7 Transaction Services

TRANSACTION SERVICES ARE USUALLY THE MAIN reason why Enterprise Services is used. So that you do not have to deal with transactions programmatically, Enterprise Services offers a way to use transactions by using attributes.

This chapter opens with an overview of transactions, and then examines how you can task transactions programmatically. With this knowledge, you will see the advantages of transaction services offered by .NET Enterprise Services. The focus of this chapter then turns to how you can use Enterprise Services transactions, and how you can access the new features offered with Windows Server 2003.

Specifically, this chapter covers the following topics:

- Transaction overview
- Programmatic transactions
- Automatic transactions
- Transactions with services without components
- Transaction isolation levels

Transaction Overview

So that you can fully understand the transaction support offered by Enterprise Services, it is important for you to have an overall understanding of transactions.

Let's start with an example of a course database and attendees registered for courses. If an attendee who is registered for the Programming ADO.NET course

wants to change to the Introduction to .NET course because he does not have the necessary prerequisites for attending the first course, he will be removed from the attendee list of the first course, and added to the attendee list of the second one. If one of these operations fails, the other should not happen either. Here a single transaction is needed. Transactions ensure that data-oriented resources are not permanently updated unless all operations complete successfully.

A transaction is defined by a unit of operations that either all succeed or all fail. If all operations complete successfully inside a transaction, the transaction is **committed**, and the updated data is written permanently. If one operation fails, however, a **rollback** is done; as a result, the data exists as it was before the transaction started.

ACID Properties

Transactions can be defined by four basic properties, easily remembered with the acronym ACID (atomicity, consistency, isolation, durability).

Atomicity

Atomicity ensures that either all updates occur or none at all. Because of the atomicity guaranteed by transactions, you do not have to write code to handle the situation where one update was successful, and another was not.

With the course example just mentioned, if removing the attendee from the first course succeeds, but adding him to the second course fails, the transaction is aborted, and the attendee is not removed from the first course.

Consistency

Consistency means that the result of a transaction leaves the system in a consistent state. Before the transaction was started, the data is in a valid state, as it is after the transaction is finished.

Consistency can be described with the same scenario as discussed with atomicity, to move an attendee from one course to the other. If removing the attendee from the first course fails, but adding him to the second course succeeds, the attendee would be registered for two courses at the same time. This is not a valid scenario; the database would not be consistent. The consistency property says that the transaction

must leave the database in a consistent state; so if one part of the transaction fails, all other parts must be undone.

Isolation

Multiple users can access the same database simultaneously. With **isolation**, it is assured that it is not possible outside of a transaction to see data that is being changed inside a transaction before the transaction is committed. It is not possible to access some in-between state that might never happen if the transaction is aborted.

If you do a query to get the list of attendees for all courses, while at the same time a transaction is running to move an attendee from one course to another, the attendee will not be listed on both courses. Isolation guarantees that an interim state of the transaction from outside of the transaction cannot be seen.

Durability

Durability means that a consistent state is guaranteed even in case of a system crash. If the database system crashes, it must be guaranteed that transactions that have been committed are really written to the database.

Distributed Transactions

Enterprise Services supports distributed transactions, where a transaction can cross multiple database systems. These database systems can also be from different vendors.

With distributed transaction resource managers, a transaction manager and a two-phase commit protocol are needed.

Resource managers, as is obvious from the term itself, manage resources. Examples of resources are databases, message queues, or simple files. SQL Server is a resource manager, as is the Oracle database, and the Message Queue server. Resource managers must support the two-phase commit protocol.

The two-phase commit consists of a prepare phase and a commit phase. When updating data, a prepare phase occurs first, during which all participants in the transaction must agree to complete the operations successfully. If one of the

participants aborts the transaction, a rollback occurs with all other participants. Agreeing to the prepare phase, the database must also guarantee that the transaction can commit after a system crash. If all participants of a single transaction had a successful prepare phase, the commit phase is started, during which all data is permanently stored in the databases.

A distributed transaction must be coordinated by a transaction manager. For Enterprise Services, the **distributed transaction coordinator** (DTC) is such a transaction manager. The DTC is a transaction manager that supports the X/Open XA Specification for Distributed Transaction, and the OLE Transactions protocol. The X/Open XA protocol is not used natively by the DTC; it must be converted to OLE Transactions by the ODBC driver or OLE DB provider. The X/Open XA Specification is not only supported by Microsoft SQL Server, but also by Oracle. The DTC is a Windows service that is part of the Windows operating system.

The distributed transaction coordinator manages transactions to multiple databases and connections from multiple clients. To coordinate work from different systems in a single transaction, the DTCs of different systems communicate with each other. Connections to resource managers that should participate in a distributed transaction must be enlisted with the DTC. To enlist database connections, the ADO.NET data providers for SQL Server and OLE DB are aware of the DTC, and the connections are enlisted automatically if transactional attributes are used with serviced components.[1]

Figure 7-1 shows a single transaction that is running across multiple systems. Here the transaction is started from component A that is running on server A. Component A has a connection to the database on server C. The connection is enlisted with the DTC on the local system (in this case, the DTC of server A). Component A invokes a method in component B that itself requires a transaction. Component B has a connection to the database on server D. The database connection is enlisted in the DTC of server B. On the database systems C and D, we also have DTCs running that manage the transactional interaction with the resource managers of the dependent systems. Because all four DTCs coordinate the transaction, anyone participating in the transaction can abort it. For example, if the resource manager of server D cannot complete the prepare phase successfully, it will abort the transaction. The DTC of server D will tell all other DTCs participating in the transaction

to do a rollback with the transaction. On the other hand, if all members of the transaction are happy with the prepare phase, the DTCs coordinate to commit the transaction.

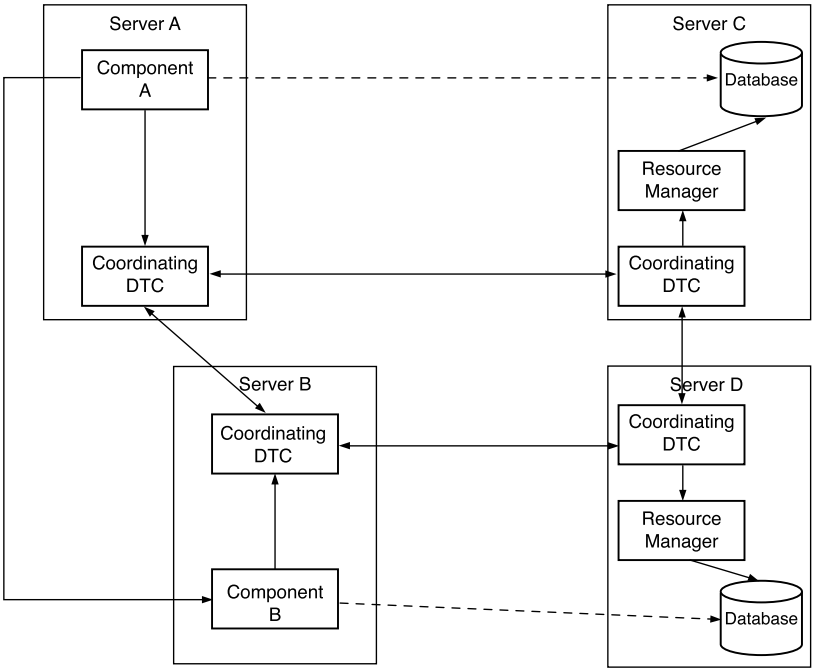

Figure 7-1 Transactions with the distributed transaction coordinator.

Programmatic Transactions

In this chapter, the database that was created earlier is reused. Just as a reminder, the database diagram is shown again in Figure 7-2.

With serviced components, you can do transactions by using attributes. Before looking into these attributes, however, let's take a look at how transactions are done programmatically so that you can easily compare the techniques.

Using the .NET data provider for SQL Server, you can do transactions programmatically with the `SqlTransaction` class. A new transaction is created with the method `BeginTransaction` of the `SqlConnection` class, which returns an object of

type `SqlTransaction`. Then the `SqlTransaction` methods `Commit` and `Rollback` can be used to commit or to undo the transaction result.

Let's immediately consider an example, with the class `CourseData`. A connection string is passed into the constructor of this class, as shown in Listing 7-1. The caller of this class can read the connection string from a configuration file if required. To avoid creating a new `SqlCommand` with every method call, a new `SqlCommand` is instantiated in the constructor, and the parameters that are needed to add a new course record are added to the command object.

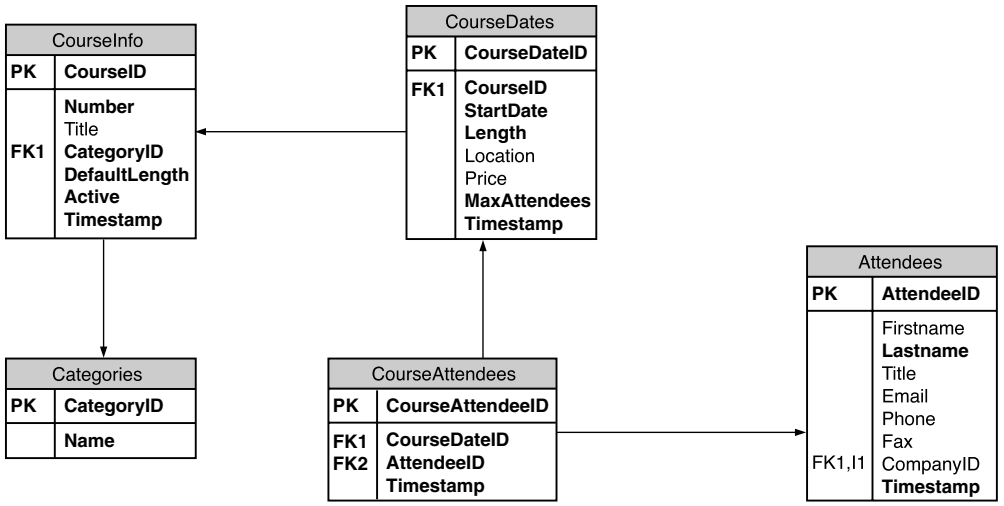

Figure 7-2 Sample database.

Listing 7-1 Programmatic Transactions

```
using System;
using System.Data;
using System.Data.SqlClient;

namespace Samples.Courses.Data
{
   public class CourseData
   {
      private string connectionString;
      private SqlCommand insertCourseCommand;
```

```
public CourseData(string connectionString)
{
   this.connectionString = connectionString;

   insertCourseCommand = new SqlCommand();
   insertCourseCommand.CommandText = "INSERT INTO " +
         "Courses (CourseId, Number, Title, Active) " +
         "VALUES(@CourseId, @Number, @Title, @Active)";

   insertCourseCommand.Parameters.Add("@CourseId",
         SqlDbType.UniqueIdentifier);
   insertCourseCommand.Parameters.Add("@Number",
         SqlDbType.NChar, 10);
   insertCourseCommand.Parameters.Add("@Title",
         SqlDbType.NVarChar, 50);
   insertCourseCommand.Parameters.Add("@Active",
         SqlDbType.Bit);
}
```

In the method `AddCourse` that is shown in Listing 7-2, the connection object is instantiated, the connection is associated with the previously created command object, and the connection is opened. Inside a try block, the `INSERT` statement is executed with the method `ExecuteNonQuery`. Before that, a new transaction is started using the `connection.BeginTransaction` method. The returned transaction object is automatically associated with the connection. If the database insert is successfully, the transaction is committed. If an exception is thrown within `ExecuteNonQuery` because of some error while issuing the database insert, the exception is caught, and a rollback is done in the catch block.

Listing 7-2 AddCourse Method with Programmatic Transactions

```
public void AddCourse(Course course)
{
   int customerId = -1;
   SqlConnection connection =
         new SqlConnection(connectionString);

   insertCourseCommand.Connection = connection;
   insertCourseCommand.Parameters["@CourseId"].Value =
         course.CourseId;
   insertCourseCommand.Parameters["@Number"].Value =
         course.Number;
   insertCourseCommand.Parameters["@Title"].Value =
```

```
            course.Title;
        insertCourseCommand.Parameters["@Active"].Value =
            course.Active;

        connection.Open();
        try
        {
            insertCourseCommand.Transaction =
                connection.BeginTransaction();
            insertCourseCommand.ExecuteNonQuery();
            insertCourseCommand.Transaction.Commit();
        }
        catch
        {
            insertCourseCommand.Transaction.Rollback();
            throw;
        }
        finally
        {
            connection.Close();
        }
    }
```

In the `Test` class of the client application (Figure 7-3), the class `CourseData` is used to create a new course. The connection string passed to the constructor is read from the configuration file using the `ConfigurationSettings` class.

Listing 7-3 Client Test Application

```
using System;
using System.Configuration;
using Samples.Courses.Data;
using Samples.Courses.Entities;

class Test
{

   static void Main()
   {
      // read the connection string from the configuration file
      string connectionString =
            ConfigurationSettings.AppSettings["Database"];

      // write a new course to the database
      CourseData db = new CourseData(connectionString);
```

```
        Course c = new Course();
        c.Number = "MS-2389";
        c.Title = "Programming ADO.NET";
        c.Active = true;

        db.AddCourse(c);
    }
}
```

Listing 7-4 shows the application configuration file that is used by the client application. In the configuration, the database connection string is defined as a child element of `<appSettings>`.

Listing 7-4 Application Configuration File to Define the Connection String

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
  <appSettings>
    <add key="Database"
        value="server=localhost;database=Courses;
        trusted_connection=true" />
  </appSettings>
</configuration>
```

As shown in the previous examples, dealing with transactions programmatically is not a hard task. However, how can the implementation be done if multiple SQL commands should be part of a single transaction? If all these commands are closely related, putting them into a single try block is not a big deal. However, a scenario in which a single transaction should cross multiple components is not so straightforward.

If multiple classes should participate in the same transaction (for example, a customer should be added, and an entry to the course map must also be done), dealing with transactions is not that easy anymore. Take a look at how this can be handled.

Figure 7-3 shows a scenario where a single transaction is useful: If one course attendee wants to change her course schedule, she must be removed from one course list and added to another one. Now it should never happen that the course attendee is added to the new course, but not deleted from the previous one, and it should never happen that the attendee is just removed from a course, but never added to the new course. Both adding and removing the attendee from the course must be part of the same transaction.

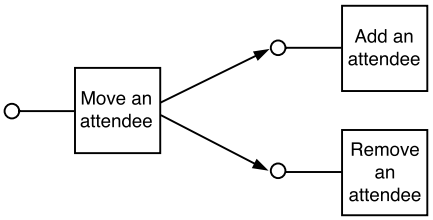

Figure 7-3 Transactions with multiple components.

You can program multiple database actions that are spread across multiple classes, and that should participate in the same transaction, by passing the connection object that is associated with the transaction as an argument to the methods. Inside the methods, the same connection object can be used to participate within the same transaction.

This can get quite complex with many classes and bigger scenarios. You have to decide whether the class should participate in the same transaction, or whether a new transaction should be started. What should happen if an object is called in between that does not use database methods at all? With such a method, it would be necessary to add an additional argument so that the connection object could be passed.

Enterprise Services offers a way to deal with this complexity: **automatic transactions**.

Automatic Transactions

With automatic transactions, you do not have to pass a transaction as an argument of a method; instead, a transaction flows automatically with the context. Using transaction attributes, you can specify whether a transaction is needed.

Using the class attribute `[Transaction]`, you can specify whether objects of the class are aware of transactions, and whether transactions should be created automatically by the Enterprise Services runtime.

The class attribute is applied to the class of the serviced component, as shown in this code segment:

```
[Transaction(TransactionOption.Required)]
public class CustomerDataComponent : ServicedComponent
{
```

Transaction Attributes

The [Transaction] attribute that can be applied to classes that implement serviced components has five different values that you can set with the enumeration TransactionOption. The values of the enumeration TransactionOption are Required, RequiresNew, Supported, NotSupported, and Disabled. Table 7-1 describes what these values mean.

Table 7-1 TransactionOption Enumeration

TransactionOption Value	Description
Required	The value Required marks that the component needs a context with a transaction. If the context of the calling object does not have a transaction, a new transaction is started inside a new context. If the calling object does have a transaction, the same transaction from the calling object is used.
RequiresNew	Similar to the value Required, with RequiresNew the component always gets a context with a transaction; but contrary to Required, a new transaction is always created. The transaction is always independent of a possible transaction of the calling object.
Supported	The value Supported means that the component is happy with or without a transaction. This value is useful if the component does not need a transaction itself, but it may invoke components that do need transactions, and it may be called by components that have already created transactions. Supported makes it possible for a transaction to cross the component, and the calling and called components can participate in the same transaction.
NotSupported	If the component is marked with NotSupported, the component never participates in a transaction. If the calling object does not have a context with a transaction, it can run inside the same context. If the calling object does have a context with a transaction, a new context is created.
Disabled	The option Disabled differs from NotSupported insofar as the transaction in the context of the calling component is ignored.

The meaning of the transaction values you can set with the attribute [Transaction] is greatly influenced by the context of the calling component. Table 7-2 helps make this behavior more clear. In this table, all five transaction values are

listed with both variants, whether the calling component is running inside a transaction or not. Column 3 shows whether the component is running in a transaction, and column 4 shows whether the component is running in a new transaction.

Table 7-2 TransactionOption Behaviors

Attribute	Calling Component Running in a Transaction	Running in a Transaction	Running in a New Transaction
`Required`	Yes No	Always	No Yes
`Requires New`	Yes No	Always	Always
`Supported`	Yes No	Yes No	Never
`Not Supported`	Yes No	Never	Never
`Disabled`	Yes No	Yes, if calling context is shared No	Never

Supported or NotSupported?

I'm often asked, "Why should a component be marked with `Supported` to support transactions, although the component does not need transactions itself?"

Figures 7-4 and 7-5 describe why the transactional value `Supported` is a useful option. In both figures, three objects that call each other are shown. In Figure 7-4, object A has the transactional option `Required`, and because the client does not have a transaction, a new transaction is created. Object B is marked with the transactional value `NotSupported`, and object C again has the value `Required`. Because object B does not support transactions, the context of object B does not have a transaction associated. When object C is called by object B, a new transaction is created. Here the transactions of object A and object C are completely independent. The transaction of object C may commit, whereas the transaction of object A may be aborted.

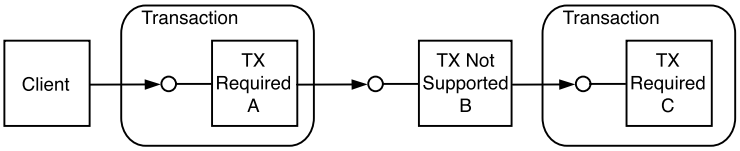

Figure 7-4 Transactional behavior with an interim object transaction NotSupported.

Figure 7-5 shows a similar scenario but with object B marked with the transactional attribute `Supported`. Here the transaction from object A is propagated to B and C. If the database access fails with object C, a rollback with object A occurs, because both objects are running inside the same transaction.

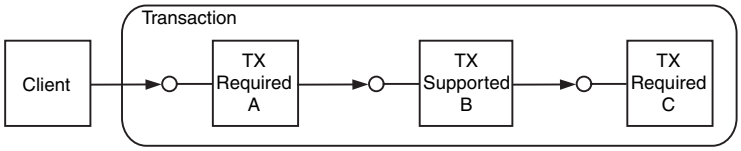

Figure 7-5 Transactional behavior with an interim object transaction Supported.

Required or Requires New?

With the transactional value `Required`, a new transaction is created if a transaction does not already exist in the calling object. If the calling object has a context with a transaction, the same transaction is used. Under some scenarios, a different behavior is required. For example, if object A in Figure 7-6 changes a course map, but object B writes the information to a log database, writing the information to the log should happen independently if the transaction with object A succeeds. To make this possible, object B is marked with the transactional configuration `RequiresNew` (a new transaction is required).

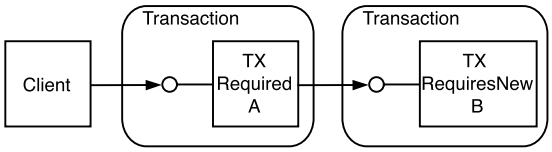

Figure 7-6 Transactional behavior with RequiresNew.

Transaction Streams

A single transaction cannot only be spread across multiple objects; it can also be spread across multiple applications that can run on multiple systems. Inside a transaction stream, all database connections are automatically enlisted in the **distributed transaction coordinator** (DTC).

A transaction stream (Figure 7-7) is started with a root object. A **root object** is the first object in the transaction stream; it starts the transaction. Depending on the configuration of the objects that are called, they will either be part of the same transaction stream, or a new transaction stream will be created.

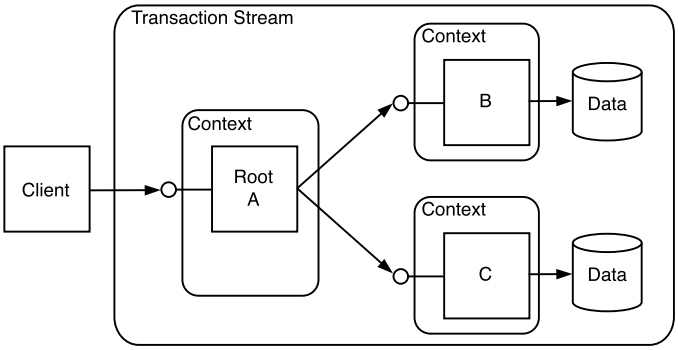

Figure 7-7 Transaction stream.

Transaction Outcomes

Every object, every resource dispenser, and every resource manager participating in a transaction can influence the outcome of a transaction. A transaction is completed as soon as the root object of the transaction deactivates. Whether the transaction should be committed or aborted depends on the done, consistent, and abort bits. The **done** and **consistent** bits are part of every context of every object that participates in the transaction. The done bit is also known as the happy bit. The **abort** bit (also known as the doomed bit) is part of the complete transaction stream. Figure 7-8 shows the done, consistent, and abort bits in conjunction with the serviced components and their contexts.

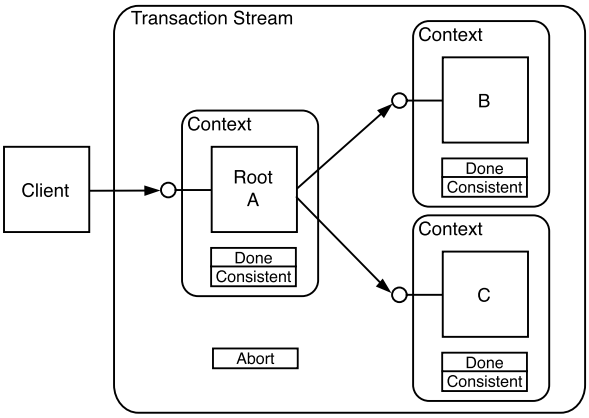

Figure 7-8 Done, consistent, and abort bits.

At object creation time, the done and consistent bits are set to false. With the help of the class `ContextUtil`, the values of these bits can be changed. Setting the done bit to `true` means that the work of the object is complete and the object can be deactivated. The consistent bit is responsible for the outcome of the transaction. If the consistent bit is `true`, the object is happy with the transactional part of its work—the transactional part succeeded.

The abort bit is set to `false` when the transaction is created. When an object is deactivated, the context of the object is lost, and so are the done and consistent bits. Here the result of the consistent bit is propagated to the abort bit. The object context is deactivated as soon as the done bit is set to `true`. Now, if the consistent bit is set to `false`, the abort bit of the transaction stream is set to `true`, so that the transaction will be aborted. If the consistent bit is set to `true`, it does not influence the value of the abort bit.

ContextUtil Methods

The `ContextUtil` class has four methods to influence the values of the done and consistent bits. The methods and their influence of the done and consistent bits are shown in Table 7-3. One thing to keep in mind is that calling these methods only influences the outcome of the method of the serviced component but does not have an immediate result.

Table 7-3 ContextUtil Methods

ContextUtil Method	Done Bit	Consistent Bit	Description
`SetComplete`	`true`	`true`	The method `SetComplete` marks the object as done and consistent. The work of the object is completed, and the transaction can be committed. When the method returns, the object is deactivated, and the vote of the object regarding the transaction is to commit the transaction.
`SetAbort`	`true`	`false`	The work of the object is completed, but it did not succeed; the transaction must be aborted. When the method returns, the object is deactivated, but the transaction vote is to abort the transaction.
`EnableCommit`	`false`	`true`	The work of the object is not completed; it should not be deactivated. However, the first phase of the transactional work was successfully completed. When the method returns, the object will not be deactivated, but the vote for the transaction is to commit the transaction. If the root object of the transaction completes the transaction before the object is called for a second time, the object is deactivated.
`DisableCommit`	`false`	`false`	Similar to `EnableCommit`, with `DisableCommit` the work of the object is not completed. State of the object will be kept, so that it can be invoked another time. However, contrary to the `EnableCommit` method, if the transaction is completed before the object is invoked a second time, the transaction vote is to abort the transaction.

The Done Bit Is Ignored If the Transaction Is Finished

Although the `EnableCommit` and `DisableCommit` methods set the done bit to `false` so that the state of the object can be kept, it can still happen that the object will be deactivated, and the state will be lost. When the transaction is completed (either committed or aborted), all objects participating in the transaction are also deactivated.

Automatic Transaction Example

Figure 7-9 shows the assemblies for the sample application. With the sample application, the assembly `Samples.Courses.Data` includes the class `CourseData` that is doing the database access code with the methods `AddCourse`, `UpdateCourse`, and `DeleteCourse`. The assembly `Samples.Courses.Components` includes the serviced component `CourseDataComponent` that uses the class `CourseData`. The assembly `Samples.Courses.Entities` includes business classes such as `Course`, `CourseData`, and `CourseDataCollection`. These classes are used all over within the application.

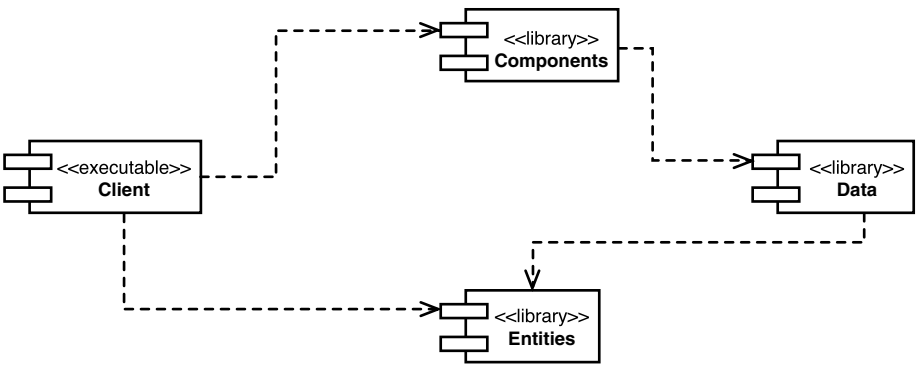

Figure 7-9 Assemblies with the transaction sample.

NOTE: Separating Assemblies

Using automatic transactions, you can use an assembly just for data access that is separated from the serviced component assembly. I always prefer this technique to reduce the code inside the serviced components to a large extent for being flexible with the serviced components technology. With .NET, there is no requirement that the classes accessing the database be configured components. (This was a requirement with COM components.)

Listing 7-5 shows the `AddCourse` method from the `CourseData` class. This method is very similar to the `AddCourse` method that was shown in Listing 7-2. But, contrary to the first implementation of `AddCourse`, here the transactional code has been removed because the transactional support will be added within the serviced component class.

Listing 7-5 AddCourse Method Without Programmatic Transactions

```
public void AddCourse(Course course)
{
   SqlConnection connection =
         new SqlConnection(connectionString);
   insertCourseCommand.Connection = connection;
   insertCourseCommand.Parameters["@CourseId"].Value =
         course.CourseId;
   insertCourseCommand.Parameters["@Number"].Value =
         course.Number;
   insertCourseCommand.Parameters["@Title"].Value =
         course.Title;
   insertCourseCommand.Parameters["@Active"].Value =
         course.Active;

   connection.Open();
   try
   {
      insertCourseCommand.ExecuteNonQuery();
   }
   finally
   {
      connection.Close();
   }
}
```

The implementation of the serviced component class is shown in Listing 7-6. The class `CourseUpdateComponent` derives from the base class `ServicedComponent` and is marked with the attribute `[Transaction(TransactionOption.Required)]`, so that a transaction stream will be created automatically when a new instance is created. The construction string[2] is passed with the method `Construct` that is overridden from the base class. Serviced component construction is enabled with the attribute `[ConstructionEnabled]`. This way the construction string can be changed later by the system administrator.

In the method `AddCourse`, the transaction outcome is influenced with the methods of the class `ContextUtil`: `SetComplete` and `SetAbort`. If the method `db.AddCourse` completes successfully, `ContextUtil.SetComplete` sets the done and consistent bits to mark a successful outcome. If an exception occurs, the exception is caught in the catch block. Here, the consistent bit is set to false by calling the method `ContextUtil.SetAbort`. The exception is rethrown, so that the calling method can get some information about the reason why the method failed. Of course, you can also create a custom exception that is thrown in case of an error.

Listing 7-6 Serviced Component Class CourseUpdateComponent

```
using System;
using System.EnterpriseServices;
using Samples.Courses.Data;
using Samples.Courses.Entities;

namespace Samples.Courses.Components
{
   public interface ICourseUpdate
   {
      void AddCourse(Course c);
      void UpdateCourse(Course c);
      void DeleteCourse(Course c);
   }

   [Transaction(TransactionOption.Required)]
   [ConstructionEnabled(true,
         Default="server=localhost;database=courses;"
         "trusted_connection=true")]
   [EventTrackingEnabled(true)]
   public class CourseUpdateComponent: ServicedComponent,
         ICourseUpdate
   {
```

```
        public CourseComponent()
        {
        }

        private string connectionString;

        protected override void Construct(string s)
        {
           connectionString = s;
        }

        public void AddCourse(Course c)
        {
           CourseData db = new CourseData(connectionString);
           try
           {
              db.AddCourse(c);
              ContextUtil.SetComplete();
           }
           catch
           {
              ContextUtil.SetAbort();
              throw;
           }
        }

        public void UpdateCourse(Course c)
        {
           //...
        }
        public void DeleteCourse(Course c)
        {
           //...
        }
     }
  }
```

After you have registered the serviced component assembly, you can see the transactional options with the Component Services Explorer. You just have to select the properties of the component and open the Transaction tab (see Figure 7-10). In this dialog box, you can see the transactional option that was defined with the attribute `[Transaction]`.

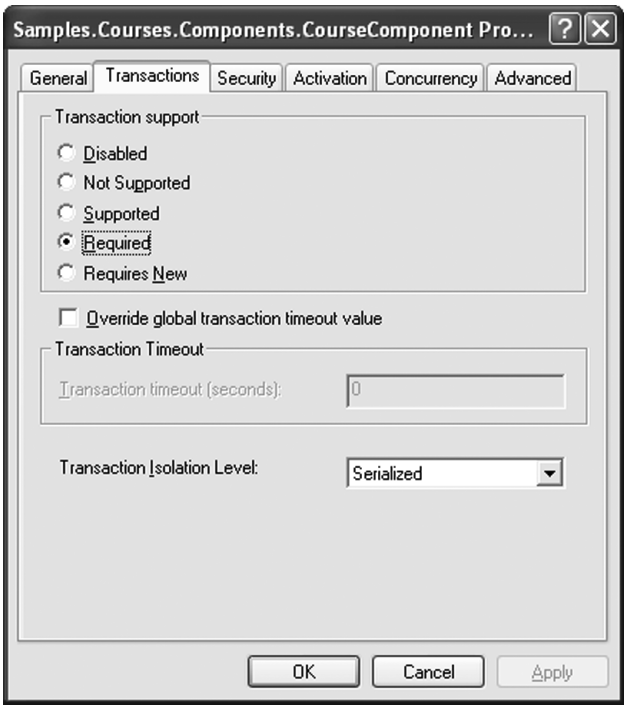

Figure 7-10 Component Services Explorer—transactional options.

If you change the option from Required to Not Supported with the Component Services Explorer, a runtime exception will occur with Windows Server 2003. During runtime, it is checked whether the programmatic configuration corresponds to the manual configuration regarding the transactional behavior.

Transaction Options and JITA

Selecting a transactional option such as `Required` and `Requires-New` also requires **just-in-time activation** (JITA)[3] and synchronization. It is not necessary to configure JITA and synchronization explicitly, but these options will be configured automatically.

Setting the Transactional Vote

Instead of setting the transactional vote with the `ContextUtil` class and the methods `SetComplete`, `SetAbort`, `EnableCommit`, and `DisableCommit` indirectly, you can set them directly. The class `ContextUtil` offers the property `MyTransactionVote` for this purpose, which you can set to one value of the enumeration `TransactionVote`: `Commit` or `Abort`. Setting `MyTransactionVote` to `Commit` sets the consistent bit to `true`, whereas setting `MyTransactionVote` to `Abort` sets the consistent bit to `false`.

The done bit is directly influenced when you set the property `DeactivateOnReturn` of the `ContextUtil` class. `ContextUtil.DeactivateOnReturn = true` sets the done bit to true.

The `AddCourse` method does not change a lot, as you can see in Listing 7-7.

Listing 7-7 AddCourse Using the Property MyTransactionVote

```
public void AddCourse(Course c)
{
   ContextUtil.DeactivateOnReturn = true;
   CourseData db = new CourseData(connectionString);
   try
   {
      db.AddCourse(c);
      ContextUtil.MyTransactionVote =
            TransactionVote.Commit;
   }
   catch
   {
      ContextUtil.MyTransactionVote =
            TransactionVote.Abort;
      throw;
   }
}
```

AutoComplete Attribute

Instead of calling `SetComplete` and `SetAbort`, all transaction handling can be done if you apply the attribute `[AutoComplete]` to a method. In this way, the implementation of the method `AddCourse` gets easier, because it is not necessary to catch and

to rethrow exceptions. Instead, if the method completes successfully, by applying the attribute `[AutoComplete]`, the done and consistent bits are set to `true`. On the other hand, if an exception is generated, the consistent bit is set to `false` at the end of the method. You can see the implementation of `AddCourse` using the `[AutoComplete]` attribute in Listing 7-8.

Listing 7-8 AddCourse Using the [AutoComplete] Attribute

```
[AutoComplete]
public void AddCourse(Course c)
{
   CourseData db = new CourseData(connectionString);
   db.AddCourse(c);
}
```

Using the `[AutoComplete]` attribute, you have to throw exceptions to the caller. The `[AutoComplete]` attribute sets the consistent bit to `false` only if an exception is thrown. If you want to handle exceptions in the `AddCourse` method, you can add a try/catch block where the exception is rethrown (see Figure 7-9). Instead of rethrowing the same exception, you can throw an exception of a custom exception type—it is just important to throw an exception in case of an error.

Listing 7-9 [AutoComplete] Attribute with Exception Handling

```
[AutoComplete]
public void AddCourse(Course c)
{
   try
   {
      CourseData db = new CourseData(connectionString);
      db.AddCourse(c);
   }
   catch (Exception ex)
   {
      // do some event logging
      // re-throw exception
      throw;
   }
}
```

Distributed Transactions

Enterprise Services transactions are enlisted in the DTC, so these transactions can run across multiple systems and across multiple databases. Figure 7-11 demonstrates such a distributed scenario. A single transaction can consist of an update to a SQL Server database, an update to an Oracle database, and writing some messages to a message queue server,[4] all on different systems.

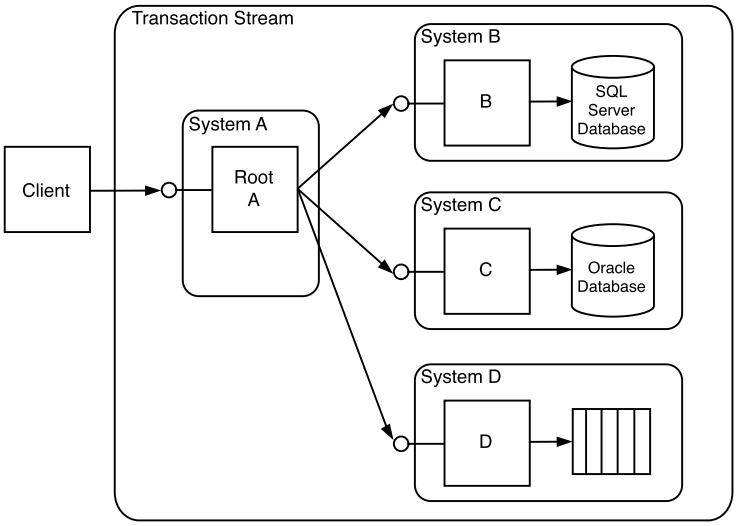

Figure 7-11 Distributed transactions.

Enabling Distributed Transactions with Windows Server 2003

With Windows Server 2003, you explicitly have to enable DTC access over the network. You can do this by selecting the Enable Network DTC Access option (see Figure 7-12) when configuring the application server components.

Transactions with Services Without Components

Starting with Windows Server 2003 and Windows XP SP2, it is possible to create contexts with transactions without defining transactional attributes with serviced

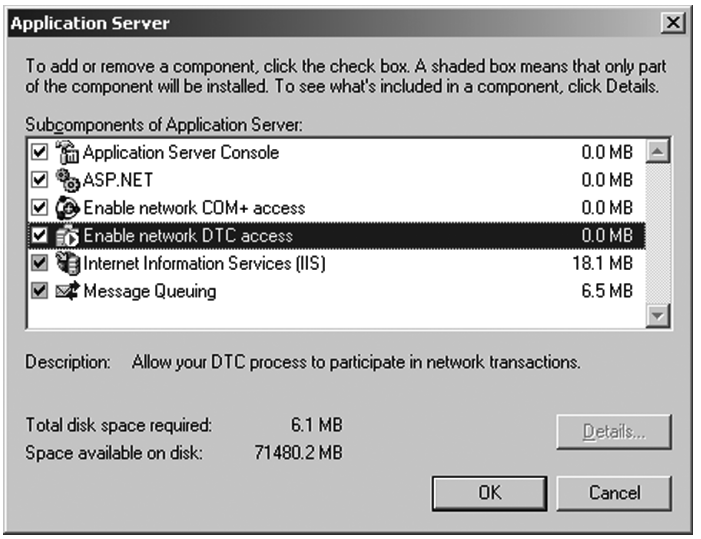

Figure 7-12 Enable DTC with Windows Server 2003.

component classes. You can create transactional contexts with the `ServiceConfig`, the `ServiceDomain`, and the `Activity` classes. This is not only useful without serviced components, but also with serviced components. With serviced components, this feature makes it possible that one method of a class uses a transaction, whereas another does not use it. For example, with the class `CourseData`, it would not be necessary to create a transaction with the method `GetCourses`, where data from the database is read-only, but with the method `AddCourse`, a transaction is necessary.

The code in Listing 7-10 demonstrates using transactions without serviced components. After reading the connection string from the application configuration file and creating a new object of type `CourseData` (`CourseData` is a normal class that does not derive from `ServicedComponent`), a new `ServiceConfig` object is created. With the `ServiceConfig` object, the context can be configured, and the transactional attributes can be set with the `Transaction` property of this class. The values that can be set with the `Transaction` property have already been shown: a value from the enumeration `TransactionOption`. Here `TransactionOption.Required` is used as the value for the `Transaction` property. The context is created as soon as we call `ServiceDomain.Enter`, passing the context configuration information. The next

method, `data.AddCourse`, is running inside the context, and the database connection that is used inside this method is enlisted with the DTC. The transaction is committed when the context is left with `ServiceDomain.Leave`, which returns the status of the transaction as a value of the enumeration `TransactionStatus`.

Listing 7-10 Transactions with Services Without Components

```
using System;
using System.EnterpriseServices;
using System.Configuration;
using Samples.Courses.Data;
using Samples.Courses.Entities;

class TransactionsWithoutComponents
{
   [STAThread]
   static void Main()
   {
      Course c = new Course();
      c.Number = "MS-2524";
      c.Title = "ASP.NET Web Services";
      c.Active = true;

      string connectionString =
            ConfigurationSettings.AppSettings["Database"];
      CourseData data = new CourseData(connectionString);

      // create a transactional context
      ServiceConfig context = new ServiceConfig();

      context.Transaction = TransactionOption.Required;

      // enter the transactional context
      ServiceDomain.Enter(context);
      data.AddCourse(c);

      // leave the transactional context
      TransactionStatus status = ServiceDomain.Leave();

      Console.WriteLine("Transaction ended with {0}", status);
   }
}
```

The result of the transaction is returned from the `ServiceDomain.Leave` method. The possible values are listed in Table 7-4.

> **NOTE: Services Without Components Platforms**
>
> If you try running the sample application on Windows XP without SP2, you will get the exception `PlatformNotSupportedException`. Services without components is only supported on Windows Server 2003 and Windows XP SP2.

Some examples of when it is helpful to use transactions without serviced components are as follows:

- Standalone applications.
- With ASP.NET Web services that use serviced components, a root transactional context can be created without serviced component configuration.
- Services without components can also be useful within serviced components: for example, if the serviced component class has one method that requires a transaction, and another method that doesn't require one.

Table 7-4 TransactionStatus Enumeration

TransactionStatus	Description
`Aborted`	The transaction is aborted. Either the database returned an error, or the method generated an exception, so the transaction is aborted.
`Aborting`	The transaction is in the process of being aborted.
`Committed`	The transaction committed successfully. Every member who was participating in the transaction voted the transaction by setting the consistent bit.
`LocallyOk`	The transaction is neither committed nor aborted. This status can be returned, if the context that is left is running inside another transactional context.
`NoTransaction`	The context did not have a context, so `NoTransaction` is returned.

Transactions Support with ASP.NET

You can create distributed transactions directly from ASP.NET. Doing so proves useful if the ASP.NET application is a client of the serviced component. With the ASP.NET transactional support, the ASP.NET page can act as the root of the transaction.

The `Page` directive of an ASP.NET page enables you to set the named property `Transaction` to a value you already know from the attribute `[Transaction]`. The allowed values of the `Transaction` property are `Required`, `RequiresNew`, `Supported`, `NotSupported`, and `Disabled`.

```
<%Page language="C#" Transaction="Required" %>
```

When you are calling serviced components from ASP.NET, the transaction can already be created; so multiple components that are called within the same method can participate with the same transaction.

> **Passing Transactions from ASP.NET to Serviced Components**
>
> Passing transactions from the ASP.NET page to the serviced component is only possible with the DCOM protocol, not with .NET remoting.[5]

Transaction Isolation

With Windows 2000, all COM+ transactions used the serialization level for transaction isolation. With Windows XP and Windows Server 2003, you can choose between different isolation levels. The full ACID properties that describe a transaction can only be fulfilled with the serializable transaction isolation level, but sometimes this level is not needed. Using a less-strict transaction isolation level can increase the scalability of the application.

With large numbers of users accessing data, turning knobs on transaction isolation can bring you a lot more performance than you can get out of fine-tuning

components. With a data-driven application database, locks can make you lose a lot of performance.

If you want to change transaction isolation levels, you have to be aware of what can happen so that you know when you can change the default values.

Potential Transaction Isolation Problems

Potential problems that can occur during transactions can be grouped into three categories: **dirty reads**, **nonrepeatable reads**, and **phantoms**:

- **Dirty reads**—A dirty read means that you can read data that was changed inside a transaction from outside of the transaction. Suppose that transaction 1 changes some customer data, but this transaction fails, so with a rollback the data is unchanged. If at the same time inside transaction 2 the same customer record is read, the data that is read never really existed, so this is called a dirty read.
- **Nonrepeatable reads**—If the same row is read multiple times, and you get different results with each read, this is called a nonrepeatable read. If you read data in transaction 1, and at the same time transaction 2 changes the data that you have read within transaction 1, and you read the data again within transaction 1, different data will be read inside the same transaction. This is a nonrepeatable read.
- **Phantoms**—A phantom is a row that matches the search criteria, but is not seen in the query. If you do a query based on a condition in transaction 1, while during transaction 2 a new record is inserted that would fit to the query of transaction 1, this row is a phantom. If all rows that match the query in transaction 1 are changed (for example, the salary of all employees that have a salary lower than $1,000 is changed to $1,000), new records that are added with a lower salary level are not updated.

Transaction Isolation Levels

Depending on the serialization level used, one of these described problems can occur, or the isolation level guarantees that the problem cannot occur. With the

isolation level, you can specify one of these values: **read uncommitted**, **read committed**, **repeatable read**, or **serializable**:

- **Read uncommitted**—With this level, transactions are not isolated from each other. This is the most scalable way for transactions; but with it, all problems described earlier can occur.
- **Read committed**—With a level of read committed, the transaction waits until all write-locked rows that are manipulated by the transaction are unlocked. It is therefore guaranteed that no read of dirty data can occur.

  The transaction holds a read lock for all rows that are read-only, and a write lock for all records that are updated or deleted. The read locks are released with a move to the next record, whereas all write locks are held until the transaction is committed or aborted.
- **Repeatable read**—Repeatable read is functionally similar to read committed. The only difference is that the read locks are held until the transaction is committed or aborted. This way, nonrepeatable reads cannot happen anymore.
- **Serializable**—Similar to read committed and repeatable read, the transaction waits until all write-locked rows that are manipulated by the transaction are unlocked. Similar to repeatable reads, read locks are done for rows that are read-only, and write locks are done for rows that are updated or deleted. What is different with this level is how ranges of data are locked. If a `SELECT * FROM Customers` statement is executed, a table lock is done. This way no new records can be inserted. With the SQL statement `SELECT * FROM Customers WHERE City = 'Vienna'`, all records with the `City` column set to `Vienna` are locked. This way, phantoms are not possible.

Table 7-5 summarizes potential problems that can occur depending on the isolation level.

When to Use Which Isolation Level

Consistency and concurrency have conflicting goals. For consistent data, locking must be done when the data is accessed. For best consistency, the transaction isolation level serializable offers the best support. Locking the data also means that no

Table 7-5 Isolation Level Behavior

Isolation Level	Dirty Reads	Nonrepeatable Reads	Phantoms
Read uncommitted	Yes	Yes	Yes
Read committed	No	Yes	Yes
Repeatable read	No	No	Yes
Serializable	No	No	No

other task may access the data simultaneously. To get better concurrency, you must select a lower isolation level.

Depending on the task to be done, you can select the isolation level accordingly. If only new records are to be inserted, you can set the isolation level to read uncommitted. If data is read once, or multiple times, and if consistency is not required (for example, you are generating a management report), you can set the isolation level to read committed.

Thinking about configuring the isolation level always requires thinking about the complete task that should be protected, what can happen during the time of the task, and how this can influence the task. With this information, you can select the required isolation level.

Specifying the Isolation Level

You can specify the transaction isolation level of a serviced component with the attribute `[Transaction]` by defining the named property `Isolation` and setting a value to one of the defined values in the enumeration `TransactionIsolationLevel`.

```
[Transaction(TransactionOption.Required,
             Isolation = TransactionIsolationLevel.Any)]
[EventTrackingEnabled]
public class CustomerData : ServicedComponent
```

The possible values and their meaning of the enumeration `TransactionIsolationLevel` are shown in Table 7-6.

Table 7-6 TransactionIsolationLevel Enumeration

TransactionIsolationLevel Value	Description
`Any`	If you set the isolation level to `Any`, the same isolation level as the calling component is used. If the object is the root object, the isolation level is `Serializable`.
`ReadUncommitted`	With `ReadUncommitted`, only shared locks are used; exclusive locks are not honored. You should use this option only for read access to generate some results that don't need to be actually up to the second.
`ReadCommitted`	With this option, shared locks are used while data is being read. After reading the data, the shared lock is released. Before the transaction is finished, the data can be changed.
`RepeatableRead`	If you set the option to `RepeatableRead`, locks are placed on all data that is used.
`Serializable`	The level `Serializable` has the best isolation. With this option, updates or inserts that belong to the range of the data that is used are not possible.

If using transactions without components, you can specify the transaction isolation level with the property `IsolationLevel` of the class `ServiceConfig`.

Monitoring the DTC

You can monitor transactions that are enlisted with the DTC by using the Component Services Explorer. Selecting Transaction List in the tree view as a child element of the Distributed Transaction Coordinator shows all current active transactions. Selecting Transaction Statistics opens up the window shown in Figure 7-13. Here you can read the current number of active transactions, the maximum number of transactions that were active concurrently, and the committed and aborted transaction count.

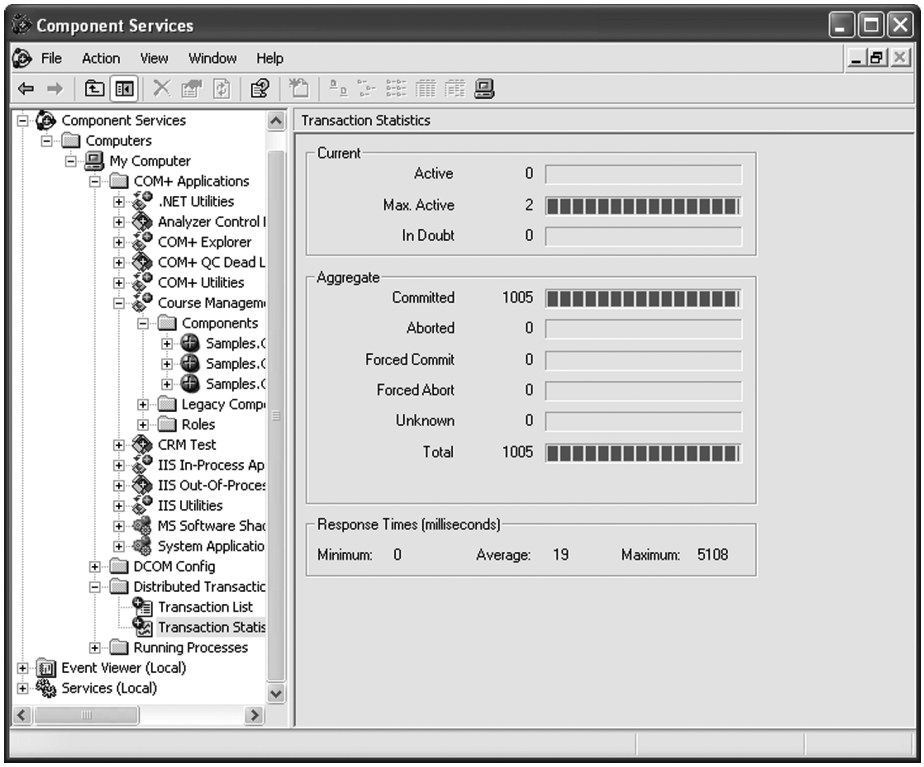

Figure 7-13 Transaction statistics.

Transactions with .NET 2.0

One disadvantage of automatic transactions with serviced components is that these transactions always make use of the DTC, even when local transactions would be enough. Of course, using DTC is more expensive than a local programmatic transaction as was used in Listing 7-2.

With .NET 2.0, a new assembly `System.Transactions` will be available that includes some new classes dealing with transactions. Particularly worth mentioning are the classes `Transaction`, `TransactionScope`, and `TransactionManager`. These classes are in the namespace `System.Transactions`. The big advantage of these classes is that the new transaction services know about local and DTC

transaction and will support the Web services standard WS-AtomicTransaction[6] in the future. Depending on the scope of the transaction, the fastest possible technology is used.

Listing 7-11 shows one way to create transactions with .NET 2.0.[7] Creating and disposing of a `TransactionScope` object defines a transactional code block. With the constructor of the `TransactionScope` object and the `TransactionScopeOption` enumeration, you can define whether a new transaction is required or whether a transaction that already exists from the outer block should be used. With the parameter of type `EnterpriseSerivcesInteropOption`, you can specify the interaction with COM+ contexts. The method `Complete()` indicates that all operations within the scope of the transaction have been successful. At the end of the `using` statement (where the `Dispose()` method gets called), the transaction outcome of the block is defined. If the `Complete` method was not called because an exception occurred, the transaction is aborted. If the scope of the transaction was successful, the transaction is committed if it is a root transaction. If the scope is not the root of the transaction, the outcome of the transaction is influenced.

Listing 7-11 Programming Transactions with .NET 2.0

```
using System;
using System.Transactions;
using Samples.Courses.Entities;
using Samples.Courses.Data;

class TxDemo
{
   static void Main()
   {
      using (TransactionScope scope = new TransactionScope(
            TransactionScopeOption.Required))
      {
         Course c = new Course();
         c.Number = "MS-2557";
         c.Title = ".NET Enterprise Services";
         c.Active = true;

         CourseData db = new CourseData(connectionString);
         db.AddCourse(c);
```

```
            scope.Complete();
        }
    }
```

With Indigo Services, you may declare transactions, as shown in Listing 7-12. This should look similar to the declarative transactions of the serviced components model. The attribute `[OperationContract(TransactionFlowAllowed=true)]` defines that a transaction is created. The attribute `[OperationBehavior(AutoEnlistTransaction=true)]` defines that the transaction is automatically enlisted with the transaction coordinator.

Listing 7-12 Declaring Transactions with Indigo Services

```
[ServiceContract]
[ServiceBehavior(InstanceMode=InstanceMode.PerCall)]
public interface IDemoService
{
   [OperationContract(TransactionFlowAllowed=true)]
   [OperationBehavior(AutoEnlistTransaction=true)]
   void AddCourse(Course course);
}

[BindingRequirements(
   TransactionFlowRequirements = RequirementsMode.Require,
   QueuedDeliveryRequirements = RequirementsMode.Disallow,
   RequireOrderedDelivery = true
)]
public class DemoService : IDemoService
{
   public void AddCourse(Course course)
   {
      //...
   }
}
```

Summary

This chapter covered the transaction features offered by Enterprise Services. Instead of doing transactions programmatically, you can task transactions automatically by

applying the attribute `[Transaction]` to specify transactional requirements. The transactional options `Required`, `RequiresNew`, `Supported`, `NotSupported`, and `Disabled` influence the Enterprise Services interception code so that a new transaction is created, an existing transaction is used, or no transaction is used at all.

The automatic transactions make use of the distributed transaction coordinator, so transactions can also flow across multiple systems.

This chapter also discussed the problems with concurrency and consistency with transaction isolation levels. Changing the isolation level can increase concurrency, whereas consistency is reduced, and vice versa.

Windows Server 2003 also offers the new feature named services without components; transactions are part of this service.

1 The connection string defines whether a connection should be enlisted with the transaction context of the thread. With the .NET data provider for SQL Server, you can define whether the transaction should be enlisted with the Enlist parameter. The `Enlist` parameter has a default value of true, so that the connection is enlisted with the transaction context. Microsoft .NET data provider for Oracle has the same `Enlist` parameter.

2 Construction strings are discussed in Chapter 2, "Object Activation and Contexts."

3 JITA is discussed in Chapter 2.

4 Transactions that include Message Queuing are shown in Chapter 10.

5 DCOM and .NET remoting are discussed in Chapter 5, "Networking."

6 WS-AtomicTransaction is discussed in Chapter 14.

7 While writing this book, I am using a Beta version of .NET 2.0. The implementation of .NET 2.0 transactions may change with the release version.