

Einer für alle, alle für einen

– verteilte Transaktionen

Dieser Teil der Serie untersucht die Enterprise Services des .NET Framework und deren Möglichkeiten, mithilfe des Distributed Transaction Coordinator (DTC) verteilte Transaktionen zu programmieren. Dabei können nicht nur verschiedene DBMS an einer Transaktion teilnehmen, sondern auch andere Datenverarbeitungssysteme wie der MS Message Queue Server.

Der erste Artikel endete mit einem Beispiel, das eine verteilte Transaktion mithilfe des SQL Server demonstrierte. Die Programmierung in TSQL, dem SQL-Dialekt des SQL Server, ist nicht für alle Aufgaben geeignet. Doch es gibt Alternativen: Wer mit einer .NET Sprache entwickeln möchte, kann mit ADO.NET Transaktionen programmieren.

Transaktionen mit ADO.NET

ADO.NET verfügt über verschiedene Connection-Objekte, eines für den SQL Server und eines für eine OLE-DB-Datenquelle. Inzwischen gibt es neue spezialisierte Connection-Objekte für Oracle und ODBC und es werden sicher weitere folgen. Die Beispiele dieses Artikels verwenden die Objekte aus dem *SqlClient*-Namensraum.

Ein *SqlConnection*-Objekt verfügt über die Methode *BeginTransaction*, die eine neue Transaktion auf dem verbundenen Datenbankserver startet. Der Methode kann ein Isolationslevel übergeben werden [1]. Standardmäßig wird *ReadCommitted* verwendet und die Methode liefert ein Objekt vom Typ *SQLTransaction* zurück. Soll ein *SqlCommand*-Objekt innerhalb der Transaktion ausgeführt werden, wird der *Transaction*-Eigenschaft des *SqlCommand*-Objektes das *SQLTransaction*-Objekt übergeben. *Commit* oder *Rollback* werden auch mithilfe des *SQLTransaction*-Objektes durchgeführt (siehe Listing 1).

Listing 1 Eine manuelle ADO.NET-Transaktion.

```
//manuelle ADO Transaktion
public string DoWork(){
    SqlConnection cnScotty;
    SqlCommand cmScotty;
    SqlTransaction Txn;
    string query = "Update authors set state = 'IN' where state = 'MI'";

    cnScotty = new SqlConnection("data source = kirk; initial catalog =
pubs; UID=sa; PWD=");
    cmScotty = new SqlCommand(query,cnScotty);
    cnScotty.Open();
    Txn = cnScotty.BeginTransaction(System.Data.IsolationLevel
.ReadCommitted);

    try{
        cmScotty.Transaction = Txn;
        cmScotty.ExecuteNonQuery();

        Txn.Commit();

        return "Ready";
    } catch (Exception e){
        Txn.Rollback();
        return "Error" + e.Message ;
    }
}
```

Die Anweisungen zur Steuerung der Transaktion bedeuten dabei zusätzliche Roundtrips zum Datenbankserver (siehe Abbildung 1). Als Alternative können Sie eine Stored Procedure schreiben, welche die gleiche Aufgabe erledigt und die Transaktion direkt steuert.

Der Aufruf dieser Stored Procedure auf dem Server bedeutet nur einen Roundtrip zum Server, der Code aus Listing 1 führt drei Roundtrips durch.

Mit reinen ADO.NET-Mitteln ist es möglich, verteilte Transaktionen zu schreiben, die Änderungen auf zwei Datenbankservern innerhalb einer Transaktion durchführen. Ein ADO.NET-Transaction-Objekt ist immer an eine Connection gebunden und kann nur mit einem Ressourcen-Manager zusammenarbeiten. Sie können über eine Connection auch SQL-Befehle senden, die sich auf einen Linked Server beziehen, wie im ersten Teil der Artikelserie beschrieben. Diese Aufgaben lassen sich aber in einer Stored Procedure besser lösen.

Als Alternative besteht die Möglichkeit, direkt mit den

SUMMARY

Auf einen Blick

Diese Artikelserie gibt eine umfassende praktische Einführung in die transaktionale Datenverarbeitung. In diesem Teil stehen die .NET Enterprise Services und der Distributed Transaction Coordinator (DTC) im Vordergrund.

Eingesetzte Anwendungen

.NET Framework, SQL Server, Win2000 oder WinXP

CD-Code

DrillDown01

Serie

Teil 2/3

Autor

Marcel Gnoth ist Senior Consultant bei der Berliner NTeam GmbH. Seine Arbeitsschwerpunkte liegen im Bereich COM, .NET und verteilte Informationssysteme, zu denen er auch Trainings durchführt. Sie erreichen ihn unter marcel@gnoth.net.

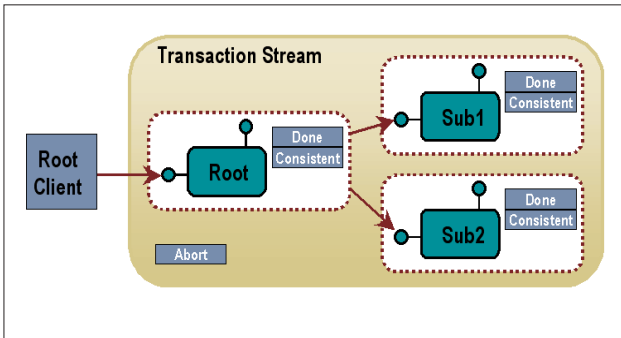


Abbildung 1 | Eine ADO.NET-Transaktion.

ODBC- oder OLE-DB-APIs zu programmieren. Diese Low-Level-Programmierung dürfte nicht der beste Weg sein und nur in besonderen Situationen in Frage kommen.

COM+ – Ein kurzer Überblick

Mit welchen Mitteln können verteilte Transaktionen programmiert werden? Die COM+-Infrastruktur von Windows bietet Komponenten, die an Transaktionen teilhaben können.

Unter dem Begriff COM+ sind eine Reihe von Systemdiensten zusammengefasst, die helfen, für die Microsoft-Plattform verteilte, komponentenbasierte Systeme zu entwickeln. Die erste Version erschien mit dem Option-Pack für NT4, seit Windows 2000 heißen diese Dienste COM+.

COM+ bietet nicht nur die Möglichkeit, Komponenten an Transaktionen teilhaben zu lassen, sondern stellt zudem die Infrastruktur für skalierbare Applikationen durch Just-In-Time-Aktivierung (JIT) oder Objektpooling bereit. Berechtigungen für die einzelnen Komponenten können deklarativ durch einen Administrator vergeben werden. Es ist möglich, Compensating Resource Manager für Ressourcen zu schreiben, die keinen eigenen Ressourcen-Manager zur Verfügung stellen, wie zum Beispiel das Dateisystem von Windows.

Code, der die Enterprise Services verwendet, wird durch die *Common Language Runtime* ausgeführt, die wiederum mit der COM+-Infrastruktur eng zusammenarbeitet. COM+ verwendet dann das COM- und Windows-API. Weitere Informationen finden Sie unter [2] und [3].

Konfiguration einer COM+-Komponente

Die Enterprise Services sind das .NET-API für COM+-Anwendungen. Eine COM+-Komponente besteht aus mindestens einer Klasse, die zu einer DLL kompiliert wird. Diese Klasse muss von *ServicedComponent* erben.

Anschließend muss diese Komponente im System registriert werden (siehe weiter unten). Die Konfiguration kann über die COM+-Verwaltung manuell erfolgen oder über Attribute im Quellcode vorgenommen werden.

ApplicationActivation-Attribut

Eine COM+-Komponente ist üblicherweise als In-Process-Komponente (Windows-DLL) realisiert. Sie haben zwei Möglichkeiten, diese Komponente auszuführen (siehe Abbildung 2). Die DLL kann im Prozessraum des Clients als Bibliotheksanwendung

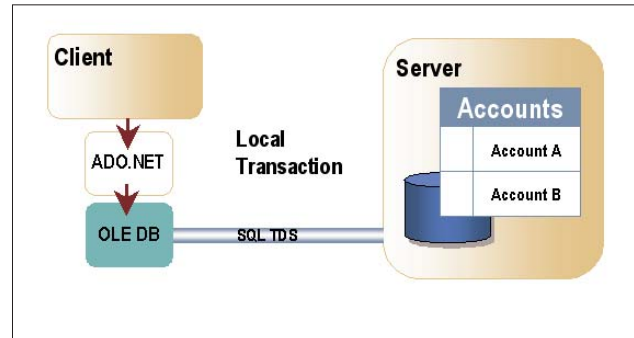


Abbildung 2 | Applikationstypen für eine COM+-Anwendung.

laufen oder sie kann als Serveranwendung in einem eigenen Mutterprozess, *DllHost.exe*, ausgeführt werden. Das *ApplicationActivation*-Attribut können Sie in der Datei *AssemblyInfo.cs* setzen.

```
[assembly: ApplicationActivation(ActivationOption.Library)]
```

Wird die Komponente im Prozessraum des Clients ausgeführt, ist die Kommunikation zwischen Client und Komponente schneller, da keine Prozessgrenzen überwunden werden müssen. Allerdings sind nicht alle Dienste von COM+ im Prozessraum des Clients verfügbar. Unter Windows XP können COM+-Komponenten auch als eigener Windows-Dienst laufen.

Transaction-Attribut

Auf welche Art eine Komponente an einer Transaktion beteiligt sein kann, wird über das *Transaction*-Attribut konfiguriert. Diese Einstellung entscheidet darüber, in welchem transaktionalen Kontext eine Komponente läuft. Benötigt sie einen eigenen, kann sie im Kontext einer vorhandenen Transaktion laufen oder sie unterstützt gar keine Transaktionen (siehe Tabelle 1).

```
[Transaction(TransactionOption.Required)]
Class MyTxClass : ServicedComponent
{...}
```

Arbeitet eine Gruppe von COM+-Komponenten innerhalb einer Transaktion, wird von einem *Transaction Stream* gesprochen (siehe Abbildung 3). Ein Client instanziiert die Root-Komponente, deren Attribut auf *Required* oder *Required New* eingestellt sein muss. Diese Komponente instanziiert wiederum zwei andere Komponenten, deren Attribute als *Required* oder *Supported* konfiguriert sein müssen. Alle drei Komponenten nehmen am gleichen *Transaction Stream* teil. Öffnet eine der Komponenten eine Connection zu einer Datenbank, so wird diese automatisch mit in den *Transaction Stream* aufgenommen. Werden Connections zu

Tabelle 1 | Konfiguration des Transaction-Attributes.

Disabled	Transaction-Attribut wird von COM+ ignoriert.
Not Supported	Komponente wird in einem Kontext ohne Transaktion aktiviert.
Supported	Komponente beteiligt sich an Transaktion, wenn vorhanden.
Required	Komponente muss in einem transaktionalen Kontext laufen.
Requires New	Für die Komponente wird immer ein neuer transaktionaler Kontext angelegt.

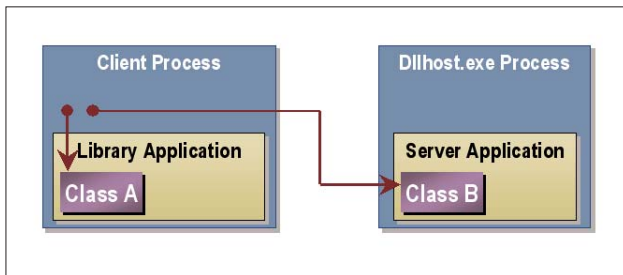


Abbildung 3 | Zusammenarbeit von drei COM+-Komponenten innerhalb eines Transaction Streams.

verschiedenen Servern geöffnet, so nehmen alle an der gleichen Transaktion teil – es handelt sich um eine verteilte Transaktion.

Das Ergebnis einer Transaktion

Jedes Objekt, das an einer Transaktion beteiligt ist, verfügt über zwei Flags, die den Zustand innerhalb der Transaktion anzeigen. Das *Consistent* oder auch *Happy* genannte Flag zeigt an, ob alle Aufgaben des Objektes erfolgreich durchgeführt wurden, ob der Zustand des Objektes konsistent ist. Bei allen an der Transaktion beteiligten Objekten muss das *Consistent*-Flag gesetzt sein, damit die Transaktion erfolgreich abgeschlossen werden kann. Ist nur eines der Flags nicht gesetzt, wird für die Transaktion ein Roll-back durchgeführt. Für den gesamten *Transaction Stream* gibt es ein *Abort*-Flag. Wird es gesetzt, kann die Transaktion nicht mehr erfolgreich beendet werden, unabhängig von den Werten der *Consistent*-Flags. Ist das *Abort*-Flag einmal gesetzt, kann es nicht mehr zurückgesetzt werden. Die Auswertung der Flags erfolgt, wenn der Aufruf vom *Root*-Objekt zum Client zurückkehrt.

Das *Done*-Flag hat nichts mit dem Ausgang der Transaktion zu tun. COM+ verfügt über Pooling-Mechanismen, die einen effizienten Einsatz von Systemressourcen ermöglichen. Wenn in Abbildung 3 das Objekt *Sub1* seine Aufgabe erledigt hat und das *Consistent*-Flag gesetzt hat, könnten die Ressourcen dieses Objektes freigegeben werden, während auf die anderen an der Transaktion beteiligten Objekte gewartet wird. Am Ende des Methodenaufrufs wird das *Done*-Flag geprüft und das Objekt wird gegebenenfalls deaktiviert. Weitere Informationen darüber finden Sie in der Dokumentation unter *Just-In-Time-Aktivierung* und *Objektpooling*. Ist ein Objekt einmal deaktiviert, kann es innerhalb der gleichen Transaktion nicht noch einmal verwendet werden. Der Zustand des Objektes geht verloren, aber die Flags bleiben bis zum Ende der Transaktion erhalten.

Steuerung der Transaktion durch den Client

Soll eine Client-Anwendung die Transaktion steuern, kann die COM+-Typbibliothek verwendet werden. Der Client kann ein Objekt vom Typ *TransactionContext* instanzieren und mithilfe dieses Objektes die Transaktion steuern (siehe Listing 2). COM+-Komponenten, die an der Transaktion teilnehmen sollen, müssen mit der *CreateInstance*-Methode erstellt werden. Bei dieser Variante muss ein Client um Code zur Transaktionssteuerung ergänzt werden. Schöner wäre es, wenn die COM+-Komponente für die Transaktionssteuerung verantwortlich wäre und der Client eine Funktion der Komponente aufruft. Der Entwickler des Clients bräuchte sich

Listing 2 Eine Client-gesteuerte Transaktion.

```

ITransactionContext tx;
tx = (ITransactionContext) new TransactionContext( );
MyTXClass txC11=(MyTXClass) tx.CreateInstance("ProgID1");
MyTXClass txC12=(MyTXClass) tx.CreateInstance("ProgID2");
txC11.DoWork( );
txC12.DoWork( );
tx.Commit( );
    
```

dann nicht mit der Transaktionsverarbeitung zu beschäftigen und könnte sich voll auf die GUI-Programmierung konzentrieren.

Steuerung der Transaktion durch die Komponente

Die Klasse *ContextUtil* aus dem *EnterpriseServices*-Namensraum bietet statische Methoden zur Steuerung einer Transaktion an (siehe Tabelle 2). Über die Eigenschaft *IsInTransaction* kann geprüft werden, ob der Code innerhalb einer Transaktion ausgeführt wird. Über die Eigenschaften *DeactivateOnReturn* und *MyTransactionVote* können das *Done*- und das *Consistent*-Flag individuell gesetzt und gelesen werden (siehe Listing 3).

Eine Quick-and-Dirty-Variante zur Transaktionssteuerung bietet das Attribut *AutoComplete*, das für eine Methode gesetzt werden kann. Läuft die Methode fehlerfrei, wird automatisch *SetComplete* aktiviert. Tritt eine Ausnahme auf, wird diese an den Aufrufer gereicht und *SetAbort* aufgerufen. Der Client sieht dann die „rohe“ Fehlermeldung. Bei einfachen und sehr kurzen Methoden mag das akzeptabel sein, aber der Einsatz dieses Attributes macht den Code nicht lesbarer. Eine explizite Steuerung des Transaktionsverlaufes im Code der Prozedur hilft, die Semantik besser zu erkennen.

```

[AutoComplete(true)]
public string TestAutoComplete(int i){
    i = i * 3;
    return "Ready";
}
    
```

Isolation von COM+-Transaktionen

Im ersten Teil des Artikels wurden die Isolationsstufen des SQL Server beschrieben. COM+ 1.0 unter Windows 2000 unterstützt nur die höchste Isolationsstufe *Serialized*. Erst mit COM+ 1.5 unter Windows XP sind auch die anderen Stufen möglich. Je höher die Isolation ist, umso stärker behindern sich die Transak-

Listing 3 Steuerung der Transaktion.

```

public void DoWork1() {
    try {
        //Tu Du something
        ContextUtil.SetComplete();
    }
    Catch (Exception e) {ContextUtil.SetAbort();}
}

public void DoWork2( ) {
    ContextUtil.DeactivateOnReturn = true;
    ContextUtil.MyTransactionVote = TransactionVote.Abort;
    ... // use ADO.NET to work with databases
    ContextUtil.MyTransactionVote=TransactionVote.Commit;
}
    
```

Tabelle 2 Methoden der Klasse ContextUtil für die Steuerung einer Transaktion.

Methoden	Done-Flag	Consistent-Flag
EnableCommit	false	true
DisableCommit	false	false
SetComplete	true	true
SetAbort	true	false
DeactivateOnReturn	true / false	-
MyTransactionVote	-	true / false

tionen gegenseitig: Arbeiten viele Anwender mit Ihrem Programm, kann die Performance stark nachlassen, obwohl die meisten Anwender überwiegend lesend auf die Daten zugreifen. Der hohe Isolationsgrad wäre nicht notwendig.

Muss die Anwendung auf einem Windows 2000 Server laufen und arbeiten Sie nur mit Datenbankservern, sollten Sie darüber nachdenken, die Transaktionsverarbeitung in Stored Procedures auszulagern, um den Durchsatz Ihrer Anwendung zu erhöhen. In TSQL können Sie alle Isolationsstufen verwenden. Im Idealfall ist ein Umstieg auf Windows Server 2003 möglich [4]. Aber Achtung, nicht alle Ressourcen-Manager unterstützen alle Isolationsstufen.

```
[Transaction(TransactionOption.RequiresNew,
    Isolation = TransactionIsolationLevel.ReadCommitted,
    Timeout = 180)]
public class cEnterprise : ServicedComponent {
    ...
}
```

Schluss mit der Theorie, ab in die Praxis

Um eine COM+-Komponente zu entwickeln, legen Sie ein neues Projekt von *ClassLibrary* in Visual Studio an. Sie müssen eine Referenz auf den *System.EnterpriseServices*-Namensraum setzen, die Klasse muss von *ServicedComponent* erben und Sie setzen das *Transaction*-Attribut für die Klasse (siehe Listing 4).

Eine Assembly, die COM+-Komponenten enthält, braucht einen *StrongName*. Mit dem Werkzeug *sn.exe*, das zum .NET Framework gehört, kann eine Datei erzeugt werden, die Informationen für einen *StrongName* enthält. Diese Datei kopieren Sie in das Projektverzeichnis. Beim nächsten Kompilieren wird für die Assembly ein *StrongName* erzeugt.

Jetzt müssen in der Datei *AssemblyInfo.cs* noch einige Attribute gesetzt werden, die später in den COM+-Katalog eingetragen werden. *AssemblyKeyFile* verweist auf die erzeugte Schlüsseldatei für den *StrongName* und *ApplicationName* setzt den Namen der Komponente im COM+-Katalog. Jetzt können Sie die Anwendung kompilieren.

```
[assembly: ApplicationActivation(ActivationOption.Library)]
[assembly: ApplicationName("TXComponentLocal")]
[assembly: AssemblyKeyFile("..\..\TXComponentLocal.snk")]
```

Registrieren der Anwendung im COM+-Katalog

Die Komponente muss als COM+-Anwendung registriert werden. Diese Informationen werden nicht in der Registry eingetragen sondern im COM+-Katalog. Dafür gibt es das Werkzeug *regsvcs.exe*, dem Sie als Parameter die kompilierte Komponente übergeben. Dieses Werkzeug untersucht die Assembly mithilfe

Listing 4 Eine COM+-Komponente.

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

namespace TXComponentLocal {
    [Transaction(TransactionOption.RequiresNew)]
    public class SQLKirk : System.EnterpriseServices.ServicedComponent {
        public SQLKirk() { } //Konstruktor

        public string DoTrans(){
            SqlConnection cnKirk;
            SqlCommand cmAuthors;
            string query;

            try{
                query = "Update authors set state = 'IN' where state = 'MI'";
                cnKirk = new SqlConnection("data source = kirk; initial catalog = pubs; UID=sa; PWD=");
                cmAuthors = new SqlCommand(query,cnKirk);
                cnKirk.Open(); //Connection nimmt automatisch an Tx teil
                cmAuthors.ExecuteNonQuery();
                ContextUtil.SetComplete();
            }
            catch (Exception exc){
                ContextUtil.SetAbort();
                throw exc;
            }
            return "ready";
        }
    }
}
```

des *Reflection*-API, findet die Attribute, die Sie im Code gesetzt haben, und trägt diese Informationen in den Katalog ein. Jetzt können Sie Ihre Komponente testen.

Die Konfiguration sehen Sie in der Verwaltung für *Komponentendienste* (*Startmenü/Programme/Verwaltung/Komponentendienste*). Es handelt sich um ein SnapIn für die MS Management Console (MMC). Sie können sich mit diesem SnapIn auch zu einem anderen Computer verbinden und dessen Konfiguration bearbeiten.

Unter dem Knoten *COM+-Anwendungen* finden Sie alle registrierten Anwendungen (siehe Abbildung 4). Weiter unten finden Sie den Knoten *Distributed Transaction Coordinator*, dort können sie eine Liste der laufenden Transaktionen sehen sowie eine Statistik, wie viele Transaktionen erfolgreich oder nicht erfolgreich durchgeführt wurden. Auch wenn Sie, wie im ersten Teil des Artikels beschrieben, über den SQL Server eine verteilte Transaktion durchführen, finden Sie unter dem DTC-Knoten Informationen über den Ausgang einer Transaktion.

Der Distributed Transaction Coordinator (DTC)

Der DTC ist ein Transaction-Monitor, der die Aufgabe hat, die Transaktionen der einzelnen Ressourcen-Manager zu einer logischen Transaktion zusammenzufassen und die einzelnen Commit- und Rollback-Anweisungen zu koordinieren (siehe Abbildung 5). Werden von den Komponenten Verbindungen zu einer Datenbank geöffnet, werden diese Connections automatisch zu der laufenden Transaktion hinzugefügt.

Der DTC wurde das erste Mal mit dem SQL Server 6.5 ausgeliefert und gehört inzwischen zum Windows-Betriebssystem. Er

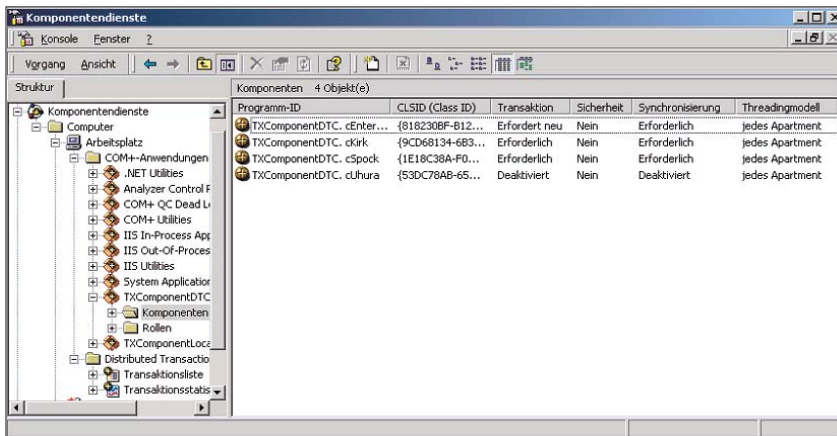


Abbildung 4 | Verwaltung für Komponentendienste.

ist eine Anwendung, die als Systemdienst läuft und auf jedem Computer, der an der Transaktion beteiligt ist, gestartet sein muss. COM+-Transaktionen kapseln die Funktionalität des DTC.

Die Steuerung der logischen Transaktion erfolgt über das *Zwei-Phasen-Commit*-Protokoll, das im dritten Teil dieser Serie näher beschrieben wird.

MS Message Queuing als Ressourcen-Manager

Es muss nicht immer ein Datenbankserver sein, der als Ressourcen-Manager an einer Transaktion teilnimmt. Von Microsoft gibt es den Message-Queue-Dienst (MSMQ), der den asynchronen Transport von Nachrichten zwischen Computern erlaubt, eine Art E-Mail für Computerprogramme ([5] und [6]). Diese Nachrichten können auch innerhalb einer Transaktion verschickt werden. Der MSMQ ist Bestandteil des Windows-Betriebssystems und wird standardmäßig nicht mitinstalliert. Sie müssen das Windows-Setup starten und den MSMQ nachträglich installieren.

Damit Nachrichten transaktional verschickt werden können, muss die Warteschlange transaktional erstellt werden. Der *Send-*

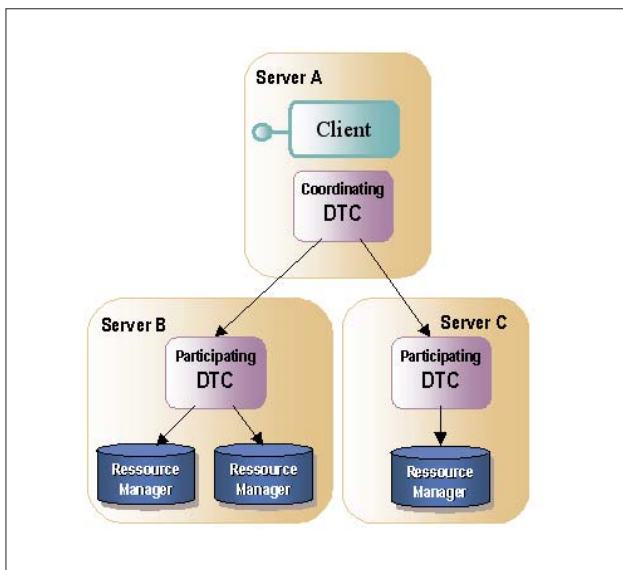


Abbildung 5 | Der DTC koordiniert die Transaktionen der beteiligten Ressourcen-Manager.

Methode muss ein entsprechender Parameter übergeben werden, damit die Nachricht innerhalb einer Transaktion versendet wird.

Komponente mit verteilten Transaktionen

Auf der Heft-CD finden Sie zwei Beispielprojekte: einen Client und ein Projekt mit einer COM+-Komponente. Die Komponente besteht aus vier Klassen. Die Klasse *cEnterprise* ist die Root-Komponente, welche die Transaktion steuert. Sie wird vom Client aufgerufen. Die Klassen *cKirk* und *cSpock* führen Datenoperationen in der *Pubs-* und der *Nordwind-*Datenbank auf zwei verschiedenen Servern durch. Sie müssen nur noch die Verbindungszeichenfolge anpassen. Auf welcher Datenbank Sie Änderungen durchführen, ist dabei für die Funktionsweise der Komponente irrelevant, Sie können also auch eigene Datenbanken und SQL-Anweisungen verwenden. Die vierte Klasse *cUhura* versendet eine Nachricht über MSMQ. Die Transaktions-Attribute der Klasse sind auskommentiert, da im Code die Nachricht transaktional versandt wird. Denken Sie daran, dass die Komponenten erst im COM+-Katalog registriert werden müssen, bevor Sie die Anwendung starten können. Setzen Sie einen Haltepunkt in der Klasse *cEnterprise* und beobachten Sie das Verhalten in der Verwaltung für Komponentendienste.

Ausblick

Sie haben gesehen, dass es nicht schwer ist, eine verteilte Transaktion mit COM+ zu programmieren. Einzig die Zusammenarbeit des DTC mit den Ressourcen-Managern kann Schwierigkeiten verursachen. Kommen diese aus dem Hause Microsoft, treten erwartungsgemäß kaum Probleme auf. Was aber, wenn ein Oracle Server einbezogen werden soll oder DB2? Oder für eine Ressource wie das Dateisystem existiert gar kein Ressourcen-Manager? Diese Probleme untersucht der nächste Teil dieser Serie. Außerdem wird das *Zwei-Phasen-Commit*-Protokoll näher beleuchtet.

|||||

- [1] Marcel Gnoth, Einer für alle, alle für einen – verteilte Transaktionen, dotnetpro 5/2003, Seite 86 ff.
- [2] COM+ Integration: How .NET Enterprise Services Can Help You Build Distributed Applications, <http://msdn.microsoft.com/msdnmag/issues/01/10/complus/default.aspx>
- [3] Microsoft Official Curriculum, Course 2557A: Building COM+ Applications Using Microsoft .NET Enterprise Services
- [4] Windows XP: Make your Components more robust with COM+ 1.5 Innovations, <http://msdn.microsoft.com/msdnmag/issues/01/08/ComXP/default.aspx>
- [5] Marcel Gnoth, Nachrichtenbasierte Informationssysteme, BasicPro 5/2001 Seite 26 ff.
- [6] Reliable Messaging with MSMQ and .NET, <http://msdn.microsoft.com/library/en-us/dnbd/html/bdadotnetasync2.asp>