

# Distributed Transactions: .NET vs J2EE

Shannon Hardt

---

## Overview

Transactions are common in the context of database programming. For example, if you are executing multiple updates against a specific database, and you would like them to either all succeed or all fail as one atomic operation, there are mechanisms at the database level that allow you to commit all operations as one unit or rollback all changes if one operation fails. But, how do you make updates to two different databases succeed or fail as a unit? This is the domain of distributed transactions. This paper will describe the facilities available in .NET and J2EE which support distributed transactions – how they are similar and how they differ.

But first, let's define what we mean by transactions. A transaction is a set of related tasks that either succeed or fail as a unit. For a transaction to succeed (or commit), all participants must guarantee that any change to data must be permanent. There are four guarantees associated with a transaction called the ACID properties. ACID is an acronym that stands for *Atomicity*, *Consistency*, *Isolation*, and *Durability*.

*Atomicity* guarantees that many operations are bundled together and appear as one contiguous unit of work, operating under an all-or-nothing paradigm – either all of the data updates are performed or nothing happens if an error occurs at any time.

*Consistency* guarantees that a transaction will leave the system in a consistent state after the transaction completes. The term consistent varies depending on the logic of the system, and it is somewhat up to the application developer to enforce the specific rules governing consistent state. Within a transaction, it is possible for some pieces to be in an inconsistent state. However, once the transactions completes – either successfully or unsuccessfully – the system must return to a consistent state. *Atomicity* helps enforce that the system will always appear to be in a consistent state.

*Isolation* protects concurrently executing transactions from seeing each other's incomplete results. Isolation allows multiple transactions to read or modify data without knowing about each other because each transaction is isolated from the others. This is achieved by using low level synchronization protocols on the underlying data. There are several levels of isolation available, each with benefits and drawbacks. More on that later.

*Durability* guarantees that updates to managed resources survive failures. Failures include machine crashes, network crashes, hard disk crashes, and power failures. Recoverable resources keep a transactional log so that the permanent data can be reconstructed by reapplying the steps in the log.

## Distributed Transactions

Distributed transaction processing systems are designed to facilitate transactions that span heterogeneous, transaction-aware resources in a distributed environment. Because they span multiple data resources, it is important that distributed transactions enforce the ACID properties to maintain data consistency across all resources. The execution of a distributed transaction requires coordination between a global transaction management system and all the local resource managers of all the involved systems. The Resource Manager and Transaction Manager (TM, also known as a transaction processing monitor – TP Monitor) are the two primary elements of any transactional system. To support advanced functionalities required in a distributed component-based system, separation of monitor from the resource managers is required.

The global Transaction Manager (TM) is responsible for managing distributed transactions by coordinating with different resource managers to access data at several different systems. Since multiple application components and resources participate in a transaction, it's necessary for the transaction manager to establish and maintain the state of the transaction as it occurs. Resource managers inform the transaction manager of their participation in a transaction by means of a process called resource enlistment. The TM keeps track of all the resources participating in a transaction and uses this information to coordinate transactional work performed by the resource managers with a two-phase commit and recovery protocol. The TM has to monitor the execution of the transaction and determine whether to commit or roll back the changes made to ensure atomicity of the transaction.

The two phase commit protocol between the TM and all the resources enlisted for a transaction ensures that either all resource managers commit or they all abort. When an application requests a transaction to commit, the TM issues a `PREPARE_TO_COMMIT` request to all resource managers. Each of these resources may in turn send a reply indicating whether it is ready for commit or not. Only when all the resource managers are ready for a commit does the TM issue a commit request to all resource managers. At this point the changes are accepted and become permanent. Otherwise, the TM issues a rollback request to all resource managers and the transaction is rolled back.

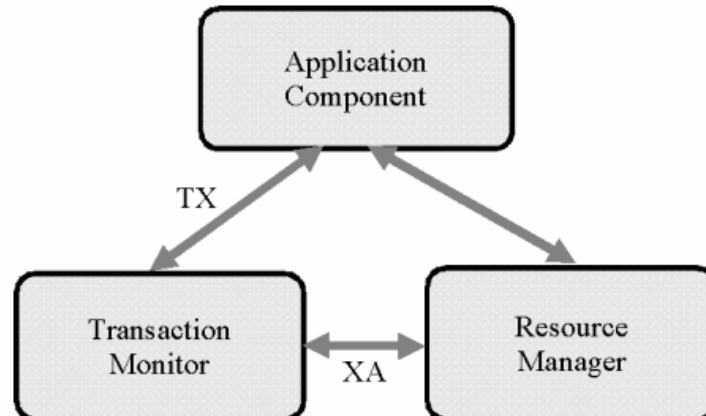
Although two-phase commit guarantees the autonomy of the transaction, the required processing load is rather heavy, creating frequent update conflicts, especially when data is duplicated across multiple sites. However, sometimes it is necessary in large-scale enterprise development to tie operations on multiple disparate systems together as one transaction.

The goal of transparency in a distributed transaction processing environment demands interoperability between TMs as well as interoperability of TMs with resource managers. Currently, the most widely used open standard is the X/Open Distributed Transaction Processing (DTP) Model. X/Open DTP was proposed by the Object Management Group, and is a standard among most of the commercial vendors providing transaction processing and relational database solutions. It is an optional CORBA service.

The two major interfaces specified in the model are the TX interface and the XA interface. The TX interface is between the application and the TM, implemented within the TM. It provides transaction demarcation services by allowing the application components to bind transactional operations within global transactions. The XA interface defines the interface between resource

---

managers and transaction managers. When both the transaction manager and resource managers support the XA interface, they can be plugged together and transaction coordination can take place between them. This is the most important interface in the standard and has wide industry acceptance. Commercial transaction management products like TXSeries/Encina, Tuxedo, and TopEnd support the TX interface. Most of the commercial databases such as Oracle, Sybase, Informix, and MS SQL Server, and messaging middleware like IBM's MQSeries and Microsoft's MSMQ Server provide an implementation of the XA interface.



X/Open DTP standard (from onjava.com article)

In .NET, a distributed transaction processing system consists of several cooperating entities, including Transaction Processing Monitors, Transaction Managers, and Resource Managers. These entities are logical and can reside on the same computer or on different computers. A Transaction Processing Monitor is software that sits between a transaction-aware application and a collection of resources. It connects multiple clients to multiple applications that potentially access multiple data resources.

In a distributed transaction, each participating resource has a local Transaction Manager to track incoming and outgoing transactions on that computer. The monitor assigns one manager the additional task of coordinating all tasks among local managers. While the Transaction Manager coordinates all transaction processing functions, it does not manage data directly. This is the responsibility of Resource Managers. A Resource Manager manages persistent data in databases, durable message queues, or transactional file systems. It stores data and performs disaster recovery.

So, the underlying architecture for distributed transactions in .NET and J2EE are very similar.

## Programmatic (Manual) Transactions

There are basically two ways to handle transactions – control everything yourself or let the framework do it for you. Both .NET and J2EE provide mechanisms for both models.

## ***J2EE Version***

J2EE supports distributed transactions through two specifications, Java Transaction API (JTA) and Java Transaction Service (JTS). JTA is a “high-level, implementation independent, protocol independent API that allows applications and application servers to access transactions”. JTS specifies the implementation of a Transaction Manager which supports JTA and implements the Java mapping of the OMG Object Transaction Service (OTS) 1.1 specification at the level below the API. JTS propagates transactions using the Internet Inter-ORB Protocol (IIOP).

JTA specifies standard Java interfaces between a transaction manager and the participants in a distributed transaction system: the resource manager, the application server, and the transactional applications.

A JTA Transaction is controlled by the J2EE Transaction Manager (TM). You may want to use a JTA transaction because it can span updates to multiple databases from different vendors. However, there is one limitation – it does not support nested transactions.

To demarcate a JTA transaction, you invoke the begin, commit, and rollback methods of the `javax.transaction.UserTransaction` interface.

A `UserTransaction` instance is available from the web tier as well as the EJB tier. The web tier (servlets and JSPs) use JNDI to look up a `UserTransaction`, while EJBs use the `EJBContext`. Web tier transactions can only be handled programmatically, while EJB transactions can be handled programmatically or declaratively. While it is possible to control transactions in the web tier, it is considered good design to encapsulate transaction processing in the business layer (or EJB) tier.

At any time during a distributed transaction, the transaction manager maintains an association between each transaction (which has a unique global ID), application threads, and connections to the resource managers. For example, a TM may associate a single transaction ID with a thread of an application, an SQL connection that has updated a table, a JMS provider waiting to transmit a message, and a resource adapter or Connector executing an external business function. A transaction context is the association of a transaction with an application component or resource manager. The transparent forwarding of a transaction context from one component to another or to a resource manager is called transaction context propagation.

---

## Distributed Transactions: .NET vs J2EE

Shannon Hardt

```
// a servlet using manual transactions
try {
    Context ctx = new InitialContext();
    UserTransaction t =
        (UserTransaction)ctx.lookup("java:comp/UserTransaction");
    t.begin();
    // make updates to multiple resources
    t.commit();
} catch (Exception e) {
    t.rollback();
}

// an EJB method using manual transactions
public void withdrawCash(double amount) {
    UserTransaction t = context.getUserTransaction();
    try {
        t.begin();
        // make updates to multiple resources
        t.commit();
    } catch (Exception e) {
        try {
            t.rollback();
        } catch ( SystemException se) {
            throw new EJBException(se.getMessage());
        }
        throw new EJBException(e.getMessage());
    }
}
```

Examples of programmatic transactions in web and EJB tier

Certain rules apply to programming distributed transactions. In a web component (such as a servlet), transactions may only be started in the service method. In a stateless session bean with bean-managed transactions, a business method must commit or roll back a transaction before returning. However, in a stateful session bean with a JTA transaction, the association between the bean instance and the transaction is retained across multiple client calls. The association is maintained until the instance completes the transaction.

Entity EJBs, Session EJBs, and Message-driven EJBs can all use container-managed transactions. However, only session beans and message-driven beans can use JTA transactions.

### **.NET Version**

The distributed transaction architecture of .NET is very similar to that of J2EE. In .NET, you can exercise full control over transaction boundaries by using a manual transaction. In this case, you can explicitly begin, commit, or rollback a transaction. However, the mechanisms at the application development level to control transactions are different.

From within one transaction boundary, you can begin a second transaction, called a nested transaction. The parent transaction does not commit until all its subordinate transactions commit. This is an improvement over the J2EE version, which does not support nested transactions.

---

ADO.NET and Message Queuing resource APIs enable manual transaction processing. While manual transactions offer explicit control, they lack some of the ease built into the automatic model. There is no automatic enlistment and coordination between data stores. Transactions do not flow from object to object. The developer must manage recovery, concurrency, security and integrity. This is similar to the J2EE programmatic model.

The Connection object will automatically enlist in an existing distributed transaction if it determines that a transaction is active. Automatic transaction enlistment occurs when the connection is opened or retrieved from the connection pool. You can disable auto-enlistment in existing transactions by specifying `Enlist=false` as a connection string parameter. If auto-enlistment is disabled, you can enlist in an existing distributed transaction using `Connection.EnlistDistributedTransaction`. This ensures that modifications made by the code at the data source will be committed or rolled back as the transaction is committed or rolled back.

`EnlistDistributedTransaction` is particularly applicable when pooling business objects. If a business object is pooled with an open connection, automatic transaction enlistment only occurs when that connection is opened or pulled from the connection pool. `EnlistDistributedTransaction` takes a single argument of type `ITransaction` that is a reference to the existing transaction.

The J2EE `UserTransaction` model is similar to the default behavior of .NET Connections – that is, if a change is made to a resource within the boundaries of a `UserTransaction`, it is automatically registered with the transaction. However, .NET allows finer control over participants in a transaction by using the `EnlistDistributedTransaction` method. More control is almost always a good thing, in my opinion. Although, of course, control must be balanced with efficiency in the real world, which is one reason why these frameworks exist in the first place.

## Declarative (Automatic) Transactions

Both the J2EE and .NET managed environments can control transactions for you. You simply declare how the transaction should be handled and the container will provide that for you.

### *J2EE Version*

J2EE attempts to separate orthogonal concerns (such as transactions and security from business logic) by using XML deployment descriptors. These are read by the container to determine how to manage the deployed component. Declarative transactions are only available to EJBs. In J2EE container-managed transactions, six different transaction attributes can be associated with an EJB method: `Required`, `RequiresNew`, `NotSupported`, `Supports`, `Mandatory`, and `Never`. The same transaction attribute can be specified for all methods of an EJB or different attributes can be specified for each method. Attributes are specified in the EJB deployment descriptor.

#### *J2EE Transaction Attributes*

If **Required** is specified, the container ensures that the method will always be invoked with a JTA transaction. If the calling client is associated with an existing transaction, the method will be invoked in the same transaction context. However, if a client is not associated with a transaction, the container will automatically begin a new transaction and try to commit the transaction when

---

## Distributed Transactions: .NET vs J2EE

Shannon Hardt

the method completes. If **RequiresNew** is specified, the container always creates a new transaction before invoking the method and commits when the method returns. If the calling client is associated with a transaction, the container suspends the association of the transaction context with the current thread before starting the new transaction. When the method completes, the container resumes the suspended transaction. If **NotSupported** is specified, the transactional context of the calling client is not propagated to the EJB. If **Supports** is specified, the client context is propagated to the method if it is associated with one. Otherwise, it acts as NotSupported. If **Mandatory** is specified, the container is required to invoke the method in a client's transaction context. If **Never** is specified, the method will not be called within a transaction context. If the client calls with a transaction context, a TransactionRequiredException is thrown.

```
<!-- specify transaction attributes for EJB method -->
<container-transaction>
  <method>
    <ejb-name>TheAccount</ejb-name>
    <method-intf>Remote</method-intf>
    <method-name>changeContactInformation</method-name>
    <method-params>
      <method-param>
        com.sun.blueprints.util.ContactInformation
      </method-param>
    </method-params>
  </method>
  <trans-attribute>Required</trans-attribute>
</container-transaction>
```

Example deployment descriptor snippet for declaring transaction attributes of EJB

In .NET, an automatic transaction manages transaction boundaries for you, based on a declarative attribute set for each component in code. A transaction automatically flows to objects instructed to participate in a transaction and bypasses objects instructed to execute outside a transaction, similar to J2EE. You cannot nest transactions when using automatic transactions, which is the same as J2EE. Furthermore, the .NET Framework class must be registered with COM+ to participate in an automatic transaction. J2EE components must be deployed to a container.

Microsoft Transaction Server, COM+, and the CLR all support the same automatic distributed transaction model. After an ASP.NET page, XML web service method, or .NET Framework class is marked to participate in a transaction, it will automatically execute within the scope of a transaction. You can control an object's transactional behavior by setting a transaction attribute value on a page, in an XML web service method, or in a class. The attribute value determines the transactional behavior of the instantiated object. The syntax varies slightly among different .NET constructs, but the principal is the same. The available attributes are Disabled, NotSupported, Supported, Required, and RequiresNew. The semantics of these attributes map directly to those found in J2EE.

In ASP.NET, you can insert a transaction directive in the code for the page. All resource management will be performed in the context of a transaction determined by the attribute value specified. This is significantly different than J2EE, in that the UserTransaction object is retrieved

---

## Distributed Transactions: .NET vs J2EE

Shannon Hardt

via JNDI. The ASP.NET version is more convenient, but you lose the ability to set the transactional boundaries.

You can declare an automatic transaction in web services by using the `TransactionOption` property of the `WebMethodAttribute` class. For example, setting the `TransactionOption` property to `TransactionOption.RequiresNew` begins a new transaction each time an XML web service client calls the web service method. J2EE doesn't provide built in support for web service deployment as easily as .NET. If you wanted to deploy an EJB as a web service, the current convention is to create a servlet as the web service endpoint, which in turn invokes methods on the EJB. In this model, the EJB transactional attributes will be enforced for web service calls.

In order for instances of a .NET Framework class to participate in automatic transactions, the developer must prepare the class. Each resource accessed by a class instance enlists in the transaction. The following process prepares a class to participate in automatic transactions:

- Apply the `TransactionAttribute` to your class
- Derive your class from the `ServiceComponentClass`, which enables it to run inside COM+
- Sign the assembly with a strong name
- To sign the assembly using attributes, create a key pair using `Sn.exe`:  
`sn -k TestApp.snk`
- Add the `AssemblyKeyFileAttribute` or `AssemblyKeyNameAttribute` specifying the name of the file containing the key pair to sign the assembly with a strong name.  
[assembly: `AssemblyKeyFileAttribute("TestApp.snk")`]
- Register the assembly that contains your class with the COM+ catalog.

This process seems a bit unweildy compared to the deployment descriptor paradigm of J2EE. Also, COM+ doesn't have concepts similar to different EJB types (session, entity, message), so you don't benefit from the specializations offered by the J2EE framework.

---

## Distributed Transactions: .NET vs J2EE

Shannon Hardt

```
// ASP.NET declaration
<%@ Page Transaction="Required" %>

// ASP.NET Web Service
<%@ WebService Language="VB" Class="Class1" %>
<%@ assembly name="System.EnterpriseServices" %>
Public Class Class1 Inherits WebService
<WebMethod(TransactionOption := TransactionOption.RequiresNew)> _
Public Function Method1()
...

// .NET Framework class in C#
using System;
using System.Runtime.CompilerServices;
using System.EnterpriseServices;
using System.Reflection;

[assembly: ApplicationName("Class1")]
[assembly: AssemblyKeyFileAttribute("class1.snk")]
[assembly: ApplicationActivation(ActivationOption.Server)]

[Transaction(TransactionOption.Required)]
public class Class1 : ServicedComponent {
    [AutoComplete]
    public void Example1() {
        // work done here
    }
}
```

Example .NET distributed transactions

### Determining Success of Transactions

In .NET, classes and ASP.NET pages can vote to commit or abort their current transaction. The absence of an explicit vote in your code casts a commit vote by default. However, this may decrease performance by lengthening the time it takes for each transaction to release expensive resources. Explicit voting also allows your class or page to abort a transaction if it encounters a significant error.

The `System.EnterpriseServices.AutoCompleteAttribute` causes an object participating in a transaction to vote in favor of completing the transaction if the method returns normally. If the method call throws an exception, the transaction is aborted. You can apply this attribute only to classes deriving from the `ServicedComponent` class. This is similar to the behavior of stateless session beans in J2EE.

You can use the `System.EnterpriseServices.ContextUtil` class, which exposes the `SetComplete` and `SetAbort` methods, to explicitly commit or abort a transaction. `SetComplete` indicates that your object votes to commit its work. `SetAbort` indicates that your object encountered a problem and votes to abort the transaction. A transaction is neither committed nor aborted until the root object of the transaction deactivates. Further, a single abort vote from any object participating in the transaction causes the entire transaction to fail. These are syntactically similar to the commit and abort methods of `UserTransaction` in J2EE.

## Concluding Remarks

Distributed transactions are an important tool for enterprise development. J2EE and .NET both include facilities for distributed transactions which are, a bit surprisingly, very similar at the lower levels. However, there are distinct differences in how to use transactions while building applications. J2EE favors a declarative model separate from the code (XML deployment descriptor) while .NET uses Attributes in code to specify behavior of automatic transactions. I prefer the declarative model, which separates orthogonal concerns and, in my opinion, makes it easier to manage transactions spanning multiple components. Furthermore, while the deployment descriptor is a bit complicated, it allows the description of deploy-time concerns in a centralized place. Furthermore, changes to transaction semantics do not require recompilation of code, as is the case in .NET. The most redeeming quality of .NET transactions, in my opinion, is that nested manual transactions are supported, while this is absent in J2EE.

---

## References

*.NET Development for Java Programmers* by Gibbons. Apress, 2002.

*Professional Java Server Programming* by Allamaraju et. al. Wrox, 2000.

*Developing Java Enterprise Applications* by Asbury & Weiner. Wiley, 1999.

*Mastering Enterprise JavaBeans* by Roman. Wiley, 1999.

*J2EE Transaction Frameworks: Distributed Transaction Primer* by Baksi. OnJava.com, 2001,  
<http://www.onjava.com/pub/a/onjava/2001/05/23/j2ee.html>.

*Java Enterprise in a Nutshell* by Flanagan, Farley, Crawford, Magnusson. O'Reilly, 1999.

*Designing Enterprise Applications with the J2EE Platform, 2<sup>nd</sup> Edition*  
[http://java.sun.com/blueprints/guidelines/designing\\_enterprise\\_applications\\_2e/transactions/transactions.html](http://java.sun.com/blueprints/guidelines/designing_enterprise_applications_2e/transactions/transactions.html)

*MSDN Online: .NET Development*  
<http://msdn.microsoft.com/library/en-us/dnanchor/html/netdevanchor.asp>

*Programming C# 3<sup>rd</sup> Edition* by Liberty. O'Reilly, 2003.

---