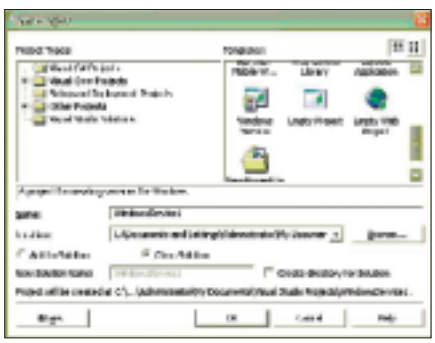
Jan Willem Overbeek
is werkzaam bij Cap Gemini Ernst & Young als softwarearchitect en ontwikkelaar. Hij is gespecialiseerd in het ontwikkelen van software voor het .NET-platform.
janwillem.overbeek@cgey.nl

Windows Services en .NET

HET WORDT ALLEEN MAAR BETER

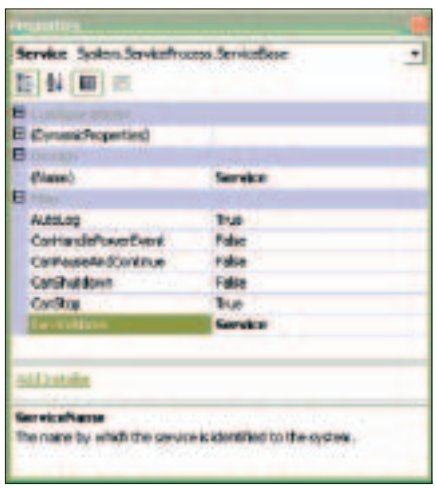
In dit artikel beschrijft auteur Jan Willem Overbeek welke aspecten er komen kijken bij het configureren en managen van een Windows Service die is ontwikkeld met behulp van het .NET Framework. Hierbij komen allerlei praktische problemen aan het licht, waarvoor een aantal oplossingsrichtingen zal worden aangedragen. Om aan te tonen dat .NET inmiddels een volwassen applicatieplatform is, zullen uitsluitend pure .NET-oplossingen worden gebruikt; dus zonder gebruik te maken van unmanaged code.

Een Windows Service is een proces dat kan draaien zonder dat een gebruiker is ingelogd. Een service kan automatisch door het besturingsysteem worden gestart en gestopt, en is meestal bedoeld om langlopende achtergrondprocessen uit te voeren, of om diensten te leveren die altijd beschikbaar moeten zijn, ongeacht of een gebruiker is ingelogd of niet. Via een netwerk bereikbare diensten zijn hiervan een goed voorbeeld. Het ontwikkelen van een Windows Service was vroeger behoorlijk complex, en kon in voorgaande versies van Visual Studio eigenlijk alleen goed met Visual C++ worden gedaan. Maar met de komst van het .NET Framework en Visual Studio.NET is dit allemaal veel eenvoudiger geworden.



Afbeelding 1. Het Windows Service projecttype

Er is binnen Visual Studio.NET een standaard projecttype voor een Windows Service gedefinieerd dat de code voor een basisimplementatie genereert; zie afbeelding 1. Verder is er ook een Installer class die het installeren van de service een stuk eenvoudiger maakt. Deze installer classes kunnen eenvoudig vanuit de Visual Studio IDE worden toegevoegd; afbeelding 2. Met deze hulpmiddelen is het mogelijk om in enkele minuten een werkende service te ontwikkelen.



Afbeelding 2. Let op de optie 'Add Installer'

De basisimplementatie van een Windows Service die in het .NET Framework beschikbaar is (ServiceBase genaamd, onderdeel van de namespace in System.ServiceProcess), schermt de ontwikkelaar van een hoop technische details af. Hierdoor hoeft er een stuk minder 'plumbing'-code geschreven te worden. Dit maakt de drempel om voor dit projecttype te kiezen een stuk lager, terwijl daar voorheen, vanwege de complexiteit en benodigde specialistische kennis, wellicht snel van af werd gezien. Het feit dat .NET deze initiële barrière wegneemt, betekent echter niet dat alle complexiteit van het ontwikkelen van een service volledig is verdwenen. Er zal, afhankelijk van het type service, nog steeds een aantal zaken zijn waarvoor een specifieke oplossing moet worden bedacht. In dit artikel zal een aantal hiervan worden besproken. Dit zijn voornamelijk zaken die te maken hebben met het configureren en managen van een service.

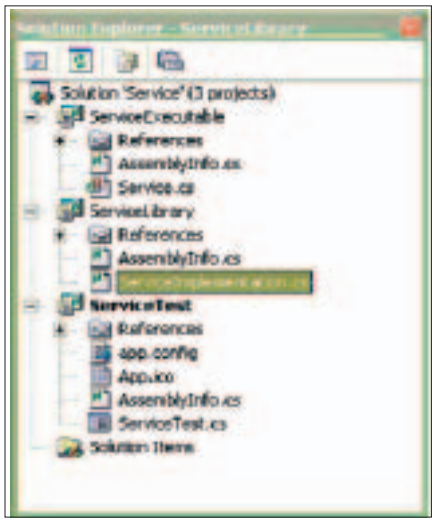
Het voorbeeldproject

Dit artikel maakt als voorbeeld gebruik van een heel eenvoudige service. Deze service doet niets anders dan een 'trace' uitvoeren op een configureerbare interval. Dat klinkt niet erg spannend, maar de focus ligt in dit artikel ook niet zo zeer op de implementatie van de

service, maar de zaken eromheen – en die zijn helaas vaak al complex genoeg. Het project is dusdanig opgezet, dat de code die in de service draait ook in een normale Windows-executable is te gebruiken. De reden hiervoor is dat het relatief lastig is om een service te debuggen, aangezien deze niet direct vanuit de Visual Studio IDE te starten is. Daarom is het vaak handiger om de code eerst in een gewone executable te hangen, en dan al het ontwikkel- en testwerk te doen. Vervolgens kan dezelfde code vanuit de service worden aangeroepen. Dan kan het laatste testwerk plaatsvinden, dat zich dan vooral op security-gerelateerde problemen zal richten, aangezien de service meestal onder een ander account zal draaien dan de testapplicatie.

Het project bestaat zodoende uit drie onderdelen; zie afbeelding 3:

- Een class library, die de implementatie voor de service bevat.
- Een normale executable, die gebruikt wordt gedurende het ontwikkeltraject.
- Een service executable, waarin de uiteindelijke code zal worden gestart.



Afbeelding 3. De opbouw van de solution

Interactie met een service

Vaak is het noodzakelijk dat een gebruiker of administrator de service op één of andere manier kan configureren, managen en monitoren. In ons simpele voorbeeld gaat het om een timer interval die moet kunnen worden ingesteld. In sommige gevallen zal het voldoende zijn om dit statisch te doen met behulp van een configuratiefile, maar vaak is er ook complexere communicatie over en weer met de service nodig; zoals met een uitgebreide service als SQL Server. In dat geval is er een managementapplicatie nodig om de gewenste interactie met de service mogelijk te maken. Hier volgt een aantal scenario's, zoals deze in de praktijk geregeld voor komen. Als eerste de simpelste variant, configuratie door middel van een .NET-configuratiebestand.

```
<?xml version="1.0" encoding="utf-8"?>
<configuration>
  <appSettings>
      <!-- User application and configured property settings go here.-->
      <!-- Example: <add key="settingName" value="settingValue"/> -->
      <add key="Service.TimerInterval" value="10000" />
  </appSettings>
</configuration>
```

Codevoorbeeld 1.

```
_timer.Interval = double.Parse(ConfigurationSettings.AppSettings[
"Service.TimerInterval"]);
```

Codevoorbeeld 2.

```
…
_timer.Interval = double.Parse(GetSetting["Service.TimerInterval"]);
…

private string GetSetting (string key)
{
   XmlDocument doc = new XmlDocument();
   string configFile = string.Format("{0}.config",
   Assembly.GetEntryAssembly().Location);
   doc.Load(configFile);
   string query =
   string.Format(@"configuration/appSettings/add[@key='{0}']", key);
   XmlNodeList setting = doc.SelectNodes(query);
   if (setting.Count > 0)
   {
      return setting[0].Attributes["value"].Value;
   }
   return null;
}
```

Codevoorbeeld 3.

Configuratie met een .config-file

.NET-applicaties maken voor hun configuratie gebruik van een config-file die in dezelfde directory staat als de applicatie. Deze configuratiefile heeft altijd dezelfde naam als de executable, aangevuld met de extensie 'config'. Een applicatie genaamd 'Applicatie.exe' heeft dus een configuratiefile genaamd 'Applicatie.exe.config'.

Deze configuratiefiles zijn gewone XML-bestanden die met een willekeurige teksteditor of XML-editor zijn te bewerken. Ze moeten uiteraard wel aan een schema voldoen, dat uitgebreid in de MSDN-documentatie staat beschreven. De configuratiefile wordt automatisch geladen tijdens het laden van de applicatie, en dus ook bij een Windows Service. We kunnen voor het configureren van onze service gebruik maken van deze file. Zo is het mogelijk om binnen de sectie appSettings onze eigen configuratieparameters toe te voegen, bijvoorbeeld:

Tijdens het starten van de service kan deze parameter dan worden gebruikt om

```
fsw = new FileSystemWatcher();
string configFile = string.Format("{0}.config",
Assembly.GetEntryAssembly().Location);
fsw.Path = Path.GetDirectoryName(configFile);
fsw.NotifyFilter = NotifyFilters.LastWrite;
fsw.Filter = Path.GetFileName(configFile);
fsw.Changed += new FileSystemEventHandler(_fsw_Changed);
fsw.EnableRaisingEvents = true;
```

Codevoorbeeld 4.

```
[InstrumentationClass(InstrumentationType.Instance)]
public class ServiceImplementation : Instance
{
   public ServiceImplementation()
   {
      Published = true;
   }

   // De waarde van deze property is via WMI uit te lezen zijn
   public string TimerInterval
   {
     get
     {
          return _timer.Interval.ToString();
     }
   }
}
```

Codevoorbeeld 5.

de timer interval van de service te configureren.

Dit is een prima methode voor de configuratie van parameters die niet te vaak wijzigen. Voor het dynamisch configureren van een service is deze methode minder geschikt, want de configuratiefile wordt slechts eenmalig gelezen en verder gecacht. Als er een waarde in de configuratiefile wordt aangepast zal deze dus pas in werking treden als de applicatie opnieuw wordt gestart. In het geval van een service houdt dit in dat deze gestopt en weer gestart moet worden. Mochten er meer services binnen dezelfde executable zijn geïmplementeerd, dan zullen deze eerst allemaal gestopt en opnieuw gestart moeten worden. Alleen het stoppen en weer starten van een individuele service is in dit geval niet voldoende, want hierdoor wordt de configuratiefile niet opnieuw geladen. In sommige situaties zal het stoppen en weer starten van de service geen acceptabel scenario zijn. In dat geval moet worden gezocht naar een alternatief. Er van uitgaande dat we zoveel mogelijk van de standaard .NET-werkwijze gebruik willen blijven maken (en dus niet terug vallen op bijvoorbeeld de registry), is het een oplossing om zelf de actuele settings uit het configuratiebestand te lezen. We kunnen dit bijvoorbeeld als volgt doen.

Nu moet er alleen nog een oplossing worden gevonden om te ontdekken dat er iets in de configuratie is aangepast. Een manier om dit te doen is door het configuratiebestand te monitoren, en de instellingen opnieuw in te lezen zodra het bestand wordt veranderd. Hiervoor kan gebruik worden gemaakt van de FileSystemWatcher class. In het volgende stukje code wordt deze class gebruikt om een event af te vuren zodra de configuratiefile wordt veranderd.

In de afhandeling van het event kunnen de instellingen dan opnieuw gelezen worden en dynamisch worden toegepast, zonder dat de service hoeft te worden gestopt. Een hele verbetering, maar misschien nog niet in alle gevallen toereikend.

Windows Management Instrumentation

Het managen van een service uitsluitend op basis van een configuratiefile zal niet altijd voldoen. In een Enterprise-scenario, waar beheerders honderden, zo niet duizenden machines moeten beheren, is het al snel geen werkbare oplossing meer. Voor dergelijke scenario's is door Microsoft 'Windows Management Instrumentation' (WMI) ontwikkeld. Het is een verzamelnaam waaronder alle technologie valt die te maken heeft met het configureren, managen en monitoren van applicaties. WMI is gebaseerd op een industriestandaard (WBEM), en wordt inmiddels toegepast in vrijwel alle onderdelen van het Windows-platform. Microsoft levert ook een Enterprise Managementtool dat van deze technologie gebruik maakt, genaamd Microsoft Operations Manager (MOM). Verder wordt WMI ook door vele third-party Enterprise Managementtools ondersteund.
Ook bij het toepassen van WMI heeft bij .NET een enorme productiviteitswinst teweeggebracht. Het implementeren van WMI-functionaliteit was voorheen een bewerkelijk en moeizaam proces, dat eigenlijk alleen in Visual C++ gedaan kon worden. In Visual Studio.NET is het echter allemaal een stuk eenvoudiger geworden, in de zin dat er opnieuw veel minder 'plumbing'-code geschreven hoeft te worden.De ondersteuning voor WMI in .NET valt uiteen in twee delen. Er is ondersteuning voor het gebruik van WMI-functionaliteit die door andere applicaties is geïmplementeerd, en er is ondersteuning voor het aanbieden (implementeren) van functionaliteit via WMI. Hiervoor is een aantal classes beschikbaar binnen het .NET Framework (in System.Management en System. Management.Instrumentation), en verder is er een WMI-uitbreiding voor de Visual Studio IDE. Deze uitbreiding moet echter wel los bij Microsoft worden gedownload en geïnstalleerd. Dat is echter zeker de moeite waard, want na installatie van deze uitbreiding wordt het moge-

lijk om vanuit de Server Explorer WMI-classes te browsen, property's te wijzigen, op WMI Event Queries op te bouwen, enzovoort (1). In feite is het met deze uitbreiding mogelijk om je hele machine te beheren, en is er geen enkele reden meer om ooit nog Visual Studio te verlaten; afbeelding 4.

Afbeelding 4. De WMI Server Explorer extensions

Een andere nuttige functie van de WMI Explorer is de mogelijkheid wrappers voor managementclasses te genereren. Deze wrappers maken het mogelijk om op een intuïtieve (strong typed) manier tegen WMI-objecten aan te praten. Van deze functionaliteit zullen we verderop ook gebruik gaan maken.

Tijd om WMI eens praktisch toe te gaan passen in het voorbeeldproject. Hiervoor moet eerst een referentie worden aangelegd naar de assembly waarin WMI is geïmplementeerd (System.Management). Vervolgens kan door het aanbrengen van een aantal attributen in de code worden aangegeven welke functionaliteit via WMI moet worden aangeboden. Het publiceren van onze service bij WMI gebeurt door de implementatieclass te laten erven van de WMI 'Instance' class. Als vervolgens de property 'Published' op true wordt gezet, zullen alle publieke property's via WMI zijn uit te lezen, mits ze van een type zijn dat WMI begrijpt. Het veiligste is hiervoor standaard het type string te gebruiken.

De eerste keer dat de applicatie opstart wordt er achter de schermen van alles gedaan. Soms zie je zelfs kort een console-window verschijnen, als de WMI-classes worden geïnstalleerd. Het resultaat hiervan is terug te vinden in directory [WINDIR]\system32\wbem\Framework\root\default.

Als het goed is zijn hier een bestand met gegenereerde C#-code en de Managed Object Format (MOF) file behorende bij de WMI-classes te vinden. Nu is onze serviceclass geregistreerd in de 'Root/default' namespace van WMI, en kan hij in de Server Explorer worden bekeken. Zoals te zien is in afbeelding 5, is de waarde van de timer interval uit te lezen in de standaard Visual Studio property-editor.

Een nadeel van de huidige implementatie van WMI in .NET Framework 1.x is dat het nog niet mogelijk is property's te wijzigen. Ook is het nog niet mogelijk om WMI method-providers te implementeren in .NET (2). Het aanroepen van een method binnen onze service via WMI is dus helaas niet mogelijk. Aangezien dit voor het managen van een service vaak wel noodzakelijk zal zijn, moeten we op zoek naar een andere manier om dit te bewerkstelligen. De meest voor de hand liggende manier om dit met .NET te doen, is door

```
public class ServiceManagement : MarshalByRefObject
{
   public ServiceManagement(ServiceImplementation implementation)
   {
       _serviceImplementation = implementation;
   }

   public override object InitializeLifetimeService()
   {
       return null;
   }

   public int TimerInterval
   {
       get
       {
           return _serviceImplementation.TimerInterval;
       }
       set
       {
           _serviceImplementation.TimerInterval = value;
       }
   }

   private ServiceImplementation _serviceImplementation;
}
```

Codevoorbeeld 6.

```
TcpChannel channel = new TcpChannel(10000);
ChannelServices.RegisterChannel(channel);
ServiceManagement serviceManagement = new ServiceManagement(this);
RemotingServices.Marshal (serviceManagement, "management");
```

Codevoorbeeld 7.

```
ServiceManagement service = (ServiceManagement)Activator.GetObject(
      typeof(ServiceManagement),
      @"tcp://localhost:10000/management");

service.TimerInterval = 3000;
```

Codevoorbeeld 8.

```
Cassini.Server cassini = new Cassini.Server(10001, "/",
   Path.GetDirectoryName(Assembly.GetEntryAssembly().Location));

cassini.Start();
```

Codevoorbeeld 9.

```
<%@ Page Language="C#" Debug="true" ClassName="WebManagement" %>
<%@ assembly name="ServiceLibrary" %>
<%@ import Namespace="ServiceLibrary" %>
<script runat="server">

    // Insert page code here
    //

    void _update_Click(object sender, EventArgs e) {
        ServiceManagement service =
        (ServiceManagement)Activator.GetObject(
                    typeof(ServiceManagement),
                    @"tcp://localhost:10000/management");

        service.TimerInterval = int.Parse(_timerValue.Text);
    }

    void Page_Load(object sender, EventArgs e) {
         if(!IsPostBack)
         {
            ServiceManagement service =
            (ServiceManagement)Activator.GetObject(
                typeof(ServiceManagement),
                 @"tcp://localhost:10000/management");

                 _timerValue.Text = service.TimerInterval.ToString();
         }
    }

</script>
<html>
<head>
</head>
<body>
    <form runat="server">
        <asp:Label id="_timerLabel" runat="server"
        width="95px">Timer Interval :</asp:Label>
        <asp:TextBox id="_timerValue" runat="server"></asp:TextBox>
         <asp:Button id="_update" onclick="_update_Click"
        runat="server" Text="Update"></asp:Button>
        <!-- Insert content here -->
    </form>
</body>
</htm
```

Codevoorbeeld 10.

gebruik te maken van de remoting functionaliteit die het .NET Framework biedt.

.NET remoting

.NET remoting is de technologie die het mogelijk maakt om tussen processen te communiceren. In ons geval kunnen we remoting toepassen om te communiceren tussen de managementapplicatie en de service. Het managen van een service met remoting hoeft niet ingewikkeld te zijn. De eenvoudigste manier om dit te

Afbeelding 5. Property's van de service-instance

implementeren is om alle managementfunctionaliteit in één class te groeperen, en eerst direct – dus zonder remoting – te testen. Vervolgens kan een instance van deze class via remoting beschikbaar worden gemaakt, en dan kan de functionaliteit ook vanuit een ander proces worden aangeroepen. In ons geval willen we de timer interval kunnen instellen, en komt de class er als volgt uit te zien.

Het publiceren van deze class kan op een aantal manieren gebeuren. Het is mogelijk met behulp van de configuratiefile van de service of direct vanuit de code. Ook kan gekozen worden om van een http- of een TCP-channel gebruik te maken. Deze keuze zal meestal afhangen van infrastructurele of performance gerelateerde randvoorwaarden, want functioneel is er weinig verschil (3).

In code ziet het publiceren van een object bij remoting er als volgt uit.

Deze code kan worden aangeroepen bij het starten van de service, waardoor de .NET remoting infrastructuur op poort 10000 zal gaan luisteren naar binnenkomende connecties. Nu is de tijd aangebroken om een clientapplicatie te ontwikkelen die hiervan gebruik kan maken.

Een management client

Voor het managen van een service kan het noodzakelijk zijn een om een specifieke managementapplicatie te ontwikkelen die toegang geeft tot functionaliteit die in de service is geïmplementeerd. Traditioneel wordt hiervoor meestal gebruik gemaakt van een Microsoft Management Console (MMC) snap-in. Het is echter met de huidige versies van het .NET Framework nog niet mogelijk om dit volledig in managed code te doen(4), en het wordt officieel ook niet door Microsoft

ondersteund. In de toekomst zal hier ongetwijfeld verandering in komen, maar voorlopig zullen we dus een andere oplossing moeten kiezen. Omdat we gebruik willen maken van de standaardmogelijkheden die .NET ons biedt, hebben we voor onze managementapplicatie de keuze uit twee oplossingen: een ASP.NET-applicatie (thin client) en een Windows Forms-applicatie (rich client). Beide hebben hun specifieke voor- en nadelen die nu aan de orde zullen komen.

Rich client

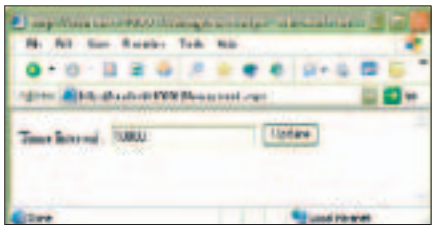
Een Windows Forms-applicatie heeft de beschikking over het rijke scala aan controls dat .NET biedt. Voor interactie met een service die complexe diensten verzorgt, zal dit een doorslaggevende factor zijn. De toepassing van dit type client vereist echter wel dat op iedere machine de managementapplicatie moet komen te draaien en minimaal een runtime-versie van het .NET Framework geïnstalleerd moet zijn. Tevens moet de assembly, waarin de classes zitten die over remoting worden aangesproken, met de applicatie worden meegeleverd. Hiermee dient bij het ontwerpen van de fysieke architectuur van de software al rekening gehouden te worden. De types die gedeeld moeten worden tussen de service en de applicatie dienen zo veel mogelijk in een eigen assembly geïsoleerd worden. Verder zijn er weinig bijzonderheden. Een Windows Forms-client kan via remoting vrijwel transparant met de service communiceren, nadat er een connectie is gelegd met het remote managementobject. Dit kan bijvoorbeeld als volgt.

Thin Client

Een thin client heeft beperktere mogelijkheden wat betreft userinterface, maar is minder arbeidsintensief met deployment. De voornaamste voorwaarde is dat ergens in de infrastructuur een webserver (IIS) beschikbaar moet zijn die toegang heeft tot de machine waarop de service draait. Indien dit niet mogelijk of wenselijk is, bestaat ook nog de mogelijkheid een webserver te embedden in de service zelf. Hiervoor kan gebruik worden gemaakt van Cassini, de lichtgewicht webserver die Microsoft in Web Matrix gebruikt. Deze component mag vrij van royalty's worden gebruikt, en ook de source-code is beschikbaar. (Voor meer details zie www.asp.net). Het embedden van Cassini is eenvoudig. Leg vanuit het project een reference aan naar de Cassini-assembly (default genaamd Cassini.dll), en neem de volgende code op.

Hiermee wordt een webserver gestart die luistert op poort 10001, en die als applicatie-directory het pad gebruikt waar de service staat. Nu is enkel nog een ASP.NET-pagina nodig om de service te managen. Hier volgt een pagina waarmee de timer interval van de service kan worden aangepast.

Als deze pagina in de directory wordt geplaatst waar ook de service ook, met als naam 'Management.aspx', dan is deze met een standaardbrowser te benaderen; afbeelding 6.



Afbeelding 6. Thin client managementinterface

Overigens moet er nog wel wat aan de code van Cassini worden getweakt om het volledig geschikt te maken voor deze toepassing. Zo weigert Cassini standaard alle requests die niet van de lokale machine komen. Maar aangezien de source-code beschikbaar is, kan iedereen dit verder naar eigen wens aanpassen.

Puur .NET

.NET is een robuust, maar vooral ook zeer productief applicatieplatform. Het is zonder enige twijfel een van de allermooiste technologieën die Microsoft tot nu toe heeft voortgebracht. En vanaf nu zal het alleen nog maar beter worden, want we staan pas aan het begin van de ontwikkeling van .NET. Maar zoals uit dit artikel ook blijkt, kan het toch voorkomen dat je tijdens het ontwikkelen tegen beperkingen of tekortkomingen van het .NET Framework aanloopt. Meestal gaat het dan om functionaliteit die wel in de Win32 API, maar nog niet in de framework classlibrary beschikbaar is. Mijn advies is om in deze situaties niet te snel terug te vallen op oude gewoonten, zoals het gebruik van unmanaged code. Vaak leidt dit er toe dat de productiviteitswinst van .NET dan snel weer verdwijnt. Er is meestal wel een pure .NET-oplossing te bedenken, ook al is die misschien technisch niet altijd 100% zoals je zou willen. Bedenk dat het slechts een kwestie van tijd is, totdat het onproductieve geworstel met obscure Win32 API-calls en frustrerende COM interop-problemen voorgoed tot het verleden behoort. En dan zul je blij zijn dat je je prachtige, cleane .NET-applicaties hier niet mee hebt opgezadeld.

Noten

1. Meer informatie over de functionaliteit van de WMI Server Explorer extensions is te vinden in het artikel "System.Management Lets You Take Advantage of WMI APIs within Managed Code" op MSDN.
2. De unmanaged WMI SDK biedt deze mogelijkheid overigens wel.
3. Voor meer informatie hierover zie de artikelen op MSDN: "ASP.NET Web Services or .NET Remoting: How to Choose" en "Performance Comparison: .NET Remoting vs. ASP.NET Web Services".
4. Er zijn overigens wel third-party frameworks om dit te doen, zie bijvoorbeeld http://www.ironringsoftware.com.