**Anko Duizer**
is werkzaam als trainer/coach bij Class-A (www.class-a.nl). Daarvoor heeft hij vijf jaar gewerkt bij Microsoft als consultant. Onder zijn klantenkring bevinden zich voornamelijk Top100-bedrijven in Nederland. Sinds begin 2001 is hij bezig met .NET. Zijn speciale interesse gaat uit naar de architectuur en het ontwerp van een gedistribueerde applicatie. Hij is bereikbaar via anko.duizer@class-a.nl

# Transactions in .NET

## De juiste transactietechnologie in een gedistribueerde .NET-applicatie

**Wanneer pas je welke transactietechnologie toe bij de ontwikkeling van een gedistribueerde .NET-applicatie?. Er zijn binnen .NET diverse mogelijkheden voor het toepassen van transacties. Voordat de mogelijkheden de revue passeren, wordt eerst ingegaan op wat een transactie eigenlijk is, en waarom transacties belangrijk zijn. Aan het eind komt de toepassing van transactietechnologie in een gedistribueerde applicatie aan de orde, waarbij de 'Enterprise Services' als transactiemechanisme worden gebruikt.**

Van het begrip transactie zijn vele definities in omloop. Voor dit artikel wordt de volgende gehanteerd: 'Een transactie is het verzoek van een client om een eenheid van werk uit te voeren, die op zich kan bestaan uit meerdere losse onderdelen. Op een transactie zijn de ACID-properties (Atomicity, Consistency, Isolation & Durability) van toepassing.' Een belangrijk onderdeel van de gehanteerde definitie zijn de zogenaamde 'ACID-properties'. Deze worden algemeen erkend en zorgen voor een duidelijke definitie waar een transactie aan moet voldoen. De betekenis van de ACID properties is beschreven in de tabel 1.

Een belangrijk onderdeel van de ACID properties is Isolation, wat meestal wordt geïmplementeerd door middel van 'locking'. Dit is een belangrijke reden waarom transacties trager zijn dan wanneer een vergelijkbare actie uitgevoerd wordt zonder een transactie. Locking zorgt er in principe voor dat, wanneer iemand bepaalde data wijzigt, anderen deze data niet gelijktijdig kunnen wijzigen. Het is zelfs mogelijk dat anderen de data niet mogen lezen. Dit betekent dat er veel administratie noodzakelijk is om de locks te registreren. Een groter probleem is het feit dat transacties op elkaar gaan staan wachten omdat ze precies dezelfde lock willen hebben, wat wachtrijen oplevert. Dit wordt 'blocking' genoemd.

Er zijn diverse Isolation levels. Afhankelijk van het level wordt er 'meer' gelocked. Dit levert dus potentieel meer blocking-situaties op. Hoe hoger het Isolation level des te slechter dit is voor de performance en schaalbaarheid van de applicatie. Toch is een hoger level soms noodzakelijk omdat een aantal potentiële foutsituaties wordt voorkomen door de keuze voor een hoger niveau van Isolation. De mogelijke transactie-levels en de probleemsituaties die ze voorkomen zijn weergegeven in tabel 2.

| ACID property | Beschrijving |
|---|---|
| Atomicity | Zorgt ervoor dat de eenheid van werk als geheel succesvol is, of dat het geheel wordt teruggedraaid naar de status van voor de transactie. Zorgt uiteindelijk voor een eenvoudig programmeermodel, het resultaat is als geheel goed of fout. |
| Consistency | Draagt zorg voor het feit dat de data na de transactie consistent is. Dus de wijziging die door de transactie wordt bewerkstelligd moet een consistent eindresultaat opleveren. Tijdens de transactie kan de resource dus tijdelijk inconsistent zijn. |
| Isolation | Zorgt ervoor dat de wereld binnen de transactie eruit ziet alsof de transactie alleen op de resources werkt. Tijdens de transactie worden andere acties tijdelijk tegengehouden. Veelal gebeurt dit via locking. |
| Durability | Zorgt ervoor dat wanneer een transactie is afgerond de status is weggeschreven op disk zodat het systeem crashes overleeft. |

Tabel 1. ACID properties

## Waarom zijn transacties belangrijk?

Op basis van het voorgaande is het duidelijk dat transacties slecht zijn voor de performance van een systeem. Waarom zijn ze dan toch belangrijk? Het doel van transacties is het correct opslaan van gegevens. Een eenheid van werk wordt als geheel wel of niet opgeslagen. Dit maakt het programmeer modelaanzienlijk eenvoudiger. Belangrijker is het feit dat sommige gegevens cruciaal zijn voor de bedrijfsvoering, van deze gegevens is het dus belangrijk om zeker te weten dat ze correct worden opgeslagen.

In het dagelijks leven komen we regelmatig transacties tegen. Bijvoorbeeld wanneer we geld pinnen, of wanneer we via

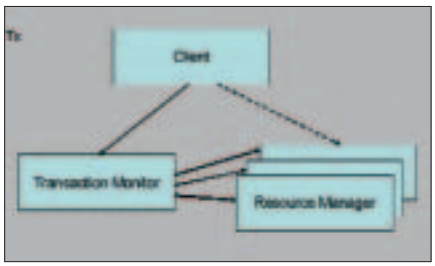
| | 0 | 1<br>Read uncommited | 2<br>Read commited | 2.99<br>Repeatable read | 3<br>Serialization |
|---|---|---|---|---|---|
| Lost update | | X | X | X | X |
| Dirty read | | | X | X | X |
| Non repeatable read | | | | X | X |
| Phantom | | | | | X |

Tabel 2. Isolation levels

Internet een ticket boeken. Voor databaseleveranciers en andere software bedrijven levert het maken van producten die goed overweg kunnen met transacties veel omzet op. Het is een van de grootste markten qua omzet in de softwarebranche. Daarnaast meten we de performance en schaalbaarheid van softwaresystemen aan de hand van het aantal transacties dat er per minuut kan worden verwerkt. Kortom transacties zijn een wezenlijk onderdeel van veel softwarearchitecturen.

## Transactie-architectuur

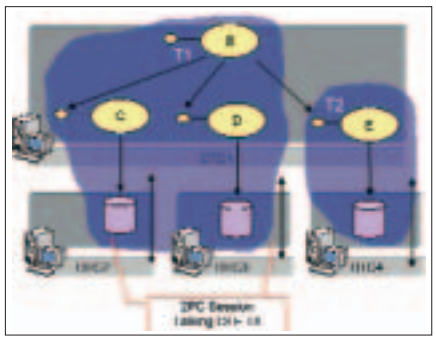
Wanneer wordt gesproken over transacties heeft men het vaak over databases. Natuurlijk zijn de meeste databases in staat om transacties af te handelen. Er zijn echter altijd drie partijen betrokken bij een transactie: 'Client', 'Transaction Monitor' (TM) en een 'Resource Manager' (RM). Dit is schematisch weergegeven in afbeelding 1.




Afbeelding 1. Transactie-architectuur

Wanneer gebruik wordt gemaakt van een transactie zijn de drie partijen in principe altijd vertegenwoordigd. Zelfs wanneer uitsluitend gebruik wordt gemaakt van SQL Server-transacties. In dit geval is SQL Server zowel de resource manager als de transaction monitor. Voorbeelden van een transaction monitor zijn Distributed Transaction Coordinator (DTC), CICS en Tuxedo. Voorbeelden van een resource manager zijn Microsoft SQL Server, Oracle, DB2, Microsoft Message Queuing (MSMQ) of een transactioneel filesysteem.

De transaction monitor bepaalt vaak de functionaliteit. Een belangrijk aspect dat de TM bepaalt is de mogelijkheid om een transactie gedistribueerd uit te voeren. Kortom, is het mogelijk gegevens transactioneel op te slaan over twee of meerdere fysieke resource managers? De DTC is in staat om inderdaad via een 'two-phase commit' protocol meerdere RM's in één transactie bij te werken. In afbeelding 2 is zichtbaar dat transactie 'T1' een tweetal resource managers beslaat.



Afbeelding 2. Gedistribueerde transactie

## Transactietechnologieën in .NET

Wanneer gebruik wordt gemaakt van .NET om een gedistribueerde applicatie te ontwikkelen, zijn er vier mogelijke transactiemechanismen die gebruikt kunnen worden. In beginsel zijn ze allemaal bedoelt om gegevens correct te verwerken en op disk te bewaren. Toch hebben ze eigen unieke eigenschappen. De volgende mogelijkheden zijn beschikbaar in .NET:

- transact-SQL-transacties;
- distributed Transact-SQL-transacties;
- ADO.NET-transacties;
- COM+-transacties (Enterprise Services).

De Transact-SQL-transacties (T-SQL) zijn reeds jaren aanwezig in Microsoft SQL Server. Deze kunnen nog steeds worden gebruikt in een .NET-applicatie, bijvoorbeeld in een stored procedure. Met expliciete statements in de code (of in een stored procedure) wordt een transactie gestart, bevestigd of teruggedraaid. Afbeelding 3 bevat voorbeeldcode die een eenvoudig T-SQL script weergeeft. Het voordeel van deze variant is het feit dat Microsoft SQL Server zowel de rol van resource managers als transaction manager op zich neemt. Dit levert belangrijke tijdswinst (performance) op.
De distributed T-SQL-variant lijkt qua syntax sterk op de pure T-SQL-syntax. Achterliggend wordt echter gebruik gemaakt van de DTC. Hierdoor ontstaat de mogelijkheid om gedistribueerd SQL-transacties uit te voeren. Omdat de DTC wordt gebruikt, wordt tijd 'verspeeld'. Deze variant is dus minder goed voor de performance van een applicatie.

```
BEGIN TRANSACTION

   INSERT INTO
   Order
   VALUES
   (1,10,getdate())

   IF @@ERROR > 0 GOTO ErrorHandler

   INSERT INTO
   OrderLine
   VALUES
   (1,12198,10,"red")

   IF @@ERROR > 0 GOTO ErrorHandler

   COMMIT TRANSACTION
   GOTO ExitHandler

ErrorHandler:
   ROLLBACK TRANSACTION

ExitHandler:
```

Afbeelding 3. Voorbeeldcode transactie in T-SQL

De ADO.NET-variant is de vervanger van ADO-transacties. Qua mechanisme komt het sterk overeen met de voorganger. Vanuit de .NET-code, bijvoorbeeld C#, wordt een transactie expliciet gekoppeld aan een connectie. De ontwikkelaar is vervolgens verantwoordelijk om expliciet een 'commit' dan wel een 'rollback' uit te voeren. Het voordeel van deze variant is het feit dat dezelfde code kan werken

tegen een andere database dan Microsoft SQL Server. Afbeelding 4 bevat een fragment C#-code waarin een ADO.NET-transactie wordt gebruikt.

De laatste mogelijkheid is de 'eigenlijke' .NET-variant. In deze vorm worden de .NET-mogelijkheden ten volle benut. Het gehele mechanisme is bijvoorbeeld gebaseerd op 'attributes'. Dit levert een fraai programmeermodel op. Achterliggend wordt gebruikgemaakt van COM+-transacties; en dit is direct de beperking. Op de server waarop de .NET class draait, moet COM+ aanwezig en geconfigureerd zijn. Voor iedere transactionele .NET class wordt op de achtergrond een 'dummy' COM+ class geregistreerd met de overeenkomstige transactie-eigenschappen. Deze vorm van transacties wordt ook wel 'enterprisetransacties' genoemd. COM+-transacties maken gebruik van de DTC als transaction monitor.

## Enterprisetransacties

Enterprisetransacties zijn een implementatie van COM+-transacties in het .NET framework. Op dit moment is alleen de API beschikbaar gemaakt in .NET. De werkelijke transactie wordt nog steeds in COM+ uitgevoerd. In de .NET-code is hiervan niets zichtbaar. Het is de bedoeling dat de transacties in de toekomst binnen de runtime gaan draaien zodat COM+ niet langer noodzakelijk is. Dit veranderd vermoedelijk niets aan het bestaande programmeermodel. Enterprisetransacties worden ook vaak 'declarative transactions' genoemd.

Het gebruik en de achterliggende logica van de enterprisetransacties is hetzelfde als van COM+-transacties. Op classniveau bepaalt de programmeur wat voor 'soort' transactie noodzakelijk is voor de desbetreffende bedrijfslogica. Vervolgens wordt de transactie impliciet gestart. De enterprisetransacties kunnen meerdere classes en databases beslaan, en zijn dus gedistribueerd. Vanuit de code kan alleen worden vermeld of dat deel van de code 'blij' of 'niet blij' is. Er kan worden 'gestemd' vanuit de code, de code stemt voor 'commit' of 'abort'. Het is dus niet mogelijk om een transactie expliciet te starten of te stoppen, dit moet worden overgelaten aan de TM. De TM verzamelt alle stemmen, en bepaalt uiteindelijk of de transactie als geheel wordt doorgevoerd (commit) of teruggedraarid (rollback). Dit betekent een eenvoudiger programmeermodel met minder kans op fouten, maar de controle over de transactie neemt sterk af. Wanneer gebruik wordt gemaakt van 'AutoComplete' dan is het niet eens meer noodzakelijk om te stemmen. In dit geval wordt eenvoudig weg een 'commit' uitgevoerd, mits er een 'exception' is opgetreden tijdens de uitvoer van de desbetreffende methode.

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

namespace TransactionADO
{

   public class Tx
   {
       public Tx()
       {

       }

       public void SomeTx()
       {

            SqlConnection _conn = new SqlConnection("connection string");
            SqlCommand _comm = new SqlCommand("sp_order",_conn);
            _conn.Open();

            SqlTransaction _transaction = _conn.BeginTransaction();
            _comm.Transaction = _transaction;

            try
            {
                 _comm.ExecuteNonQuery();
                 _transaction.Commit();
            }
            catch
            {
                 _transaction.Rollback();
            }
       }
   }
}
```

Afbeelding 4. Voorbeeldcode transactie in ADO.NET

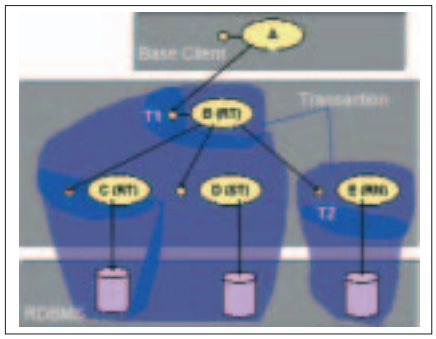
| Transactie setting | Omschrijving |
|---|---|
| Disabled (D) | Negeert iedere transactie binnen de huidige context |
| Not Supported (NS) | De class draait nooit in een transactie |
| Supported (S) | De class kan zowel binnen als buiten een transactie draaien. Afhankelijk van de aanroepende class zal de class binnen dan wel buiten een transactie draaien. Wanneer wenselijk kan er mee worden gestemd voor de uitkomst van de transactie. |
| Required (R) | De class draait altijd binnen een transactie. Wanneer reeds een transactie aanwezig is, dan wordt die gebruikt. Anders wordt een nieuwe transactie gestart. |
| Requires New (RN) | Is altijd een 'Transaction Root' omdat altijd een nieuwe transactie wordt gestart |

Tabel 3. Transactiesettings

Tabel 3 bevat vijf transactiesettings die via een attribuut aan een class kunnen worden toegekend.

Afbeelding 5 toont voorbeeldcode waarmee zichtbaar is hoe de transactieproperties kunnen worden gebruikt in C#-code.

Afhankelijk van de settings kan een andere transactieflow ontstaan. Wanneer een class het attribuut 'Supported' heeft gespecificeerd kan het zelfs zo zijn dat, afhankelijk van de aanroeper, de desbetreffende class de ene keer wel en de andere keer niet binnen een transactie draait. In afbeelding 6 is het een en ander schematisch weergegeven.




Afbeelding 6. Transactieflow

In afbeelding 6 neemt class 'B' een speciale plaats in. Deze class start transactie 'T1'. Deze class wordt de 'transaction root' genoemd. In het schema komt een tweede 'transaction root' voor, namelijk 'E'. In principe is de transactie aanwezig totdat de root uit de lucht gaat. Op dat moment worden de stemmen bekeken. Wanneer één negatieve stem is uitgebracht dan wordt de transactie teruggedraaid. De stem die wordt uitgebracht is niet meer dan het omzetten van een bit in het geheugen. Hiervoor biedt de class 'ContextUtil' een aantal methoden.

Een andere manier waarop de transactie kan worden beëindigd is wanneer de transactie timeout optreedt. Op dat moment wordt altijd alles teruggedraaid. De transactie timeout kan worden aangepast, maar dit is vaak niet verstandig. Zolang een transactie loopt zijn er namelijk locks. Dit kan betekenen dat anderen staan te wachten. Wanneer de transactie te lang duurt is dit dus slecht voor de totale performance van de applicatie. Het verhogen van de transactie timeout is dus meestal niet wenselijk; de tijd moet eerder worden verkort.

```
using System;
using System.EnterpriseServices;

namespace EnterpriseTx
{

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

      }

      public void ControlledAction()
      {
          try
          {
              // Add some logic
              ContextUtil.SetComplete();
          }
          catch
          {
              ContextUtil.SetAbort();
          }
      }

      [AutoComplete()]
      public void AutomaticAction()
      {
          // Add some logic
      }
  }
}
```

Afbeelding 5. Voorbeeldcode transactie via Enterprise Services

## Transaction border pattern

In veel applicaties zijn transacties belangrijk. Het is dus ook zaak om gedurende het ontwerp van de applicatie na te denken over het gebruik van transacties. Hierbij moet de doelstelling van een transactie niet uit het oog worden verloren: het correct opslaan van gegevens. Dit betekent soms dat er concessies moeten worden gedaan aan de performance van een applicatie. Aan de andere kant hoeft niet altijd alles transactioneel uitgevoerd te worden. Het ophalen van gegevens kan meestal zonder een transactie gebeuren.

Van de eerder genoemde transactiemogelijkheden is het in de praktijk vaak wenselijk om een combinatie te gebruiken. Vaak is dit goed mogelijk. Een ADO.NET SqlClient kan bijvoorbeeld goed participeren in een enterprisetransactie. Een ADO.NET OleDb-transactie overigens niet!

Persoonlijk denk ik dat een combinatie van een pure T-SQL-transactie met een enterprisetransactie een eenduidig schaalbaar programmeermodel kan opleveren Wanneer een stored procedure wordt geschre-

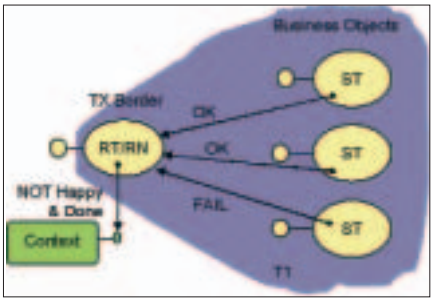
```
DECLARE @theTxCount int
SET @theTxCount = @@TRANCOUNT

IF @theTxCount = 0
  BEGIN
   SET TRANSACTION LEVEL SERIALIZABLE
   BEGIN TRANSACTION
  END

INSERT INTO
Order
VALUES
(1,10,getdate())

IF @@ERROR > 0 GOTO ErrorHandler

INSERT INTO
OrderLine
VALUES
(1,12198,10,"red")

IF @@ERROR > 0 GOTO ErrorHandler

IF @theTxCount = 0 AND @@TRANCOUNT > 0
  BEGIN
    COMMIT TRANSACTION
  END

GOTO ExitHandler

ErrorHandler:
  IF @theTxCount = 0 AND @@TRANCOUNT > 0
    BEGIN
      ROLLBACK TRANSACTION
    END
  ELSE
    BEGIN
      RAISERROR('Error in stored procedure',16,1)
    END

ExitHandler:
```

**Afbeelding 7. Voorbeeldcode Stored Procedure, geschikt voor enterprisetransacties**

ven volgens de voorbeelcode in afbeelding 7, kan deze worden gebruikt in combinatie met een .NET declaratieve transactieclass. Wanneer de achterliggend logica van de declarative transactions wordt meegenomen is het vaak verstandig om gebruik te maken van een model zoals zichtbaar is in afbeelding 8. In dit model is er één class die de transactie start: de transaction root. De rest heeft het attribuut 'Supported' aanstaan. Deze classes stemmen niet, maar geven het resultaat van de desbetreffende actie terug via een goed of fout code aan de root. Deze bepaalt wanneer alle resultaten terug zijn of de transactie goed of fout is verlopen en brengt als enige een stem uit. Op deze manier heeft de transaction root alsnog controle over de uitkomst van de transactie.

| | Handmatig of Automatisch | Distributed of Local | Performance verschillende RM's? | Portable naar | Controle |
|---|---|---|---|---|---|
| Enterpise transaction | A | D | 1 | Ja | 1 |
| ADO.NET | H | L | 2 | Ja | 2 |
| T-SQL | H | L | 3 | Nee | 3 |
| Distributed T-SQL | H | D | 2 | Nee | 3 |

**Tabel 4. Transactiematrix**




**Afbeelding 8. Transactie pattern**

## Flexibele architectuur heeft de voorkeur

Binnen .NET-code kan er van diverse soorten transactie mechanismen gebruik worden gemaakt. Voor alle mechanismen is een plaats. Afhankelijk van de gewenste functionaliteit en performance kan voor een ander mechanisme worden gekozen. In tabel vier zijn de vier mechanismen afgezet ten opzichte van de mogelijke eisen. Wanneer een kolom uit cijfers bestaat dan is 3 het beste en 1 het minste.

Uiteindelijk blijft het dus zaak om gedurende de ontwerpfase de eisen boven tafel te krijgen en een juiste keuze te maken voor een transactietechnologie. Als het kan heeft een flexibele architectuur, waarin de keuze voor een andere transactietechnologie mogelijk blijft, altijd de voorkeur.

**Nuttige internetadressen**

- www.tpc.org
- http://msdn.microsoft.com/library/default.asp?url=/nhp/Default.asp?contentid=28000519