De ToolBus in een web-enabled omgeving

Bert Huijben

Afstudeerscriptie Informatica
Faculteit der Natuurwetenschappen, Wiskunde en Informatica (FNWI)
Universiteit van Amsterdam
Afstudeerrichting: Programmatuur
Afstudeerdocent: Prof. Dr. Paul Klint
Datum: 27 september 2001

Inhoud

I. Inleiding 5
- Motivatie 5
- Structuur 6

II. Introductie Web-omgevingen 7
- Geschiedenis 7
- Uitgangspunten en Systeemmodel 7
- Webserver Architectuur 9
- Beveiliging & Verificatie 10

III. Windows DNA 13
- Geschiedenis 13
- Uitgangspunten 14
- Programmeermodel 15
- Beveiliging & Verificatie 17
- Webkoppeling: ASP - Active Server Pages 17
- Case Studie 18
- Vooruitzichten 18

IV. DotNet 19
- Geschiedenis 19
- Uitgangspunten & programmeermodel 19
- Beveiliging & Verificiatie 22
- Webkoppeling 23
- Case Studies 24
- Vooruitzichten 24

V. J2EE - Enterprise Java 26
- Geschiedenis 26
- Uitgangspunten 26
- Programmeermodel 27
- Beveiliging & Verificatie 29
- Webkoppeling 29
- Case Studie 29
- Toekomstperspectief 30

VI. De ToolBus 31
- Geschiedenis 31
- Uitgangspunten 31
- Programmeermodel 32
- Beveiliging en Verificatie 34
- Webserver koppeling 34

VII. ToolBus - Webserver koppeling 35
- Structuur 35
- Connectie opbouw 35
- Term uitwisseling 38
- Autorisatie 39
- Case Studie – PHP adapter 41
- Case Studie – ToolBus port naar Windows 43
- Case Studie – DotNet Adapter 45
- Case Studie – DCOM Adapter 47

VIII. Conclusies 48
De ToolBus in web-enabled toepassingen 48
ToolBus gebruik vanuit een webserver 49
DotNet Adapter voor de ToolBus 49
ToolBus voor Windows 49
A. C# ATerm Factory 50
Publieke interface 50
Implementatie 50
B. Bronvermelding 69

I. Inleiding

Motivatie
In de informatietechnologie wordt de laatste jaren bij het ontwikkelen van grote systemen steeds vaker gekozen om de communicatie binnen software-systemen te structureren met abstracte standaardtechnologieën. De zelf ontwikkelde koppelingen maken hierbij plaats voor een systeem met goed vastgelegde eigenschappen dat het mogelijk maakt om de modules als losse componenten te zien die hergebruikt kunnen worden in andere systemen, of los van de rest van het systeem aangepast kunnen worden zonder dat dit consequenties heeft in de rest van het systeem.

In het begin van de jaren 90 kwamen Corba en COM op als de technieken die dit mogelijk moesten gaan maken, maar uiteindelijk is gebleken dat deze systemen te ingewikkeld waren om het in alle soorten systemen toe te kunnen passen. De vakgroep programmatuur van de UvA onderkende al snel dat deze systemen te ingewikkeld waren voor de systemen die zij gebruikten en hebben daarom een eigen techniek ontwikkeld.

De ToolBus [Bergstra98] die ontwikkeld werd voor de renovatie van de Asf+Sdf Meta-Environment [Brand01] is daarom een stuk eenvoudiger opgezet, maar bouwt voort op de bestaande technieken. In tegenstelling tot COM en Corba is de ToolBus echter geen op het client-server of drie-laag model gebaseerde techniek, maar op een model waarin alle componenten gelijkwaardig zijn en geen kennis hebben van de opbouw van de andere componenten.

Intussen zijn we een aantal jaar verder en zijn de ideeën over het bouwen van component-systemen veranderd. De toepasbaarheid in een Internet omgeving is intussen een van de belangrijkste eisen geworden voor de componenttechnologieën en dat heeft een paar grote verschuivingen te weeg gebracht. COM en Corba zijn intussen vervangen door Windows DNA en Enterprise Java en het ziet er niet naar uit dat de markt zich op korte tijd zal stabiliseren.

De ToolBus heeft de stap naar de Internet wereld nog niet gemaakt. Deze afstudeer scriptie is bedoeld als een eerste stap in deze richting. Ik probeer een antwoord te geven op de vraag: *Wat zijn de mogelijkheden en onmogelijkheden van de ToolBus in web-enabled toepassingen?*

Om dit te realiseren heb ik me verdiept in de technieken achter Windows DNA, Microsoft DotNet, Enterprise Java en PHP[1]. Daarnaast heb ik de ToolBus en de aanverwante technieken bestudeerd door het porten van de ToolBus naar Windows en het ontwikkelen van ToolBus-adapters voor COM en DotNet.

Uiteindelijk heeft dit geleid tot een groot aantal kleine testapplicaties in de diverse systemen om de werking te kunnen bestuderen en in een aantal modules die het mogelijk maken om gebruik te maken van de ToolBus vanuit diverse soorten Windows applicaties.

---

[1] Een goed voorbeeld van een programmeertaal die bedoeld is om dynamische webpagina's op te bouwen, maar standaard niet gekoppeld is aan een componenttechnologie. Door gebruik te maken van een aantal nu nog redelijk experimentele extensies biedt PHP wel de mogelijkheid om gebruik te maken van DCOM, DotNet en Enterprise Java.

Structuur
Voordat het mogelijk is om de ToolBus in een web-enabled omgeving te bestuderen is het noodzakelijk om te bepalen waaruit een web-enabled omgeving bestaat. Hoofdstuk II geeft een overzicht van de eigenschappen en biedt een inzicht in de mogelijkheden die de gangbare technieken bieden.

In de hoofdstukken III tot en met V beschrijf ik achtereenvolgens Windows DNA, Microsoft's DotNet en Enterprise Java. In hoofdstuk VII beschrijf ik de ToolBus en de mogelijkheden die de ToolBus zou kunnen bieden in een web-enabled omgeving. De hoofdstukken beginnen met een introductie over het ontstaan van de technologieën, gevolgd door een uitleg over hoe het model in elkaar zit. Vervolgens worden de mogelijkheden in een web-omgeving bestudeerd. De hoofdstukken worden afgesloten met een case studie en een vooruitzicht over de toekomstige mogelijkheden.

In hoofdstuk VII bepaal ik vervolgens de mogelijkheden die de ToolBus heeft in een web-enabled omgeving, door te bestuderen wat er moet gebeuren om de ToolBus te koppelen aan een webserver. Daarnaast bespreek ik hier de projecten die ik heb uitgevoerd om de toepasbaarheid van de ToolBus in een web-enabled omgeving te vergroten. In hoofdstuk VIII worden tenslotte de conclusies die getrokken kunnen worden uit dit project gepresenteerd.

Appendix A geeft tenslotte een gedeelte van de implementatie van de ATerm bibliotheek voor DotNet in C#.

De voornaamste resultaten van dit afstudeerproject zijn:
- Aanbevelingen om de ToolBus beter geschikt te maken voor het gebruik in een web-enabled omgeving.
- Een implementatie van de ATermen [Brand2000] voor Microsoft's DotNet.
- Een DotNet adapter voor de ToolBus.

II. Introductie Web-omgevingen

Geschiedenis
In het verleden zijn veel systemen langzaam uitgegroeid tot systemen die uit een groot aantal modules bestaan zonder een duidelijke samenhang. Het onderhouden van dit type systemen wordt alleen steeds lastiger tot er een duidelijke structuur aangebracht wordt.

Het afgelopen decennium zijn er een aantal nieuwe technieken geïntroduceerd die aan deze problemen een einde moeten maken. Door het systeem in duidelijke componenten in te delen die onderling via een duidelijk vastgelegd protocol met elkaar communiceren ontstaat er een veel duidelijkere structuur en wordt het mogelijk om de componenten als losse modules te ontwikkelen. Bij een goed ontwerp is het daarnaast mogelijk om dit soort modules in meerdere systemen te gebruiken of juist modules in zijn geheel te vervangen door andere modules zonder dat de rest van het systeem daar last van heeft.

Uitgangspunten en Systeemmodel
Van applicaties in een web-enabled omgeving wordt vereist dat ze niet alleen bereikbaar zijn vanaf lokale systemen maar ook vanuit een webbrowser. Hiervoor zijn dynamische websites nodig die kunnen communiceren met lokale applicaties. De besproken technieken zijn van oorsprong allemaal bedoeld om modules binnen applicaties met elkaar te verbinden, maar zijn in de afgelopen jaren uitgebreid met ondersteuning voor het gebruik in web-scripts.

Oorspronkelijk werden de meeste dynamische websites gebouwd in scripttalen die vooral geschikt zijn om homepages te genereren en niet bedoeld zijn voor gebruik in grote systemen zoals PHP en Perl. Deze talen bevatten standaard een grote runtime library met mogelijkheden om die uit te breiden met functies op diverse gebieden. Voor eenvoudige systemen die niet veel meer doen dan eenvoudige database transacties uitvoeren, bieden deze talen op zichzelf al voldoende mogelijkheden.

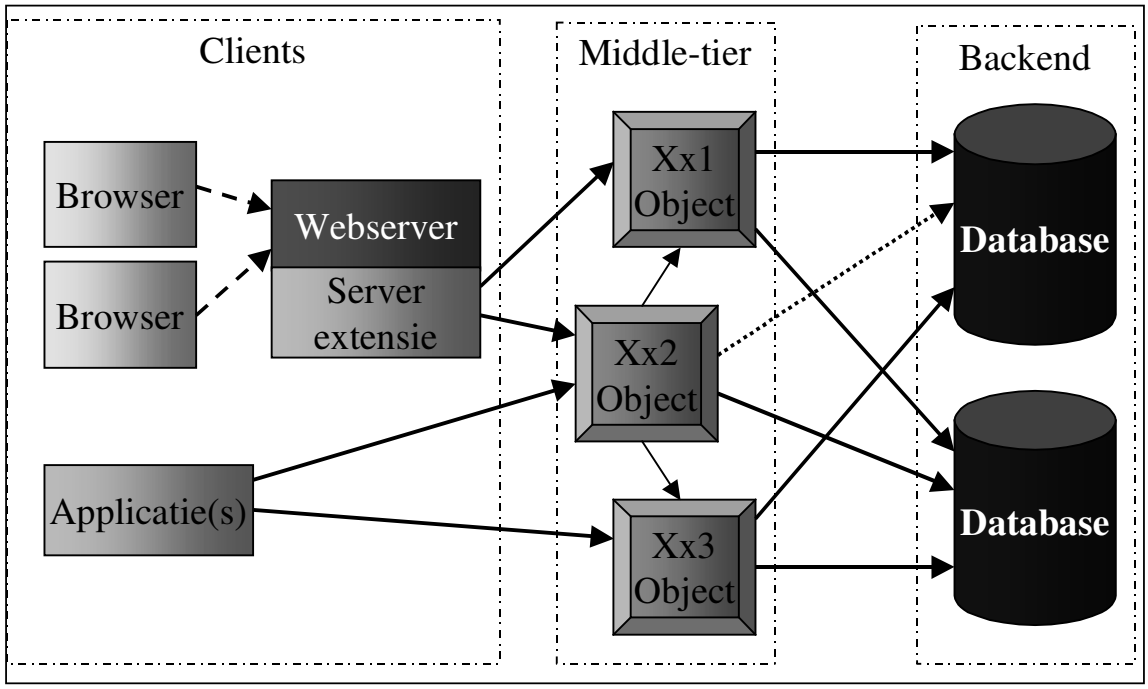
Voor grotere web-systemen wordt steeds vaker gekozen voor drie- of meer-laags architecturen, omdat deze systemen, mits goed ontworpen, veel betere mogelijkheden bieden op het gebied van schaalbaarheid, onderhoudbaarheid en stabiliteit. Het is opvallend dat er in dit segment maar twee grote partijen gesteund worden binnen de software industrie: Enterprise Java en Windows DNA. De Windows georiënteerde bedrijven steunen vooral DNA of DotNet en de meer Unix/Linux georiënteerde bedrijven steunen Enterprise Java.

De meeste bedrijven die tot een paar jaar geleden onder de Corba aanhangers vielen hebben intussen gekozen hebben voor Enterprise Java, omdat Corba niet zo goed van de grond gekomen is als verwacht. Een aantal grote bedrijven zoals IBM hebben geen keuze voor een techniek gemaakt en willen zich in de nabije toekomst op beide technieken te richten.

Enterprise Java en Windows DNA baseren zich op een vergelijkbare drie-laags architectuur maar hebben die op een verschillende manier opgebouwd. In deze architectuur worden toepassingen opgebouwd in drie lagen:

- Een backend, dat meestal gevormd wordt door een of meer database management systemen.
- Een object-georiënteerde middelste laag die de applicatielogica bevat.

- Client applicaties die een connectie opbouwen met de middelste laag om hun functionaliteit aan te bieden aan de gebruiker.



Om goed te laten zien dat ze gericht zijn op een web-enabled omgeving laten de modellen die door Sun en Microsoft gepresenteerd worden duidelijk zien dat er in de applicatie laag onderscheid gemaakt wordt tussen gewone applicaties en web-applicaties. De systemen zijn zo geïmplementeerd dat alle lagen van het model uitgevoerd kunnen worden als een losse computer of door een groep computers.

Afhankelijk van welk model er gebruikt wordt zijn er op alle plaatsen in het model applicatie-namen in te vullen die respectievelijk Microsoft en Sun het liefste op die plaats zouden zien.

	Windows DNA	**Enterprise Java**
Web-Client	Internet Explorer die een verbinding opbouwt naar de Internet Information Server (Microsoft IIS).	Een browser die een verbinding opbouwt naar een server die Java Server Pages ondersteunt.
App-Client	Een Windows of DotNet applicatie	Een Java applicatie of een Java applet
Middle Tier	COM Objecten op een of meer Windows Servers.	Objecten binnen een of meer Java Virtual Machines met Enterprise Java Bean ondersteuning.
Backend	Een server(farm) met Microsoft SQL Server.	Een Server met Java Database Connectivity (JDBC) ondersteuning.

In het Windows DNA model worden alle lagen gerepresenteerd door Microsoft systemen, maar in de Enterprise Java wereld is dat veel minder strikt vastgelegd. In het Enterprise Java model bestaat iedere tier, behalve de webbrowser uit een Java specifieke technologie, terwijl in het Microsoft Model iedere laag op vele verschillende manieren geïmplementeerd kan worden.

Webserver Architectuur
Om de koppeling tussen de architecturen en webservers te bestuderen is het nodig om onderscheid te maken in een aantal soorten webservers, omdat die verschillende eisen stellen aan koppelingen.

De huidige webservers zijn qua implementatie in te delen in drie categorieën. Single-threaded, multi-threaded en multi-task webservers. Single-threaded servers zijn vooral geschikt voor het zo efficiënt mogelijk opvragen van beschikbare pagina's zonder dat er pagina's dynamisch gegenereerd worden. In deze categorie vallen een aantal hele eenvoudige webservers die alleen bedoeld zijn voor kleine groepen gebruikers en vele servers in deze categorie zijn niet in staat meerdere webbrowsers tegelijkertijd te ondersteunen. Er zijn echter ook een aantal op efficiency en capaciteit geoptimaliseerde webservers in deze categorie te vinden die vooral gebruikt worden door bedrijven die extreem grote aantallen gebruikers exact dezelfde pagina's willen aanbieden. Meestal worden in dat geval de statische pagina's door dit type server aangeboden en de dynamische pagina's door een ander type webserver. De performance vermindering door de synchronisatie van de taken en de overhead van het creëren van de taken staan in dat geval de performance van het systeem minder in de weg.

Webservers die oorspronkelijk ontwikkeld zijn om onder Unix systemen te werken maken meestal gebruik van meerdere zelfstandige processen om HTTP verzoeken af te handelen. De eerste generatie servers creëerde per verzoek een nieuw proces dat het verzoek volledig afhandelde en daarna stopte, maar tegenwoordig is er meestal een centraal proces dat verzoeken over de al aanwezige vrije taken verdeeld en alleen als dat echt noodzakelijk is nieuwe processen start. De term die hiervoor gebruikt wordt is task pooling.

Webservers voor Windows systemen zijn meestal multi-threaded, omdat Windows op een heel andere manier met child processen omgaat dan Unix. Onder Windows en de meeste moderne Unix systemen is de multi-threading implementatie de laatste jaren uitgegroeid tot een veel efficientere methode om met requests om te gaan dan zelfstandige processen. Dat meerdere threads hetzelfde geheugen direct kunnen benaderen blijkt hierin meer mee te spelen dan de nadelen van reentrance. Wanneer de Apache gebruikers naar de binnenkort te lanceren 2.0 versie upgraden, zal deze categorie webservers gangbaarder worden dan de multi-task categorie.

Aan webserver extensies worden in een multi-threaded omgeving andere eisen gesteld dan in een multi-task of single-threaded server. Extensies in single-threaded servers kunnen over het algemeen net zo geschreven worden als gewone losse programma's, want de extensies kunnen gewoon gebruik maken van statische variabelen om gegevens tussen verschillende requests op te slaan. In servers die geïmplementeerd zijn als verschillende losse processen is, dat niet zondermeer mogelijk, omdat het in dat geval niet bekend is of het vervolgverzoek uitgevoerd wordt door hetzelfde proces of door een geheel ander proces. Er zijn webservers die de verzoeken zo optimaliseren dat verzoeken die gebruik maken van dezelfde extensies zo veel mogelijk uitgevoerd worden door dezelfde processen, omdat dit de opstarttijd van de child

processen ten goede komt; dit verschil is vooral goed te merken bij dynamisch geladen extensies.

Bij het ontwikkelen van extensies voor multi-threaded webservers is het noodzakelijk om zo min mogelijk gegevens op te slaan in statische variabelen, want zonder het gebruik van thread synchronisatie is het niet goed mogelijk deze variabelen tegelijkertijd te benaderen vanuit verschillende threads. De meeste webservers in deze categorie hebben daarom memory managers die het eenvoudig mogelijk maakt om thread local variabele blokken aan te maken. Apache biedt daarnaast nog de mogelijkheid om geheugenblokken te reserveren die na het uitvoeren van het verzoek automatisch weer vrijgegeven worden.

Beveiliging & Verificatie

Omdat veel via webservers beschikbare diensten beveiligd zijn, ligt de taak voor beveiliging van pagina's in eerste instantie bij de webserver. In veel gevallen delegeert deze de taken naar extensies die daar speciaal voor ontwikkeld zijn of naar speciaal voor dat doel geschreven scripts. PHP is een van de scripttalen die overweg kan met dit type beveiliging. Om dit mogelijk te maken doet PHP zich aan een webserver voor als meerdere types extensies. Onder Microsoft's Internet Information Server (IIS) zijn dat een protocol filter, om alle headers van het request te kunnen lezen, en een request handler om de gegenereerde pagina op te bouwen.

Voor tools die in een Internet omgeving ingezet moeten worden gelden andere eisen dan voor componenten die binnen grotere applicaties gebruikt worden. Homepages worden alleen op verzoek van een webbrowser opgevraagd van de webserver. De pagina's zijn hierdoor statisch en moeten door een op zichzelf staand verzoek worden opgebouwd. Zo'n verzoek bestaat uit de op te vragen informatie, maar wordt vaak aangevuld met gebruiker identificerende gegevens, die in de meeste gevallen behandeld worden als cookies.

Een cookie is een klein datablok dat een webserver kan instellen na het uitvoeren van een verzoek. Na dit instellen wordt dit datablok bij ieder nieuw verzoek aan deze webserver mee gestuurd. Een cookie kan een vervaltijd hebben en het is ook mogelijk om het meesturen van cookies alleen te limiteren tot subdomeinen of de cookies alleen te gebruiken voor een bepaalde subdirectory van de webserver. Omdat cookies het opslaan van privacy gevoelige gegevens op webservers mogelijk maakt, bieden webbrowsers die cookies ondersteunen ook mogelijkheden om cookies te negeren. In navolging van Sun's Java Server Pages technologie bieden veel webserver tools tegenwoordig ook mogelijkheden om dynamisch een gebruikers identificatie toe te voegen aan alle URL's naar pagina's op dezelfde server die een bepaalde HTML pagina bevat.

Omdat de voornaamste taak van een webserver het afhandelen van verzoeken van webbrowsers is, wordt van extensies vereist dat ze goed overweg kunnen met korte op zichzelf staande verzoeken. Wanneer dit niet het geval is komt dit direct naar voren als lagere responstijden van de website. Daarnaast moeten ze geschikt zijn om in beveiligde omgevingen gebruikt te kunnen worden. Hierbij is het zowel van belang dat er rekening wordt gehouden met autorisatie mogelijkheden als met het beperken van misbruik zoals denial of service attacks.

Beveiliging van webservers bestaat uit autorisatie en versleuteling. Voor het gebruiken van een login met een wachtwoord is er de basic authentication [rfc2617] uitbreiding van de HTTP standaard die door zo goed als alle webbrowsers en het merendeel van de webservers

ondersteund wordt. In dit systeem wordt bij het opvragen van een pagina waarvoor de gebruiker nog niet de juiste rechten heeft aan de webbrowser gevraagd nog een keer de pagina op te vragen, maar dan met een gebruikersnaam en wachtwoord. De webbrowser zal in dat geval de gebruiker een login en een wachtwoord vragen. Deze beveiliging biedt in deze standaardvorm geen mogelijkheden om het wachtwoord en de gebruikersnaam te versleutelen alvorens ze naar de webserver te sturen. Er zijn een aantal voorstellen voor betere beveiligings methodes geopperd zoals digest, maar er is nog geen echte standaard ontstaan. In plaats daarvan wordt voor beveiligde toegang gebruik gemaakt van een protocol dat de hele connectie, inclusief alle headers versleutelt.

De meest gangbare methode om de verbinding te versleutelen is op dit moment Secure HTTP (HTTPS), dat gebaseerd is op de Secure Socket Layer (SSL) en de nieuwere Transport Layer Security standaarden[1] [openssl]. Deze standaards definiëren een raamwerk waarin diverse beveiligingsprotocollen gebruikt kunnen worden nadat de uiteindes van de connectie daarover onderling overeenstemming hebben bereikt. In de meeste gevallen wordt een public key systeem gebruikt om de applicaties veilig aan elkaar te koppelen en wordt er daarna overgeschakeld op een eenvoudiger te coderen, niet op een publieke sleutel gebaseerd codering schema, om de processorbelasting voor het coderen en decoderen te minimaliseren. SSL en TLS bieden naast beveiligde verbindingen ook de mogelijkheid om certificaten uit te wisselen waarmee servers en/of toepassingen kunnen aantonen dat ze zijn wie ze beweren te zijn, als beveiliging tegen “man-in-het-midden” aanvallen.

Bij het introduceren van SSL in Netscape hebben de ontwikkelaars er voor gekozen om deze certificaten te gebruiken om webservers zich te laten identificeren, door te vereisen dat alleen certificaten van een aantal certificeringbureaus worden geaccepteerd die op de vereiste plaats de naam van de server hebben staan. Het doel hiervan is dat je als gebruiker er op moet kunnen vertrouwen dat de gegevens die naar een beveiligde website worden gestuurd echt bij de juiste ontvanger terechtkomen.

De SSL en TLS protocollen zijn zo ontworpen dat ze op veel plaatsen te hergebruiken zijn en met veel verschillende codering schema's overweg kunnen, maar dat gaat ten koste van de tijd die nodig is om een verbinding op te bouwen. Voor een gemiddeld HTTP verzoek aan een webserver zijn maar enkele honderden bytes van de client naar de server noodzakelijk voordat de webserver data terug kan gaan sturen. Bij Secure HTTP moet eerst aangegeven worden welke versie van het SSL protocol gebruikt wordt, daarna stuurt een partij zijn codering mogelijkheden naar de andere partij die daarop de keuze van het protocol maakt. Ten slotte wordt dan het gewone HTTP protocol over de beveiligde verbinding gebruikt. Met het certificaat uitwisselen er bij zijn er 4 extra synchronisatie stappen nodig bij het opbouwen van de verbinding. Het HTTP protocol kan vanaf versie 1.0 in veel gevallen echter wel de eenmaal opgebouwde verbinding hergebruiken voor volgende webverzoeken, maar omdat dit meestal alleen lukt voor statische pagina's en beveiligde pagina's juist vaak dynamisch zijn levert dit in dit geval niet zo veel verbetering op[2].

---

[1] De door Netscape geïntroduceerde SSL 2 en 3 zijn op dit moment het meest gangbaar.
TLS is een door het World Wide Web Consortium gestandaardiseerde opvolger die op termijn SSL moet gaan vervangen. (Zie [openssl]

[2] HTTP 1.0 [rfc1945] vereist voor het hergebruiken van de connectie dat voor aan de pagina het exacte aantal bytes staat dat een pagina lang is. HTTP 1.1 [rfc2616] heeft ook een andere mogelijkheid om de connectie te hergebruiken, waarbij de data dynamisch verdeeld wordt in kleinere blokken voorafgegaan door de lengte van het blok.

Voordat een systeem geschikt is om aan een andere webserver gekoppeld te worden moet het aan een aantal eisen voldoen. Verzoeken van een webserver hebben namelijk andere eigenschappen dan die van andere applicaties. Een webbrowser en een webserver staan niet continu met elkaar in verbinding en wisselen alleen op verzoek van de webbrowser informatie uit. Dat betekent dat de server applicaties ten alle tijden klaar moet staan om verzoeken van webbrowsers af te handelen, maar dat ze er nooit van uit kunnen gaan dat de webbrowsers ook inderdaad nog een verbinding opbouwen.

Voor gebruikers is het tegenwoordig echter heel vanzelfsprekend dat een webserver op de gebruiker toegespitste informatie levert. Hiervoor bieden de meeste webomgevingen hun eigen mogelijkheden die in de meeste gevallen meer toegespitst zijn op de architectuur van de webontwikkelomgeving dan op een portable methode. De implementaties verschillen ook qua persistentie van de gegevens. Microsoft gebruikt voor zijn ASP en ASPX technologieën een object dat in het geheugen blijft tot een vastgestelde periode na het laatste verzoek van de webbrowser. Het is mogelijkheid om zelf objecten toe te kennen aan een dynamische array op het sessie-object, die daardoor ook bewaard blijven tot na het verlaten van de website. Als de timeout verstreken is wordt het object automatisch vrijgegeven. Een groot nadeel van deze techniek is dat er slechts een webpagina tegelijkertijd gebruik kan maken van deze sessie objecten zonder dat de programmeur zich bezig moet houden met extra synchronisatie.

III. Windows DNA

Geschiedenis
In 1991 introduceerde Microsoft 'Object Linking and Embedding' (OLE) [Brockschmidt96] als een manier om gegevens uit Microsoft Excel te kunnen gebruiken in Microsoft Word en andersom. Hierbij werd de interface zo opgezet dat hij ook door andere applicaties te gebruiken was. Kort na de introductie groeide dit systeem tot de standaard manier om documenten van Windows applicaties te kunnen hergebruiken.

De ontwikkelaars van OLE bedachten dat deze manier van documenten samenstellen een specifieke vorm was van het gebruik van software componenten. OLE 2.0, dat op basis van deze ideeën ontwikkeld werd, was daardoor meteen de opvolger van een aantal andere eerder ontworpen specifieke component-technieken, zoals Visual Basic Controls (VBX) en Dynamic Data Exchange (DDE). Omdat de OLE componenttechnologie ontworpen werd als een uitbreidbaar en herbruikbaar systeem, was het onnodig om de OLE specificaties op zich verder uit te breiden in latere versies.

De strikt gedefinieerde componenten die met deze technologie gebouwd konden worden, hadden naast de verbeterde structuur ook het voordeel dat ze geschikt waren voor de geleidelijke migratie van een 16 bit naar een 32 bits architectuur zoals Microsoft dat voor ogen had met de introductie van Windows '95. In Windows '95 kreeg OLE een belangrijke plaats als de opvolger van de reeds beschikbare dynamisch geladen bibliotheken, doordat het besturingssysteem voor een groot aantal functies op de achtergrond veel gebruik maakt van dit soort componenten. Omdat de componenttechnologie niet langer alleen gebruikt werd voor het hergebruiken van document objecten werd de componenttechnologie achter OLE rond de introductie van Windows '95 omgedoopt tot COM.

Met de introductie van het server-besturingssysteem Windows NT 4.0, werd COM uitgebreid met de mogelijkheid om transparant objecten over een netwerk te kunnen gebruiken. Distributed COM (DCOM) maakt het mogelijk om op NT servers objecten te registreren die daarna op andere Windows of Windows NT computers gebruikt kunnen worden.

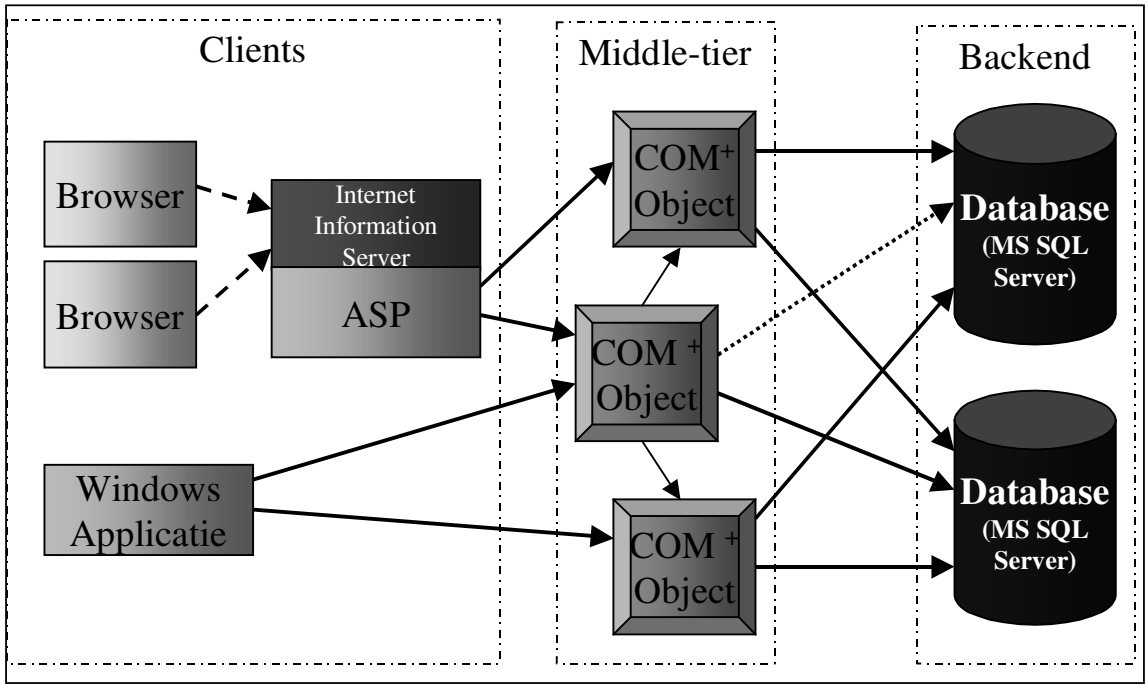
Met de opkomst van het Internet en de webbrowser Internet Explorer in het bijzonder heeft Microsoft de rol van COM binnen de protocol ondersteuning en user interfaces van Windows vergroot, om een volledig aanpasbare webbrowser te maken. Om deze mogelijkheden te benadrukken draagt COM in de web-enabled context vanaf dat moment ook de naam ActiveX.

Eind 1999 is Microsoft een combinatie van nieuwe Windows producten gaan positioneren als het enterprise systeem Windows Digital Nervoussystem Architecture[1] (Windows DNA). Dit model, dat in de standaardconfiguratie volledig uit Microsoft producten bestaat, moet het mogelijk maken om goed schaalbare oplossingen te bieden, omdat het op alle lagen van het model mogelijk is om het systeem uit te breiden zonder dat het systeem wezenlijk verandert.

Een groep NT servers biedt in dit schema zijn COM diensten aan een groep clients aan. De clients bestaan in dit schema uit COM applicaties of een webserver die via een scripttaal gebruik kan maken van DCOM objecten. De componenten kunnen op hun beurt via de reeds

[1] 'Digitaal zenuwstelsel architectuur'.

bestaande interfaces gebruik maken van het database backend. De schaalbaarheid van de middle-tier wordt gerealiseerd door de COM+ uitbreidingen op DCOM die in Windows 2000 server geïntroduceerd werden, waardoor een COM object niet meer noodzakelijk op een specifieke host hoeft te worden gecreëerd, maar ook op een middels load balancing geselecteerde host kan worden gecreëerd.



De huidige Windows programma's maken zo goed als allemaal wel ergens gebruik van COM, omdat dat bijna vereist is om een modern Windows programma te maken. Omdat COM binnen een applicatie vaak zelfs sneller is dan het gebruik van dynamische bibliotheken, wordt het zelfs gebruikt voor snelheidskritische functies zoals het direct aansturen van video hardware met DirectX. Onder grotere Windows applicaties is er daarnaast ook een tendens zichtbaar, dat er steeds vaker interfaces naar buiten worden aangeboden voor gebruik door andere applicaties of add-ins. Helaas worden deze mogelijkheden de laatste jaren ook steeds vaker gezien als lekken in de beveiliging.

Uitgangspunten
COM is ontwikkeld als een standaard voor samenwerking tussen Windows programma's in de tijd dat op PC's de meeste applicaties voor bijna alle functies hun eigen bibliotheken gebruikten om de computer zo efficiënt mogelijk te kunnen benutten. Voor het basisontwerp van COM waren daarom snelheid, herbruikbaarheid van de code en een generiek herbruikbare interface de belangrijkste punten.

COM maakt daarom onderscheid in objecten die fysiek aanwezig zijn binnen het proces dat ze gebruikt en objecten die alleen via een proxy object beschikbaar zijn. Een object binnen hetzelfde proces bevindt zich meestal in een dynamische bibliotheek die door de COM infrastructuur geladen wordt op het moment dat een applicatie gebruik wil maken van een component in die bibliotheek. Voor een object buiten het fysieke proces wordt op basis van de type-informatie van het object dynamisch een proxy object aangemaakt dat er voor de

applicatie uitziet als een gewoon com-object, maar alle functie aanroepen doorstuurt naar het eigenlijke object[1].

Om COM eenvoudig in bestaande systemen te kunnen integreren werd gekozen voor een ontwerp waarbij een COM object overeenkomt met een C++ class met alleen virtual functions. Een COM interface is daardoor niets meer dan een abstracte C++ class, met de functies QueryInterface, AddRef en Release plus de interface specifieke functies.

Deze drie functies, die samen ook de IUnknown interface vormen die ieder object moet ondersteunen, maken het mogelijk dat objecten meerdere interfaces beschikbaar kunnen stellen en zelf hun levensduur kunnen bepalen door reference counting.

De interface definities ondersteunen naast de bij de meeste RPC protocollen gangbare waarde types als integers, strings en databuffers ook typed enums, interface references en structures.

Programmeermodel
Een nieuwe interface krijgt bij het definiëren een wereldwijd unieke 128 bit sleutel, de Interface identifier (IID), die applicaties kunnen gebruiken om te verwijzen naar die specifieke interface definitie. Deze identifier wordt vervolgens in het windows register opgeslagen zodat andere applicaties informatie op kunnen vragen over deze interface. Volgens de specificaties mag er daarom nooit meer iets aan het object veranderd worden zonder dat die sleutel verandert, maar in de praktijk gebeurt dat helaas wel.

Ook classes krijgen bij het definiëren een 128 bit sleutel, de class identifier (CLSID) die gebruikt kan worden door andere applicaties. Voor classes wordt met deze sleutel vastgelegd welke interfaces een object van deze class ten minste ondersteunt. Een Windows applicatie kan na het vastleggen van deze sleutel in de registry objecten van deze class aanmaken door het aanroepen van de API functie CoCreateInstance(), met als argumenten een class identifier en de identifier van de interface die de applicatie wil gebruiken. Op de achtergrond zal COM er dan voor zorgen dat de juiste applicaties en/of bibliotheken geladen worden om het object te gebruiken.

Omdat C++ classes en functie pointers in C niet in alle omgevingen goed te gebruiken zijn, biedt COM ook een meer op scripttalen gerichte mogelijkheid om objecten te kunnen gebruiken. De IDispatch interface biedt een functie waaraan je een functienummer en een pointer naar een lijst van argumenten kunt meegeven. Als er voldoende type informatie voor handen is kan COM zelf de functieaanroepen omzetten in het C++ formaat, maar in andere gevallen zal de ontwikkelaar hier veel werk voor moeten doen.

De type informatie wordt door de meeste Windows ontwikkelomgevingen automatisch gegenereerd, maar het is ook mogelijk om zelf in de IDL taal definities te schrijven. Microsoft heeft zoveel mogelijkheden uitgedacht bij het definiëren van de type informatie dat de meeste programma's het merendeel van de eigenschappen simpelweg negeren.

In het C++ voorbeeld op de volgende pagina wordt kort weergegeven wat je in C++ moet doen om een eenvoudig COM object te gebruiken. Deze applicatie vraagt eerst aan de COM bibliotheek een nieuw 'Dictionary' object. Wanneer dit gelukt is wordt onderzocht of dit object de ISpell interface ondersteunt. Tenslotte wordt een functie op dit object aangeroepen

[1] Om een maximale performance te behalen is het mogelijk zelf een proxy class te maken.

om de spelling van een stuk tekst te controleren. Wanneer je dit voorbeeld vergelijkt met het stukje Javascript code dat bij het webkoppeling onderdeel staat is goed te zien dat het gebruik

```
/* C++ sample code to create and use a simple COM Object */

// Check the specified text for spell errors
bool CheckSpelling(BSTR pText)
{
    IUnknown* unk;
    ISpell* spl;
    bool ok;

    if (!SUCCEEDED(CoCreateInstance(CLSID_Dictionary, NULL, CLSCTX_ALL, IID_IUnknown, &unk))
        abort("Dictionary not installed");

    if (!SUCCEEDED(unk->QueryInterface(IID_ISpell, &spl))
    {
        unk->Release(); // Release reference before abort
                        // (This object might exist outside this application)

        abort("Dictionary doesn't support ISpell Interface");
    }

    unk->Release();

    ok = (S_OK == spl->CheckText(pText));
    spl->Release();

    return ok;
}

// Note: CoCreateInstance can call QueryInterface for you when you change the 4th parameter
//       to the Identifier of the Interface you need
```

van COM een stuk eenvoudiger is vanuit talen die er standaard ondersteuning voor hebben, dan vanuit talen waarin je zelf verantwoordelijk bent voor de reference counting.

Om het gebruikt van COM eenvoudig te maken heeft Microsoft in de laatste jaren een scala aan programmeertalen gelanceerd. Zo kunnen natuurlijk de oorspronkelijk voor gewone applicaties bedoelde Visual Basic (VB) en Visual C++ (VC++) overweg met deze interfaces. Daarnaast hebben de office applicaties allemaal hun eigen programmeertalen gehad, maar die zijn intussen allemaal vervangen door het veel meer op Visual Basic lijkende Visual Basic for Applications (VBA).

Voor het gebruik in Internet Explorer en de Microsoft Internet Information Server (IIS) heeft Microsoft daarnaast ook nog een Ecmascript interpreter en een derde Visual Basic dialect genaamd Visual Basic Script (VBS) geïntroduceerd. Deze talen zijn echter in tegenstelling tot de eerder genoemde talen niet bedoeld voor gewone applicaties, maar als interpreters die binnen andere applicaties gebruikt kunnen worden.

In een poging om Windows te voorzien van een moderne scripttaal heeft Microsoft voor deze laatste talen een Windows Scripting Host (WSH) gebouwd die qua mogelijkheden niet onder doet voor Unix shell scripts. Met wat meer moeite is het zelfs mogelijk dit soort programma's van een eenvoudige grafische gebruikersinterface te voorzien. De mogelijkheden die deze applicatie bood zijn echter nooit algemeen ingezet als taal voor het bouwen van eenvoudige tools doordat de documentatie niet meegeleverd werd. Door het eenvoudige gebruik van ActiveX automation is deze applicatie het laatste jaar uitgegroeid tot een taal die vooral gebruikt wordt door virusbouwers die Outlook Express willen misbruiken om virussen te verzenden.

Beveiliging & Verificatie
Omdat het merendeel van de componenten nooit gebruikt worden buiten de directe gebruikersomgeving laat COM de beveiliging over aan de componenten zelf en biedt het alleen hulp bij het identificeren van de gebruiker. Objecten die in een distributed omgeving worden gebruikt, zoals bij DCOM of die gebruikt worden door niet vertrouwde code kunnen aan COM informatie opvragen over de aanroepende code. Eventueel kan er dan door COM ook aan de client om extra informatie gevraagd worden voordat bepaalde acties worden uitgevoerd. Een ontwikkelaar kan hierdoor in principe zijn eigen verificatieschema bouwen, maar meestal wordt gebruikt van de standaard Windows beveiliging die alleen op de login van de gebruiker let.

Wanneer een service wegvalt kan de proxy code dit aan de applicatie doorgeven. Functies met een gestandaardiseerde resultaatcode als returnwaarde geven in dit geval een error code terug die aangeeft dat het object niet langer beschikbaar is. Wanneer andersoortige functies aangeroepen worden, wordt teruggevallen op een besturingssysteem exceptie om de applicatie op de hoogte te stellen.

Webkoppeling: ASP - Active Server Pages
Voor het dynamisch generen van websites biedt Microsoft de Active Server Pages (ASP) technologie aan. Dit is een scripttaal die net als bij PHP er van uit gaat dat het merendeel van het bestand uit kale HTML code bestaat die op bepaalde plaatsen aangevuld wordt met elementen die door een stuk sourcecode opgebouwd worden.

Standaard ondersteunt ASP de eerder genoemde geïnterpreteerde talen VBScript en EcmaScript, maar door de implementatie als gespecificeerde COM bibliotheken zijn er andere taalmodules toe te voegen. De bekendste ActiveX Script engine die ook de vereiste ASP extensies ondersteunt is de Windows Perl implementatie van ActiveState.

Bij het ontwerpen van ASP heeft Microsoft er voor gekozen om een centrale parser de code te laten splitsen voor de verschillende talen en die parser is dan verantwoordelijk voor het verspreiden van de code onder de verschillende script interpreters. Dit heeft als voordeel dat verschillende stukken code door verschillende talen kunnen worden geparsed en er dus meerdere programmeertalen binnen een bestand ondersteunt worden. De verschillende parse-niveaus zorgen er echter wel voor dat er tijdens het executeren van de code telkens een groot aantal context switches gemaakt moeten worden tussen de verschillende interpreters voordat de HTML pagina compleet beschikbaar is voor de webserver.

Op de volgende pagina staat een stukje ASP code dat laat zien hoe Microsoft's webserver IIS het mogelijk maakt om COM objecten te gebruiken voor het genereren van HTML pagina's[1]. Er wordt in dit voorbeeld gebruik gemaakt van een server object dat het script van de informatie voorziet die nodig is om een advertentie weer te geven.

[1] ASP makt het ook mogelijk dat COM objecten direct tekst naar de webserver sturen, maar die functie zou dit voorbeeld overbodig maken.

```
<!-- ASP Adrotator Voorbeeld -->
<html>
<head>
    <title>ASP Demo- Advertentie test</title>
</head>
<body bgcolor=white>
<h2>Welkom</h2>
<!-- Advertentie header -->
<center>
<script language=JavaScript RunAt=Server>
    var ad;

    ad = Server.CreateObject("MyLib.AdRotator");
    ad.SelectRandom();
    Response.Write("<a href=\"" + ad.GetAdLocation() + "\" target=\"_blank\">");
    Response.Write("<img src=\"" +ad.GetAdImg()+ "\" width=468 height=60>");
    Response.Write("</a>");
</script>
</center>
<!--/Advertentie header -->
<p><br></p>

[pagina body]
</body>
</html>
```

Case Studie
Omdat DCOM geen techniek is die specifiek bedoeld is voor het ontwikkelen van web-toepassingen en ik vooraf al de nodige ervaring had met COM heb ik voor dit afstudeerproject op zichzelf niet veel met COM gewerkt. Voor het Programmeeromgevingen II college heb ik ongeveer een half jaar voor het starten van dit project een DCOM adapter voor de ToolBus gebouwd. Ter volledigheid wordt deze adapter beschreven int hoofdstuk VII.

Vooruitzichten
COM is een techniek die de fundering vormt van een zeer groot deel van alle Microsoft Software. Het ziet er daarom niet naar uit dat COM snel zal verdwijnen, maar de techniek zal wel langzaam naar de achtergrond verdwijnen als de DotNet technologie aanslaat.

Voor het ontwikkelen van op componenten gebaseerde applicaties wil Microsoft steeds meer gebruik gaan maken van DotNet, maar dat betekend niet dat COM verdwijnt. De Windows versie van DotNet maakt namelijk intern erg veel gebruik van COM. Alle DotNet objecten zijn in principe COM objecten met meer type informatie en deze objecten zijn daarom ook zeer eenvoudig te gebruiken vanuit oudere applicaties die nog niet in de DotNet omgeving draaien.

IV. DotNet

Geschiedenis
Microsoft heeft een paar jaar terug besloten om het aantal technologieën voor het bouwen en koppelen van componenten te beperken, want het onderhouden van drie Visual Basic varianten, enkele andere scripttalen en diverse bibliotheken voor Visual C++ werd een steeds grotere aanslag op de resources. Met de overgang van een 32 bit naar een 64 bit architectuur in zicht hebben ze er voor gekozen de principes van het ontwikkelen van Windows software nog eens onder de loep te nemen. In de eerste helft van 2000 heeft Microsoft hun nieuwste ontwikkelfilosofie, Microsoft.NET voor het eerst aan de pers getoond.

Uitgangspunten & programmeermodel
Met de invoering van DotNet wil Microsoft het voor ontwikkelaars makkelijker maken om op componenten gebaseerde software te bouwen en tegelijkertijd de herbruikbaarheid van code vergroten. Om dit mogelijk te maken is er een gestandaardiseerde virtual machine ontwikkeld, die geoptimaliseerd is om object-georiënteerde code vanuit diverse programmeertalen tegelijkertijd te kunnen gebruiken. Bij het ontwerp heeft de nadruk gelegen op compatibiliteit, herbruikbaarheid, versiebeheer, beveiliging, snelheid en eenvoud.

De DotNet architectuur bestaat uit een aantal lagen, die door Microsoft tegelijkertijd voor standaardisatie ingediend zijn bij het Europese standaardisatie instituut Ecma. Aan de basis van het totale systeem ligt het Common Language Typesystem (CLT). Dit is een systeem-model waarin programmatuur volgens diverse paradigma's kan worden uitgedrukt.

Applicaties die voor dit systeem ontwikkeld zijn, worden door de compiler niet omgezet in gewone machine code zoals bij een gewone compiler, maar in Common Intermediate Language (CIL) instructies. De bytecodes van deze CIL zijn zo ontworpen dat ze eenvoudig naar machinecode zijn te converteren, wanneer de programmacode uitgevoerd kan worden. Het idee hier achter is dat de processorspecifieke optimalisaties bij het uitvoeren van het programma kunnen worden uitgevoerd, in plaats van slechts bij het compileren van het programma.

De Common Language Runtime (CLR) heeft daarom op iedere PC architectuur een verschillend Virtual Execution System (VES), dat specifiek ontwikkeld is voor het systeem waar het op draait. Microsoft heeft aangekondigd zelf de runtime omgeving uit te gaan brengen in tenminste vijf verschillende versies. De DotNet CLR komt in eerste instantie beschikbaar in een aantal varianten voor de 32 en 64 bit Windows versies voor Windows CE. Daarnaast gaat Microsoft in samenwerking met Corel een shared-source[1] inplementatie voor FreeBSD ontwikkelen, die speciaal bedoeld is als referentie-implementatie en als studie model voor educatieve instellingen. Daarnaast zijn er intussen drie verschillende open source initiatieven opgezet om ook een CLR te krijgen onder de gangbare Unix versies.

Door het gebruik van de CIL codes heeft de runtime omgeving niet alleen de beschikking over de bytecode, maar ook over type informatie en zogenaamde metadata die extra eigenschappen van de code kan definiëren. Deze metadata wordt binnen het CLT gebruikt

[1] Microsoft's variant van open source, waarbij de originele eigenaar van de sourcecode meer vrijheden heeft dan bij de gangbare vormen van open source.

voor diverse toepassingen in de verschillende stadia van de executie en bevat ook informatie die vooral nuttig is voor compilers.

De CIL code biedt de runtime-omgeving veel vrijheid bij het implementeren van het Virtual Execution System, door programmastructuren als het aanroepen van functies en de geheugen-locatie van variabelen volledig aan de runtime-omgeving over te laten. Deze definitie maakt runtime stack traces en een op de callstack gebaseerde beveiliging mogelijk. Op systemen die voldoende resources hebben kan de VES voor de vaak gebruikte code zoals de standaard-bibliotheken de processor-afhankelijke instructies opslaan in een diskcache.

Het CLT heeft hoogstwaarschijnlijk veel meer mogelijkheden dan iedere zelfstandige programmeertaal op zich zal benutten. Er is daarom binnen het typesysteem een subset gedefinieerd die de eisen bevat waaraan code minimaal moet voldoen om binnen de runtime omgeving samen te kunnen werken met andere code die aan deze standaard voldoet. Deze Common Language Specification (CLS) bevat allerlei regels waar de compiler- en library-bouwers zich aan moeten houden om CLS compatible te blijven.

Bijna alle eisen van het CLS gaan over de interconnectie met andere componenten binnen het VES en niet over de implementatie van de componenten zelf. Het CLS bevat naast deze eisen ook een uitgebreide standaardbibliotheek die aan deze eisen voldoet en daardoor te gebruiken is onder alle CLS compatible talen.

De CLS bibliotheek is zo ontworpen dat per component de beveiliging in niveaus regelbaar is. Voor een functie-aanroep die alleen door vertrouwde code uitgevoerd mag worden wordt bij de aanroep van de betreffende functie de stack onderzocht om te achterhalen welke code verantwoordelijk is voor de aanroep. Er wordt net zo lang gezocht tot er op de stack een referentie naar een CodeAccessPermission class wordt gevonden van de juiste soort.

De CLS biedt voor het bouwen van callbacks naast de uit Java bekende interfaces ook events en delegates[1]. Een delegate is het beste te vergelijken met een typesafe variant van een functie-variabele uit C die zowel naar een statische functie als naar een instance functie van een class kan wijzen. Een event is een speciaal variabele type dat een of meer delegates kan bevatten en deze dan met een aanroep allemaal aan kan roepen als een soort broadcast.

Op het CLS niveau bestaan er in tegenstelling tot in de Java Virtual Machine geen value types zoals integers en strings, maar alleen objecten. De intermediate language IL is echter zo gedefinieerd dat de VES de verwachtte nadelen van deze keuze eenvoudig weg kan optimaliseren. Deze keuze maakt het mogelijk om op een gestructureerde manier runtime informatie over alle objecten beschikbaar te maken en maakt het daarnaast voor debuggers en de beveiligingscode eenvoudiger om stack frames te onderzoeken.

Microsoft heeft er bij de DotNet implementatie voor gekozen om hun versie van de CLS library binnen de CLR zelf te ontwikkelen omdat code binnen de CLR sneller ontwikkeld kan worden en om er direct voor te zorgen dat deze code ook onder hun andere CLR implementaties werkt. Marketing technisch is dit ook handig omdat ze daarmee aangeven zelf vertrouwen te hebben in hun nieuwe ontwikkelingen[2].

---

1 Deze taalconstructie die eerder in de Microsoft versie van de Java Runtime Environment werd geïntroduceerd is mede de oorzaak van het conflict over het bouwen van Java runtime omgevingen.
2 Sun heeft de Java runtime library voor het grootste gedeelte in C geïmplementeerd.

```
/* Event sample in C# - Tested in C# Beta 1  */
using System;

public class EventTest
{
       // Event definitie
       delegate void MyDelegate(int arg);     // Functie type
       event MyDelegate myEvent;              // event variable

       // Een instance functie met een integer als argument
       void OnMyMemberEvent(int x) {
             Console.WriteLine("OnMyMemberEvent:  {0}", x);
       }

       // Een statische functie met een integer als argument
       static void OnMyStaticEvent(int x) {
           Console.WriteLine("OnMyStaticEvent:  {0}", x);
       }

    public static int Main(string[] args)
    {
             // Instantieer een event variabele
             EventTest tst = new EventTest();

             // Voeg delegates aan het event toe
             tst.myEvent += new MyDelegate(OnMyStaticEvent);
             tst.myEvent += new MyDelegate(tst.OnMyMemberEvent);

             // Roep de functies aan
             tst.myEvent(12);

        return 0;
    }
}
/*
 * Uitvoer:
 * OnMyStaticEvent:  12
 * OnMyMemberEvent:  12
 */
```

Binnen de DotNet omgeving is het direct mogelijk om COM objecten aan te spreken alsof het CLS objecten zijn. Om nieuwe object definities geschikt te maken voor de CLR is het alleen nodig om een conversietool de typelibrary te laten converteren in een class met attributen die de VES kan gebruiken om de juiste koppeling tot stand te brengen. Andersom is het ook direct mogelijk om CLS objecten buiten de CLR omgeving te gebruiken als gewone COM objecten.

Aan de DCOM specificatie zijn de laatste jaren zo veel toeters en bellen toegevoegd dat Microsoft er voor gekozen heeft om een aantal van de mogelijkheden niet aan te bieden binnen het CLS, omdat anders de implementatie eisen voor de verschillende compilers te groot zouden worden.

Objecten binnen de CLS moeten aan striktere eisen voldoen dan standaard COM objecten, omdat de CLS omgeving de herbruikbaarheid over verschillende versies wil kunnen garanderen. De mogelijkheden die COM via de IDispatch interface beschikbaar stelde voor standaard argumenten en standaard properties vervallen ten gunste van een vorm van functie

overloading zoals die in Java beschikbaar is[1]. Er is wel een mogelijkheid om functies een variabel aantal parameters te geven, maar de CLS eist dat die niet gebruikt wordt voor standaard libraries.

Met de introductie van de CLR heeft Microsoft de moeilijke keuze gemaakt om de omgeving in tegenstelling tot standaard COM mark and sweep garbage collected te maken. Veel gebruikers van de oudere Microsoft programmeeromgevingen vinden dit een onvergeeflijke stap omdat dit het voorheen deterministische gedrag van hun code doorbreekt. Microsoft vond dit echter een fundamentele keuze die gemaakt moest worden omdat dit de betrouwbaarheid van nieuwe code ten goede komt en een betere beveiliging tegen fouten mogelijk maakt.

In een pure garbage collected omgeving behoren geheugenlekken tot het verleden en is het niet mogelijk reeds vrijgegeven objecten alsnog te adresseren. Maar helaas zijn er kunstgrepen noodzakelijk om de garbage collected wereld binnen de CLR en de reference counted wereld buiten de CLR op een juiste manier in twee richtingen te kunnen koppelen zonder memory leaks.

CLR Objecten bevatten een finalizer functie die aangeroepen wordt wanneer er niet meer naar het object gerefereerd wordt. Omdat de volgorde van het garbage collecten niet vast ligt is het dan echter niet meer zeker of de objecten waar het object zelf aan refereerde nog bestaan. Voor veel objecten die naar resources buiten de CLR verwijzen blijft het dus helaas nog van belang dat ze door de programmeur op volgorde worden vrijgegeven als niet zeker is dat dit ook automatisch op de juiste manier en op het juiste moment gebeurt. Ook voor schaarse operating system resources zoals window en file handles is het in sommige gevallen handiger om ze direct zelf vrij te geven in plaats van te wachten tot garbage collection optreedt. Dit probleem is helaas niet op te lossen zonder dat het hele operating system en alle er aan gelinkte systemen mark and sweep garbage collected zijn.

Binnen de CLR library worden tal van web standaarden geïmplementeerd om de functies daarvan aan alle applicaties beschikbaar te stellen. XML, SOAP, XSL, HTTP, FTP en gecompileerde reguliere expressies zijn een paar van de kernwoorden waarmee de Microsoft marketeers duidelijk willen maken dat de CLR een moderne functie bibliotheek bevat. Naast deze library bevat de Micrsoft versie van de CLR een aantal extensies die het mogelijk maken om de managed omgeving binnen bestaande applicaties te kunnen gebruiken en om de user interface van nieuwe applicaties via de gewone Windows API te kunnen benaderen.

Beveiliging & Verificiatie

Binnen de CLS specificatie wordt er onderscheid gemaakt tussen twee soorten objecten: verifieerbare en niet-verifieerbare. De verifieerbare objecten maken alleen gebruik van een eenvoudig statisch controleerbare subset van de IL-code en kunnen daardoor alleen gebruik maken van CLS objecten. Omdat de classloader voor de executie precies kan controleren wat het object doet is het mogelijk om objecten in deze klasse in een beveiligde omgeving uit te voeren zonder dat ze daar schade kunnen berokkenen. Dit soort objecten is dus bijvoorbeeld net als Java applets veilig te downloaden van een webserver om te executeren binnen een webbrowser, maar het is ook toepasbaar om de computer beter te beveiligen tegen gewone toepassingen.

---

1 Programmeertalen zijn vrij om default argumenten toe te staan, maar het is niet mogelijk om deze functionaliteit generiek te implementeren binnen de CLS.

Niet verifieerbare objecten kunnen naast IL code ook processor specifieke machine code bevatten. Dit type objecten is speciaal bedoeld voor het koppelen van de CLR aan andere omgevingen buiten de VES, maar ook objecten die de runtime informatie van andere objecten gebruiken om die indirect uit te voeren vallen hier onder. Daarnaast is het om diverse redenen mogelijk om IL te genereren die niet eenvoudig verifieerbaar is en daarom niet onder de andere categorie kan vallen. De eerste versie van Microsoft Visual C++ met IL ondersteuning is bijvoorbeeld niet in staat om verifieerbare code te produceren, omdat er een aantal niet verifieerbare instructies gebruikt wordt om een C++ compatible stack te bouwen[1]. Het is voor een compiler niet noodzakelijk om verifieerbare objecten te kunnen genereren om aan de CLS specificatie te voldoen.

Wanneer een object niet langer beschikbaar is omdat het applicatie domein of de verbinding wegvalt zullen aanroepen naar het niet langer bestaande object een exceptie tot gevolg hebben.

In de media lijkt het of DotNet vooral wordt gepresenteerd als een methode om software abonnementen mogelijk te maken, maar dat is zoals hiervoor beschreven is maar een van de vele mogelijkheden. Met het aankondigen van een product onder de codenaam 'Hailstorm' laat Microsoft zien dat zij veel verwachten van deze technologie. Hailstorm moet het op termijn mogelijk maken om diverse diensten op een beveiligde manier via het Internet af te nemen. Een client applicatie die in een beveiligde omgeving op de pc van de gebruiker draait werkt in dat geval als een interface naar een back-office applicatie die ergens op een server draait. Microsoft zou het liefste zien dat dit soort initiatieven gebruik zouden maken van de Passport® dienst van Microsoft, maar Sun en Gnu zijn al begonnen aan het ontwikkelen van alternatieve systemen.

Webkoppeling

Ook voor het dynamisch genereren van website content heeft Microsoft met DotNet nieuwe mogelijkheden gedefinieerd. De oude ASP technologie die gebruikt werd met de talen Visual Basic Script en EcmaScript (dat beter bekend is onder de naam Java Script) is vervangen door een nieuwe ASPX technologie die volledig onder de .NET engine draait. Hierbij heeft Microsoft er voor gekozen VBScript niet verder te ontwikkelen en op alle plaatsen te vervangen door de 'echte' Visual Basic compiler. Een aantal mogelijkheden die VBScript wel had, maar de gewone Visual Basic compiler nog niet ondersteunde zijn op dat moment alsnog aan toegevoegd. EcmaScript is voor DotNet opgewaardeerd tot een volledig CLS compatible taal en er is ook ondersteuning voor C# toegevoegd aan ASPX.

Er wordt verwacht dat de meeste ASP code niet direct te gebruiken is onder ASPX zonder de nodige aanpassingen en Microsoft heeft er dan ook voor gekozen om standaard tegelijkertijd het oudere ASP en het nieuwere ASPX te ondersteunen op hun webserver IIS.

Als een pagina voor de eerste keer opgehaald wordt dan wordt het .aspx bestand gecompileerd tot IL code die daarna direct naar de JIT-Compiler en de IL cache gaat. Vanaf dat moment kan ieder verzoek naar die pagina afgehandeld worden met dezelfde snelheid als een stuk normale IL code. Het is op dit moment nog niet toegestaan de snelheid van dit systeem te vergelijken met die van andere systemen omdat Microsoft dit expliciet verbiedt in de licentie voorwaarden van de beta versie van de .NET omgeving.

---

1 In de beta versie wordt standaard ook de C Runtime library nog geinitialiseerd door assembly code, maar volgens het Microsoft Visual C++ compiler team is dat niet de reden waarom ze gekozen hebben voor de niet verifieerbaarheid.

Case Studies
Van de DotNet omgeving was in de tijd dat ik er mee experimenteerde net de eerste beta beschikbaar. Het was daarom nog niet mogelijk om boeken te gebruiken voor het bestuderen van de werking van deze omgeving. Ik had het geluk dat ik al kort na het uitkomen van de eerste Beta van Visual Studio.NET kon beschikken over deze ontwikkelomgeving, omdat deze omgeving al over een grote hoeveelheid voorlopige informatie beschikte[1].

Om inzicht te krijgen in de werking van de DotNet omgeving heb ik wat experimenten gedaan met de beta versies van C#, Visual Basic.NET en Microsoft Visual C++.NET. Met wat kleine demo programma's was redelijk te achterhalen wat de mogelijkheden van de standaard-bibliotheek waren.

Om meer inzicht te krijgen in de werking van de CLR heb ik daarnaast ook de technische documentatie bestudeerd waarin beschreven wordt hoe de CIL code in elkaar zit. Het meest opvallende hierin vond ik de overeenkomsten met de Intel 64 bitarchitectuur IA-64 [ia64]. Om in de IA-64 architectuur efficient gebruik te kunnen maken van de dynamisch gereserveerde registers moeten namelijk bij de aanroep van een functie een aantal constanten gedefinieerd zijn, die bij de CIL precies in de functie definitie opgenomen zijn.

Na voldoende inzicht te hebben gekregen in de werking van deze techniek heb ik de Java adapter voor de ToolBus geconverteerd naar C#. De beschrijving van deze conversie wordt beschreven als een van de projecten in hoofdstuk VII.

Vooruitzichten
Het is op dit moment nog niet goed te voorzien, hoe DotNet zijn plaats in de markt gaat vervullen. Er is een groeiende interesse uit de markt voor wat DotNet te bieden heeft, maar er zijn buiten Microsoft met de kort geleden vernieuwde MSDN website nog geen bedrijven die DotNet toepassen in grote systemen.

Steeds meer applicaties kiezen er nu al voor om bestaande op COM gebaseerde producten beter geschikt te maken voor gebruik vanuit DotNet applicaties, maar het is nog te vroeg om een verschuiving naar DotNet te zien.

De laatste weken voor het afronden van deze scriptie zijn een aantal open source projecten gestart die onderdelen van de DotNet als open source code willen gaan aanbieden. Het Amerikaanse bedrijf Ximian is op dit moment het verste met het realiseren van hun plannen. Zij proberen in een aantal fases een complete implementatie van de CLR en de CLS te bouwen. Voor dit 'Mono' project [mono] worden ondertussen ook al compilers voor C# en MonoBasic gebouwd.

Intel is als research project begonnen met het implementeren van de CLS standaard bibliotheek [ocl]. Bij dit project is het plan een platformonafhankelijke basis te bouwen, waarin experimentele compilertechnieken getest kunnen worden. Op dit moment is er nog geen systeem beschikbaar dat in staat is om deze bibliotheek te gebruiken. Waarschijnlijk heeft Intel een eigen virtuele machine voor Java die ze op dit moment aan het geschikt maken zijn voor de CIL bytecodes. De sourcecode verwijst op een aantal plaatsen naar

[1] Intussen is ook de documentatie van Beta 2 en de tekst zoals hij ingediend is bij Ecma beschikbaar op de website van Microft, maar dit project is nog gebaseerd op de eerste beta versie.

compatibiliteit met Java, maar er is op dit moment nog geen informatie beschikbaar over de plannen die ze bij Intel hebben.

Tenslotte is ook Gnu nu begonnen met een systeem waar een CLR een onderdeel van moet gaan vormen, genaamd ‘DotGNU Portable.NET’ [portable.net]. Het doel van dit project is een systeem dat overweg kan met vele soorten bytecodes. In eerste instantie wordt support ingebouwd voor Java bytecodes en CIL, maar het is de bedoeling in een later stadium nog meer just in time compilers op te nemen in het systeem.

V. J2EE - Enterprise Java

Geschiedenis
Sun heeft sinds het uitbrengen van Java geprobeerd om Java op alle terreinen in de software ontwikkeling binnen te laten dringen. Na de introductie van ontwikkelomgevingen voor eenvoudige systemen en embedded systemen is Enterprise Java (J2EE[1]) geïntroduceerd als de Java oplossing om gedistribueerde applicaties te bouwen.

Uitgangspunten
Sun's Enterprise Java specificatie definieert een architectuur waarmee gedistribueerde Java server toepassingen ontwikkeld kunnen worden. De architectuur wordt in deze specificatie globaal gestandaardiseerd om het mogelijk te maken dat verschillende bedrijven compatible implementaties kunnen bouwen.

Het design is gebaseerd op de al eerder door Sun geïntroduceerde JavaBeans. Een JavaBean is een component bestaat uit één of meer Java classes die aan een aantal eisen voldoen, om ze op een gestandaardiseerde manier benaderbaar te maken vanuit zowel een ontwikkelomgeving als de runtime omgeving. De normale JavaBeans worden vooral gebruikt in applicatie frameworks om op een eenvoudige wijze losse extensies in een applicatie op te nemen.

In veel Java ontwikkelomgevingen, worden deze mogelijkheden benut om automatisch code te genereren. Met de informatie die een JavaBean aanbiedt over events en properties die betrekking hebben op een object is het mogelijk design tools te bouwen die ook met niet-standaard aanwezige componenten overweg kunnen. De Java gebruikers die alleen van commandoregel Java compilers gebruik maken en niet van een speciaal op Java gerichte ontwikkelsuite, gebruiken op dit moment nog niet vaak JavaBeans, omdat die in deze situatie weinig voordelen hebben. In ontwikkelomgevingen zijn de JavaBeans echter de belangrijkste functie die Rapid Application Development (RAD) mogelijk maken.

Om een Java object te veranderen in een eenvoudige JavaBean is het in de meeste gevallen niet nodig om veel code te veranderen. Op zich is een JavaBean namelijk niet meer dan een Java class die een of meer specifieke interfaces uit het java.beans package implementeert. In veel gevallen is het echter ook noodzakelijk om een aantal functies van naam te veranderen, omdat er dan extra functionaliteit beschikbaar komt. Zo kunnen JavaBean objecten instelbare properties op objecten representeren door 'get_<propertyNaam>()' en 'put_<propertyNaam>()' functies te gebruiken[2]. Deze properties komen hierdoor beschikbaar in property browsers in diverse omgevingen.

Enterprise JavaBeans bestaan in tegenstelling tot gewone JavaBeans uit meer dan alleen aanpassingen binnen Java zelf. Een minimale EJB bestaat uit twee specifieke interfaces, een implementatie class die deze interfaces implementeert en een bestand met informatie over hoe de EJB geïnstalleerd moet worden. Voor het gebruik van Enterprise Java is echter meer nodig dan het ontwikkelen van deze code.

---

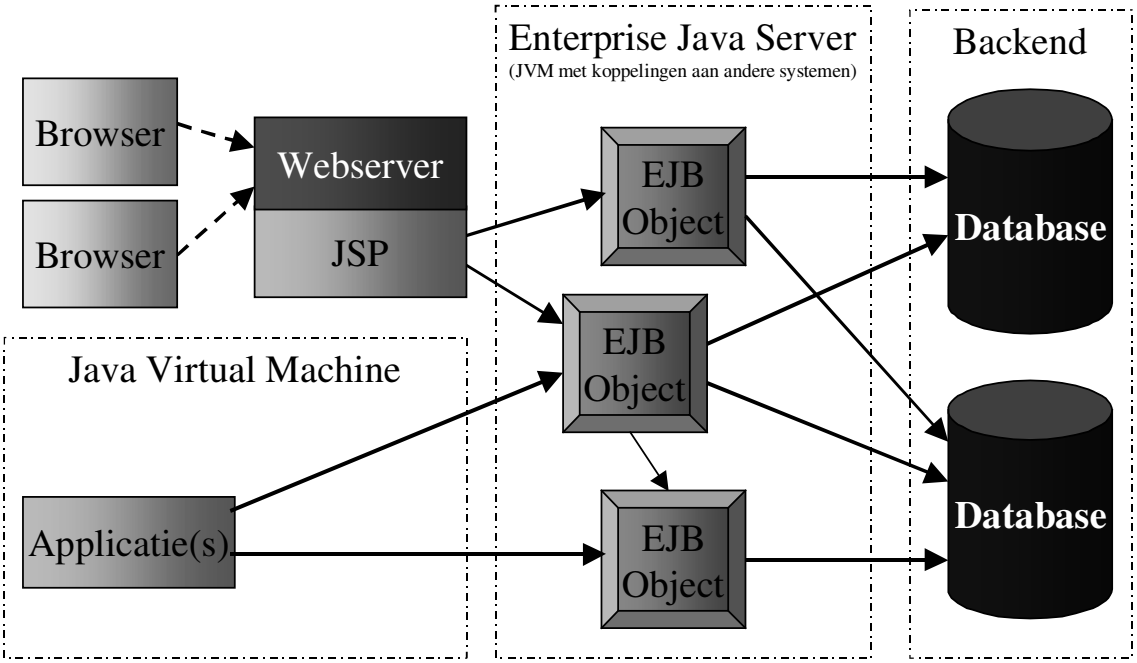
[1] Java 2 platform, Enterprise Edition.
[2] In de Java versies na het publiceren van de JavaBean specificatie heeft Sun in de Java runtime-library een aantal functies hernoemt om aan dit patroon te kunnen voldoen.

Enterprise Java is helaas geen standaard, waar je zo maar gebruik van kan maken, onder Sun's motto: 'Write once, run everywhere.' Enterprise Java is namelijk geen binaire standaard, maar alleen een enorme hoeveelheid documenten die beschrijven hoe je een Enterprise Java kunt bouwen. De specificaties zijn de afgelopen jaren geïmplementeerd door diverse producenten als IBM, BEA en Oracle. Helaas zijn deze systemen door de grote vrijheid die de Enterprise Java specificatie aan de ontwikkelaars gaf hooguit op source code niveau met elkaar compatible.

Omdat Sun dit probleem zag ontstaan hebben ze er voor gekozen om op een later tijdstip alsnog een aantal van deze gaten in de specificatie te vullen. De meeste producenten hebben echter gekozen om op de door hen ingeslagen weg verder te gaan, en hun systemen niet aan te passen aan de nieuwe specificaties. In de praktijk betekent dit dat een EJB die ontwikkeld is voor een versie van een Enterprise Java server niet direct overweg kan met andere servers, anders dan door aanpassingen aan de code.

Programmeermodel
De Enterprise Java architectuur is zo ontworpen dat objecten die beschikbaar zijn in de Enterprise Server ook in andere Java virtual machines te gebruiken zijn. Een architectuur-overzicht bestaat daarom uit tenminste twee Java vitual machines die met elkaar verbonden zijn via een remote procedure protocol. Een Java of JSP applicatie[1] kan door gebruik te maken van een aantal diensten die de Enterprise omgeving biedt een connectie opbouwen met een enterprise server. Over deze verbinding kan de applicatie nieuwe objecten op de server aanmaken of verbinding maken met bestaande objecten.

Voor deze objecten worden dan op de client dynamisch proxy objecten gedefinieerd die het mogelijk maken om functies op de achterliggende objecten op te roepen alsof het object zich in de omgeving zelf bevindt. Hoe deze verbinding tot stand gebracht wordt hangt af van de implementatie van de Enterprise server, omdat dit een van de niet gedefinieerde punten van de



[1] JSP: Java Server Pages. Meer informatie over deze technologie volgt onder het kopje webkoppeling.

specificatie is. In de meeste J2EE implementaties wordt gebruik gemaakt van vereenvoudigde varianten van Corba, omdat het dan mogelijk is bestaande niet-Java producten te integreren in de Enterprise Java omgeving zonder ze te converteren naar Java.

Een Enterprise Java server bestaat uit een aantal onderdelen die het mogelijk maken om op een gestructureerde manier objecten aan te maken. Een centrale registratiedatabase registreert welke objecten de server aan kan maken en welke gebruikers er dan gebruik van moegen maken. Met behulp van een losse applicatie is het mogelijk om een object met alle bij-behorende interfaces te registreren.

Een Enterprise Bean bestaat uit minstens 3 onderdelen, maar bevat er in de praktijk meestal meer. Het belangrijkste is de Sessie Bean, die het eigenlijke object in het server proces vormt. Dit object bevat de implementatie van de functies die door de client applicatie gebruikt kunnen worden. Naast dit object moeten er twee interfaces gedefinieerd worden: Een 'remote interface' die de functies bevat die de Bean beschikbaar stelt naar de externe servers en een 'home interface' die een of meer create() functies bevat voor de Bean die gebruikt kunnen worden als een soort constructor.

Deze interfaces worden in het client proces dynamisch geïmplementeerd met functies die alle functie aanroepen doorsturen naar de Server Bean.

Hieronder wordt in een voorbeeld weergegeven welke stappen er in een Java applicatie genomen moeten worden om een server object te gebruiken. Eerst wordt een server context aangemaakt die de connectie met de server representeert. Daarna wordt aan de server een home interface van het server object gevraagd. Met de functies op deze home interface is het tenslotte mogelijk om een object aan te maken dat gebruikt kan worden als een Java object dat lokaal aanwezig is in de Java virtual machine.

```
import javax.ejb.*;
import javax.naming.*;
import java.rmi.*;
import java.util.Properties;

public class DemoClient {
  public static void main(String[] args) {
      Context ctx = new InitialContext(...);  // Include settings to locate
                                              // enterprise server
      DemoHome dhome = (DemoHome)
        ctx.lookup("demo.DemoHome");

     // We have a reference to the DemoHome interface
     // Now use it to ask the server to create an instance of our Bean

     Demo demo = dhome.create();

     System.out.println(demo.DoSomething());
  }
}
```

Op veel J2EE servers zijn naast stateless beans ook entity beans beschikbaar. Een entity bean kan zijn state bewaren als de connectie met de client verbroken is. De Enterprise Java specificatie definieert hiervan een eenvoudig en een geavanceerd type. Het eenvoudige type laat het opslaan van de state volledig aan de server over, maar bij het geavanceerdere type is de programmeur verantwoordelijk voor het opslaan van de state. Voor alle twee de varianten

geldt helaas dat ze op alle Enterprise servers verschillend werken en daarom vaak aangepast moeten worden.

Beveiliging & Verificatie
De beveiliging van Enterprise Java is niet vastgelegd in de specificatie en daarom bij iedere variant anders geregeld. De referentie implementatie door Sun waar ik mee geëxperimenteerd heb, had alleen een beveiliging die IP adressen controleerde waarvandaan een verbinding opgebouwd kon worden.

Wanneer voor de connectie tussen de server en de client gebruik gemaakt wordt van een protocol dat op zichzelf geen beveiligingsproblemen heeft, dan is het goed mogelijk om de beveiliging over te laten aan de home interface. De argumenten die meegegeven kunnen worden aan create kunnen in dat geval de informatie bevatten die nodig is om toegang te verkrijgen tot de rest van het systeem.

Webkoppeling
Om gebruik te kunnen maken van Enterprise Beans vanuit webservers heeft Sun de Java Server Pages (JSP) techniek ontwikkeld. Deze techniek maakt het net als de ASP techniek van Microsoft en PHP mogelijk om programmacode in een HTML pagina op te nemen. Deze programmacode wordt door de server dan uitgevoerd net voordat de uitvoer naar de webbrowser gestuurd wordt om het mogelijk te maken gedeeltes van de pagina dynamisch op te bouwen.

De Java server pages bouwen zoals de naam al doorgeeft voort op de Java technologie. Sun's implementatie van de Java server pages bestaat uit een Java virtual machine die gekoppeld wordt aan een webserver. Op het moment dat de webserver een .jsp pagina wil weergeven wordt het verzoek doorgestuurd aan de virtuele machine die daarop het script compileert tot een Java class en die vervolgens uitvoert. De populariteit van de Java Server pages heeft tot nu toe nogal te lijden onder de traagheid waarmee Java server pages uitgevoerd worden. In nieuwere releases is de snelheid al een stuk verbeterd, maar in ongeveer alle op het Internet gepubliceerde test komen de Java server pages nog steeds als langzaamste uit de bus.

Omdat de scripts in de Java Virtual Machine draaien kan er direct gebruik gemaakt worden van alle Java functies, waaronder ook de Enterprise Java Beans. Er worden bij JSP ook een aantal speciale classes meegeleverd die helpen bij de meest voorkomende administratieve taken in webservers zoals user administratie, advertenties en formulieren. Dit soort classes die bedoeld zijn om taken te vervullen in websites worden door Sun 'servlets' genoemd, omdat ze het best te vergelijken zijn met server componenten zoals die gebruikt worden door gewone applicaties.

Toen Sun net uitkwam met JSP was het sessiebeheer van dit systeem met voorsprong het gebruikersvriendelijkste systeem om gebruikersinformatie in websites te beheren. Intussen zijn de meeste van deze functies ook al standaard in de andere systemen.

Case Studie
Omdat ik bij de start van mijn afstudeerproject niet de beschikking had over een werkend Enterprise Java omgeving heb ik slechts enkele test programma's gemaakt die gebruik maakten van deze technologie. Daarnaast was het niet nodig om de ToolBus aan deze omgeving te koppelen omdat hiervoor de ATerm adapter op zichzelf al prima geschikt is.

Tijdens het schrijven van dit verslag ben ik er achter gekomen dat Sun intussen niet meer hun eigen implementatie van Enterprise Java adviseert om kennis te maken met Enterprise Java. De voorbeeld programma's op hun website gaan nu uit van de demoversie van de BEA omgeving.

Toekomstperspectief

Op de vraag wat de toekomst gaat brengen voor Enterprise Java geeft iedere instantie op dit moment een verschillend antwoord. Sun zal zeggen dat het *de* standaard wordt voor het ontwikkelen van Enterprise systemen in de web omgeving.

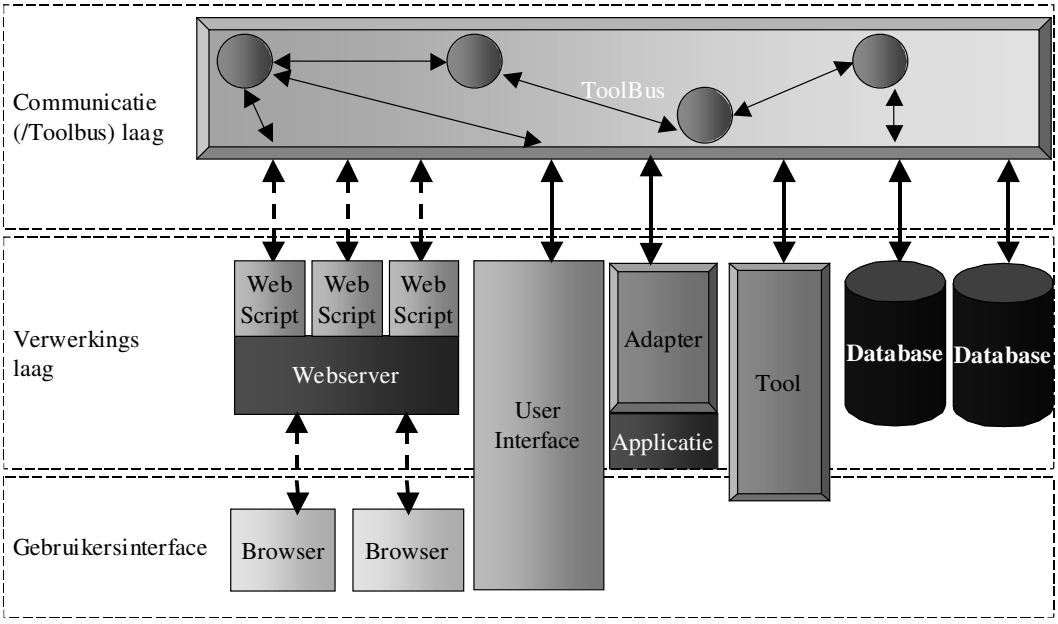
Veel andere bedrijven zien voor Enterprise Java best een plaats naast de bestaande systemen, maar zien geen grote overstap naar deze techniek. De matige ondersteuning van niet-java systemen blijft een zwak punt van deze techniek. Op dit moment wordt Enterprise vooral gebruikt in proef projecten van grote bedrijven en als backend voor webservers.

Enterprise Java krijgt vermoedelijk de komende jaren veel concurrentie van de met DotNet geïntroduceerde omgeving. De introductie van deze omgeving op een groot aantal platformen maakt het waarschijnlijk mogelijk om heterogene systemen op een veel eenvoudigere manier aan elkaar te koppelen dan dat dit in Enterprise Java mogelijk is, omdat er niet voor alle systemen een Java frontend ontwikkeld hoeft te worden. Misschien dat Sun hierdoor alsnog overgehaald wordt om Java beter te laten samenwerken met andere programmeertalen.

VI. De ToolBus

Geschiedenis
De ToolBus is net als de andere componenttechnologieën gebouwd als een technologie om software systemen te modulariseren. De ToolBus is speciaal ontworpen om gebruikt te worden in software renovatie projecten en niet alleen voor nieuwe systemen.

Uitgangspunten
De ToolBus is in tegenstelling tot DCOM en Enterprise Java niet ontwikkeld als een interface waarbij applicaties gebruik kunnen maken van componenten, maar als een bus architectuur. Alle componenten zijn hierbij zelfstandige programma's die alleen met de ToolBus communiceren. De werking van deze databus wordt bestuurd door een ToolBus-script dat bestaat uit een proces-algebra specificatie van de werking van het systeem.



De globale structuur van een toepassing gebaseerd op de ToolBus is te zien in het bovenstaande schema. De ToolBus is in dit model de plaats waar de samenwerking tussen de tools geregeld wordt. De ToolBus scripts beschrijven deze samenwerking op door binnen de bus ToolBus-processen te definieren die pseudo-parallel worden uitgevoerd. Deze-ToolBus processen kunnen met elkaar en met de externe tools communiceren met messages die bestaan uit termen. Binnen de ToolBus zelf is het ook mogelijk om gebruik te maken van een broadcast message die door alle geïnteresseerde ToolBus-processen te ontvangen is.

Door dit model is het eenvoudig toepassingen te definiëren als verschillende state-machines, die onafhankelijk van elkaar met de tools kunnen communiceren. Het grootste voordeel van deze manier van specificeren is dat er eenvoudig inzicht verkregen kan worden in de werking van een systeem en dat het zelfs mogelijk is om de correctheid van dergelijke systemen te bewijzen.

De belangrijkste reden dat bij de ontwikkeling van de ToolBus voor het bus model is gekozen is dat het in dit model mogelijk is om alle componenten als losse toepassingen te zien, omdat alle afhankelijkheden tussen de componenten vastgelegd zijn in de ToolBus scripts. Dit maakt het mogelijk om componenten te vervangen of te splitsen zonder dat in de rest van het systeem gecontroleerd hoeft te worden wat de afhankelijkheden van deze wijziging zijn.

Programmeermodel
Alle data in de ToolBus wordt gezien als termen. Een ToolBus-term is een variabele van het type integer, real, string, binary data, placeholder of een combinatie van meerdere termen opgebouwd als een lijst of als een functie. Wanneer dat mogelijk is wordt gespecificeerd welk type bepaalde data heeft, maar het is ook mogelijk data te specificeren als een willekeurige term. Een placeholder is een bijzonder type dat een term van een bepaald type representeert of voor het type term, dat iedere willekeurige term representeert.

De termen zijn opgebouwd om eenvoudig vergelijken en matchen mogelijk te maken. Bij dit matchen worden de placeholders gebruikt om onbekende delen van de te matchen term af te splitsen voor gebruik. Termen worden in de meeste gevallen opgebouwd door hun tekst representatie aan een functie mee te geven, maar in de meeste termbibliotheken is het ook mogelijk om met een laag-niveau interface directer met de termen om te gaan.

Voor de meeste in de Unix wereld populaire programmeertalen zijn intussen termbibliotheken beschikbaar. Voor veel talen is dit een van de standaard C term libraries met een aantal koppel versies, maar voor een aantal talen is er een eigen term library ontwikkeld.

De eerste versie van de term library voor C en C++ bleek in het gebruik nogal ingewikkeld, omdat er telkens extra handelingen gedaan moesten worden voor het memory management. Er is in het ontwerp rekening gehouden met het automatisch vrijgeven van termen na een ToolBus request door een eenvoudige garbage collector, maar voor termen die langer in het geheugen blijven moest een protect functie aangeroepen worden die dit registreert.

In de nieuwere ATerm library [Brand2000] is daarom gekozen voor een geheel andere aanpak. Er is hierbij gekozen voor een datalayout met maximum sharing en een luxere garbage collector. Om er voor te zorgen dat alle termen maximaal één keer in het geheugen aanwezig zijn werd de library ook voorzien van een hashing systeem. Een systeem met maximum sharing heeft grote voordelen bij een groot aantal handelingen met termen in diverse datastructuren en vermindert het geheugengebruik, maar heeft nadelen voor andere soorten handelingen.

Het exact vergelijken van datastructuren is de handeling die het meeste profijt heeft van maximum sharing, want dat wordt hierbij gereduceerd tot het vergelijken van twee pointers. Ook het matchen van termen op de hierboven beschreven wijze wordt een stuk eenvoudiger, omdat het merendeel van de vergelijkingen veel eenvoudiger is op te lossen en daarnaast zijn een aantal optimalisaties uit te voeren voor de placeholders die in dit geval bestaan uit vooraf bekende pointerwaarden.

Het grote nadeel van maximum shared termen is dat elementen van lijsten of bomen niet meer snel veranderd kunnen worden zonder een hele nieuwe structuur op te bouwen. Voor de parsergeneratie omgeving waar de ToolBus en de ATerm library voor ontwikkeld zijn, bleek uit experimenten die in [Brand99] beschreven worden dat de performance voordelen ruimschoots opwegen tegen de performance nadelen. Het is tot op heden niet onderzocht of

dat met de datastructuren die in andere takken van de software engineering gebruikt worden ook zo duidelijk het geval is.

De eenvoudige garbage collector van de oude term library is in de ATerm versie vervangen door een uitgebreidere versie die op de C-stack kan zoeken naar ATerm verwijzingen[1]. Hierdoor is het mogelijk om ook op andere momenten ongebruikte termen uit het geheugen te verwijderen. Voor ATermen die niet op de stack opgeslagen worden moet nu direct de protect functie aangeroepen worden, omdat de garbage collector nu zelf bepaalt wanneer er geheugen vrijgemaakt moet worden, in plaats van alleen na een verzoek aan de collector na het afhandelen van een ToolBus verzoek.

In de ToolBus architectuur bestaan alle componenten uit zelfstandige applicaties die op een of meerdere computers draaien. De koppeling wordt verzorgd via Unix of TCP/IP sockets en het hier voor gebruikte communicatieprotocol bestaat volledig uit termen.

```
/*
 * C sample code to generate and match a few ATerms using the
 *     ATerm library (HTTP://www.cwi.nl/projects/MetaEnv/aterm/)
 */

ATerm FromToolBus = ATmake("eval(calculate(\"12 + 5 / 3\"))");

/* ... */

ATerm func;
char* val

if(ATmatch(FromToolBus, "eval(<term>)", &func))
{
    if(ATmatch(func, "calculate(<str>)", &val))
        Calculate(val);
    else
        ATerror("Unknown eval");
}
```

In het bovenstaande codevoorbeeld wordt weergegeven hoe het matchen van termen in zijn werk gaat. Allereerst wordt er een term aangemaakt zoals die ook van de ToolBus ontvangen zou kunnen komen. Daarna wordt met ATmatch gekeken of de term voldoet aan het patroon dat gegeven wordt. In dit geval voldoet hij aan dit patroon en plaatst daarom in de variabele func de subterm die op de plaats van de eerste placeholder <term> gevonden wordt. Uiteindelijk wordt wanneer deze term een calculate verzoek met een string argument is, dit argument doorgestuurd naar de functie Calculate voor verdere processing.

[1] Deze techniek werkt goed op de huidige X86 en Sparc architecturen, maar er is op dit moment nog geen ervaring met ondersteuning voor processorarchitecturen met een aparte register renaming stack, zoals die in de IA-64 architectuur van de Intel Itanium processoren.

Beveiliging en Verificatie
Bij het ontwikkelen van de ToolBus is nog geen rekening gehouden met de aspecten van beveiliging en verificatie. Voor de toepassingen waarbij de ToolBus tot nu toe is ingezet waren deze aspecten nog niet van belang. In het volgende hoofdstuk wordt een overzicht gegeven van de beperkingen en tekortkomingen op dit terrein.

Uit dit overzicht wordt duidelijk dat er nog een hoop aan de beveiliging moet gebeuren voordat de ToolBus inzetbaar is in een omgeving waar niet vertrouwde processen een rol spelen.

Webserver koppeling
Een eerste inventarisatie van de mogelijkheden van de ToolBus gekoppeld aan een webserver, leverde een hoop mogelijkheden op. Voordat ik aan dit project begon waren er echter nog geen adapters die het mogelijk maakte de ToolBus aan een webserver te koppelen.

Het idee hier achter was dat het in de ToolBus mogelijk is er op een veel logischere manier met de sessie informatie om te gaan door voor iedere gebruiker een apart ToolBus-proces[1] aan te maken. Deze processen kunnen dan via messages communiceren met alle ToolBus processen die de sessie informatie willen gebruiken. De standaard delay functie die geïntroduceeerd werd met de discrete time ToolBus kan er zelfs voor zorgen dat de state informatie op tijd opgeruimd wordt. Ook transactie processen zijn in de ToolBus veel eenvoudiger te beschrijven dan in de concurrerende systemen, omdat het door de proces algebra specificatie het heel eenvoudig maakt om te beschrijven wat er in bepaalde gevallen moet gebeuren.

In eerste instantie was het de bedoeling om Apache via PHP aan de ToolBus te koppelen, om daarmee de mogelijkheden in de praktijk te onderzoeken. Helaas bleek dit veel ingewikkelder dan verwacht, door de problemen met het integreren van de geheugenbeheerders.

[1] Er wordt hier een ToolBus-proces bedoeld; niet een via een adapter gekoppelde applicatie.

VII. ToolBus - Webserver koppeling

Structuur
Om een beeld te krijgen over het gebruik van de ToolBus in een web-enabled omgeving heb ik geprobeerd PHP uit te breiden met een ToolBus adapter. De huidige interfaces bleken echter niet geschikt om direct gekoppeld te worden. Het voltooien van deze PHP extensie bleek uiteindelijk alleen mogelijk door ook de ATermbibliotheek en de webserver aan te passen. Deze aanpassingen heb ik uiteindelijk niet uitgevoerd omdat deze te specifiek gericht waren op een specifieke versie van Apache en PHP en daardoor niet bruikbaar zouden zijn voor toekomstige ToolBus distributies.

Om toch het benodigde inzicht te krijgen in de mogelijkheden van de ToolBus heb ik een aantal andere software modules voor de ToolBus ontwikkeld. De C# adaptor die in de case studie van dit hoofdstuk beschreven wordt kan wanneer Mono beschikbaar komt is vanaf het beschikbaar komen van de Microsoft DotNet omgeving al geschikt om op Windows systemen PHP aan de ToolBus door de DotNet-extensie in PHP. Ik verwacht dat over ongeveer een jaar Mono dezelfde mogelijkheden biedt voor de gangbare Unix platforms.

Om inzicht te krijgen in de werking van de ToolBus zelf heb ik de ToolBus sourcecode geport naar een Windows ’9X/NT omgeving. Omdat dit project vooral bestond uit het oplossen van problemen die ontstonden door de verschillen in de twee architecturen en verder niets te maken heeft met de componentarchitecturen die ik tijdens mijn afstuderen onderzocht voeg ik geen beschrijving toe van dit project zelf. Een groot gedeelte van de benodigde kennis over de werking van de ToolBus, die noodzakelijk was om de mogelijkheden van de ToolBus in deze context te kunnen beoordelen komt echter wel voort uit dit project.

Connectie opbouw
Zoals al in de introductie ter sprake is gekomen, wordt in een web-applicatie vooral gebruik gemaakt van korte verzoeken aan de componentarchitectuur, die zo snel mogelijk tot een resultaat leiden dat teruggestuurd kan worden naar de webbrowser van de gebruiker. Het huidige ToolBus protocol is echter ontworpen om gebruikt te worden door componenten die langdurig met elkaar in verbinding staan. Dat is vooral terug te vinden in de handshake fase en in de manier waarop in de ToolBus met datatransport wordt omgesprongen.

De handshake van de ToolBus is op dit moment niet geschikt voor het gebruik van hele korte connecties, zoals die in web-toepassingen gangbaar zijn, omdat tijdens de connectiefase de ToolBus geblokkeerd wordt voor het gebruik door andere componenten. Dat de handshake in sommige situaties het transport van veel data vereist, zorgt er voor dat dit probleem de ToolBus ongeschikt maakt voor diverse toepassingen.

Op het eerste gezicht is het mogelijk om de ToolBus te ontlasten door gebruik te maken van een adaptor die de verbindingen die de webserver-scripts normaal gesproken met de ToolBus zouden opbouwen. Zo'n proxy-adapter verandert de ToolBus ingrijpend, omdat de ToolBus scripts dan op de hoogte moeten zijn van de netwerk structuur waarmee tools aan de ToolBus gekoppeld zijn, terwijl de ToolBus architectuur juist ontworpen is om deze kennis te verbergen. In alle berichten die tussen de ToolBus en de proxy-adapter onderling verstuurd worden zouden namelijk subtool-identifiers moeten worden opgenomen voordat de proxy adapter in staat is de gegevens naar de juiste tools door te sturen.

Dit heeft echter een aantal nadelen die deze oplossing lastig realiseerbaar maken en ook alleen eenvoudig oplosbaar zijn door de ToolBus aan te passen. De belangrijkste is dat de hoeveelheid data die getransporteerd wordt tussen de adapter en de ToolBus aanmerkelijk groeit. Dit is, zoals beschreven wordt in de volgende sectie, een ander zwak punt van de ToolBus. De tijd die nodig is om zo'n proxy-adapter te bouwen kan beter geïnvesteerd worden in het bouwen van een nieuwe netwerk laag voor de ToolBus-server, omdat de snelheidswinst dan niet alleen ten goede komt van ToolBus-toepassingen die geschreven zijn voor het gebruik.

De handshake tussen de ToolBus en een adaptor is in de huidige ToolBus releases als volgt:

Adapter		**ToolBus**
Opbouwen connectie	➔	
	⇔	Accepteren connectie (=TCP/IP handshake)
		Wachten op 512 bytes (*slapend*)
Versturen naam, hostnaam en tool-nummer, aangevuld tot een blok van 512 bytes.	➔	
		Initialiseer de tool structuren
Wachten op toolnummer (*slapend*)		
	➔	Stuur toolnummer naar de adapter
Accepteren toolnummer		
Wachten op rec-do(signature(<term>, <term>)) (slapend)		
	➔	Stuur rec-do(signature(<term>, <term>)) met de signaturen van alle mogelijke messages die de ToolBus aan deze adapter kan sturen en welke messages de ToolBus van deze adapter accepteert.
Accepteren van de signaturen en vervolgens wanneer de tool dat ondersteunt alle signaturen matchen met deze lijst		Wachten op resultaat (*niet slapend*)
Resultaat terug sturen naar de ToolBus	➔	
		Het resultaat verwerken (c.q. negeren)

Op plaatsen waar gewacht wordt met het doorgaan van de toepassing, voordat het resultaat verwerkt wordt heb ik dat gemarkeerd met ‘slapend’. De ToolBus server maakt na het versturen van de signaturen gebruik van de ToolBus netwerk laag om te kijken of er data beschikbaar is. Vanaf dat moment is het mogelijk dat andere ToolBus-processen verder gaan tot het resultaat beschikbaar is.

Het versturen van de signaturen van de mogelijke termen vanuit de ToolBus naar de adapters zorgt bij het opbouwen van korte connecties voor een grote overhead ten opzichte van de benodigde data voor alleen het uitwisselen van termen. Omdat voor web-toepassingen geldt dat wanneer er meer mogelijkheden geboden worden, er meer signaturen verstuurd moeten worden, wordt juist bij grote projecten de hoeveelheid signaturen groter. Omdat juist bij deze

toepassingen de performance van de ToolBus-server een grotere rol gaat spelen, is het van belang dat er voor deze bottleneck een oplossing komt[1].

Met minimale aanpassingen aan zowel de ToolBus server als de adapters is het mogelijk om het protocol zo aan te passen, dat bij communicatie tussen nieuwe adapters en een nieuwe ToolBus de signatuur uitwisselfase overgeslagen kan worden, maar toch nog steeds het oude protocol ondersteunt blijft. Het is namelijk zondermeer mogelijk om extra parameters aan het 512 bytes grote blok toe te voegen, omdat de ToolBus sourcecode dit blok leest met sscanf. Een tool hoeft in deze situatie alleen de rec-do(signature(<term>, <term>)) als een gewoon ToolBus verzoek af te handelen, voor het geval er gebruik gemaakt wordt van een oude ToolBus[2].

De meeste andere architecturen gebruiken een unieke code die het communicatieschema identificeert en die door beide partijen begrepen wordt, maar veel minder transactietijd kost dan het communiceren van alle mogelijke termen. Dit is natuurlijk een erg efficiënte methode, maar deze methode doet geen recht aan de mogelijkheden van de ToolBus.

Een voor de ToolBus fraaiere oplossing is mogelijk door in het nieuwe protocol de adapter een controle getal naar de server te laten sturen die aangeeft welk communicatie protocol de adapter verwacht. Wanneer dit communicatiegetal overeenkomt met het controlegetal van de ToolBus-applicatie is het dan niet langer nodig om de informatie uit te wisselen.

De manier waarop het controle getal uitgerekend wordt is op zich niet van belang voor de adapters, omdat die dat getal aangeleverd kunnen krijgen vanuit de ToolBus via het .tifs bestand of kunnen opslaan na een eerste connectie met de ToolBus[3]. Voor een eenvoudige implementatie in de ToolBus code is het bijvoorbeeld mogelijk om als controlegetal te kiezen voor een MD5-hash van de signaturen als een tekststring. Om de flexibiliteit van de ToolBus zo groot mogelijk te laten zou ik er voor kiezen om de hashcodes '*' en '0' te reserveren voor een altijd respectievelijk nooit overeenkomend controlegetal, zodat adapters ook kunnen aangeven of ze de signaturen wel of niet willen ontvangen[4].

Het bovenstaande leidt tot de volgende geoptimaliseerde communicatie tussen de adapter en de ToolBus:

---

[1] Het is niet noodzakelijk het 512 bytes grote blok te optimaliseren, omdat deze 512 bytes precies in een standaard tcp/ip netwerk packet passen.

[2] De signatuur controle moet er wel rekening mee houden dat de rec-do(signature(<term>, <term>)) signatuur niet in de lijst mogelijke signaturen hoeft te staan.

[3] Voor deze mogelijkheid is het vereist dat de rec-do(signature(<term>, <term>)) uitgebreid wordt met een derde parameter die de hashcode bevat.

[4] Bij deze aanpassing is het verstandig om tegelijkertijd te overwegen of het nuttig is het protocol ook te voorzien van een versienummer om adapters mogelijk te maken die ook met toekomstige versies van het protocol eenvoudig overweg kunnen.

Adapter		ToolBus
Opbouwen connectie	➔	
	⇔	Accepteren connectie (=TCP/IP handshake)
		Wachten op 512 bytes (*slapend*)
Versturen naam, hostnaam, toolnummer en hashcode aangevuld tot een blok van 512 bytes.	➔	
		Initialiseer de tool structuren
Wachten op toolnummer (*slapend*)		
	➔	Stuur toolnummer naar de adapter
Accepteren toolnummer		
	➔	Wanneer de hashcode '*' is of niet overeenkomt met de hashcode van de messages: Stuur rec-do(signature(<term>, <term>, <str>)) met de signaturen van alle mogelijke messages die de ToolBus aan deze adapter kan sturen, welke messages de ToolBus van deze adapter accepteert en de hashcode.
Afhandelen van rec-do(signature(<term>,<term>,<str>)) als een gewone ToolBus message		

Term uitwisseling
Het uitwisselen van de grote signatuur termen is mede een probleem, omdat de ToolBus het verzenden en ontvangen van een term als een atomaire actie ziet en ondertussen geen andere acties van hetzelfde of andere ToolBus-processen kan uitvoeren. Dit probleem treedt ook op wanneer een grote term getransporteerd wordt tussen twee adapters via de ToolBus, want tijdens het communiceren van de termen liggen in die situatie alle overige ToolBus-processen stil, terwijl dit volgens de proces algebra definitie van de ToolBus niet vereist is.

Wanneer er alleen termen kleiner dan maximaal enkele kilobytes getransporteerd worden zorgt het besturingssysteem er voor dat alle gegevens gebufferd worden voor ze via het netwerk verstuurd kunnen worden. Wanneer de termen echter groter worden zal de socket bibliotheek wachten met het versturen tot er weer ruimte vrijkomt in de netwerk buffers. Om ook het communiceren van grote termen, zoals lijsten van database records of parse trees mogelijk te maken zonder dat deze de ToolBus blokkeren is het nodig om de socket laag in een asynchrone mode te zetten. Hierdoor zal het netwerk alleen de data accepteren die direct verstuurd kan worden. Wanneer de data die niet direct geaccepteerd kan worden, wordt de foutcode EWOULDBLOCK teruggegeven, zodat de applicatie weet dat de buffers vol zijn.

Dit maakt het mogelijk om een tweede bufferlaag te bouwen binnen de ToolBus-server, die terwijl er gewacht wordt op de datatransmissie andere ToolBus-processen de mogelijkheid kan bieden om verder te gaan.

De ToolBus definitie bevat afzonderlijke verzend- en ontvangprimitieven, die wanneer ze na elkaar gebruikt zouden worden nu al onderbroken kunnen worden door andere acties uit parallel lopende processen binnen de ToolBus. Om de ToolBus door deze buffering efficiënter te maken, maar verder hetzelfde te laten functioneren is het daarom alleen noodzakelijk om te eisen dat er geen twee communicatie primitieven naar een bepaalde tool tegelijkertijd uitgevoerd kunnen worden. Ondertussen zouden echter ToolBus-processen die niet afhankelijk zijn van deze term al door kunnen gaan. Omdat een communicatieprimitieve volgens de ToolBus algebra altijd termineert is het mogelijk om dit zonder het creëren van nieuwe potentiele deadlocks te realiseren.

De bufferlaag is redelijk eenvoudig te realiseren door het inlezen en schrijven van de termen te laten verzorgen door een onderbreekbare state machine. Het schrijven en lezen van de termen kan hierbij eenvoudig gebruik maken van de sharing in de termen bibliotheek in plaats van data buffers die de termen als tekst opslaan. De state op zichzelf is ook op te slaan met de Termbibliotheken, omdat die een efficiënte stack aanbieden en daarnaast een afzonderlijk memory management van de nog te schrijven of de reeds gelezen termen overbodig maken[1].

Autorisatie

Verbindingen van de ToolBus met tools kunnen tot stand komen door te wachten tot een tool zelf een verbinding opbouwt, of doordat de ToolBus zelf een tool opstart. In beide gevallen wordt er door de tool een TCP of Unix socket connectie opgebouwd met een openstaande poort van de ToolBus op een vooraf bekend poortnummer. Na het opbouwen van de verbinding stuurt de tool zijn naam naar de ToolBus zodat die kan bepalen welk type tool er gestart is en de daar bijbehorende acties in het ToolBus-script kan uitvoeren[2].

Webdiensten die gebruik maken van een standaard TCP/IP poortnummer voor het aanbieden van diensten, moeten gebruik maken van beveiliging om te voorkomen dat ze ongewenst gebruikt worden door hackers. Dit soort ongewenst gebruik is te onderscheiden in misbruik dat het verlenen van de gewone diensten onmogelijk maakt (DOS-attacks[3]), en misbruik dat gericht is op het misbruiken van de dienst zelf.

De ToolBus is op dit moment erg vatbaar voor denial-of-service attacks, omdat tijdens het opbouwen van de verbinding op meerdere plaatsen de ToolBus stil ligt terwijl gewacht wordt op gegevens van de adapters. Het openen van een socketverbinding naar de poort van de ToolBus en daarna niets versturen naar deze poort is al voldoende om alle communicatie via de ToolBus zo stil te leggen dat er geen events meer kunnen plaatsvinden tot de ToolBus herstart wordt. Zo lang de ToolBus alleen verbindingen hoeft te verwerken van lokale computers is daarom de veiligste oplossing om verbindingen naar de ToolBus van buiten het lokale netwerk al op de firewall te blokkeren.

De eenvoudigste manier om de ToolBus uit te breiden met een vorm van verificatie is bij het opbouwen van de verbinding controleren of het ip-adres dat de verbinding opbouwt in een

[1] De termen moeten anders met ATprotect beveiligd worden tegen vrijgeven door de garbage collector.
[2] Bij het opstarten door de ToolBus is dit noodzakelijk omdat in ToolBusscripts meerdere tools tegelijkertijd gestart kunnen worden, maar er niet gewacht wordt met het uitvoeren van de ToolBusscripts tot deze tools de verbinding met de poort opgebouwd hebben.
[3] Denial of service attack

lijst toegestane ip-adressen staat. Hiermee is het gevaar van misbruik beperkt tot systemen op het eigen netwerk[1].

Op veel Unix netwerken wordt de op ip-adressen gebaseerde beveiliging uitgebreid met een 'ident' beveiliging [rfc1413]. Op systemen die ident ondersteunen kan de computer waar een verbinding mee wordt opgebouwd aan de andere computer vragen onder welke gebruikersrechten het verbindende proces draait. Op netwerken waar er tussen de computers een vertrouwensrelatie bestaat is dit een afdoende beveiliging voor de meeste software. Wanneer er echter computers zonder ident ondersteuning of van buiten de vertrouwde omgeving mee communiceren is deze beveiliging vaak zeer eenvoudig te omzeilen.

De voorgaande verbeteringen van de beveiliging zijn mogelijk zonder dat het communicatie-protocol tussen de tools aangepast wordt, maar zijn niet voldoende voor netwerken waar computers in aanwezig zijn die niet vertrouwd worden. Om de authenticiteit bij een zwaardere beveiliging te waarborgen moet er gebruik gemaakt worden van wachtwoorden of certificaten die aantonen dat de applicatie met de ToolBus mag communiceren. Ook is het in veel omgevingen noodzakelijk om naast het vaststellen van de authenticiteit van de verbinding ook de uitgewisselde informatie te versleutelen, omdat het anders mogelijk is om de verificatie informatie te onderscheppen.

Voor connecties op het Internet wordt hiervoor in toenemende mate gebruik van het standaard versleutel en verificatie protocollen SSL[2] en TLS[3]. Voor deze versleutelsystemen zijn diverse betrouwbare standaard implementaties beschikbaar, zoals Openssl [openssl], die met een aantal kleine aanpassingen te integreren zijn in de ToolBus en de adapters.

Om de beveiliging optimaal te maken zou het verstandig zijn om de ToolBus en de adapters te voorzien van certificaten die gecontroleerd worden door SSL/TLS in plaats van wacht-woorden, om ook man-in-the-middle attacks onmogelijk te maken. Het genereren en beheren van certificaten maakt het gebruik echter zo veel lastiger dat het verstandig is deze vorm van beveiliging alleen te gebruiken in situaties waar dit niveau van beveiliging echt noodzakelijk is. Voor alle overige verbindingen, zoals die op locale systemen of systemen binnen gesloten netwerken is dit niveau van beveiliging overkill en zorgt het verplichten er alleen voor dat er manieren worden gezocht om het te omzeilen.

Wanneer de ToolBus ergens als algemene component-technologie ingezet wordt is het misschien handiger om in plaats van met certificaten te werken, een centrale server de beschikking te geven over alle public keys van de tools. Alle processen kunnen dan met behulp van deze server controleren of tools de juiste rechten hebben. Deze techniek die ook toegepast wordt bij Kerberos authenticatie [rfc1510] maakt het mogelijk om door slechts een server goed te beveiligen alle applicaties te beveiligen. Het grootste voordeel van dit model is dat de private keys niet ergens centraal opgeslagen zijn, maar in iedere applicatie voor zich. Deze verspreiding van de informatie maakt het onmogelijk om door het kraken van een systeem toegang te verkrijgen tot andere systemen.

[1] Voor IP-spoofing zullen andere maatregelen genomen moeten worden.
[2] Secure Socket Layer, geïntroduceerd door Netscape.
[3] Transport Layer Security, de door het World Wide Web Consortium gestandaardiseerde versie van SSL.

Case Studie – PHP adapter

Om de mogelijkheden van de ToolBus in een web enabled omgeving te kunnen bestuderen ben ik begonnen aan het bouwen van een PHP4-adapter. De PHP engine is zo geschreven dat hij aan een zo groot mogelijk aantal webservers gekoppeld kan worden en heeft daarvoor een eigen geheugenbeheer laag die per webserver verschillend geïmplementeerd is. Doordat de Atermbibliotheek speciale eisen stelt aan het geheugen bleek de standaard methode om voor een taal een adapter te bouwen niet toepasbaar in een standaard PHP-Apache installatie.

Binnen PHP4 zijn alle variabelen reference counted, net als in een standaard COM installatie. Er zijn alleen een aantal oplossingen bedacht om de problemen van circular references te verbergen. Zo wordt geheugenruimte die gebruikt wordt door PHP standaard vrijgegeven zodra het PHP script afgehandeld is door de webserver.

De garbage collector van de ATermbibliotheek gaat er van uit dat de meeste ATermen opgeslagen worden op de stack of als lijsten van ATermen in het geheugen, maar dit zou in PHP niet het geval zijn. PHP slaat namelijk alle variabelen op in hash-tabellen die op de heap opgeslagen worden. Een ander probleem is dat de ATermbibliotheek voor het beheer van de termen de hele stack wil kunnen scannen, maar de basis van de stack alleen bekend is binnen Apache en niet binnen PHP.

Om de ATermen bibliotheek in zijn huidige vorm te kunnen koppelen zouden zowel Apache en PHP als de ATermbibliotheek zelf ingrijpend gewijzigd moeten worden. Dit zou ook betekenen dat de ondersteuning niet overdraagbaar is naar nieuwere versies van de Apache PHP combinatie en ook niet overdraagbaar is naar nieuwere ATerm versies[1].

Om de ATermbibliotheek te kunnen gebruiken vanuit PHP onder Apache 1.3.x moet allereerst de garbage collector de basispositie van de Stack hebben, zoals die was bij het starten van het Apache proces. Daarnaast moeten de geheugenlocaties van variabelen die ATermen bevatten geregistreerd worden in de ATermbibliotheek om te zorgen dat de Termen niet vrijgegeven worden wanneer er nog naar verwezen wordt.

Omdat PHP4 de aanname doet dat alle resource handles onbeperkt gekopieerd kunnen worden is de enige manier om dit gestructureerd te realiseren alle ATermen in een tabel opslaan, waar naar verwezen wordt vanuit de PHP variabelen. Wanneer het PHP script afgelopen is zal de PHP engine de resource handles proberen vrij te geven, waarop de ATermen in de tabel verwijderd kunnen worden. Pas op dat moment kan de garbage collector achterhalen dat de termen niet meer gebruikt worden en het geheugen vrijgeven.

Om de Garbage collector al direct te kunnen gebruiken in PHP moeten alle variabelenlocaties binnen PHP geregistreerd worden in de ATermbibliotheek. Hiervoor moet de ATerm-bibliotheek worden uitgerust met een functie gebaseerd op de ATprotect functie, maar de geheugenlocatie die geregistreerd wordt niet ziet als een locatie waar een ATerm staat, maar als een locatie waar een ATerm *kan staan*. Daarnaast moet PHP op diverse plaatsen aangepast worden, om bij het aanmaken van de geheugenlocaties voor het opslaan van de variabelen. Helaas is dit een zo goed als onmogelijke opgave, omdat PHP extensies deze structuren direct benaderen[2].

---

[1] Zo lang de benodigde uitbreidingen niet ingebouwd worden in de standaardversie.

[2] Zelfs als dit allemaal zou lukken, dan werkt deze PHP versie slechts tot het uitkomen van Apache 2.0, omdat die gebruik gaat maken van threads in plaats van losse processen voor het verzoeken van webbrowsers.

Om de ToolBus toch direct vanuit PHP te kunnen gebruiken is het noodzakelijk om gebruik te maken van een nieuwe termen bibliotheek, die meer inspeelt op het gebruik in een reference counted omgevingen[1] en/of omgevingen die geheugenblokken direct vrijgeven na een bepaalde tijd. Het is echter de vraag of PHP deze inspanning waard is, omdat PHP steeds meer mogelijkheden krijgt voor het gebruik van externe object bibliotheken. De C# adapter die hierna beschreven wordt, is namelijk nu al geschikt om de ToolBus te gebruiken vanaf Windows systemen.

[1] Een dergelijke termen bibliotheek zou eenvoudig herbruikbaar zijn in een efficiëntere termen bibliotheek voor Tcl 8.0 en hoger.

Case Studie – ToolBus port naar Windows

Om inzicht te verkrijgen in de werking van de ToolBus en tegelijkertijd een al lang bestaande vraag te vervullen heb ik de sourcecode van de ToolBus geschikt gemaakt om gebruikt te worden als een Windows Console applicatie. Ik heb er hierbij voor gekozen om de ToolBus te compileren met Microsoft's Visual Studio in plaats van een Unix emulatie systeem, omdat al de gangbare emulatie systemen de nodige eisen stellen aan de computer-installatie van de gebruikers en Visual C++ applicaties op iedere Windows computer meteen werken.

Zoals bekend bij de meeste Unix/Linux ontwikkelaars is het niet eenvoudig mogelijk om Unix applicaties geschikt te maken voor Windows, omdat er een aantal grote verschillen zijn in de implementatie van bepaalde systeemfuncties. De belangrijkste verschillen bij de conversie van de ToolBus waren:

- Het merendeel van de standaard Unix C headers is onder Windows niet beschikbaar[1].
- Filestreams en Sockets zijn onder Windows niet van hetzelfde type.
- De console is onder Windows geen volwaardige stream.
- Signalen van het besturingssysteem naar de applicatie worden anders doorgegeven.
- De fork() functie is onder Windows niet beschikbaar.
- Er is geen standaard C preprocessor beschikbaar onder Windows.

Om de conversie mogelijk te maken moesten er voor al deze problemen oplossingen gezocht worden.

Ik ben begonnen met in een directory alle include bestanden aanmaken die Visual Studio niet had, maar de ToolBus wel nodig had. De inhoud van deze files bestond alleen uit een include van een standaard header file waaraan ik later de windows header files zou kunnen toevoegen die nodig waren om alles correct te laten compileren.

De volgende stap was om de TCP/IP sockets geschikt te maken voor gebruik onder Windows. Zoals ik al in een ander project gemerkt had gaat dat het makkelijkste door een paar #define's op te nemen die read(…), write(…) en close(…) vervangen door respectievelijk recv(…, 0), send(…, 0), en closesocket(…). De ToolBus maakt naast de TCP/IP sockets ook gebruik van Unix sockets, die onder Windows niet worden ondersteund. Om de rest van de ToolBus sourcecode zo min mogelijk aan te passen heb ik er voor gekozen om deze code rechtstreeks te vertalen naar een tcp/ip connectie naar localhost.

Om de console als een volwaardige stream te kunnen benaderen was ik van plan in een background thread de console aan een socket te koppelen, zodat de ToolBus de console op dezelfde manier zou kunnen benaderen. Bij het testen van de ToolBus om de exacte eigenschappen van de console onder Solaris te bepalen bleek dat er nog een bug in de console code van de ToolBus zit. Deze bug maakt het onmogelijk om lees operaties van de console af te breken met de timeout primitieve, ook al geeft de documentatie van de ToolBus aan dat deze functionaliteit wel aanwezig is. Na overleg bleek dat deze functionaliteit tot nu toe nergens werd gebruikt en het repareren daarom geen hoge prioriteit had[2].

De ToolBus vangt standaard een groot aantal Unix signalen af, maar met de meeste wordt niets gedaan. De meeste Unix signalen die de ToolBus onder Unix expliciet negeerde kunnen onder Windows niet voorkomen en het was daardoor simpelweg mogelijk deze signalen te

[1] Alleen de door Ansi en ISO gestandaardiseerde headers zijn direct beschikbaar.

[2] Het probleem is niet triviaal te repareren zonder andere functionaliteit verloren te laten gaan, omdat er in een versie zonder deze bug geen gebruik meer kan worden gemaakt van de C input functie.

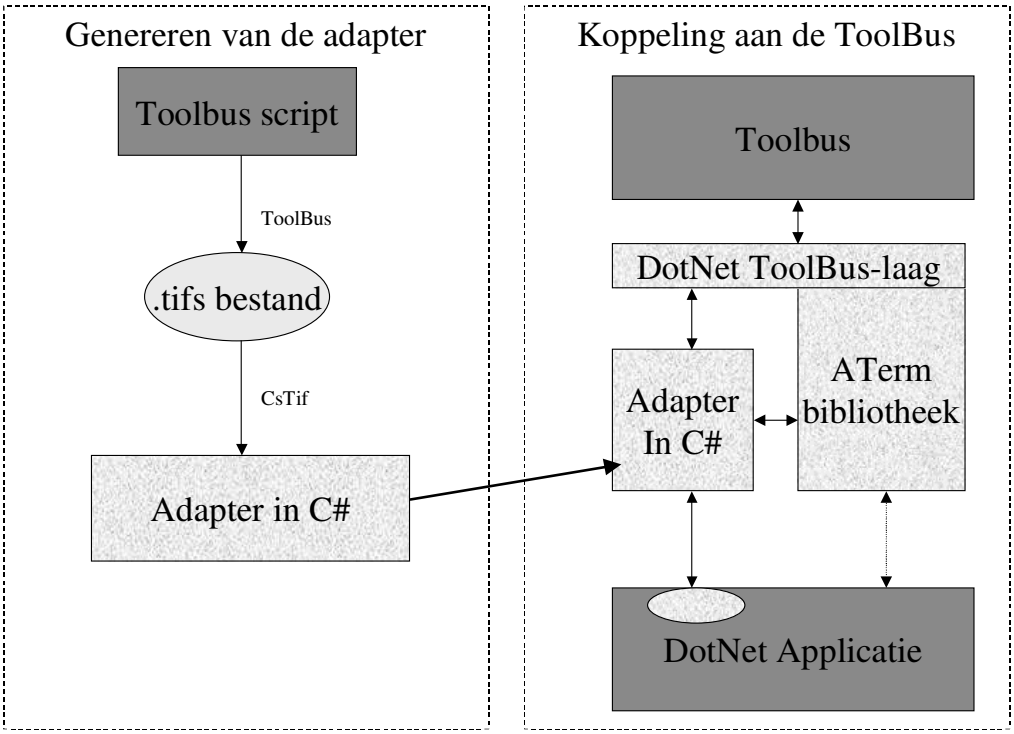
negeren. De alarm timer, die intensief door de discrete-tijd versie van de ToolBus gebruikt wordt is onder Windows niet beschikbaar en moest daarom nagebootst worden. De multimedia timer heeft ongeveer dezelfde eigenschappen als het Unix alarm en was daarom eenvoudig om te bouwen tot een alarm timer met dezelfde interface als de Unix timer.

Uiteindelijk moest alleen de op fork() gebaseerde functie voor het starten van child processen herschreven worden om te kunnen werken onder Windows. Gelukkig was deze functie zo opgebouwd dat de fork() aanroep en de kort daarop volgende aanroep van execvp() eenvoudig vervangen konden worden door een _spawnvp() aanroep.

Omdat deze ToolBus implementatie nog niet bedoeld was om in de standaard ToolBus distributie opgenomen te worden heb ik er voor gekozen om de c-preprocessor stap van de ToolBus uit te schakelen in plaats van ondersteuning voor een andere preprocessor in te bouwen. Zo kon ik het uitvoeren van scripts toch al testen, door ze eerst op een Unix systeem door de preprocessor te halen.

Case Studie – DotNet Adapter
Om een beeld te krijgen van de verschillen tussen Java en C# heb ik de standaard Java adapter van de ToolBus zoals die bijgeleverd wordt bij de ATerm bibliotheek geport naar C#. Deze adapter maakte het niet alleen mogelijk om C# in de praktijk te testen, maar maakte het ook mogelijk om de verschillen tussen Java en C# in dit specifieke geval te bestuderen. Een leuke bijkomstigheid is dat deze adapter niet alleen in C# te gebruiken is, maar vanuit alle talen die de DotNet bibliotheek kunnen gebruiken. (zie hieronder)

De DotNet adapter bestaat net als de Java adapter uit een aantal onderdelen die het met elkaar mogelijk maken om code op een eenvoudige manier aan de ToolBus te koppelen. Uit de informatie die de ToolBus aanbiedt is het mogelijk om C# code te genereren die de communicatie met de ToolBus afhandelt. Deze code maakt hierbij gebruik van de ATerm bibliotheek voor DotNet en de ToolBus laag[1].



Het CsTif tool genereert een ‘Tool’ class met match code, een ‘Tif’ interface met de functies door de ToolBus opgeroepen kunnen worden en een ‘Bridge’ class die de ‘Tool’ class aan de ‘Tif’ interface koppelt. De class bevat alle match code die nodig is om termen die de ToolBus stuurt om te zetten in oproepen naar functies.

Omdat DotNet het mogelijk maakt om interfaces die gedefinieerd worden in een programmeertaal te implementeren in een willekeurige programmeertaal voor DotNet is het ook mogelijk om de gegenereerde C# code te gebruiken voor applicaties in bijvoorbeeld Visual Basic of Python door in die taal de Tif interface te implementeren.

[1] De ToolBus laag is de C# code die de socket communicatie met de ToolBus omzet in ATermen.

Het converteren van Java sourcecode naar C# is een redelijk triviale stap, omdat de syntactische structuren van de programmeertalen erg op elkaar lijken. Om de sourcecode om te zetten in syntactisch correct C# was het alleen noodzakelijk om de class en interface definities aan te passen en op sommige punten wat specifieker te definiëren, omdat de objectstructuur van het Common Language Typesystem (CLT) wat strikter is dan het Java model.

In de CLT mogen interfaces geen member variabelen bevatten en moeten alle member functies public toegankelijk zijn. Om toch member variabelen te kunnen definiëren is het mogelijk om een abstracte class te gebruiken of door de variabelen als een property te definiëren. Verder dwingt de CLT, in tegenstelling tot Java, de programmeur om overerfde functies tenminste dezelfde toegankelijkheid te geven als de originele functie, zodat de toegankelijkheid van functies altijd eenduidig is.

Na het syntactisch converteren van de sourcecode was de volgende stap om de aanroepen naar de standaardbibliotheek te vervangen door die van de Common Language Specificatie (CLS). Dit bleek veel eenvoudiger te zijn dan verwacht, omdat bijna alle Java standaard classes onder een net iets andere naam ook beschikbaar zijn in de CLS bibliotheek en ook de member functies op die classes meestal overeen kwamen.

Na het aanpassen van de sourcecode zodat die door de compiler kwam, heb ik de C# classes gecontroleerd met de tests die ontwikkeld waren voor de Java versie van de ATerm-bibliotheek. Uiteindelijk kwamen hierdoor nog een aantal duidelijke verschillen tussen de bibliotheken tevoorschijn. Het bleek noodzakelijk om de hashfuncties en een aantal string-operaties te herschrijven, omdat die te veel gebruik maakten van een paar eigenschappen van de Java objecten om ze direct te kunnen hergebruiken[1,2].

Het converteren van de socket laag naar C# was meer werk, omdat de netwerk architectuur van de CLS anders en flexibeler opgebouwd is dan die van Java. Zo ondersteunt de CLS naast de ook in Java ondersteunde Tcp connecties ook andere protocollen zoals IPX.

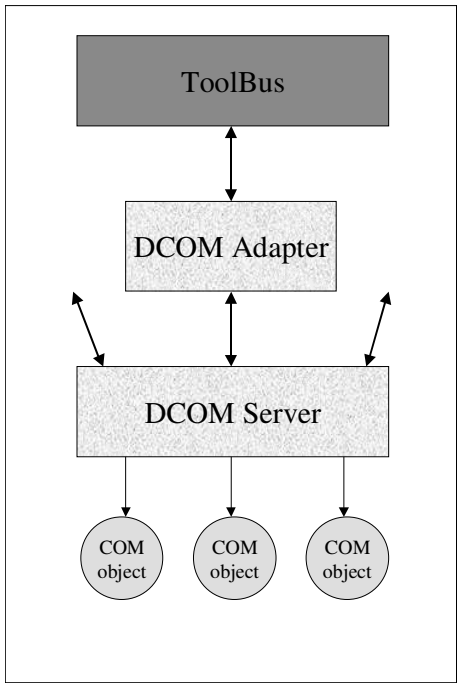
In Appendix A is de sourcecode opgenomen van de hoofdmodule van de ATerm bibliotheek voor DotNet. De ATerm factory in deze module is verantwoordelijk voor het geheugenbeheer en de sharing van de termen.

[1] Er kwamen hierbij twee niet eerder ontdekte bugs in de Java implementatie van de ATerm bibliotheek naar boven. Één in het parsen van hexadecimale nummers en één die er in bepaalde gevallen voor kon zorgen dat ATermen niet meer maximaal geshared werden, wanneer er andere termen door de garbage collector werden opgeruimd.

[2] Ik heb de code in eerste instantie geschikt gemaakt voor de eerste beta van de Microsoft.NET Software Development kit. In een later stadium heb ik de sourcecode op een aantal details aangepast om de code ook te kunnen gebruiken in de tweede beta release.

Case Studie – DCOM Adapter

De DCOM adapter was in tegenstelling tot de meeste andere ToolBus-adapters niet bedoeld als een nieuwe mogelijkheid om een programmeeromgeving aan de ToolBus te koppelen, maar als mogelijkheid om bestaande DCOM componenten te kunnen hergebruiken vanuit de ToolBus. Om de ontwikkeling van deze adapter te vereenvoudigen heb ik hem opgebouwd uit twee modules.



De belangrijkste module is de DCOM server module, die de COM objecten creëert en functies en properties op deze objecten kan benaderen. Deze module is gebouwd als een server proces voor Windows dat TCP/IP verbindingen van de andere module accepteert.

De DCOM adapter is een gewone ToolBus adapter die met de standaard ATerm bibliotheek aan de ToolBus gekoppeld wordt en draait op de computer waar de ToolBus op draait. Wanneer deze adapter gestart wordt door de ToolBus, maakt de adapter direct een verbinding met de andere module om het COM object te creëren dat op de commandoregel is meegegeven.

De DCOM Adapter module zet de op termen gebaseerde berichten van de ToolBus om in een zeer eenvoudig protocol dat door de andere module wordt omgezet wordt in aanroepen naar COM objecten via de IDispatch interface.

Deze twee laag interface maakt het mogelijk om vanuit de ToolBus een groot aantal standaard bestaande Windows applicaties te besturen via de ActiveX automation interface. Voor het bouwen van nieuwe Windows toepassingen die gebruik maken van de ToolBus is het echter verstandiger om direct de ATerm bibliotheek te gebruiken, omdat de extra laag extra overhead bij de communicatie oplevert en de mogelijkheden van de ToolBus beperkt.

Omdat ik deze adapter meer als testomgeving heb ontwikkeld, dan als een adapter die veel gebruikt zal worden heb ik alle veiligheidsaspecten van deze infrastructuur genegeerd. In de praktijk komt het er op neer dat als je de DCOM server module op een Windows computer draait, het mogelijk is om de volledige computer over te nemen[1].

[1] Alle beveiligingsproblemen uit het ToolBus hoofdstuk zijn ook van toepassing op deze adapter. Alleen biedt deze adaptor op Windows '9X en NT systemen standaard toegang tot een filesysteem object, dat shell toegang aanbiedt.

VIII. Conclusies

In de inleiding presenteerde ik de vraag die centraal staat in mijn afstudeerwerk: *Wat zijn de mogelijkheden en onmogelijkheden van de ToolBus in web-enabled toepassingen?*

Om een antwoord te zoeken op deze vraag heb ik tal van technieken rond web-enabled toepassingen onderzocht. Ik heb geprobeerd de kennis die ik daarbij opgedaan heb, te gebruiken in een aantal kleine projecten die de herbruikbaarheid van de ToolBus in een web-enabled omgeving ten goede komen.

Toen ik net aan mijn onderzoek begon was het de bedoeling om de mogelijkheden van de ToolBus in Enterprise Java te onderzoeken, maar dat bleek geen haalbare doelstelling. De belangrijkste reden hiervoor was dat de J2EE omgeving van Sun de eerste maanden van mijn onderzoek nog te experimenteel was om goed onder de Solaris omgeving te kunnen werken. Toen uiteindelijk een nieuwe versie uitkwam die wel werkte onder Solaris bleek het systeem onder Solaris te veeleisend om op de ingeslagen weg verder te gaan[1,2].

Toevallig kwam ik rond deze tijd in aanraking met de eerste berichten over de toekomstige DotNet omgeving van Microsoft. Omdat dit mooi aansloot op mijn eerdere ervaring met Windows DNA en Enterprise Java ontstond het plan om het onderzoek te verleggen naar de technologie achter deze technieken.

Enterprise Java, Windows DNA en DotNet zijn componenttechnologieën die tegenwoordig gepresenteerd worden als een manier om gestructureerd grote applicaties te bouwen die te benaderen zijn via zowel normale applicaties als via webbrowsers. De ToolBus is net als deze technologieën ontwikkeld als een componentarchitectuur, maar het opnemen van de ToolBus in een web-toepassing was tot nu toe nog niet bestudeerd.

Om me een goed beeld te kunnen vormen van de web-technologieën heb ik Enterprise Java, Windows DNA en DotNet in deze context bestudeerd. Ik heb me daarbij vooral gericht op DotNet, omdat deze techniek de meeste nieuwe op het web gerichte mogelijkheden biedt en de technische achtergrond van deze componenttechnologie mij het meeste aansprak.

De ToolBus in web-enabled toepassingen
De ToolBus is, zoals verwacht nog niet voorbereid op het gebruik in web-enabled toepassingen. Een groot aantal tekortkomingen is echter redelijk eenvoudig te verhelpen zonder grote inspanningen.

Omdat web applicaties te maken hebben met het dynamisch aan en afkoppelen van componenten is de beperkte snelheid van de ToolBus in dit type applicaties een grotere bottleneck dan in gewone ToolBus-toepassingen. Deze bottleneck is het beste op te lossen door de socket implementatie in de ToolBus-server geschikt te maken voor asynchrone communicatie. Om de ToolBusserver veilig te kunnen gebruiken op een openbare webserver

[1] De minimale werkomgeving voor Enterprise Java onder Solaris bestond uit 3 modules die per stuk tussen de 45 en 65 megabyte ram nodig hadden.
[2] Bij een test onder Windows 2000 was het geheugengebruik veel minder groot, ook al zou je van software die door Sun ontwikkeld wordt eerder andersom verwachten.

is het ook noodzakelijk om de beveiliging in de ToolBus te verbeteren, omdat de ToolBus door de synchrone communicatie op dit moment erg vatbaar is voor denial of service attacks.

ToolBus gebruik vanuit een webserver
De ATermbibliotheek die in bijna alle ToolBus-adapters gebruikt wordt voor het beheren van de term structuren en de communicatie met de ToolBus is niet geschikt om direct als een module in webservers te gebruiken. Omdat de meeste webservers hun extensies gebruik laten maken van een eigen geheugenbeheersysteem dat geoptimaliseerd is op het tegelijkertijd gebruiken van meerdere extensies in een multi-threaded omgeving is het niet eenvoudig om de garbage collector van de ATermbibliotheek te hergebruiken.

Om de ToolBus toch vanuit webservers te kunnen gebruiken is het noodzakelijk om gebruik te maken van een andere termen bibliotheek. De eenvoudigste oplossing is om vanuit de scripts van de webserver de koppeling met Java of DotNet te gebruiken en die de koppeling met de ToolBus te laten regelen. Een andere oplossing is een andere termenbibliotheek te ontwikkelen die wel in deze situatie bruikbaar is.

DotNet Adapter voor de ToolBus
Om de Java omgeving goed te kunnen vergelijken met de DotNet omgeving heb ik de Java ATermbibliotheek naar C# geport. Globaal lijken de systemen erg op elkaar te lijken, wat het converteren van software eenvoudig moet maken. Op details is er bij de ontwikkeling van DotNet echter vaak gekozen voor andere oplossingen om de bibliotheek beter geschikt te maken voor het gebruik vanuit verschillende programmeertalen.

De DotNet adapter maakt het mogelijk om in alle programmeertalen die voor deze omgeving zijn gebruik te maken van ATermen en de ToolBus. Ook ziet het er naar uit dat steeds meer Windows applicaties gebruik gaan maken van DotNet objecten voor scripting ondersteuning en plugins.

ToolBus voor Windows
Het porten van de ToolBus naar Windows was een project dat een hoop inzicht in de werking van de ToolBus opleverde. Daarnaast is het een eerste stap voor het beschikbaar maken van de meta environment voor Windows systemen.

Om de ToolBus onder Windows goed te kunnen testen is het nog noodzakelijk om ook de tcl/tk adaptor naar Windows te porten, zodat de viewer beschikbaar komt.

A. C# ATerm Factory

Deze appendix geeft een impressie van de C# implementatie van de Aterm bibliotheek.. De volledige implementatie bestaat uit de volgende interfaces, classes en enums:
(van de cursief opgenomen classes volgt hierna de sourcecode)

Type	Naam	Extend en Implementeert
Publieke interface		
I	ATerm	
I	ATermAppl	ATerm
I	ATermBlob	ATerm
I	ATermInt	ATerm
I	ATermList	ATerm
I	ATermPlaceholder	ATerm
I	ATermReal	ATerm
I	Visitable	
I	ATermVisitable	Visitable
I	ATermVisitor	
E	ATermType	
I	AFun	
Implementatie		
C	ATermVisitableImpl	ATermVisitable
C	ATermImple	ATermVisitableImpl, ATerm
C	ATermApplImpl	ATermImpl, ATermAppl
C	ATermBlobImpl	ATermImpl, ATermBlob
C	ATermIntImpl	ATermImpl, ATermInt
C	ATermListImpl	ATermImpl, ATermList
C	ATermPlaceholderImpl	ATermImpl, ATermPlaceholder
C	ATermRealImpl	ATermImpl, ATermReal
C	*ATermReader*	
C	ATermWriter	ATermVisitor
C	*HashedWeakRef*	System.WeakReference
C	*PureFactory*	
I	Visitor	
C	TopDown	Visitor
I	Visitable	

```
namespace aterm.Pure
{
	using System;
	using System.IO;
	using System.Collections;
	using System.Text;
	using aterm;

	[CLSCompliant(true)]
	public class PureFactory : ATermFactory
	{
		private const int DEFAULT_TERM_TABLE_SIZE = 43117;
		private const int DEFAULT_AFUN_TABLE_SIZE = 2003;
		private const int DEFAULT_PATTERN_CACHE_SIZE = 4321;

		private int term_table_size;
		private HashedWeakRef[] term_table;

		private int afun_table_size;
		private HashedWeakRef[] afun_table;

		static public ATermList Empty;

		static bool IsBase64(int c)
		{
			return Char.IsLetterOrDigit((char)c) || c == '+' || c == '/';
		}

		static public int AbbrevSize(int abbrev)
		{
			int size = 1;

			if (abbrev == 0)
				return 2;

			while (abbrev > 0)
			{
				size++;
				abbrev /= 64;
			}

			return size;
		}

		public PureFactory()
			: this(DEFAULT_TERM_TABLE_SIZE, DEFAULT_AFUN_TABLE_SIZE)
		{
		}

		public PureFactory(int term_table_size, int afun_table_size)
		{
			this.term_table_size = term_table_size;
			this.term_table = new HashedWeakRef[term_table_size];

			this.afun_table_size = afun_table_size;
			this.afun_table = new HashedWeakRef[afun_table_size];

			Empty = new ATermListImpl(this);
			term_table[Empty.GetHashCode() % this.term_table_size] =
					new HashedWeakRef(Empty, null);
		}

		public ATermInt MakeInt(int val)
		{
			return MakeInt(val, Empty);
		}
```

```
public ATermInt MakeInt(int val, ATermList annos)
{
	ATerm term;
	int hnr = ATermIntImpl.hashFunction(val, annos);
	int idx = hnr % term_table_size;

	HashedWeakRef prev, cur;
	prev = null;
	cur  = term_table[idx];

	while (cur != null)
	{
		term = (ATerm)cur.Target;
		if (term == null)
		{
			// Found a reference to a garbage collected term,
			// remove it to speed up lookups.
			if (prev == null)
				term_table[idx] = cur.next;
			else
				prev.next = cur.next;
		}
		else if ((term is ATermInt) &&
			((ATermInt)term).GetInt() == val &&
				term.GetAnnotations() == annos)
		{
			return (ATermInt)term;
		}
		else
			prev = cur;

		cur = cur.next;
	}

	// No integer term with 'val' found, so let's create one!
	term = new ATermIntImpl(this, val, annos);
	term_table[idx] = new HashedWeakRef(term, term_table[idx]);

	return (ATermInt)term;
}

public ATermReal MakeReal(double val)
{
	return MakeReal(val, Empty);
}

public ATermReal MakeReal(double val, ATermList annos)
{
	ATerm term;
	int hnr = ATermRealImpl.HashFunction(val, annos);
	int idx = hnr % term_table_size;

	HashedWeakRef prev, cur;
	prev = null;
	cur  = term_table[idx];
	while (cur != null)
	{
		term = (ATerm)cur.Target;
		if (term == null)
		{
			// Found a reference to a garbage collected term,
			// remove it to speed up lookups.
			if (prev == null)
				term_table[idx] = cur.next;
			else
				prev.next = cur.next;
		}
```

```
            else if ((term is ATermReal) &&
                                ((ATermReal)term).GetReal() == val &&
                                term.GetAnnotations() == annos)
            {
                return (ATermReal)term;
            }
            else
                  prev = cur;

            cur = cur.next;
      }

      // No real term with 'val' found, so let's create one!
      term = new ATermRealImpl(this, val, annos);
      term_table[idx] = new HashedWeakRef(term, term_table[idx]);

      return (ATermReal)term;
}

public ATermList MakeList()
{
      return Empty;
}

public ATermList MakeList(ATerm singleton)
{
      return MakeList(singleton, Empty, Empty);
}

public ATermList MakeList(ATerm first, ATermList next)
{
      return MakeList(first, next, Empty);
}

public ATermList MakeList(ATerm first, ATermList next,
                          ATermList annos)
{
      ATerm term;
      int hnr = ATermListImpl.hashFunction(first, next, annos);
      int idx = hnr % term_table_size;

      HashedWeakRef prev, cur;
      prev = null;
      cur  = term_table[idx];
      while (cur != null)
      {
            term = (ATerm)cur.Target;
            if (term == null)
            {
                  // Found a reference to a garbage collected term,
                  // remove it to speed up lookups.
                  if (prev == null)
                        term_table[idx] = cur.next;
                  else
                        prev.next = cur.next;
            }
            else
            {
                  if (term is ATermList)
                  {
                        ATermList list = (ATermList)term;
```

```
                        if (list.GetFirst() == first &&
                              list.GetNext() == next
                              && term.GetAnnotations() == annos)
                        {
                              return list;
                        }
                  }
                  prev = cur;
            }
            cur = cur.next;
      }

      // No existing term found, so let's create one!
      term = new ATermListImpl(this, first, next, annos);
      term_table[idx] = new HashedWeakRef(term, term_table[idx]);

      return (ATermList)term;
}

public ATermPlaceholder MakePlaceholder(ATerm type)
{
      return MakePlaceholder(type, Empty);
}

public ATermPlaceholder MakePlaceholder(ATerm type, ATermList annos)
{
      ATerm term;
      int hnr = ATermPlaceholderImpl.hashFunction(type, annos);
      int idx = hnr % term_table_size;

      HashedWeakRef prev, cur;
      prev = null;
      cur  = term_table[idx];
      while (cur != null)
      {
            term = (ATerm)cur.Target;
            if (term == null)
            {
                  // Found a reference to a garbage collected term,
                  // remove it to speed up lookups.
                  if (prev == null)
                        term_table[idx] = cur.next;
                  else
                        prev.next = cur.next;
            }
            else
            {
                  if (term is ATermPlaceholder)
                  {
                        ATermPlaceholder ph =(ATermPlaceholder)term;
                        if (ph.GetPlaceholder() == type &&
                              term.GetAnnotations() == annos)
                        {
                              return ph;
                        }
                  }
                  prev = cur;
            }
            cur = cur.next;
      }

      // No existing term found, so let's create one!
      term = new ATermPlaceholderImpl(this, type, annos);
      cur = new HashedWeakRef(term, term_table[idx]);
      term_table[idx] = cur;

      return (ATermPlaceholder)term;
}
```

```
public ATermBlob MakeBlob(byte[] data)
{
	return MakeBlob(data, Empty);
}

public ATermBlob MakeBlob(byte[] data, ATermList annos)
{
	ATerm term;
	int hnr = ATermBlobImpl.HashFunction(data, annos);
	int idx = hnr % term_table_size;

	HashedWeakRef prev, cur;
	prev = null;
	cur  = term_table[idx];
	while (cur != null)
	{
		term = (ATerm)cur.Target;
		if (term == null)
		{
			// Found a reference to a garbage collected term,
			// remove it to speed up lookups.
			if (prev == null)
				term_table[idx] = cur.next;
			else
				prev.next = cur.next;
		}
		else
		{
			if (term is ATermBlob)
			{
				ATermBlob blob = (ATermBlob)term;
				if (blob.GetBlobData() == data &&
					term.GetAnnotations() == annos)
				{
					return blob;
				}
			}
			prev = cur;
		}
		cur = cur.next;
	}

	// No existing term found, so let's create one!
	term = new ATermBlobImpl(this, data, annos);
	cur = new HashedWeakRef(term, term_table[idx]);
	term_table[idx] = cur;

	return (ATermBlob)term;
}

public AFun MakeAFun(string name, int arity, bool IsQuoted)
{
	AFun fun;
	int hnr = AFunImpl.HashFunction(name, arity, IsQuoted);
	int idx = hnr % afun_table_size;

	name = string.Intern(name);
	HashedWeakRef prev, cur;
	prev = null;
	cur  = afun_table[idx];
	while (cur != null)
	{
		fun = (AFun)cur.Target;
		if (fun == null)
		{
			// Found a reference to a garbage collected term,
			// remove it to speed up lookups.
```

```
                        if (prev == null)
                                afun_table[idx] = cur.next;
                        else
                                prev.next = cur.next;
                }
                else
                {
                        if (fun.GetName() == name && fun.GetArity() ==
                                arity && fun.IsQuoted() == IsQuoted)
                                return fun;

                        prev = cur;
                }
                cur = cur.next;
        }

        // No similar AFun found, so build a new one
        fun = new AFunImpl(name, arity, IsQuoted);
        afun_table[idx] = new HashedWeakRef(fun, afun_table[idx]);

        return fun;
}

public ATermAppl MakeAppl(AFun fun, ATerm[] args)
{
        return MakeAppl(fun, args, Empty);
}

public ATermAppl MakeAppl(AFun fun, ATerm[] args, ATermList annos)
{
        ATerm term;
        int hnr;
        int idx;
        HashedWeakRef prev, cur;

        if (fun.GetArity() != args.Length)
        {
                throw new ArgumentException("arity does not match " +
                                        "argument count: " +
                                        fun.GetArity()+ " != " +args.Length);
        }

        hnr = ATermApplImpl.HashFunction(fun, args, annos);
        idx = hnr % term_table_size;

        prev = null;
        cur  = term_table[idx];
        while (cur != null)
        {
                term = (ATerm)cur.Target;
                if (term == null)
                {
                        // Found a reference to a garbage collected term,
                        // remove it to speed up lookups.
                        if (prev == null)
                                term_table[idx] = cur.next;
                        else
                                prev.next = cur.next;
                }
                else
                {
                        if (term is ATermAppl)
                        {
                                ATermAppl appl = (ATermAppl)term;
                                if (appl.GetAFun() == fun)
                                {
                                        ATerm[] appl_args =
                                                appl.GetArgumentArray();
```

```
                                    if (appl_args.Length == args.Length &&
                                          term.GetAnnotations() == annos)
                                    {
                                          bool found = true;
                                          for (int i=0; i<args.Length;
                                                                      i++)
                                          {
                                                if(appl_args[i] !=
                                                            args[i])
                                                {
                                                      found = false;
                                                      break;
                                                }
                                          }
                                          if (found)
                                                return appl;
                                    }
                              }
                        }
                        prev = cur;
                  }
                  cur = cur.next;
            }

            // No existing term found, so let's create one!
            term = new ATermApplImpl(this, fun, args, annos);
            term_table[idx] = new HashedWeakRef(term, term_table[idx]);

            return (ATermAppl)term;
      }

      public ATermAppl MakeAppl(AFun fun, ATermList args)
      {
            ATerm[] arg_array;

            arg_array = new ATerm[args.GetLength()];

            int i = 0;
            while (!args.IsEmpty())
            {
                  arg_array[i++] = args.GetFirst();
                  args = args.GetNext();
            }

            return MakeAppl(fun, arg_array);
      }

      public ATermAppl MakeAppl(AFun fun)
      {
            return MakeAppl(fun, new ATerm[0]);
      }

      public ATermAppl MakeAppl(AFun fun, ATerm arg)
      {
            ATerm[] args = { arg };
            return MakeAppl(fun, args);
      }

      public ATermAppl MakeAppl(AFun fun, ATerm arg1, ATerm arg2)
      {
            ATerm[] args = { arg1, arg2 };
            return MakeAppl(fun, args);
      }

      public ATermAppl MakeAppl(AFun fun, ATerm arg1, ATerm arg2,
                                                            ATerm arg3)
      {
            ATerm[] args = { arg1, arg2, arg3 };
```

```
	return MakeAppl(fun, args);
}

public ATermAppl MakeAppl(AFun fun, ATerm arg1, ATerm arg2,
                                        ATerm arg3, ATerm arg4)
{
	ATerm[] args = { arg1, arg2, arg3, arg4 };
	return MakeAppl(fun, args);
}

public ATermAppl MakeAppl(AFun fun, ATerm arg1,
                                        ATerm arg2, ATerm arg3,
                                        ATerm arg4, ATerm arg5)
{
	ATerm[] args = { arg1, arg2, arg3, arg4, arg5};
	return MakeAppl(fun, args);
}

public ATermAppl MakeAppl(AFun fun, ATerm arg1, ATerm arg2,
                                        ATerm arg3, ATerm arg4,
                                        ATerm arg5, ATerm arg6)
{
	ATerm[] args = { arg1, arg2, arg3, arg4, arg5, arg6 };
	return MakeAppl(fun, args);
}

public ATermAppl MakeAppl(AFun fun, ATerm arg1, ATerm arg2,
                                        ATerm arg3, ATerm arg4,
                                        ATerm arg5, ATerm arg6,
                                        ATerm arg7)
{
	ATerm[] args = { arg1, arg2, arg3, arg4, arg5, arg6, arg7 };
	return MakeAppl(fun, args);
}

public ATermList GetEmpty()
{
	return Empty;
}

private ATerm ParseAbbrev(ATermReader reader)
{
	ATerm result;
	int abbrev;

	int c = reader.read();

	abbrev = 0;
	while (IsBase64(c))
	{
		abbrev *= 64;
		if (c >= 'A' && c <= 'Z')
		abbrev += c - 'A';
		else if (c >= 'a' && c <= 'z')
			abbrev += c - 'a' + 26;
		else if (c >= '0' && c <= '9')
			abbrev += c - '0' + 52;
		else if (c == '+')
			abbrev += 62;
		else if (c == '/')
			abbrev += 63;
		else
			throw new ApplicationException("not a base-64"+
					" digit: " + c);

		c = reader.read();
	}
```

```
        result = reader.getTerm(abbrev);

        return result;
    }

    private ATerm ParseNumber(ATermReader reader)
    {
        StringBuilder str = new StringBuilder();
        ATerm result;

        do
        {
            str.Append((char)reader.getLastChar());
        }
        while(char.IsDigit((char)reader.read()));

        if(reader.getLastChar() != '.' &&
            reader.getLastChar() != 'e' &&
            reader.getLastChar() != 'E')
        {
            int val;
            try
            {
                val = int.Parse(str.ToString());
            }
            catch (Exception e)
            {
                throw new ParseErrorException("malformed int", e);
            }
            result = MakeInt(val);
        }
        else
        {
            if(reader.getLastChar() == '.')
            {
                str.Append('.');
                reader.read();
                if(!char.IsDigit((char)reader.getLastChar()))
                    throw new ParseErrorException(
                                "digit expected");
                do
                {
                    str.Append((char)reader.getLastChar());
                }
                while(char.IsDigit((char)reader.read()));
            }
            if(reader.getLastChar() == 'e' ||
                reader.getLastChar() == 'E')
            {
                str.Append((char)reader.getLastChar());
                reader.read();
                if(reader.getLastChar() == '-' ||
                    reader.getLastChar() == '+')
                {
                    str.Append((char)reader.getLastChar());
                    reader.read();
                }

                if(!char.IsDigit((char)reader.getLastChar()))
                    throw new ParseErrorException(
                            "digit expected!");
                do
                {
                    str.Append((char)reader.getLastChar());
                }
                while(char.IsDigit((char)reader.read()));
            }
            double val;
```

```
            try
            {
                  val = double.Parse(str.ToString());
            }
            catch (Exception e)
            {
                  throw new ParseErrorException("malformed real", e);
            }
            result = MakeReal(val);
      }
      return result;
}

private string ParseId(ATermReader reader)
{
      int c = reader.getLastChar();
      StringBuilder buf = new StringBuilder(32);

      do
      {
            buf.Append((char)c);
            c = reader.read();
      }
      while (char.IsLetterOrDigit((char)c) || c == '_' || c == '-');

      return buf.ToString();
}

private string ParseString(ATermReader reader)
{
      bool escaped;
      StringBuilder str = new StringBuilder();

      do
      {
            escaped = false;
            if(reader.read() == '\\')
            {
                  reader.read();
                  escaped = true;
            }

            if(escaped)
            {
                  switch(reader.getLastChar())
                  {
                        case 'n':    str.Append('\n');  break;
                        case 't':    str.Append('\t');  break;
                        case 'b':    str.Append('\b');  break;
                        case 'r':    str.Append('\r');  break;
                        case 'f':    str.Append('\f');  break;
                        case '\\':   str.Append('\\');  break;
                        case '\'':   str.Append('\'');  break;
                        case '\"':   str.Append('\"');  break;
                        case '0': case '1': case '2': case '3':
                        case '4': case '5':case '6': case '7':
                          str.Append(reader.ReadOct());
                          break;
                        default:
                              str.Append('\\')
                               .Append((char)reader.getLastChar());
                              break;
                  }
            }
            else
                  if(reader.getLastChar() != '\"')
                        str.Append((char)reader.getLastChar());
      } while(escaped || reader.getLastChar() != '"');
```

```
        return str.ToString();
}

private ATermList ParseATerms(ATermReader reader)
{
        ATerm[] terms = ParseATermsArray(reader);
        ATermList result = Empty;
        for (int i=terms.Length-1; i>=0; i--)
        {
                result = MakeList(terms[i], result);
        }

        return result;
}

private ATerm[] ParseATermsArray(ATermReader reader)
{
        ArrayList list = new ArrayList();
        ATerm term;

        term = ParseFromReader(reader);
        list.Add(term);
        while (reader.getLastChar() == ',') {
                reader.readSkippingWS();
                term = ParseFromReader(reader);
                list.Add(term);
        }

        ATerm[] array = new ATerm[list.Count];
        int index = 0;

        foreach(object o in list)
        {
                array[index++] = (ATerm)o;
        }

        return array;
}

private ATerm ParseFromReader(ATermReader reader)
{
        ATerm result;
        int c, start, end;
        string funname;

        start = reader.getPosition();
        switch(reader.getLastChar())
        {
        case -1:
                throw new ParseErrorException(
                        "premature EOF encountered.");

        case '#':
                return ParseAbbrev(reader);

        case '[':
                // Read a list

                c = reader.readSkippingWS();
                if (c == -1)
                        throw new ParseErrorException(
                                "premature EOF encountered.");

                if(c == ']')
                {
                        c = reader.readSkippingWS();
                        result = Empty;
```

```
                        }
                        else
                        {
                                result = ParseATerms(reader);
                                if(reader.getLastChar() != ']')
                                        throw new ParseErrorException(
                                                "expected ']' but got '" +
                                                (char)reader.getLastChar() + "'");

                                c = reader.readSkippingWS();
                        }
                        break;

                case '<':
                        // Read a placeholder

                        c = reader.readSkippingWS();
                        ATerm ph = ParseFromReader(reader);

                        if (reader.getLastChar() != '>')
                                throw new ParseErrorException(
                                        "expected '>' but got '" +
                                        (char)reader.getLastChar() + "'");

                        c = reader.readSkippingWS();

                        result = MakePlaceholder(ph);

                        break;

        case '"':
                        // Read a quoted function

                        funname = ParseString(reader);

                        c = reader.readSkippingWS();
                        if (reader.getLastChar() == '(')
                        {
                                c = reader.readSkippingWS();

                                if (c == -1)
                                        throw new ParseErrorException(
                                                "premature EOF encountered.");

                                if (reader.getLastChar() == ')')
                                        result = MakeAppl(MakeAFun(funname, 0,
true));
                                else
                                {
                                        ATerm[] list = ParseATermsArray(reader);

                                        if(reader.getLastChar() != ')')
                                                throw new ParseErrorException(
                                                        "expected ')' but got '" +
                                                        reader.getLastChar() + "'");

                                        result = MakeAppl(MakeAFun(funname,
                                                        list.Length, true), list);
                                }

                                c = reader.readSkippingWS();
                                if (c == -1)
                                        throw new ParseErrorException(
                                                "premature EOF encountered.");
                        }
                        else
                                result = MakeAppl(MakeAFun(funname, 0, true));
```

```
break;

   case '-':
   case '0':    case '1': case '2': case '3': case '4':
   case '5':    case '6': case '7': case '8': case '9':
         result = ParseNumber(reader);
         c = reader.skipWS();
         break;

   default:
         c = reader.getLastChar();
         if (char.IsLetter((char)c))
         {
               // Parse an unquoted function

               funname = ParseId(reader);
               c = reader.skipWS();
               if (reader.getLastChar() == '(')
               {
                     c = reader.readSkippingWS();

                     if (c == -1)
                           throw new ParseErrorException(
                                 "premature EOF encountered.");

                     if (reader.getLastChar() == ')')
                           result = MakeAppl(MakeAFun(funname, 0,
                                       false));
                     else
                     {
                           ATerm[] list =
                                 ParseATermsArray(reader);

                           if(reader.getLastChar() != ')')
                                 throw new ParseErrorException(
                                       "expected ')' but got '"
                                       + reader.getLastChar()
                                       + "'");

                           result = MakeAppl(MakeAFun(funname,
                                 list.Length, false), list);
                     }
                     c = reader.readSkippingWS();
               }
               else
                     result = MakeAppl(MakeAFun(funname, 0,
                                 false));
         }
         else
               throw new ParseErrorException("illegal character: "
                           + reader.getLastChar());
         break;
   }

   if(reader.getLastChar() == '{')
   {
         ATermList annos;
         // Parse annotation

         if(reader.readSkippingWS() == '}')
         {
               reader.readSkippingWS();
               annos = Empty;
         }
         else
         {
               annos = ParseATerms(reader);
```

```
                if(reader.getLastChar() != '}')
                        throw new ParseErrorException(
                                "'}' expected");

                reader.readSkippingWS();
            }
            result = result.SetAnnotations(annos);

        }

        // Parse some ToolBus anomalies for backwards compatibility

        if(reader.getLastChar() == ':')
        {
            reader.read();
            ATerm anno = ParseFromReader(reader);
            result = result.SetAnnotation(Parse("type"), anno);
        }

        if(reader.getLastChar() == '?')
        {
            reader.readSkippingWS();
            result = result.SetAnnotation(Parse("result"),
                        Parse("true"));
        }

        end = reader.getPosition();
        reader.storeNextTerm(result, end-start);

        return result;
    }

    public ATerm Parse(string trm)
    {
        ATermReader reader = new ATermReader(new StringReader(trm));
        reader.readSkippingWS();
        ATerm result = ParseFromReader(reader);

        return result;
    }

    public ATerm Make(String trm)
    {
        return Parse(trm);
    }

    public ATerm Make(string pattern, ArrayList args)
    {
        return Make(Parse(pattern), args);
    }

    public ATerm Make(string pattern, params object[] args)
    {
                    // implicit conversion
        return Make(pattern, /*new ArrayList(*/ args /*)*/);
    }

    public ATerm Make(ATerm pattern, ArrayList args)
    {
        return pattern.Make(args);
    }

    public ATerm ParsePattern(string pattern)
    {
        // <TODO>: cache patterns
        return Parse(pattern);
    }
```

```
public bool IsDeepEqual(ATermImpl t1, ATerm t2)
{
      throw new NotImplementedException("not yet implemented!");
}

private ATerm ReadFromSharedTextFile(ATermReader reader)
{
      reader.InitializeSharing();
    return ParseFromReader(reader);
}

private ATerm ReadFromTextFile(ATermReader reader)
{
      return ParseFromReader(reader);
}

public ATerm ReadFromTextFile(Stream stream)
{
      ATermReader reader = new ATermReader(new StreamReader(stream));
      reader.readSkippingWS();

      return ReadFromTextFile(reader);
}

public ATerm ReadFromSharedTextFile(Stream stream)
{
      ATermReader reader = new ATermReader(new StreamReader(stream));
      reader.readSkippingWS();

      if (reader.getLastChar() != '!')
      {
            throw new IOException("not a shared text file!");
      }

      reader.readSkippingWS();

      return ReadFromSharedTextFile(reader);
}

public ATerm ReadFromBinaryFile(Stream stream)
{
      throw new NotImplementedException("not yet implemented!");
}

public ATerm ReadFromFile(Stream stream)
{
      ATermReader reader = new ATermReader(new StreamReader(stream));
      reader.readSkippingWS();

      int last_char = reader.getLastChar();
      if (last_char == '!')
      {
            reader.readSkippingWS();
            return ReadFromSharedTextFile(reader);
      }
      else if (char.IsLetterOrDigit((char)last_char) ||
       last_char == '_' || last_char == '[' || last_char == '-')
      {
            return ReadFromTextFile(reader);
      }
      else
            throw new ApplicationException(
                  "BAF files are not supported by this factory.");
}

public ATerm ReadFromFile(string file)
{
```

```
		return ReadFromFile(new FileStream(file, FileMode.Open));
	}

	public ATerm ImportTerm(ATerm term)
	{
		throw new NotImplementedException("not yet implemented!");
	}
}

class HashedWeakRef : WeakReference
{
	internal protected HashedWeakRef next;

	public HashedWeakRef(object ob, HashedWeakRef next) : base(ob)
	{
		this.next = next;
	}
}

class ATermReader
{
	private const int INITIAL_TABLE_SIZE = 2048;
	private const int TABLE_INCREMENT    = 4096;
	private TextReader reader;
	private int last_char;
	private int pos;

	private int nr_terms;
	private ATerm[] table;

	public ATermReader(TextReader reader)
	{
		this.reader = reader;
		last_char   = -1;
		pos = 0;
	}

	public void InitializeSharing()
	{
		table = new ATerm[INITIAL_TABLE_SIZE];
		nr_terms = 0;
	}

	public void storeNextTerm(ATerm t, int size)
	{
		if (table == null)
			return;

		if (size <= PureFactory.AbbrevSize(nr_terms))
			return;

		if (nr_terms == table.Length)
		{
			ATerm[] new_table = new
				ATerm[table.Length+TABLE_INCREMENT];

			table.CopyTo(new_table, 0);
			table = new_table;
		}

		table[nr_terms++] = t;
	}

	public ATerm getTerm(int index)
	{
		if (index < 0 || index >= nr_terms)
			throw new ArgumentException("illegal index");
```

```
        return table[index];
}

public int read()
{
        last_char = reader.Read();
        pos++;
        return last_char;
}

public int readSkippingWS()
{
        do
        {
                last_char = reader.Read();
                pos++;
        }
        while (char.IsWhiteSpace((char)last_char));

        return last_char;
}

public int skipWS()
{
        while (char.IsWhiteSpace((char)last_char))
        {
                last_char = reader.Read();
                pos++;
        }

        return last_char;
}

public bool IsOctDigit(char c)
{
        return (c >= '0') && (c <= '7');
}

public int ReadOct()
{
        char c2, c3;

        c2 = (char)read();
        if(!IsOctDigit(c2))
                throw new ParseErrorException(
                        "octal must have 3 octdigits.");

        c3 = (char)read();
        if(!IsOctDigit(c3))
                throw new ParseErrorException(
                        "octal must have 3 octdigits.");

        int nr =     (int)c3 - (int)'0';
        nr += 8   * ((int)c2 - (int)'0');
        nr += 8*8 * ((int)last_char - (int)'0');

        return nr;
}

public int getLastChar()
{
        return last_char;
}
```

```
		public int getPosition()
		{
			return pos;
		}
	}
}
```

B.Bronvermelding

[Bergstra98] The ToolBus Coordination Architecture
J.A. Bergstra and P. Klint, In P. Ciancarini and C. Hankin (eds) Proc. 1st Int. Conf. on Coordination Models and Languages, LNCS 1061, 75--88, 1996.

[Bergstra99] Programs, Interfaces and Components
Jan Bergstra & Marijke Loots, Universiteit Utrecht, Preprint nr. 6, July 1999

[Bettini01] A Java 2 Network Class Loader
Lorenzo Bettini e.a., Dr. Dobbs Journal, Februari 2001

[Brand00] Efficient Annotated Terms
M.G.J. van den Brand, H.A. de Jong, P. Klint and P.A. Olivier, Software, Practice & Experience, 30(3):259--291, 2000.

[Brand01] The ASF+SDF Meta-Environment: a Component-Based Language Development Environment.
M.G.J. van den Brand, A. van Deursen, J. Heering, H.A. de Jong, M. de Jonge, T. Kuipers, P. Klint, L. Moonen, P.A. Olivier, J. Scheerder, J.J. Vinju, E. Visser, and J. Visser, in: R. Wilhelm (ed). Proceedings of Compiler Construction (CC'01), LNCS 2027, 365--370, 2001.

[Brand99] M.G.J. van den Brand, P. Klint and P.A. Olivier, Compilation and Memory
Management for ASF+SDF. in: Proceedings of Compiler Construction (CC'99) (ed. S. Jähnichen), volume 1575 of LNCS, 198--213, Springer-Verlag, 1999

[Brockschmidt96] What OLE Is *Really* About
Kraig Brockschmidt, OLE Team, Microsoft Corploration, July 1996
http://msdn.microsoft.com/library/en-us/dnolegen/html/msdn_aboutole.asp

[chappell00] Presentaties van David Chappell op Tech Ed 2000, Amsterdam
Vergelijking van Windows DNA en Enterprise Java

[dotgnu] DotGNU Portable.NET
http://www.southern-storm.com.au/portable_net.html

[Eady01] C# versus Java
Marc Eaddy, Dr. Dobbs Journal, Februari 2001

[Farley01] Microsoft .NET vs. J2EE: How Do They Stack Up?
Jim Farley, 2001

[Grune00] Modern Compiler Design
Dick Grune, Henri E. Bal, Ceriel J.H. Jacobs and Koen G. Langendoen
Free University, Amsterdam
Wiley & Sons, 2000

[ia64] Intel IA-64 Software Conventions and Runtime Architecture Guide
Intel Corporation, September 2000

[Klint93] A Meta-Environment for Generating Programming Environments
P. Klint, ACM Transactions on Software Engineering and Methodology, 2(2):176-201,1993.

[Leidundguth01] A Universal Model for Designing the Entity EJB Layer
Reducing development time and improving maintainability.
J. Leidundguth, M. Pease and A. Povodyrev - Dr. Dobbs Journal, June 2001

[mono] The Mono Project, an effort to create an Open Source implementation of the .NET Development Framework.
http://www.go-mono.com/

[ocl] Intel Labs C# implementation of portions of the CLI
http://www.sourceforge.net/projects/ocl/

[Olivier00] A Framework for Debugging Heterogeneous Applications
Pieter A. Olivier, Dissertation, Universiteit van Amsterdam, 2000
http://www.cwi.nl/~paulk/dissertations/Olivier.ps.gz

[openssl] OpenSSL – The Open Source toolkit for SSL/TLS
http://www.openssl.org/

[pcint5] PC Intern 5
Easy Computing N.V., 1996
(Vertaling van de Duitse versie van Data Becker Gmbh, 1996)

[Plainfosse94] Distributed Garbage Collection and Referencing Management in the Soul Object Support system
David Plainfossé, Université Paris VI, 1994

[soap] Simple Object Access Protocol (SOAP) 1.1
W3C, Mei 2000
http://www.w3.org/TR/SOAP/

[Visser01] A ToolBus Prototype in Java
C. Visser, Master Thesis, University of Amsterdam
http://www.cwi.nl/~paulk/theses/CVisser.pdf

[rfc1413] RFC 1413 - Identification Protocol (Ident)
[rfc1510] RFC 1510 - The Kerberos Network Authentication Service
[rfc1945] RFC 1945 - Hypertext Transfer Protocol - HTTP/1.0
[rfc2616] RFC 2616 - Hypertext Transfer Protocol - HTTP/1.1
[rfc2617] RFC 2617 - HTTP Authentication: Basic and Digest Access Authentication
[rfc2795] RFC 2795 - The Infinite Monkey Protocol Suite (IMPS)

Het merendeel van de systemen waar ik mee gewerkt heb waren tijdens dit project nog zo nieuw nog geen papieren boeken over waren verschenen. De volgende websites waren daarom van groot belang om inzicht te verkrijgen in de werking van de systemen:

- **HTTP://msdn.microsoft.com/net/**
  (Hierop is alle documentatie die bij de DotNet Sdk geleverd wordt te vinden)
- **HTTP://java.sun.com/j2ee/**
  (Alle Enterprise Java specificaties en voorbeelden)
- **HTTP://www.codeguru.com/**
  (Veel artikelen over DotNet en COM)

Voordat de publicatie van artikelen over DotNet op gang kwam werd er op een aantal plaatsen op het Internet al druk gediscussieerd over de mogelijkheden en onmogelijkheden van DotNet.

Daarnaast was de voorlopige documentatie van Visual Studio.NET, zoals die bijgeleverd werd bij de Beta 1 versie van dit systeem, van grote waarde bij het onderzoeken van de werking van DotNet. De archieven van deze groepen geven een aardig beeld van de gebruikerswensen van de betatesters van dit systeem.

- Nieuwsgroep: microsoft.public.beta.dotnet.visualc
  (Later hernoemt tot: microsoft.public.dotnet.languages.vc)
- Nieuwsgroep: microsoft.public.beta.dotnet.csharp
  (Later hernoemt tot: microsoft.public.dotnet.languages.csharp)
- Mailinglist: dotnet@discuss.develop.com
  (Had veel subscribers die al voor de officiële bekendmaking gebruik maakten van wat later DotNet zou gaan heten)