    Debug.Assert (Object.ReferenceEquals (sender,e.Transaction));
    Transaction transaction = e.Transaction;
    switch (transaction.TransactionInformation.Status)
    {
        case TransactionStatus.Aborted:
        {
            Trace.WriteLine("Transaction Aborted!");
            break;
        }
        case TransactionStatus.Committed:
        {
            Trace.WriteLine("Transaction Committed!");
            break;
        }
    }
}
}

```

Example 13: Using the transaction completed event.

While developers know when an LTM transaction starts (when the scope is constructed), your code may also want to know when that LTM transaction is promoted to a distributed OleTx transaction. The static class `TransactionManager` provides the event `TransactionStarted` defined as:

```

public delegate void
    TransactionStartedEventHandler(object sender,
        TransactionEventArgs e);

public static class TransactionManager
{
    public event TransactionStartedEventHandler ←
        DistributedTransactionStarted;
    //Rest of the members
}

```

The `DistributedTransactionStarted` event is raised whenever a distributed transaction starts. You can subscribe to both a distributed transaction's start and completion events, as shown in Example 14.

```
public void DoTransactionalWork()
{
    TransactionManager.DistributedTransactionStarted +=
        OnDistributedStarted;

    using(TransactionScope scope = new
        TransactionScope())
    {
        Transaction transaction = Transaction.Current;
        transaction.TransactionCompleted += OnCompleted;
        /* Perform transactional work here */
        scope.Complete();
    }
}

void OnDistributedStarted(object
    sender, TransactionEventArgs e)
{...}

void OnCompleted(object sender, TransactionEventArgs e)
{...}
```

Example 14: Subscribing to the transaction start events.

Be mindful of the work you do in the distributed transaction's start event handlers. The event handling methods should be of short duration, because the distributed transaction will not start until all the event subscribers have been notified.

Code-Access Security

An application that uses an LTM transaction can consume resources from at most a single durable resource such as SQL Server 2005. This, however, is not the case with a distributed transaction, which can interact with multiple resources, potentially across the network. This opens the way for both denial-of-service attacks by malicious code, or even just accidental excessive use of such resources. To prevent that, `System.Transactions` defines the `DistributedTransaction` security permission. Whenever a transaction is promoted from an LTM to OleTx transaction, the code that triggered the promotion will be verified to have the `DistributedTransaction` permission. Verification of the security permission is done like any other code-

access security verification, using a stack walk, demanding from every caller up the stack the `DistributedTransaction` permission. Note again that the security demand will affect the code which triggered the promotion, not necessarily the code that created the LTM transaction in the first place (although that can certainly be the case if they are on the same call stack).

This permission demand is of particular importance for Smart Client applications deployed in a partial trust environment, such as the `LocalInternet` zone, that want to perform transactional work against multiple resources. None of the pre-defined partial trust zones grant the

`DistributedTransaction` permission. You will have to grant that permission using a custom code group, or manually list that permission in the application's `ClickOnce` deployment manifest. Another solution altogether is to introduce a middle-tier between the client application and the resources, and have the middle tier encapsulate accessing these resources transitionally.

Concurrency Management and Cloning

Imagine a transactional client that creates a worker thread to perform work concurrently to the client. The client would like to compose the work of the worker thread into its own transaction, meaning, only if the work done on the worker thread is consistent, should the client's transaction commit. However, there are two problems with this scenario. The first is the classic problem of mixing transactions and multithreading – if concurrent work is allowed in a transaction, then there could be a situation where one thread tries to commit the transaction while another tries to abort it. Second, the ambient transaction of the client is stored in the TLS, and as a result, it will not propagate to the worker thread. `System.Transactions` has built-in support to address the issues involving transactional concurrent work.

The `Transaction` class provides the method `DependentClone()`, defined as:

```
Public enum DependentCloneOption
{
    BlockCommitUntilComplete,
    RollbackIfNotComplete,
}
[Serializable]
public class Transaction : ITransaction, ISerializable
{
    public DependentTransaction ←
    DependentClone(DependentCloneOption cloneOption);
```

```
//Rest of the members  
}
```

`DependentClone()` returns an instance of the sealed class

`DependentTransaction` defined as:

```
[Serializable]  
public sealed class DependentTransaction : Transaction  
{  
    public void Complete();  
}
```

`DependentTransaction` derives from `Transaction`, and its sole purpose is to indicate (potentially to the creator of the transaction) that the work done with the cloned transaction is complete and is ready to be committed.

Example 15 demonstrates cloning a dependent transaction and passing it to a worker thread.

```
public class WorkerThread  
{  
    public void DoWork(DependentTransaction  
        dependentTransaction)  
    {  
        Thread thread = new Thread(ThreadMethod);  
        thread.Start(dependentTransaction);  
    }  
    public void ThreadMethod(object transaction)  
    {  
        DependentTransaction dependentTransaction;  
        dependentTransaction = transaction as DependentTransaction;  
        Debug.Assert(dependentTransaction != null);  
        Transaction oldTransaction = Transaction.Current;  
        try  
        {  
            Transaction.Current = dependentTransaction;  
            /* Perform transactional work here */  
            dependentTransaction.Complete();  
        }  
        finally  
        {  

```

```

        dependentTransaction.Dispose();
        Transaction.Current = oldTransaction;
    }
}
//Client code
using(TransactionScope scope = new TransactionScope())
{
    Transaction currentTransaction = Transaction.Current;
    DependentTransaction dependentTransaction;
    dependentTransaction =
        currentTransaction.DependentClone(←
        DependentCloneOption.BlockCommitUntilComplete);
    WorkerThread workerThread = new WorkerThread();
    workerThread.DoWork(dependentTransaction);
    /* Do some transactional work here, then: */
    scope.Complete();
}

```

Example 15: Using dependent clone by another thread.

The client code creates a transactional scope which also sets the ambient transaction. Although the ambient transaction is a committable transaction, you don't want to pass that transaction to the worker thread. Instead, the client clones the current (ambient) transaction by calling `DependentClone()` on the current transaction. `DependentClone()` creates a dependent-transaction object – the underlying returned object is

`DependentTransaction` not `CommittableTransaction`. Note that the `Transaction` class also provides a raw `Clone()` method, which returns a true clone of the transaction, including `CommittableTransaction` (if applicable). Avoid passing that dangerous clone to worker threads for the reasons listed at the beginning of this section.

The class `WorkerThread` provides the `ThreadMethod()` that will execute on the new thread. The client starts a new thread passing the dependent transaction as the thread method parameter. The problem now is synchronization on completion – what if the client reaches the end of the transactional scope before the worker thread is done? How could the client try to commit its transaction in that case?

To address this, the transaction object keeps track of all the dependent clones it created. The `DependentClone()` method accepts a parameter called `cloneOption` of the enum of the type `DependentCloneOption`. If

`cloneOption` is equaled to

`DependentCloneOption.BlockCommitUntilComplete`, when the client tries to commit the transaction, the client will be blocked until all dependent transactions are completed. In Example 15 the client will be blocked trying to dispose of the transaction object at the end of the `using` statement. If `cloneOption` is `DependentCloneOption.RollbackIfNotComplete`, the client will not be blocked trying to commit. Instead, if there are still active dependent transactions and the client tries to commit, the transaction is aborted and a `TransactionAborted` exception is thrown. Not only that, but the worker thread is not affected, and it continues to work on the transaction in vain. Unless you have a need for some custom manual synchronization, always have `cloneOption` set to `DependentCloneOption.BlockCommitUntilComplete`.

Even with `cloneOption` set to

`DependentCloneOption.BlockCommitUntilComplete`, there are a few additional concurrency issues you need to be aware of:

- If the worker thread rolls back the transaction but the client tries to commit it, a `TransactionAborted` exception is thrown.
- The worker thread can only call `Complete()` once, and will get `InvalidOperationException` exception otherwise.
- Make sure to create a new dependent clone for each worker thread in the transaction. Never pass the same dependent clone to multiple threads, because only one of them can call `Complete()` on it.
- If the worker thread spawns a new worker thread, make sure to create a dependent clone from the dependent clone and pass that to the new thread.

Interoperability

`System.Transactions` supports natively .NET Enterprise Services - any serviced component can take advantage of the new transaction managers and transaction promotion, without having the developers do anything different about it (see Example 2).

The interesting questions are what happens when a transactional serviced component creates a `TransactionScope` object, or when a transactional scope creates serviced components. If the `TransactionScope` object executes in an Enterprise Services transactional context, should it use the Enterprise Services transaction as the ambient transaction? Which Enterprise Services context should the `TransactionScope` object use? What issues arise from such mix-and-match scenarios? The Enterprise Services

transaction programming model is coupled to the object lifecycle and state management, and combining that with transactional scopes that are not even object-based may lead to some complicated side effects.

`System.Transactions` defines three levels of interoperability between itself and Enterprise Services: None, Automatic, and Full. The enum `EnterpriseServicesInteropOption` defined as:

```
public enum EnterpriseServicesInteropOption
{
    Automatic,
    Full,
    None
}
```

allows you to specify the interoperability level. The `TransactionScope` class provides constructors that accept `EnterpriseServicesInteropOption`, for example:

```
public TransactionScope(
    TransactionScopeOption scopeOption,
    TransactionOptions transactionOptions,
    EnterpriseServicesInteropOption interopOption
);
```

`EnterpriseServicesInteropOption.None`, as the name implies, means that there is no interoperability between Enterprise Services contexts and transactional scopes. Such transactional scopes will completely ignore the transactional context or their creating client, and will use their own ambient transaction. The ambient transaction will be distinct from the Enterprise Services managed transaction. As a result, you could have the transactional scope's transaction aborts while the Enterprise Services transaction around it commits, as shown in Example 16.

```

[Transaction]
public class MyService : ServicedComponent
{
    [AutoComplete]
    public void DoSomething()
    {
        TransactionOptions options = new
        TransactionOptions();
        options.IsolationLevel =
        IsolationLevel.Serializable;
        options.Timeout =
        TransactionManager.DefaultTimeout;
        using(TransactionScope scope = new
        TransactionScope(
            TransactionScopeOption.Required,
            options,
            EnterpriseServicesInteropOption.None)
        {
            //No call to scope.Complete(), yet the COM+
            //transaction still can commit
        }
    }
}

```

Example 16: Using `ComplusInteropOption.None`.

`EnterpriseServicesInteropOption.None` eliminates any side effects resulting from mixing Enterprise Services transactions and ambient transactions, and as such it is the safest option.

`EnterpriseServicesInteropOption.None` is the default used by `TransactionScope` with all constructors that do not accept a `EnterpriseServicesInteropOption` value.

If you do want to combine `EnterpriseServices` transactions with your ambient transaction, you need to use either

`EnterpriseServicesInteropOption Automatic` or

`EnterpriseServicesInteropOption.Full`. Both these values rely on serviced domains, and therefore require running on Windows XP Service Pack 2 or Windows 2003 Server.

`EnterpriseServicesInteropOption.Full` will try to behave as much as possible as a serviced component would. If the `TransactionScope` object needs a transaction (either joining an existing ambient transaction or creating a new one), `EnterpriseServicesInteropOption.Full` will create a new Enterprise Services transactional context, flow into it any existing Enterprise Services transaction (or create a new Enterprise Services transaction) and the ambient transaction used will be the same transaction used by that Enterprise Services context. If the `TransactionScope` object does not require a transaction, the scope will be placed in the default Enterprise Services context.

`EnterpriseServicesInteropOptionAutomatic` combines `EnterpriseServicesInteropOption.None` and `EnterpriseServicesInteropOption.Full`. `EnterpriseServicesInteropOptionAutomatic` checks whether the scope is constructed in the default Enterprise Services context or not.

If the scope is constructed in the default Enterprise Services context, then there is no Enterprise Services transaction.

`EnterpriseServicesInteropOptionAutomatic` will behave like `EnterpriseServicesInteropOption.None` and will not create a new Enterprise Services context. If the scope is constructed in any Enterprise Services context besides the default context, then `EnterpriseServicesInteropOptionAutomatic` will behave like `EnterpriseServicesInteropOption.Full`.

I believe that `EnterpriseServicesInteropOptionAutomatic` is flawed. Instead of examining the default context it should have checked whether the creating context is transactional or not.

Tables 2 and 3 below illustrate the Enterprise Services context and the ambient transaction used by a `TransactionScope` object which requires a transaction.

Table 2: Enterprise Services contexts and a transactional scope that requires a transaction.

ES Context	None	Automatic	Full
Default context	Default context	Default context	Create new transactional context
Non-default context	Maintain client's context	Create new transactional context	Create new transactional context

Table 3: Ambient transaction a transactional scope that requires transaction.

ES Context	None	Automatic	Full
Default context	ST	ST	ES
Non-default context	ST	ES	ES

ST – Scope's ambient transaction is managed by `System.Transactions`, separate from any transactions with Enterprise Services contexts that may be present.

ES – Scope's ambient transaction is same as the Enterprise Services context's transaction.

Conclusion

`System.Transactions` is an innovative and practical architecture that is long overdue. It separates the application programming model from transaction management, it enables automatic promotions of transaction managers for optimized performance, and it is an extensible architecture that will accommodate future transaction managers. Upcoming releases of `System.Transactions` will make it easier to build custom resource managers by providing isolation locks and recovery management, including volatile resource managers. Technologies such as Indigo and platforms such as Longhorn will rely on `System.Transactions` as the foundation for consistent transaction management, offering new transaction managers in a pluggable provider model. The first release of `System.Transactions` in the .NET Framework v2.0 is an important addition to your development arsenal in building robust, high-performance enterprise applications.

About the Author

Juval Lowy is a software architect and the principal of *IDesign* (www.idesign.net), specializing in .NET architecture consulting and advanced .NET training. Juval is Microsoft's [Regional Director](#) for the Silicon Valley, working with Microsoft on helping the industry adopt .NET. His latest book is *Programming .NET Components*, published by O'Reilly. Juval participates in the Microsoft internal design reviews for future versions of .NET. Juval published numerous [articles](#) regarding almost every aspect of .NET development, and is a frequent presenter at development [conferences](#). Microsoft recognized Juval as a *Software Legend*—one of the world's top .NET experts and industry leaders.