Gijs de Jong is Principal Consultant bij Microsoft Services

BETROUWBARE SERVICES BOUWEN

WINDOWS COMMUNICATION FOUNDATION VOOR HET BOUWEN VAN SERVICEGEORIENTEERDE APPLICATIES

Windows Communication Foundation (codenaam 'Indigo') is een geünificeerd framework voor het bouwen van servicegeoriënteerde applicaties op het Windows-platform. Het biedt een communicatie-infrastructuur voor het ontwikkelen van veilige, betrouwbare en platformonafhankelijke services. Dit artikel bouwt voort op het artikel 'Programmeren met Indigo' dat gepubliceerd is in editie 8 van het .NET Magazine. In dit artikel zal specifiek worden ingegaan op mogelijkheden die Windows Communication Foundation (WCF) biedt voor het bouwen van robuuste gedistribueerde systemen. Aspecten die aan bod zullen komen, zijn ***reliable messaging, queuing*** en ***transacties***. Security zal in een volgend artikel aan de orde komen.

Dit artikel is gebaseerd op de publiek beschikbare bètaversie 1 van WCF. Vanwege de bètastatus van de technologie bestaat de mogelijkheid dat er verschillen zijn met de finale versie. Om de codevoorbeelden te kunnen gebruiken, zullen de volgende componenten geïnstalleerd moeten zijn op een Windows XP SP2, Windows 2003 of Windows Vista Beta 1 pc:

- .NET Framework Beta 2
- WinFx Components Beta 1
- WinFx Extensions Beta 1 for Visual Studio
- WCF Beta 1 MSMQ Support Package
- COM+ Hotfix for WCF

Reliable Messaging

Serviceoriëntatie is een architectuuraanpak voor het ontwerpen van gedistribueerde applicaties. Services abstraheren de onderliggende implementatie en communiceren op basis van berichten. Services kunnen zich in hetzelfde datacenter bevinden, maar kunnen ook geografisch verspreid zijn. Er bestaat dus een kans dat de communicatie over minder stabiele netwerkverbindingen plaatsvindt, zoals het internet. Weer andere toepassingen maken gebruik van steeds populairder wordende wireless technologie. Dit betekent voor de bouw van moderne systemen dat niet altijd op een betrouwbaar communicatiemedium vertrouwd kan worden. WCF biedt de mogelijkheid om systemen op robuuste wijze met elkaar te integreren over minder betrouwbare netwerkverbindingen, door middel van *reliable messaging* (RM).

De eerste generatie webservices maakte nagenoeg alleen maar gebruik van http als het transportprotocol. WCF maakt het mogelijk om services beschikbaar te stellen over verschillende protocollen; http, tcp, Named Pipes en MSMQ. De WCF-architectuur is echter zo opgezet dat deze set aan transportprotocollen verder aangevuld kan worden met eigen implementaties. Afhankelijk van het specifieke scenario zal dus een bijpassend transportprotocol gekozen kunnen worden. Elk transportprotocol heeft zijn eigen betrouwbaarheidskarakteristieken. Om op transportprotocolonafhankelijke wijze toch dezelfde betrouwbaarheidskarakteristieken te kunnen bieden, implementeert WCF dit op SOAP-berichtniveau. Tcp zorgt er bijvoorbeeld voor dat IP-pakketten altijd eenmalig en in de juiste volgorde bij de ontvanger aankomen. Wat Tcp voor IP biedt, biedt WCF *Reliable Messaging voor SOAP-berichten*.

Een ander voordeel bij het realiseren van RM op SOAP-niveau is *'end-to-end'* betrouwbaarheid. Het is mogelijk dat tussen de client en de werkelijke service verscheidene zogenaamde tussenstations of 'intermediaries' bestaan. Tussen deze tussenstations kan er over verschillende protocollen gecommuniceerd worden. Om *'end-to-end'* betrouwbaarheid te kunnen garanderen, zal deze 'Quality of Service' (QoS) op SOAP-niveau gerealiseerd moeten worden en niet op het niveau van het transportprotocol. WCF realiseert betrouwbaarheid op SOAP-niveau door de implementatie van de *WS-ReliableMessaging*-specificatie. Deze specificatie beschrijft gestandaardiseerde *headers* die aan het SOAP-bericht worden toegevoegd. Ook wordt beschreven hoe deze *headers* geïnterpreteerd en verwerkt moeten worden. Naast de voordelen die hierboven al zijn genoemd, biedt een gestandaardiseerd protocol voor *RM* ook platformonafhankelijkheid. Dit betekent dat op betrouwbare wijze gecommuniceerd kan worden tussen elk platform of webserviceruntime dat het *WS-ReliableMessaging-protocol ondersteunt*.

```
[ServiceContract]
public interface IOrderService
{
   [OperationContract (IsOneWay = true)]
   void SubmitOrder(Order order);
}

[DataContract]
public class Order
{
   [DataMember] public int Id;
   [DataMember] public DateTime Date;
   [DataMember] public double Amount;
}
```

Codevoorbeeld 1.
Service- en datacontract

```
public class OrderService : IOrderService
{
public void SubmitOrder(Order order)
   {
      // Sla de order op in de database
      InsertOrder(order);
   }
}
```

Codevoorbeeld 2.
Service-implementatie

```
<system.serviceModel>
  <services>
    <service
      serviceType=»dotNETMagazine.Samples.OrderService»>
    <endpoint
      address=»http://localhost/OrderService»
      bindingSectionName=»wsProfileBinding»
      bindingConfiguration=»wsBinding»
      contractType=»dotNETMagazine.Samples.IOrderService» />
    </service>
  </services>
  <bindings>
    <wsProfileBinding>
      <binding
        configurationName="wsBinding"
        reliableSessionEnabled="true"
        orderedSession="false"
        flowTransactions="Ignore"
        securityMode ="None"/>
    </wsProfileBinding>
  </bindings>
</system.serviceModel>
```

Codevoorbeeld 3.
EndPoint in app.config

```
class Program
{
  Static void Main(string[] args)
  {
    // Creëert een instantie van de ServiceHost en laadt automatisch
      de bijbehorende configuratie
    ServiceHost<OrderService> host = new ServiceHost<OrderService>();
    host.Open();

    Console.WriteLine("Host is running ...");
    Console.ReadLine(); // Blokkeer thread, houdt de applicatie in leven

    host.Close();
  }
}
```

Codevoorbeeld 4.
ServiceHost

WCF ondersteunt dus RM, maar wat betekent dit concreet:

- WCF maskeert de onderliggende netwerkverbinding. Wanneer de verbinding wegvalt, zal WCF deze automatisch, volledig transparant voor de applicatie, opnieuw opzetten.
- WCF verstuurt een bericht opnieuw wanneer de client binnen een configureerbare time-outwaarde geen ontvangstbevestiging ('Ack') van de service heeft ontvangen. Dit is ook volledig transparant voor de applicatie.
- WCF past de tijdsduur tussen het versturen van berichten en ontvangstbevestigingen dynamisch aan de karakteristieken aan (bijvoorbeeld responsetijden) van het netwerk.
- WCF garandeert dat binnen de scope van de RM-sessie een bericht maar eenmalig bij de service wordt afgeleverd, ook al is deze meermalen verstuurd, omdat er nog geen ontvangstbevestiging ontvangen was.
- WCF garandeert dat binnen de scope van de RM-sessie berichten in dezelfde volgorde bij de service worden afgeleverd zoals ze door de client zijn verstuurd. Dit is optioneel configureerbaar.
- WCF garandeert dat een bericht bij de service is afgeleverd als de client hier een ontvangstbevestiging ('Ack') voor ontvangen heeft. (Dit is vooral voor OneWay-berichten van belang.) Als geen ontvangstbevestiging is binnengekomen, is het niet duidelijk in hoeverre het bericht wel of niet bij de service is afgeleverd.

RM-aspecten worden bepaald door bindings. Een binding kan gezien worden als de configuratie van de WCF-infrastructuur die in grote mate bepaalt hoe een service te benaderen is. Een binding definieert bijvoorbeeld het transportprotocol, de bericht-encoding en SOAP-protocollen gerelateerd aan security, reliable messaging en gedistribueerde transacties. Bindings worden over het algemeen bepaald in configuratie, maar zijn ook aan te passen vanuit code. De wsProfileBinding, netProfileTcpBinding en netProfileNamedPipeBinding zijn standaard bindings die ondersteuning voor RM bieden. Het is natuurlijk ook mogelijk zelf een custom binding samen te stellen die RM ondersteunt.

```
svcutil.exe http://localhost/OrderService /namespace:, /out:
  OrderServiceProxy.cs /config:app.config
```

Codevoorbeeld 5.
Proxy- en configuratiegeneratie door svcutil.exe

Hoe implementeer ik RM?

In het nu volgende voorbeeld wordt geïllustreerd welke stappen nodig zijn om een WCF-service te creëren die ondersteuning biedt voor RM over http. Allereerst beginnen we met het definiëren van de service- en datacontracten en de concrete service-implementatie.

Nu de contracten en de concrete service-implementatie gebouwd zijn, kan door configuratie een **EndPoint** aan de service worden toegevoegd. Codevoorbeeld 3 laat de *ServiceModel*-sectie zien uit het *app.config*-configuratiebestand van de service-host. Het *reliableSessionEnabled*-attribuut van het *binding*-element bepaalt of gebruik gemaakt moet worden van RM. Het *orderedSession*-attribuut geeft aan in hoeverre berichten ook in dezelfde volgorde bij de service afgeleverd moeten worden als ze door de client verstuurd zijn. De *securityMode* is voor dit voorbeeld op *None* gezet, omdat de nadruk van dit artikel op betrouwbaarheid ligt en niet op security. In een productieomgeving is dit uiteraard niet wenselijk.

Als host voor de service gebruiken we voor het gemak een console-applicatie.

De configuratiesectie uit codevoorbeeld 3 wordt automatisch geladen bij het creëren van een instantie van het *ServiceHost*-object. Er zal in het configuratiebestand gekeken worden naar het service-element waarbij het *serviceType*-attribuut overeenkomt met 'OrderService'. De service is nu volledig geïmplementeerd en klaar om gebruikt te worden. Om te verifiëren of alles goed werkt, kan de helppagina van de service in een browser worden opgevraagd. Dit kan door de browser naar de volgende url te laten verwijzen http://localhost/OrderService. Door middel van het *metadataPublishing behavior* kan gecontroleerd worden of de helppagina getoond moet worden en hoe metadata gepubliceerd moet worden (http-get en/of *WS-MetadataExchange*).
Nu de service volledig is geïmplementeerd, kan de client gebouwd worden. In eerste instantie zullen een proxy en bijbehorende configuratie gegenereerd moeten worden. Hiervoor kan de *svcutil.exe*-tool worden gebruikt. Deze tool leest alle metadata van de service in en genereert de bijbehorende proxy en configuratie voor de client. De metadata van de contracten bestaat uit WSDL en XSD Schema. De RM-specifieke metadata worden gepubliceerd door middel van Policy. Codevoorbeeld 5 illustreert hoe de tool aangeroepen dient te worden.

De gegenereerde code en configuratiebestanden kunnen vervolgens aan een nieuw Visual Studio 2005-project worden toegevoegd. In dit voorbeeld is voor een console- applicatie gekozen.

Is een RM-channel alleen voldoende?

Al communiceren de client en de service over een RM-channel, er kunnen nog steeds fouten optreden. Een RM-bevestiging ('Ack') geeft aan dat een bericht ontvangen is door de service. Op het moment dat de client geen bevestiging ontvangt, is het onduidelijk of het bericht wel of niet bij de service is aangekomen. Het is namelijk mogelijk dat de service het bericht wel degelijk heeft ontvangen, maar dat tijdens het terugsturen van de bevestiging iets fout is gegaan. Dit zou bijvoorbeeld veroorzaakt kunnen zijn door het uitvallen van het netwerk. Door middel van configuratie kan aangegeven worden hoe vaak WCF een bericht opnieuw moet versturen. Als een bericht meermalen verstuurd wordt, zal WCF garanderen dat binnen de scope van de RM-sessie de daadwerkelijke service slechts eenmalig wordt aangeroepen. Mocht de hoeveelheid retries en/of time-outwaarden overschreden worden, dan zal er op de client een exceptie optreden en wordt de RM-sessie beëin-

digd. Op dit punt is het voor de client onduidelijk of het bericht wel of niet door de service is verwerkt. Het is de verantwoordelijkheid van de client om de exceptie af te handelen en te bepalen hoe verder om te gaan met de situatie. De client kan ervoor kiezen om het bericht opnieuw te versturen. Is dit laatste het geval, dan kan het zijn dat het bericht misschien voor een tweede keer door de service wordt verwerkt.

Codevoorbeeld 6 laat zien dat er bij het aanroepen van de service rekening gehouden wordt met mogelijke foutscenario's door het gebruik van *try-catch*-blokken. Er kan onderscheid gemaakt worden tussen twee typen excepties, *ChannelConnectException* en *SessionFaultedException*. *ChannelConnectException* treedt op wanneer het maken van de initiële connectie met de service mislukt. Als de initiële connectie met de service wel tot stand is gekomen wordt een RM-sessie gecreëerd. Na het aanroepen van een of meer operaties op het proxy-object wordt de *Close*-methode aangeroepen. Deze methode blokkeert net zo lang totdat alle berichten verstuurd zijn, of de *retries* en/of time-outwaarden overschreden zijn. Op dat moment zal de RM-sessie afgebroken worden en wordt de *SessionFaultedException* gegenereerd. De statusverandering van de RM-sessie kan ook gedetecteerd worden door het *proxy.InnerChannel.OutputSession.Faulted* event.

Transacties

Reliable messaging zorgt voor betrouwbare communicatie tussen modules in een gedistribueerd systeem, maar zegt niets over de verwerking van de berichten. Voor het betrouwbaar verwerken van de afgeleverde berichten biedt WCF ondersteuning in de vorm van (gedistribueerde) transacties.

```
private void button1_Click(object sender, EventArgs e)
{
using (OrderServiceProxy proxy = new OrderServiceProxy("OrderService"))
  {
    Order order = new Order();
    order.Id = 1;
    order.Date = DateTime.Now;
    order.Amount = 100;

    try
    {
      proxy.SubmitOrder(order);
      proxy.Close();
    }
    catch (ChannelConnectException ex)
    {
      // De initiële connectie met de remote service is mislukt
    }
    catch (SessionFaultedException ex)
    {
      // De RM sessie is afgebroken door WCF
    }
    catch (Exception ex)
    {
      // Handel alle overige excepties af
    }
  }
}
```

Codevoorbeeld 6.
Client-implementatie

```
[OperationBehavior(AutoEnlistTransaction = true, AutoCompleteTransaction
= true)]
void SubmitOrder(Order order)
{
InsertOrder(order); // Methode voegt de order toe aan de database

  // Creëert een instantie van de BillingService proxy
using (BillingServiceProxy proxy = new BillingServiceProxy("billingSer
vice"))
  {
    proxy.BillCustomer(order); // Roep de BillingService aan
    proxy.Close();
  }
}
```

Codevoorbeeld 7.
ProcessOrder-implementatie

Er zijn grofweg twee typen transacties te onderkennen: atomaire en compenserende transacties.

- Atomaire transacties zorgen ervoor dat verschillende fysieke acties als één logische actie worden uitgevoerd. Hierbij worden alle acties doorgevoerd. In het geval van een foutsituatie, wordt er geen een doorgevoerd.
- Bij een compenserende transactie worden alle acties uitgevoerd, er vanuit gaande dat deze slagen. Mocht een actie mislukken, dan zal voor de overige acties compenserende logica moeten worden uitgevoerd/aangeroepen. De wijzigingen worden dan niet teruggedraaid, maar de compenserende actie zal de resource terug moeten brengen in een geldige staat. Compenserende transacties zijn over het algemeen complexer dan atomaire transacties.

Dit artikel zal zich richten op atomaire transacties. In veel gevallen zal het aanroepen van een serviceoperatie resulteren in het opslaan of wijzigen van gegevens in een of meer achterliggende datastores. Om consistentie in de datastore(s) te waarborgen moeten de wijzigingen allemaal wel, of allemaal niet worden doorgevoerd. Dit principe wordt uiteraard al jaren gehanteerd. Het .NET Framework 2.0 biedt ondersteuning voor het implementeren van atomaire transacties. Deze functionaliteit is gehuisvest in de *System.Transactions*- namespace. Het gebruik van transacties maakt het eenvoudig de staat van verscheidene datastores, of andere transactionele resourcemanagers, in een logische actie te wijzigen. Hetzelfde geldt voor serviceoverstijgende scenario's. Het zou erg eenvoudig zijn als alle acties in elke service als eenheid, wel of niet, worden doorgevoerd. Voordat we ingaan op hoe dit in WCF valt te realiseren, is het belangrijk een goed beeld te hebben wanneer beter wel en wanneer niet voor serviceoverstijgende transacties gekozen kan worden.

Serviceoriëntatie is een architectuurprincipe waarmee wordt gestreefd naar een hoge mate van ontkoppeling tussen de verschillende modules. Deze modules kunnen zich in verschillende geografische locaties bevinden en onder verschillende beveiligingsautoriteiten functioneren. Het gebruik van gedistribueerde transacties stelt zeer hoge eisen aan de *latency* tussen deze verschillende services en de mate van vertrouwen. Bij een gedistribueerde transactie worden de te wijzigen gegevens in bijvoorbeeld een database geblokkeerd gedurende de levensduur van de transactie. De duur van de transactie heeft dus direct invloed op de beschikbaarheid van het systeem. Ook is er een hoge mate van vertrouwen nodig om een externe service rechtstreeks controle te geven over de *locks* op de database. In een proces waar een gedistribueerde transactie over verscheidene services wordt gebruikt, wordt de beschikbaarheid en performance bepaald door de 'zwakste schakel'. Als een database, die door een van de services wordt gebruikt, niet beschikbaar is zal dit ervoor zorgen dat de gehele transactie wordt teruggedraaid.

Het gebruik van gedistribueerde transacties over services heen moet dus goed overwogen worden. De noodzaak voor transactie-flow is vaak sterk gerelateerd aan de granulariteit van de services. Wanneer verscheidene 'kleinere' operaties nodig zijn om een bepaalde businessoperatie te kunnen afhandelen, wordt de noodzaak groter om deze operaties in de context van dezelfde transactie uit te voeren. De toepasbaarheid van gedistribueerde transacties over meer services heen, geldt vooral voor services die sterk aan elkaar gerelateerd zijn. Dit kan zijn omdat ze bijvoorbeeld tot hetzelfde functionele domein behoren, of omdat ze als eenheid beheerd worden. WCF biedt ondersteuning voor transactie-flow over services met het *WS-AtomicTransaction*-protocol. Transacties in WCF zijn dus integreerbaar met transacties op elk willekeurig platform en webservice-runtime, mits deze ook *WS-AtomicTransaction* (WS-AT) ondersteunen. Het WS-AT-protocol beschrijft, net zoals alle andere WS*- protocollen, een standaard set aan SOAP-*headers* en hoe deze geïnterpreteerd

en verwerkt moeten worden. Omdat dit slechts een set aan *headers* in het SOAP-bericht is, kan het WS-AT-protocol prima worden gebruikt in combinatie met *WS-ReliableMessaging*. Zoals in codevoorbeeld 3 te zien is, biedt de configuratie van de wsProfileBinding settings voor beide protocollen.

Hoe bouw ik transactionele services?

Om te illustreren hoe gebruik gemaakt kan worden van transacties in WCF, breiden we codevoorbeeld 2 uit de reliable messaging-sectie verder uit. Om het voorbeeld werkend te krijgen zal eerst ondersteuning van WS-AT voor de MSDTC-service eenmalig aangezet moeten worden. Dit kan door de xws_reg.exe-tool, die meegeleverd wordt in de WinFX SDK. Het volgende commando zorgt hiervoor: *xws_reg.exe -wsat+*

WCF biedt grofweg een drietal features rondom transacties:

- Transactioneel versturen van berichten – meer OneWay-berichten worden door de client verstuurd binnen de scope van een transactie.
- Transactioneel uitvoeren van een operatie – dit is een service-implementatie-aspect dat door middel van een behavior wordt bepaald.
- Transactie-flow over services heen – is een service-extern aspect dat door middel van een binding wordt bepaald.

Het voorbeeld breiden we verder uit zodat de *SubmitOrder*-operatie van de *OrderService* naast het uitvoeren van de databaseoperatie ook een tweede service aanroept. Deze tweede service (*BillingService*) zal daarentegen ook een databaseoperatie uitvoeren. De databaseoperaties van beide services worden uitgevoerd in de context van dezelfde transactie en zullen dus beide wel of niet doorgevoerd worden.

Wat in codevoorbeeld 7 opvalt, is het *OperationBehavior*-attribuut en de bijbehorende transactieparameters.
AutoEnlistTransaction = true geeft aan dat als er een transactie is meegekomen vanaf de aanroepende partij, de operatie hierin moet participeren. Als er geen transactiecontext is meegekomen, zal een nieuwe transactie gecreëerd worden. Dit is een declaratieve manier om een transactiecontext te creëren. Aangezien de client geen transactiecontext heeft meegegeven, zal *SubmitOrder* de root van een nieuwe transactie worden.
AutoCompleteTransaction = true geeft aan dat de transactie automatisch doorgevoerd moet worden als de methode gewoon (zonder exceptie) retourneert. Als er wel een exceptie optreedt, zal de transactie automatisch worden teruggedraaid. Wanneer *AutoCompleteTransaction* op *false* gezet wordt, zal de *OperationContext.Current.SetTransactionComplete-methode* expliciet moeten worden aangeroepen om de transactie door te voeren. Als deze methode niet wordt aangeroepen, zal de transactie worden teruggedraaid.

Nu moet ervoor gezorgd worden dat bij het aanroepen van de *BillingService* de nieuw gecreëerde transactie ook daadwerkelijk meegenomen wordt naar de *BillingService*. Hiervoor zal de *binding*-configuratie van de *OrderService* aangepast moeten worden. In de configuratie, die in codevoorbeeld 3 is getoond, zal het *flowTransactions*-attribuut veranderd moeten worden (van 'Ignore') in 'Allowed'. De *BillingService* zal net zoals een willekeurig andere service gehost moeten worden. In het voorbeeld zou dezelfde console-applicatie gebruikt kunnen worden als de *OrderService*. Om de *BillingService* aan te roepen vanuit de *OrderService* moet uiteraard eerst een proxy- en configuratiebestand worden gegenereerd met *svcutil.exe*. De configuratie zal toegevoegd moeten worden aan het *App.config*-bestand van de *OrderService*-host. Hierin zal aangegeven moeten worden dat als de *BillingService* binnen een transactie wordt uitgevoerd de transactiecontext meeflowt. In codevoorbeeld 8 worden de relevante secties van de gegeneerde configuratie getoond.

```
<client>
  <endpoint
    address=»http://localhost/BillingService»
    bindingConfiguration="IBillingService"
    bindingSectionName="customBinding"
    contractType="IBillingService"
    configurationName="BillingService">
    ...
  </endpoint>
</client>
<bindings>
  <customBinding>
    <binding
      configurationName="IBillingService">
      <contextFlow
        transactions="Allowed"
        transactionHeaderFormat="Wsat"
        ... />
      <httpTransport
        ... />
      <textMessageEncoding
        ... />
    </binding>
  </customBinding>
</bindings>
```

Codevoorbeeld 8.
Client-side configuratie voor de BillingService

```
[ServiceContract]
interface IBillingService
{
  [OperationContract]
  void BillCustomer(Order order);
}

[OperationBehavior(AutoEnlistTransaction = true, AutoCompleteTransaction
= true)]
void BillCustomer(Order order)
{
  InsertBill(order); // Methode voegt de rekening toe aan de database
}
```

Codevoorbeeld 9.
BillingService-contract en implementatie

Wat ook opvalt, is dat er een *customBinding* is gegenereerd en geen *wsProfileBinding*. De svcutil-tool kent het concept van standaard/named bindings niet. De tool leest de metadata van de service in en genereert de configuratie per element binnen een *customBinding*.

Voorbeeld 9 illustreert hoe de 'BillCustomer'-operatie is geïmplementeerd. Het *OperationBehavior*-attribuut bepaalt hier ook de transactie-eigenschappen. *AutoEnlistTransaction* zorgt er hier voor dat er automatisch geparticipeerd moet worden in de transactie die meegekomen is vanuit de *OrderService*.

De *BillCustomer*-operatie retourneert *void*, toch is deze operatie niet als *OneWay* geïmplementeerd. Transactie-flow vereist namelijk dat een response-(out)-bericht naar de aanroepende partij wordt gestuurd. De configuratie van de 'BillingService' wordt weergegeven in codevoorbeeld 10. Het *flowTransaction*-attribuut geeft in de binding-sectie aan dat binnenkomende transacties zijn toegestaan.

WCF-transactiefunctionaliteit wordt dus beïnvloed door een combinatie van *bindings, behaviors* en het feit of de aanroepende partij een transactie mee laat gaan. Tabel 1 geeft een overzicht van de verschillende combinaties en het resulterende gedrag.

Een ontwikkelaar bepaalt dus door *AutoEnlistTransaction* of een methode transactioneel moet worden uitgevoerd. Normaliter wordt de code op een dusdanige manier geschreven dat je er vanuit kunt gaan dat iets transactioneel gebeurt. Dit is niet iets dat achteraf geconfigureerd wordt. Transactie-flow is echter wel iets dat achteraf geconfigureerd moet kunnen worden. Het wordt dan mogelijk om voor dezelfde service verscheidene EndPoints te definiëren met verschillende transactiekarakteristieken. Er zou bijvoorbeeld voor gekozen kunnen worden om een bepaalde EndPoint wel

```
<services>
   <service
        serviceType="dotNETMagazine.Samples.BillingService">
      <endpoint
         address="http://localhost/BillingService"
         bindingSectionName="wsProfileBinding"
         bindingConfiguration="wsBindingBillingService"
         contractType="dotNETMagazine.Samples.IBillingService" />
   </service>
</services>
<bindings>
   <wsProfileBinding>
      <binding
         configurationName="wsBindingBillingService"
         reliableSessionEnabled="false"
         orderedSession="false"
         flowTransactions="Allowed"
         securityMode ="None"/>
   </wsProfileBinding>
</bindings>
```

Codevoorbeeld 10.
Configuratie van de BillingService

binnenkomende transacties te laten accepteren, omdat alleen interne clients deze aanroepen. Als de service op een gegeven moment ook door een externe client gebruikt dient te worden, kan een EndPoint worden toegevoegd die geen transactie-flow ondersteunt.

Queuing

Tot nu toe hebben we gezien hoe we op betrouwbare wijze tussen een client en een service kunnen communiceren op basis van reliable messaging. Ook hebben we gezien hoe services op betrouwbare wijze verscheidene acties kunnen uitvoeren in de context van een atomaire transactie. Bij RM wordt er vanuit gegaan dat zowel de client als de service op hetzelfde moment beschikbaar zijn. Voor transacties geldt dit nog sterker, want hierbij wordt er vanuit gegaan dat alle achterliggende transactionele resourcemanagers waar de service gebruik van maakt ook beschikbaar zijn. Deze vorm van koppeling is in sommige gevallen niet wenselijk of helemaal niet mogelijk. Het gebruik van queues tussen de client en service creëert een ontkoppeling waardoor beide partijen niet noodzakelijkerwijs op hetzelfde moment beschikbaar hoeven zijn. Een tweede voordeel van deze ontkoppeling is dat de service alleen de gemiddelde lading van berichten hoeft te kunnen verwerken en niet de piekladingen die clients op een specifiek moment kunnen genereren. Op het moment dat clients meer berichten in de queue stoppen dan de server kan verwerken, zal de hoeveelheid berichten in de queue gewoon oplopen. Op het moment dat clients minder berichten versturen dan de server kan verwerken, zal de hoeveelheid berichten in de queue afnemen, totdat alle berichten in de queue uiteindelijk verwerkt zijn. Er vindt dus nivellering van werklast plaats. Als er meer verwerkingskracht nodig is om de berichten te verwerken, kunnen meer services uit dezelfde queue berichten halen. Op deze manier faciliteert queuing ook in de schaalbaarheid (scale-out) van een systeem.

Naast de ontkoppeling die queues tussen de twee communicerende modules creëren, zorgen queues ook voor een verhoogde betrouwbaarheid. *Reliable Messaging* zorgt er alleen maar voor dat een bericht aankomt bij zijn eindbestemming, maar wat gebeurt er wanneer de server crasht net nadat het bericht ontvangen is? Dan raken we het bericht kwijt. Het bericht zal in een persistente opslag bewaard moeten worden, zodat het eventuele server/servicecrashes kan overleven. WCF maakt voor zijn queuing-functionaliteit gebruik gemaakt van MSMQ. Dit wordt weergegeven in afbeelding 1 en 2. Hierin is te zien dat wanneer een applicatie een bericht naar een queue verstuurt, het bericht eerst in een lokale queue wordt geplaatst. MSMQ zorgt er vervolgens voor dat het bericht naar de uiteindelijk remote queue wordt toegestuurd, wanneer deze beschikbaar is. De communicatie tussen de lokale queue en de remote queue gebeurt op basis van MSMQ's eigen communicatieprotocol. Deze communicatie is veilig en robuust, maar is niet gebaseerd op de WS*-protocollen zoals *WS-ReliableMessaging*.

FlowTransactions (Binding)	AutoEnlist Transaction (Behavior)	Client geeft transactie mee	Resultaat
False	False	Nee	Methode wordt zonder transactie uitgevoerd
False	True	Nee	Methode wordt in een nieuwe transactie uitgevoerd
False	True / False	Ja	Een SOAP fault wordt geretourneerd i.v.m. de transaction header
True	False	Ja	Methode wordt zonder transactie uitgevoerd
True	True	Ja	Methode wordt in de meegekomen transactie uitgevoerd

Tabel 1. Transactiescenario's

Het gebruik van queues in WCF is net zo transparant als elk willekeurig ander transportmechanisme, het programmeermodel blijft dus exact hetzelfde. De enige kanttekening bij het gebruik van het *Queued Channel* is dat alleen het *OneWay*-message exchange pattern gebruikt kan worden. Dit is inherent aan het asynchrone communicatiemodel dat queuing met zich meebrengt. Aangezien *Queued Channels* de MSMQ- infrastructuur gebruiken, kunnen beheerders deze queues beheren zoals ze gewend zijn te doen. Op deze manier is voor zowel de ontwikkelaars als de beheerders de *learning-curve* voor het gebruik en beheer van queues in WCF tot een minimum beperkt.

MSMQ gerelateerde Bindings

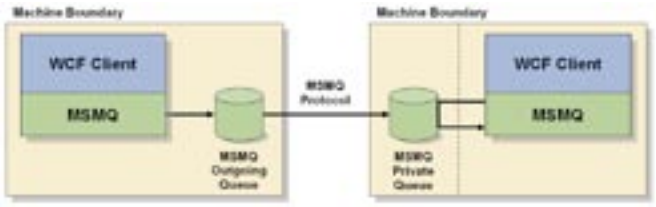
Het gebruik van het *Queued Channel* wordt net als alle andere transportprotocollen bepaald door een binding. Er zijn twee type MSMQ-gebaseerde bindings: *netProfileMsmqBinding* en *msmqIntegrationBinding*. Bij het gebruik van de *netProfileMsmqBinding* is zowel de client als de service gebaseerd op WCF. Wanneer de client de service aanroept zal het bericht allereerst in een lokale queue geplaatst worden. MSMQ zorgt vervolgens dat het bericht van de lokale queue naar de remote queue wordt verstuurd, zodra deze beschikbaar is. Op het moment dat de service wordt geactiveerd, registreert deze zich voor events uit de queue. Wanneer een bericht in de queue wordt geplaatst, zal deze er door WCF uitgehaald worden en zal de desbetreffende operatie op de service aangeroe-

```
string queueName = "OrderService";

if (!MessageQueue.Exists(".\private$\" + queueName))
{
   MessageQueue.Create(".\private$\" + queueName, true);
}
```

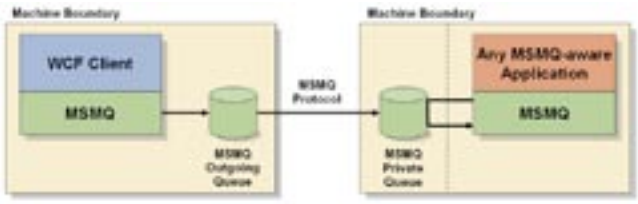
Codevoorbeeld 11.
Aanmaken van een queue

```
<endpoint
   address="net.msmq://localhost/private$/OrderService"
   bindingSectionName="netProfileMsmqBinding"
   bindingConfiguration="MyQueuedBinding"
   contractType="dotNETMagazine.Samples.IOrderService" />

<bindings>
   <netProfileMsmqBinding>
      <binding
         configurationName="MyQueuedBinding"
         msmqAuthenticationMode="None"
         msmqProtectionLevel="None"
         maxRetries="3"
         maxRetryCycles="1"
         retryCycleDelay="0:0:10"
         rejectAfterLastRetry="false"/>
   </netProfileMsmqBinding>
</bindings>
```

Codevoorbeeld 12.
Queuing EndPoint in configuratie



Afbeelding 1. NetProfileMsmqBinding



Afbeelding 2. MsmqIntegrationBinding

pen worden. Voor de ontwikkelaar is het volledig transparant dat MSMQ onder water gebruikt wordt. Afbeelding 1 illustreert dit model.

De *msmqIntegrationBinding* is bedoeld voor samenwerking met elke willekeurige applicatie die gebruikmaakt van MSMQ. Een WCF-client verstuurt in dit geval een bericht naar de queue die vervolgens door bijvoorbeeld een Visual Basic 6-applicatie verwerkt wordt. Afbeelding 2 illustreert dit model.

Hoe configureer ik een queued EndPoint?

Om weer te geven hoe een **'queued' service** geïmplementeerd kan worden, nemen we opnieuw het *OrderService*-voorbeeld als uitgangspunt. Allereerst zullen we een transactionele queue in MSMQ moeten aanmaken. Dit kan door middel van de MMC snap-in of programmatisch door de *System.Messaging* API. Codevoorbeeld 11 laat zien hoe in code een queue aangemaakt kan worden, nadat eerst gevalideerd wordt of de specifieke queue al bestaat.

Nu de queue is aangemaakt, kunnen we een EndPoint toevoegen aan de *OrderService*. Voor het adres van een private queue gebruiken we het volgende scheme: *net.msmq://<MachineNaam>/private$/<QueueNaam>*. Voor de *binding* gebruiken we *netProfileMsmqBinding*. De bijbehorende configuratiesettings van de *binding* worden in de *Bindings*-sectie gedefinieerd. Het contracttype blijft gewoon hetzelfde.

Aan het servicecontract, de service-implementatie of de host hoeft dus totaal niets gewijzigd te worden. Het is slechts een kwestie van het toevoegen van een nieuw *EndPoint* in het configuratiebestand. Zoals we hebben kunnen zien in codevoorbeeld 7, zorgt de *AutoEnlistTransaction*-eigenschap van het *OperationBehavior*-attribuut dat de serviceoperatie uitgevoerd wordt in de context van een transactie. Bij het gebruik van de *netProfileMsmqBinding* zal ook het uit de queue lezen van het bericht binnen de context van deze zelfde transactie gebeuren. Met andere woorden, mocht een exceptie optreden, dan zullen dus niet alleen beide databasewijzigingen worden teruggedraaid, maar zal ook het bericht teruggeplaatst worden in de queue. Op deze manier garandeert WCF dat een bericht nooit verloren gaat, ook al gaat de service/server halverwege de transactie down.

Na de foutsituatie zal na verloop van tijd opnieuw geprobeerd worden het bericht bij de service af te leveren. Het is exact te configureren om de hoeveel tijd dit moet gebeuren en hoeveel keren dit proces herhaald moet worden. Als de *maxRetryCycles*-limiet is bereikt, zal het bericht in de zogenaamde *Poison Message Queue* (PMQ) geplaatst worden. Dit is een sub-queue van de originele queue, met de vaste naam 'Poison'. In het geval van de *OrderService* zou het adres van de subqueue worden: *'net.msmq://localhost/private$/OrderQueue;Poison'*. Deze queue kan als een willekeurige queue benaderd worden om de gefaalde berichten te verwerken. In Windows Vista zal de naam van de *Poison*-queue zelf te definiëren zijn. De PMQ is vergelijkbaar met de MSMQ DeadLetter-queue.

```
ServiceHost<OrderService> host =
new ServiceHost<OrderService>("http://localhost/OrderService");
host.Open();
```

Codevoorbeeld 13.
ServiceHost en baseAddress

De DeadLetter-queue wordt echter gebruikt wanneer de MSMQ-infrastructuur een bericht van de client niet binnen de gestelde randvoorwaarden (time-outs enzovoort) kan afleveren bij de queue op de remote server. Dan wordt het bericht in de DeadLetter-queue op de client-machine opgeslagen.

In de codevoorbeelden tot nu toe hebben we kunnen zien hoe door *svcutil.exe* proxy- en configuratiebestanden gegenereerd konden worden. In WCF Beta 1 is het opvragen van metadata echter alleen nog maar over het http(s)-protocol mogelijk. Aangezien het *Queued Channel* ook geen WS*-protocollen ondersteunt, kunnen we de proxy- en configuratiebestanden niet op dezelfde wijze genereren. Door de *ServiceHost* een http(s)-gebaseerd *baseAddress* mee te geven kunnen hierover de metadata wel opgehaald worden. De metadata zijn op deze manier over http(s) op te vragen, maar de servicefunctionaliteit zal nog steeds alleen over het *Queued Channel* te benaderen zijn. Codevoorbeeld 13 laat zien hoe het *baseAddress* aan de *ServiceHost* moet worden meegeven.

Door de url in de browser in te voeren is de helppagina en de WSDL te zien. Het *baseAddress* kan vervolgens gebruikt worden als input voor de *svcutil.exe*-tool. Aangezien de rest van de stappen voor het opzetten van de client identiek zijn aan eerder beschreven voorbeelden worden deze hier achterwege gelaten.

Infrastructuur voor het bouwen van gedistribueerde systemen

Het bouwen van betrouwbare gedistribueerde applicaties zorgt voor verscheidene uitdagingen. WCF neemt een groot deel van deze complexiteit weg door de ondersteuning van *reliable messaging* en transacties. Voor maximale ontkoppeling tussen client en service kan gebruik gemaakt worden van queuing-technologie. Hierbij geldt wel de beperking dat zowel de client als de service gebruik moet maken van MSMQ. In toekomstige releases van WCF zal queuing wellicht wel in combinatie met WS*-protocollen te gebruiken zijn.

Gijs de Jong is Principal Consultant bij Microsoft Services. Zijn e-mailadres is gijsdj@microsoft.com

Nuttige Internetadressen

- Windows Communication Foundation MSDN-homepage: http://msdn.microsoft.com/webservices/indigo/default.aspx
- WinFX Developer Center: http://msdn.microsoft.com/winfx/
- Reliable Messaging in a Service Oriented Architecture: http://msdn.microsoft.com/library/en-us/dnglobspec/html/ws-rm-soa.asp