Faculteit Toegepaste Wetenschappen
Vakgroep Informatietechnologie
Voorzitter: Prof. Dr. Ir. P. Lagasse

Grondige performantiestudie van JAVA en .NET technologieën voor gedistribueerd software-ontwerp

door

Sofie Van Hoecke

Promotoren: Prof. Dr. Ir. B. Dhoedt, Dr. Ir. F. De Turck
Scriptiebegeleider: Ir. T. Verdickt

Afstudeerwerk ingediend tot het behalen van de academische graad van
Burgerlijk Ingenieur in de Computerwetenschappen

Academiejaar 2002–2003

Toelating tot bruikleen

De auteur geeft de toelating dit afstudeerwerk voor consultatie beschikbaar te stellen en delen van het afstudeerwerk te copiëren voor persoonlijk gebruik. Elk ander gebruik valt onder de beperkingen van het auteursrecht, in het bijzonder met betrekking tot de verplichting de bron uitdrukkelijk te vermelden bij het aanhalen van resultaten uit dit afstudeerwerk.

Sofie Van Hoecke

mei 2003

Dankwoord

Wat ooit als een verre droom begon, krijgt met het afwerken van deze scriptie langzamerhand vorm. Zonder de steun en vriendschap van een aantal ongelofelijke mensen, zou het echter altijd een droom gebleven zijn. Ik kan er uren over schrijven, met verbloemende woorden, pagina's vullend. Maar uiteindelijk volstaat één woord voor ieder onder jullie:

Bedankt!

Daarom wil ik eerst en vooral mijn promotoren, Prof. Dr. Ir. Bart Dhoedt en Dr. Ir. Filip De Turck, bedanken voor het aanbieden van een dergelijk interessant eindwerk, het ter beschikking stellen van de Atlantis-infrastructuur en de regelmatige opvolging van dit eindwerk.
Natuurlijk dank ik ook mijn thesisbegeleider, Tom Verdickt, voor zijn nooit aflatende blijk van interesse, zijn hulp en de wekelijkse opvolging.
Ook dank ik alle medewerkers van Atlantis die steeds bereid waren hulp te verschaffen.
In het bijzonder gaat mijn dank uit naar mijn ouders en zus zonder wiens onvoorwaardelijke steun dit allemaal nooit mogelijk was geweest.
Ik dank ook Kurt. Zijn opmerkingen en tips stimuleerden een grondige uitwerking van mijn onderzoek.
Ook gaat mijn dank uit naar mijn BC-buddies Sebastian, Wim, Stijn, Peter en Bruno zonder wie BC veel minder leuk was geweest.
En natuurlijk dank ik Elke, de beste vriendin die iemand zich kan wensen.
En tenslotte al mijn overige vrienden die ik niet bij naam kan noemen omdat dit dankwoord anders veel te lang zou worden. Dankzij hen waren de voorbije jaren een fijne en onvergetelijke tijd.

Sofie Van Hoecke
mei 2003

Grondige performantiestudie van JAVA en .NET technologieën voor gedistribueerd software-ontwerp

door

Sofie Van Hoecke

Afstudeerwerk ingediend tot het behalen van de academische graad van Burgerlijk Ingenieur in de Computerwetenschappen

Academiejaar 2002–2003

Promotoren: Prof. Dr. Ir. B. Dhoedt, Dr. Ir. F. De Turck
Scriptiebegeleider: Ir. T. Verdickt
Faculteit Toegepaste Wetenschappen
Universiteit Gent

Vakgroep Informatietechnologie
Voorzitter: Prof. Dr. Ir. P. Lagasse

Samenvatting

Eén van de meest gedebateerde vragen in de industrie vandaag handelt over de keuze voor Microsoft .NET of J2EE. Aangezien beide technologieën sterk op elkaar lijken, brengt een pure featurevergelijking niet echt grote verschillen aan het licht. Bijgevolg zal in dit afstudeerwerk een evaluatie en vergelijkende performantiestudie gemaakt worden van deze twee technologieën. Hierbij zal de aandacht vooral gaan naar databanktoegang via webapplicaties. Maar ook andere domeinen zullen kort verkend worden.

Trefwoorden

Gedistribueerde software, performantie, J2EE, .NET, servlet, enterprise javabean, JDBC, ASP, ASP.NET, ADO, ADO.NET, ODBC, OLEDB, native .NET Data Provider

Gebruikte afkortingen

ADO	ActiveX Data Object
API	Application Programming Interface
ART	Average Response Time
ASP	Active Server Pages
CLR	Common Language Runtime
CLS	Common Language Specification
COM	Component Object Model
CTS	Common Type System
DAO	Data Access Objects
DBMS	Database Management System
DLL	Dynamic Link Library
DSN	Data Source Name
EIS	Enterprise Information System
GSM	Global System for Mobile communication
GUI	Graphical User Interface
HTML	HyperText Markup Language
HTTP	HyperText Transfer Protocol
IIS	Internet Information Services
IL	Intermediate Language
JDBC	Java Database Connectivity
JIT	Just in Time
JMS	Java Message Service
JSP	Java Server Pages
JVM	Java Virtual Machine
ODBC	Open Database Connectivity
OLEDB	Object Linking and Embedding DataBase
PDA	Personal Digital Assistant
PE	Portable Executable

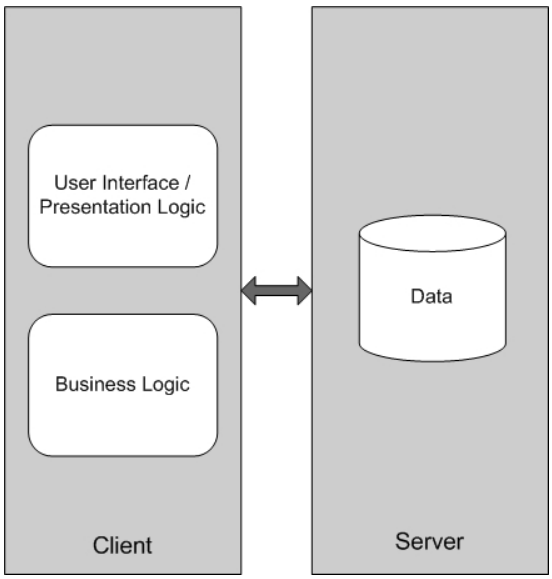
SDK	Software Development Kit
SMB	Small and Midsize Businesses
SPI	Service Provider Interface
SQL	Structured Query Language
TCP	Transmission Control Protocol
TPS	Transactions Per Second
URL	Uniform Resource Locator
VB	Visual Basic
WWW	World Wide Web
XML	Extensible Markup Language

Inhoudsopgave

Hoofdstuk 1

Inleiding

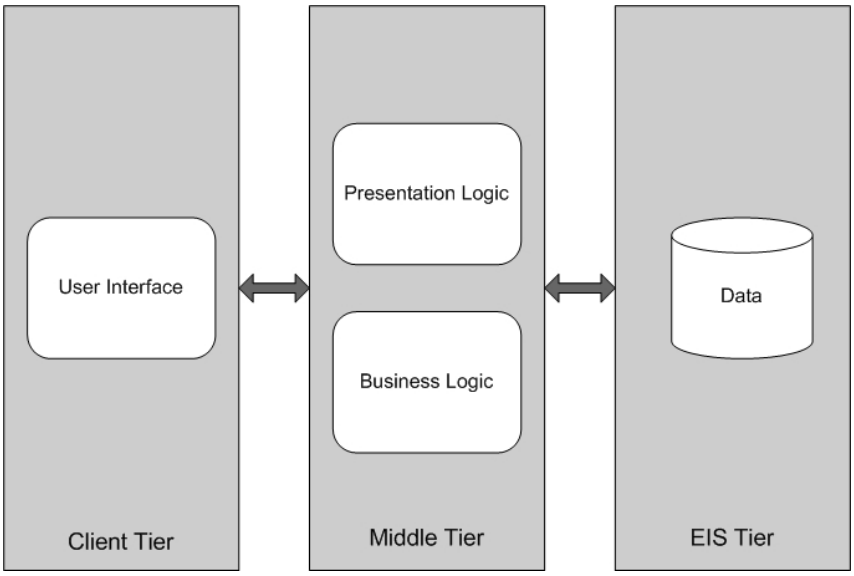
Begin de jaren 1990 gebruikten informatie-systemen vaak de client-server architectuur (zie Figuur 1.1). De gebruikersinterface naar de applicatie liep hierbij op de clientlaag. Terwijl de enterprise data op de server gestockeerd werd.



Figuur 1.1: Client-server architecture

Ervaring leerde echter dat het bouwen en instandhouden van een flexibel gedistribueerd systeem met deze architectuur heel moeilijk was. Bijvoorbeeld een upgrade moest op elke client uitgevoerd worden, wat een nogal omslachtige manier van werken met zich meebracht.

Als oplossing hiervoor kwam het meerlagenmodel (zie Figuur 1.2). Hier loopt de logica omtrent de gebruikersinterface naar de applicatie, alsook de business logica op de middelste laag. Wanneer er bijgevolg dus veranderingen nodig zijn, kunnen deze op één plaats geupdated worden in plaats van op elke client. Op dit moment zijn er twee verschillende



Figuur 1.2: Multi-tier model

kandidaten die voldoen aan de eisen die aan een platform voor moderne softwaresystemen worden gesteld. Aan de ene kant heeft men Microsoft met zijn .NET initiatief, terwijl men aan de andere kant Sun met J2EE heeft. De vraag of een ontwikkelaar met het oog op de toekomst het beste voor Microsoft .NET of voor Java kan kiezen, kan aan de hand van een pure featurevergelijking nauwelijks worden bepaald: daarvoor lijken de toepassingsgebieden die onder hun bereik vallen en de oplossingen gewoon te veel op elkaar. Bijgevolg zal in dit afstudeerwerk een evaluatie en vergelijkende performantiestudie gemaakt worden van deze twee principes en technologieën. Hierbij zal de aandacht vooral gaan naar databanktoegang via webapplicaties. Maar ook andere domeinen zullen kort verkend worden.

Doorheen dit afstudeerwerk zal de lezer eerst vertrouwd gemaakt worden met de twee gebruikte technologieën en op zoek gaan naar de eisen op vlak van performantie, om vervolgens applicaties op beide platformen van naderbij te bestuderen en te testen. Daarna wordt een bespreking gegeven van de bekomen testresultaten om zo de performantie van

beide platformen te kunnen vergelijken.

Het afstudeerwerk wordt als volgt opgedeeld:

Hoofdstuk 1: Inleiding Hoofdstuk 1 vormt de inleiding van dit afstudeerwerk.

Hoofdstuk 2: Basistechnologieën In dit hoofdstuk wordt de lezer vertrouwd gemaakt met de twee te onderzoeken technologieën. Zowel J2EE als .NET worden onder de loep genomen, waarna een vergelijkend overzicht tussen beiden gegeven wordt.

Hoofdstuk 3: Performantie Na het definiëren van performantie, wordt er dieper ingegaan op connection pooling en caching, twee methodes die kunnen gebruikt worden om de performantie van applicaties met databanktoegang te verbeteren.

Hoofdstuk 4: J2EE-applicaties In dit hoofdstuk worden eerst de mogelijkheden tot databanktoegang besproken waarbij dieper ingegaan wordt op JDBC. Vervolgens worden de ontwikkelde J2EE-applicaties, meer bepaald de servlet en de session enterprise javabean, meer in detail besproken.

Hoofdstuk 5: .NET-applicaties Startend met een korte schets over ADO en ADO.NET, wordt er overgegaan naar de verschillende connectiemogelijkheden in .NET: ODBC, OLEDB en de native .NET Data Provider. Hierna worden de ontwikkelde .NET applicaties nader bekeken voor de verschillende connectievarianten.

Hoofdstuk 6: Testresultaten Dit hoofdstuk zal beginnen met een beschouwing van de gebruikte testopstelling en het testscript. Vervolgens zullen de verkregen resultaten kritisch vergeleken worden op vlak van performantie.

Hoofdstuk 7: Besluit Rekening houdend met het volledige onderzoek, de testresultaten, maar ook de learning curve en serverconfiguraties, wordt er tot een besluit gekomen. Vervolgens worden nog enkele aspecten toegelicht die nog onderzocht kunnen worden.

Hoofdstuk 2

Basistechnologieën

2.1 J2EE

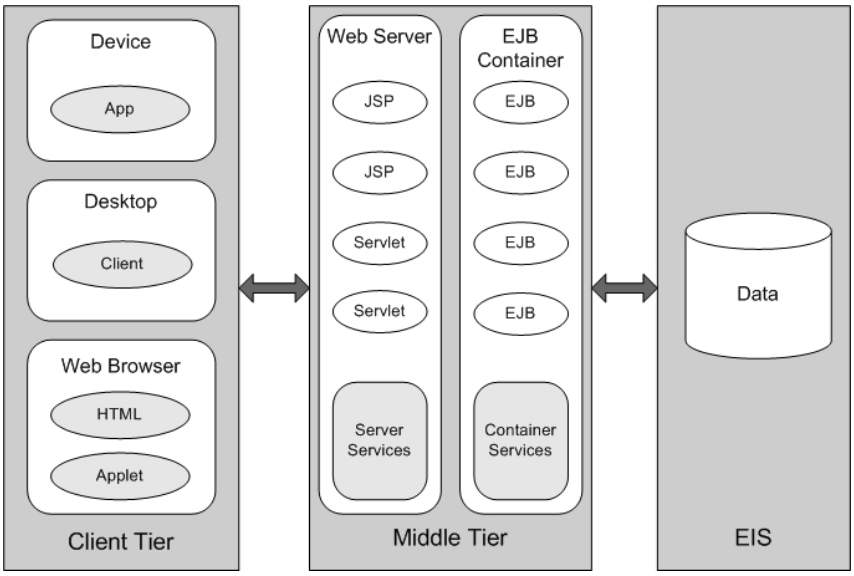
Het Java 2 Platform Enterprise Edition (J2EE) is een op Java gebaseerd ontwikkelingsplatform voor gedistribueerd software-ontwerp.
Het J2EE platform gebruikt een meerlagenmodel (zie Figuur 2.1). J2EE applicaties zijn opgemaakt uit componenten met elk hun eigen functie. Deze componenten kunnen op verschillende machines geïnstalleerd staan in overeenkomst met de laag in het applicatiemodel. Een J2EE-component is een ingebouwde functionele software-eenheid met gerelateerde klassen en bestanden, die communiceert met andere componenten.
In J2EE onderscheidt men volgende verschillende componenten:

- Client applicaties en applets zijn componenten die op de client lopen.
- Java Servlets en JavaServer Pages zijn web componenten die op de server lopen.
- Enterprise JavaBeans zijn business componenten die ook op de server lopen.

Client applicaties, applets, servlets, JSP en EJB zijn allemaal J2EE-componenten die geassembleerd zullen worden in een J2EE-applicatie.

J2EE applicaties worden gecompileerd tot Java Byte Code, een platform-onafhankelijke tussencode. Deze Java Byte Code wordt dan door een Java Virtual Machine (JVM) geïnterpreteerd. De Java Byte Code kan dus perfect gebruikt worden op verschillende

systemen, zolang deze maar over een JVM beschikken.



Figuur 2.1: J2EE Multi-tier model

Bekijkt men J2EE nu laag per laag:

- De client in een J2EE-applicatie kan een applet in een browser zijn, een Java-applicatie op een desktop machine, of zelfs een Java client op één of ander draagbaar toestel zoals een PDA of mobiele telefoon.
- De middelste laag kan Java Server Pages (JSP) of servlets runnen op een webserver. Terwijl een EJB Container op de middelste laag een runtime environment biedt voor de Enterprise JavaBeans (EJB). Er zijn verschillende soorten Enterprise JavaBeans:
  - Session Beans:

    Een session bean stelt één enkele client binnen de J2EE server voor. Om toegang te krijgen tot een applicatie ontwikkeld op de server, roept de client de methodes van de session bean op. Deze session bean voert dan werk uit voor de client zodat de client niet geconfronteerd wordt met de complexiteit van de taken op de server. Wanneer de client stopt met uitvoeren, verdwijnt de session bean en zijn data. Men onderscheidt twee soorten session beans:

    * Stateless Session Beans (meerdere clients)
    * Stateful Session Beans (enkele client)

  – Entity Beans:

    Een entity bean stelt persistente data voor, opgeslagen in een rij van een databanktabel. Wanneer de client of de server stopt, garanderen de onderliggende services dat de data van de entity bean bewaard blijft. Ook hier bestaan er twee soorten:

    * Container-managed Entity Beans
    * Bean-managed Entity Beans

  – Message-driven Beans:

    Deze beans combineren de mogelijkheden van een session bean en een Java Message Service (JMS) message listener. Deze bean maakt dat een J2EE applicatie asynchroon JMS berichten ontvangen. Bij session en entity beans is het immers enkel mogelijk JMS berichten synchroon te verzenden en ontvangen. In tegenstelling tot session en entity beans, krijgt men bij message-driven beans geen toegang via een interface.

- Op de figuur merkt men nog een derde laag op, de Enterprise Information System (EIS) laag, zij is verantwoordelijk voor de data van het bedrijf, bijv. in een relationele databank.

2.2 .NET

Maar er is naast J2EE nog een ander platform voor gedistribueerd software-ontwerp, nl. .NET van Microsoft.

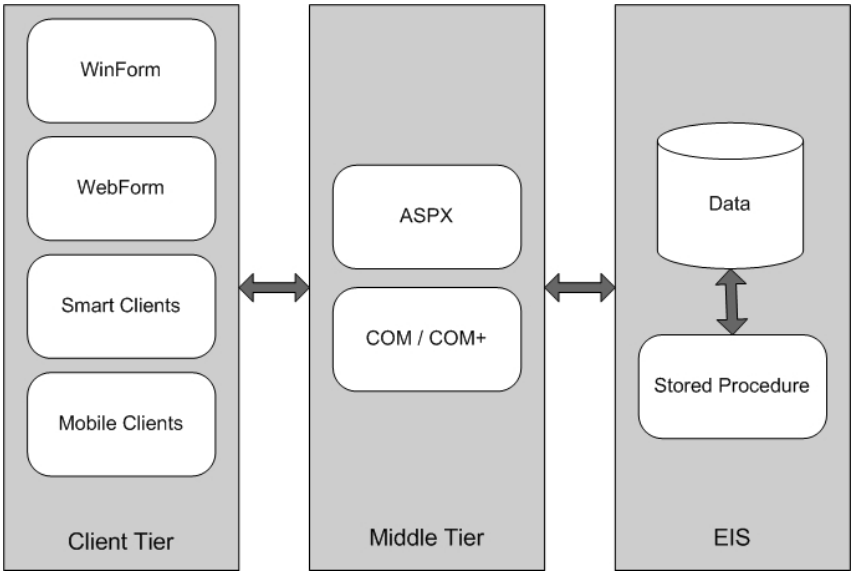
Microsoft maakt gebruik van een techniek die op Java lijkt: om ervoor te zorgen dat de .NET-software niet enkel op Windows platforms loopt, heeft men een soort virtuele machine gecreëerd, nl. de CLR (Common Language Runtime). Vandaag de dag bestaat er een CLR voor zowel Windows als Linux besturingssystemen. Er zijn ook CLR's op komst voor een aantal niet-PC toestellen. Alhoewel, het blijft niet bij het exporteren van een

CLR. Veel van de basiscomponenten en al de applicatie frameworks zijn direct verbonden met Windows. Dit betekent dat om het even welke niet-triviale applicatie gebouwd op het .NET platform, onder Windows moet draaien. Bijgevolg zal het extreem moeilijk zijn om een niet-triviale .NET applicatie naar een ander besturingssysteem over te dragen.
De basis van de CLR is een gezamelijke standaard voor objectgeoriënteerde programmeertalen (Common Language Specification, CLS) en hun typesysteem (Common Type System, CTS). Zo is in .NET elk object van een specifieke klasse, afgeleid van de root klasse System.Object. Omdat elke compiler moet voldoen aan een minimaal aantal regels bepaald door de CLS en code genereren conform met de CTS, kunnen verschillende .NET talen door elkaar gebruikt worden.
Aangezien elke compiler een Portable Executable (PE) file, geschreven in de 'Intermediate Language' (IL) van de CLR, creëert, is dit concept onafhankelijk van de programmeertaal. Deze .NET executables verschillen ten opzichte van de typische Windows executables in het feit dat ze naast code en data, ook metadata bevatten. In .NET bevat metadata type definities, versie informatie, externe assembly referenties en andere standaard informatie. Aangezien alle .NET PE files IL en metadata bevatten, maar geen native code, zal een 'Just in time' compiler (JIT) de IL converteren naar native code zodat ze kan uitgevoerd worden op het betreffende systeem. De JIT is een onderdeel van de CLR en compileert dus code geoptimaliseerd voor het systeem. Op deze manier kan dus in principe eenzelfde PE file op een totaal verschillend systeem gebruikt worden, bijv. Windows, Linux, zolang het systeem maar een CLR heeft.

Bekijkt men nu ook .NET laag per laag (zie Figuur 2.4):

- .NET ondersteunt verschillende types clients :
  - WinForm Clients
  - WebForm Clients
  - Smart Clients
  - Mobile Clients

  De WinForm Clients zijn de gewone Windows applicaties met een rijke GUI. Deze



Figuur 2.2: .NET Multi-tier model

clients moeten echter op elke client-pc geïnstalleerd worden en ze maken ook volledig gebruik van de capaciteiten van de client. Door de vereiste installatie op elke client, maakt dit dat het onderhoud niet gemakkelijk verloopt, elke client heeft bijv. aparte updates nodig.

Dan zijn er ook nog de WebForm Clients. Deze clients zijn als het ware een .NET oplossing om de WinForms op het web te brengen. Ze zijn gebaseerd op de ASP.NET server controllers en laten de ontwikkelaar toe webapplicaties te creëren in een event gebaseerd ontwikkelingssysteem. Voordeel hiervan is dat er geen installatie op de client-pc vereist is. Nadeel is echter dat er slechts beperkte GUI mogelijkheden zijn, alsook dat HTML beperkingen met zich meebrengt.

Als derde client zijn er de Smart Clients. Deze vormen een uitbreiding van de WinForm Clients. Het zijn WinForm applicaties die niet manueel geïnstalleerd moeten worden. Wanneer het nodig is, wordt de applicatie, of een deel ervan, van een centrale server gedownload. Ook wanneer er een update op de server plaatsvindt, worden de client applicaties automatisch geupdated bij de volgende run. Nadeel van deze methode is wel dat het downloaden voor een vertraging zorgt.

En als laatste heeft men dan nog de Mobile Clients die gebruikt worden op toestellen zoals bijvoorbeeld een PDA of een GSM.

- Op de middelste laag vindt men weerom de business logica. Hier onderscheidt men volgende methodes :
  - ASP.NET pagina's
  - Componenten : COM, COM+
  - Stored procedures die in de DBMS lopen

  Het Microsoft .NET platform biedt momenteel build-in support bij ASP.NET voor drie talen: C#, Visual Basic, and JScript. Het is geen goed idee om veel van de business logica te implementeren via ASP.NET pagina's omdat hierbij gebruik gemaakt wordt van scripting. Bijgevolg moet de code elke keer ze opgevraagd wordt, geïnterpreteerd worden wat tot lagere performantie leidt. Een betere manier is dus om de bedrijfslogica te implementeren met behulp van COM/COM+ componenten. Als alternatief is er ook de mogelijkheid om code te genereren die als stored procedure zal lopen in het databank management systeem (DBMS). Het schrijven en debuggen van stored procedures is vaak wel moeilijker dan van COM objecten. Ook het hergebruik van code bij COM componenten is gemakkelijker. Aan de andere kant brengt het gebruik van stored procedures een betere performantie met zich mee.

- Net als bij J2EE is ook bij .NET de derde laag verantwoordelijk voor de data van het bedrijf en vindt men hier dus ook bijv. de databanken terug.

2.3 J2EE vs. .NET

.NET en J2EE bieden in principe quasi hetzelfde pakket van mogelijkheden aan, zij het op verschillende manieren. Wanneer men beide technologieën met elkaar globaal vergelijkt, komt men tot volgende resultaten :

- Ondanks het feit dat er een CLR voor Linux bestaat, werkt .NET grotendeels op **Windows** platforms, terwijl J2EE op **meerdere platforms** draait.

- Het Java platform betekent ook **vrijheid van keuze**. Er zijn verschillende Java platform implementaties ontwikkeld door Sun, IBM, Symantec, Apple, open source

projecten,... Ontwikkelaars hebben de keuze uit verschillende tools. Terwijl .NET Microsoft betekent en als tool Visual Studio. .NET is dus een **verkoper-specifieke** oplossing waar keuze beperkt is.

- Java-compilers generen **Java bytecode**. Microsoft heeft zijn eigen tussentaal, de **IL**, gegenereerd door de CLR. Beiden ondersteunen object georiënteerd programmeren.

- Het runtime-systeem van beiden lijkt sterk op elkaar. Bij J2EE heeft men de **Java Virtual Machine** (JVM), terwijl men bij .NET de **Common Language Runtime** (CLR), met daarin de JIT, heeft. Een verschilpunt is echter dat de JVM enkel op zijn **eigen taal** afgestemd is, terwijl de CLR de mogelijkheid biedt met **meerdere talen** om te gaan: Managed C++, VB.NET, C# en J#.

- In .NET stamt elke klasse af van **System.Object**. Er bestaan twee types, nl. **value types en reference types**. Value types stellen waarden voor gealloceerd op de stack (passed by value). De reference values bevatten referenties naar objecten op de heap (passed by reference). In Java stamt elke klasse af van **java.lang.Object**. Java kent **primitieve types en klassen**. In .NET is er geen verschil tussen primitieve types en klassen.

- Beide platformen ondersteunen **quasi dezelfde statements**.

- .NET gebruikt **assemblies**, terwijl Java **jar**-files heeft. Op deze manier bundelt men meerdere bestanden in één enkele file wat onder andere een eenvoudige portabiliteit tot gevolg heeft.

- Als componenten heeft J2EE de beans. Er zijn vier verschillende soorten **JavaBeans**: stateful session beans, stateless session beans, entity beans en message driven beans. Bij .NET heeft men de **COM+** services.

- Database toegang gebeurt in .NET via **DAO, ADO of ADO.NET**. DAO is echter verouderd. ADO.NET is connectieloos of **offline**, terwijl ADO wel met een connectie werkt. ADO.NET is gebaseerd op XML. Bij Java kan men een relationele

databank raadplegen via **JDBC**, maar er zijn nog andere API's. In tegenstelling tot ADO.NET is JDBC gebaseerd op een connectie of dus **online**.

- Wat webgebaseerde applicaties betreft, voorziet .NET **ASP.NET**, terwijl men bij J2EE de **Java Server Pages (JSP) en Servlets** heeft.
- Beiden voorzien **Web Services**.
- Zowel Java als .NET voorzien de verwachte **XML** ondersteuning. Bij Java heeft men de JAXP. En bij .NET gebruiken de componenten XML voor hun data representatie.
- De twee technologieën hebben **error/exception handling**.
- Beiden doen aan **garbage collection**.

Hoofdstuk 3

Performantie

3.1 Definiëren van performantie

Er bestaat geen uniforme definitie van performantie. In sommige gevallen zal performantie betekenen: de mogelijkheid een groot aantal gebruikers toe te laten, terwijl het ook weer kan betekenen dat het programma zo snel mogelijk loopt. Performantie zal enerzijds afhangen van het soort applicatie en anderzijds van de parameters die zo goed mogelijk moeten zijn.
Men kan eigenlijk twee soorten applicaties onderscheiden :

- Interactieve applicaties
- Batch of back-end applicaties

Bij de interactieve applicaties is er constante interactie tussen de client en de applicatie, in tegenstelling tot de back-end applicatie waar dit niet het geval is.
Bij interactieve applicaties zal de performantie dus eerder bepaald worden door grootte en capaciteit, meer bepaald het ondersteunen van een groot aantal gebruikers terwijl voldaan is aan parameters als maximum antwoordtijd.
Bij back-end applicaties, die niet gebruikersgeoriënteerd zijn, is de belangrijkste performantiemaat echter throughput, uitgedrukt in Transactions Per Second (TPS).
Aangezien er interactieve applicaties zullen getest worden, zal de antwoordtijd een belangrijke performantiemaat zijn.

Onder antwoordtijd verstaat men de tijd nodig door de applicatie om een gegeven gebruikersvraag te beantwoorden. Dit vormt van de gebruikerskant gezien, de belangrijkste performantiemaat omdat voor hen de kwaliteit van een applicatie evenredig met de antwoordtijd is. Het is zelfs zo dat de functionaliteit en het gebruiksgemak minder invloed hebben op de gebruikersperceptie van de kwaliteit van de applicatie.
Aangezien de antwoordtijd rechtstreeks beïnvloed wordt door het aantal gebruikers die interageren met de applicatie op dat moment, moet men het aantal gelijktijdige gebruikers in betrekking nemen wanneer men de antwoordtijd bekijkt.
Men kan de antwoordtijd opsplitsen in meerdere componenten:

- verwerkingstijd (tijd vanaf dat de request de server bereikt tot wanneer het antwoord klaar is om verstuurd te worden)
- transmissietijd (tijd die de request er over doet om van de client naar de server en terug te gaan)
- renderingstijd (tijd die de client nodig heeft om het antwoord weer te geven of te verwerken)

3.2 Testen van performantie

Aangezien er twee verschillende platformen op de markt zijn voor gedistribueerd softwareontwerp, J2EE en .NET, is het van groot belang beiden te vergelijken aan de hand van zo uniform mogelijke performantietesten.
De Grinder is de performantie meettool die zal gebruikt worden bij het uitvoeren van deze testen.
De performantie-testen zullen telkens uit vier stappen bestaan:

- Definiëren van de performantiematen
- Definitie van test scripts, simuleren van het applicatie-gebruik.
- Definiëren van de test methode : grootte, duur, ...
- Uitvoeren van de test

Elke performantietest heeft verschillende test runs, elkeen gebruik makend van één of meerdere test scripts (real-life gebruik van de applicatie). Een test script is opgesteld uit verschillende requests. Er is per test run een vast aantal gebruikers. Een variabel aantal gebruikers heeft invloed op de resultaten en zou een statistisch verkeerd resultaat geven bij het vinden van het maximum aantal gebruikers.

Het is het eenvoudigst bij testen om alle requests in een script direct na elkaar uit te voeren, zonder tijd ertussen. In realiteit gebeurt dit echter zo niet. Daarom kan men een tijdsinterval tussen twee requests instellen, thinktime. Deze denktijd is heel cruciaal en zal een enorme invloed hebben op de geobserveerde antwoordtijden en throughput voor een gegeven gebruikersbelasting. Denktijd kan variëren van enkele seconden (de tijd om op een knop te drukken) tot enkele minuten (de tijd om banktransacties te bekijken). Omdat het gebruik van denktijd de testen verlengt, zal men steeds testen met zero think time en pas als men het meest optimale systeem heeft, dit nog eens hertesten met real think time. Het gebruik van zero think time zal wel leiden tot minder goede resultaten omdat de server onder een veel hogere stress zal staan. Dit wil niet zeggen dat de performantie slechter is. Het betekent gewoon testen onder andere en meer gestresseerde omstandigheden. Zero think time testen komen dus als het ware overeen met "worst-case scenario". Uit deze testen kan men leren wat de bottlenecks zijn die het het eerst zullen begeven onder hoge ongewone stresstoestanden.
Bij onze testen zullen ook alle gebruikers tegelijkertijd de scripts beginnen uitvoeren. Omdat dit allesbehalve realistisch is, kan men gebruik maken van een "initial spread". Op deze manier starten ze allen binnen deze periode in plaats van allen op hetzelfde moment.

Aangezien het inschatten van een realistische "thinktime", alsook van een realistische "initial spread", afhangt van de gebruikers, zal er steeds getest worden onder de hoogste stress. De resultaten zullen overeenstemmen met een worst-case scenario ten gevolge van de zero think time en omdat alle clients simultaan werken in plaats van met een initiële spreiding. Ook zullen er meerdere cycles doorlopen worden zodat men steeds een gemiddelde van de antwoordtijden kan nemen. Ook al voldoet deze Average Response Time

(ART) aan de performantievereisten, moeten we er rekening mee houden dat in sommige gevallen deze limiet niet bereikt zal worden omdat de antwoordtijd onderhevig is aan schommelingen.

3.3 Connection Pooling

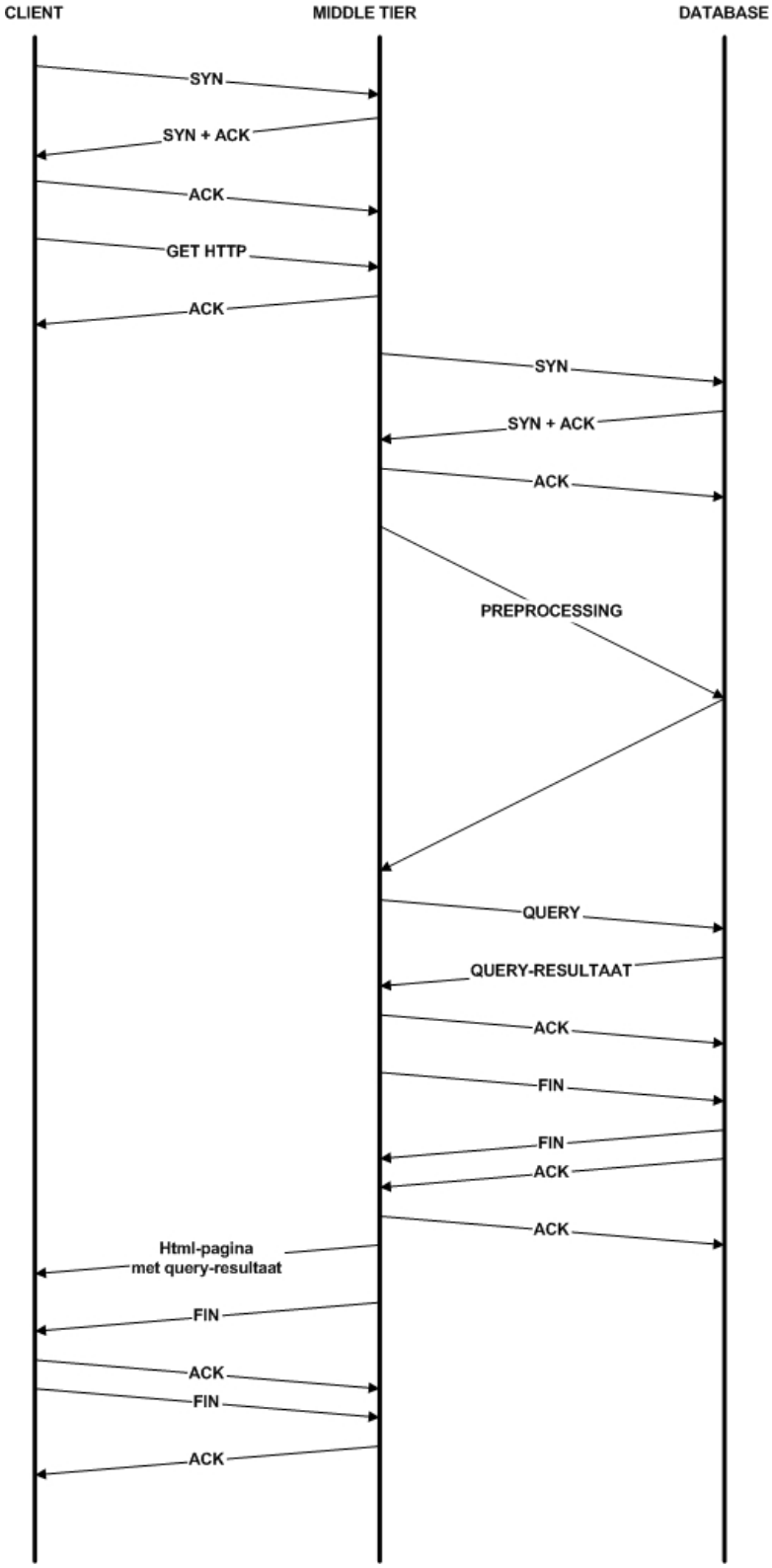
3.3.1 Algemeen

Connection pooling is een methode die gebruikt wordt om de performantie van applicaties met databanktoegang te verbeteren. Alvorens men een commando kan uitvoeren op de databank, moet er eerst een connectie gelegd worden. Soms echter is het creëren en sluiten van deze connectie tijdsrovender dan het uitvoeren van het commando zelf. Om deze reden worden connection pools gecreëerd. Een connection pool is een verzameling van databankconnecties die beschikbaar zijn voor de applicatie om te gebruiken. Nadat een connectie gecreëerd is, wordt ze in de connection pool geplaatst. Nieuwe connecties moeten dan niet meer gemaakt worden en worden gewoon gebruikt vanuit de pool. Wanneer echter alle connecties uit de pool in gebruik zijn, worden er nieuwe connecties gecreëerd en beschikbaar voor de pool gemaakt. Op die manier wordt een verhoogde performantie verkregen.

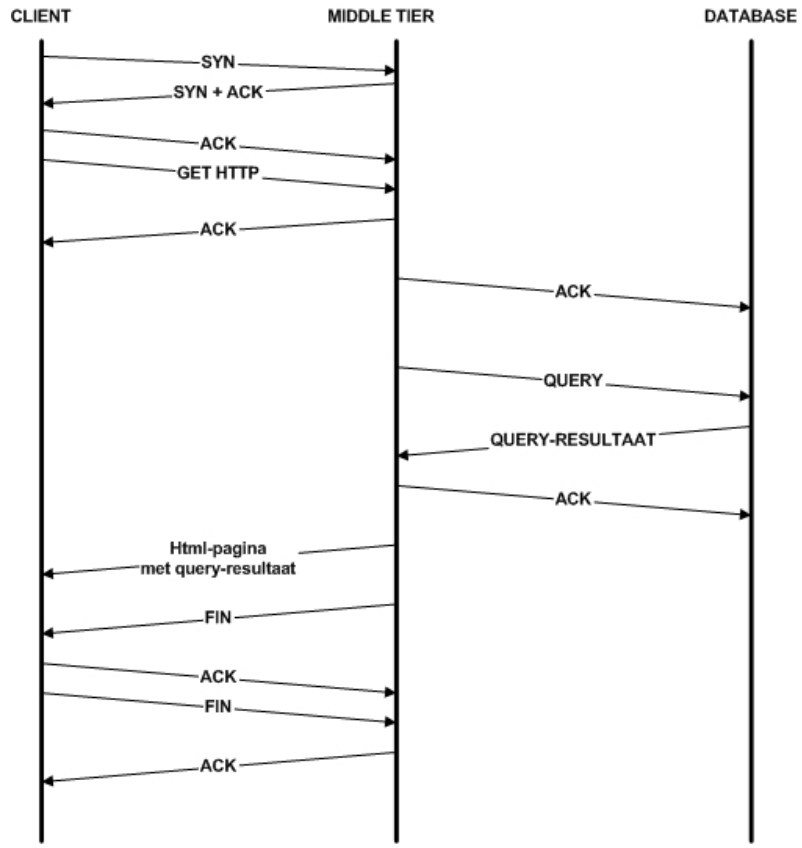
3.3.2 Message sequence

Het effect van connection pooling is heel duidelijk te zien bij de message sequence (figuren 3.1 en 3.2)[1]. Wanneer men beide figuren vergelijkt, vallen er een aantal zaken op.
Ten eerste ziet men op beide figuren duidelijk de three-way handshake tussen de client en de applicatieserver. TCP gebruikt deze three-way handshake om een vertrouwbare connectie te bekomen, alsook om deze te verbreken (zie figuur 3.3). Host 1 stuurt hierbij een segment met het SYN bit aan en een random sequence nummer. Hierop antwoordt host 2 met een segment waarbij ook de SYN bit aan staat, een acknowledgement (ACK) naar host 1 en een random sequence nummer. Host 1 antwoordt hier terug op met een

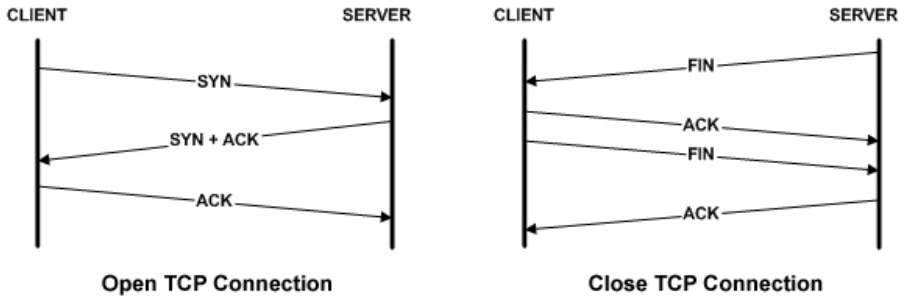
[1]In de figuren zijn een aantal packetflows vereenvoudigd. Zo zal bijv. het queryresultaat uit meerdere TCP packetten bestaan.



Figuur 3.1: Message sequence zonder connection pooling



Figuur 3.2: Message sequence bij connection pooling



Figuur 3.3: Three-way handshake

acknowledgement. Op deze manier komt de connectie tot stand. Bij het sluiten van de connectie gebeurt iets gelijkaardigs.

Ten tweede merkt men op de eerste figuur preprocessing op. De preprocessing houdt onder andere in het kiezen van de juiste databank, inloggen en het opvragen van de variabelen.

Dus naast het aanmaken van de connectie zorgt deze preprocessing nog eens voor extra overhead.
Nog een verschilpunt tussen de beide figuren is het al dan niet optreden van een three-way handshake tussen de applicatieserver en de databank. Wanneer er geen connection pooling optreedt, moet voor elke request een connectie geopend worden via een three-way handshake tussen applicatieserver en databank, alsook gesloten worden. Wanneer er echter connection pooling plaatsvindt, is deze three-way handshake overbodig aangezien men een vrije connectie uit de pool kan nemen in plaats van een nieuwe aan te maken. Wanneer men beide figuren vergelijkt, merkt men dat connection pooling een enorme tijdswinst in de antwoordtijd met zich meebrengt, wat dus leidt tot een verhoogde performantie.

Om de netwerktrafiek van J2EE met die bij .NET te vergelijken, maken we gebruik van Tcpdump, een tool die ons in staat stelt netwerkpacketjes te ontvangen en daaruit gegevens te halen. Voorwaarde hiervoor is echter dat Tcpdump loopt op een host die alle netwerkverkeer (zowel van de applicatieserver als van de databank) kan onderscheppen. Hiervoor gebruiken we het commando:

```
tcpdump -s 1500 -X -ttt host ip-adres-webserver > file-naam
```

De optie -s 1500 geeft packetgrootte 1500 weer. In combinatie met -X - gebruikt om ASCII te verkrijgen - krijgt men dan het volledige packet te zien. De optie -ttt maakt dat men telkens de tijd tussen twee opeenvolgende berichten ziet in plaats van de absolute tijd. Het resultaat wordt dan in file-naam geschreven. Wanneer men dit zowel voor J2EE als .NET doet, dan kan men uit beide data-files een grafiek opstellen die de message sequence weergeeft. Het effect van connection pooling komt echter nog beter tot uiting wanneer men in plaats van één client, men een tcpdump doet van 150 simultane clients. Omdat het opstellen van een grafiek met de message sequence dan allesbehalve duidelijk zou zijn, kan men het aantal connecties meten met het volgend commando:

```
cat file-naam | grep " S " | wc -l
```

Grep doorzoekt hierbij de file naar lijnen die de string "S "bevatten. Deze string komt enkel voor in de tcpdump-outputfile wanneer er een connectie gecreëerd wordt met de

three-way handshake. Het commando wc -l telt dan dit aantal lijnen. Bij elke three-way handshake komen er echter twee SYN's voor. Bijgevolg moet het getal bekomen door dit commando nog gedeeld worden door twee.

```
cat file-naam | grep "pythia86.test.mysql: S " | wc -l
```

Bij dit commando doorzoekt grep de file naar de string "pythia86.test.mysql: S "die enkel bij databankconnecties optreedt. Op die manier verkrijgt men het aantal connecties met MySQL. Om het aantal connecties met MS SQL te bekomen, kan men een gelijkaardige unieke string vinden die in de tcpdump enkel optreedt bij de databankconnectie met MS SQL.

```
watch --interval 1 'netstat -na | grep 3306 | wc -l'
```

Met bovenstaand commando kan men bijv. zien hoe het aantal MySQL databankconnecties varieert tijdens het testen.

Met uitzondering van de gebruikte MySQL native .NET Data Provider[2] (ByteFX.Data), ondersteunen zowel J2EE als .NET connection pooling. Wanneer men echter met bovenvermelde commando's de tcpdump-outputfiles doorzoekt, dan komt men tot een aantal opmerkelijke bevindingen.
Bij J2EE heeft men per client een connectie tussen client en applicatieserver alsook een connectie tussen applicatieserver en databank. Dus ondanks het feit dat J2EE connection pooling ondersteunt, wordt er niet automatisch gebruik van gemaakt.
Bij .NET kwamen er opmerkelijk minder connecties tussen applicatieserver en databank voor, tot soms een factor 6 verschil. Dit is een duidelijke aanwijzing dat er bij .NET wel automatische connection pooling optreedt.
Onderzoek van de tcpdump-outputfile toonde ook aan dat preprocessing bij J2EE ongeveer dubbel zo veel tijd in beslag neemt als bij .NET. Hierdoor is J2EE minder performant dan .NET. Echter het niet automatisch gebruiken van connection pooling heeft een veel negatievere invloed op de performantie van J2EE.

[2]ByteFX.Data is nog in ontwikkeling en zal met de tijd ook connection pooling ondersteunen

Om een zo juist mogelijke vergelijking tussen J2EE en .NET te bekomen, is het dus noodzakelijk dat beiden wel of beiden niet aan connection pooling doen. En aangezien het niet de bedoeling is een systeem minder performant te maken, is het dus nodig connection pooling aan de praat te krijgen bij J2EE. Ondanks het feit dat J2EE connection pooling ondersteunt, is er meer nodig dan een aanpassing in de programmeercode. In de code moet men in plaats van connect() getConnection() gebruiken, dit is echter niet voldoende. Ook aan de serverconfiguratie moet er het één en ander gewijzigd worden [12]. In server.xml moet volgende code komen tussen de </Context> tag van de context van de voorbeelden en de </Host> tag die de localhost definitie sluit:

```
<Context path="/DBTest" docBase="DBTest"
       debug="5" reloadable="true" crossContext="true">
 <Logger className="org.apache.catalina.logger.FileLogger"
           prefix="localhost_DBTest_log." suffix=".txt"
           timestamp="true"/>
 <Resource name="jdbc/TestDB"
             auth="Container"
             type="javax.sql.DataSource"/>
 <ResourceParams name="jdbc/TestDB">
   <parameter>
     <name>factory</name>
     <value>org.apache.commons.dbcp.BasicDataSourceFactory</value>
   </parameter>
   <!-- Maximum number of dB connections in pool. -->
   <parameter>
     <name>maxActive</name>
     <value>100</value>
   </parameter>
   <!-- Maximum number of idle dB connections to retain in pool.
        Set to 0 for no limit.
        -->
```

```
<parameter>
  <name>maxIdle</name>
  <value>30</value>
</parameter>
<!-- Maximum time to wait for a dB connection to become available
     in ms, in this example 10 seconds. An Exception is thrown if
     this timeout is exceeded.  Set to -1 to wait indefinitely.
     -->
<parameter>
  <name>maxWait</name>
  <value>10000</value>
</parameter>
<!-- dB username and password for dB connections  -->
<parameter>
 <name>username</name>
 <value>sofie</value>
</parameter>
<parameter>
 <name>password</name>
 <value>svhoecke</value>
</parameter>
<!-- Class name for JDBC driver -->
<parameter>
   <name>driverClassName</name>
   <value>org.gjt.mm.mysql.Driver</value>
</parameter>
<!-- The JDBC connection url for connecting to your dB. -->
<parameter>
  <name>url</name>
  <value>
```

```
        jdbc:mysql://10.10.10.86:3306/svhoeckeDB?autoReconnect=true
      </value>
    </parameter>
  </ResourceParams>
</Context>
```

In web.xml moeten volgende zaken toegevoegd worden:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
   <!DOCTYPE web-app PUBLIC
   "-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN"
   "http://java.sun.com/dtd/web-app_2_3.dtd">
<web-app>
   <description>MySQL Test App</description>
   <resource-ref>
      <description>DB Connection</description>
      <res-ref-name>jdbc/TestDB</res-ref-name>
      <res-type>javax.sql.DataSource</res-type>
      <res-auth>Container</res-auth>
   </resource-ref>
</web-app>
```

Het effectief connecteren in de servlet gebeurt dan op de volgende manier:

```
Context ctx = new InitialContext();
if(ctx == null )
   throw new Exception("No Context");
DataSource ds =
         (DataSource)ctx.lookup("java:comp/env/jdbc/TestDB");
if (ds != null)
   Connection conn = ds.getConnection();
```

Een tweede manier om connection pooling in J2EE te gebruiken, is het gebruik van de class ConnectionPool[3]. Deze manier van werken vereist niet al de voorgaande aanpassingen in de serverconfiguratie maar werkt met een class die het volledige connection pool gebeuren regelt [13].
In de servlet moet de connection pool gecreëerd en geïnstantieerd worden, alsook de eigenschappen ervan ingesteld worden.

```
ConnectionPool connectionPool = null;
String jdbcDriver = "org.gjt.mm.mysql.Driver";
String dbURL = "jdbc:mysql://10.10.10.86:3306/svhoeckeDB";
String user = "sofie";
String passwd = "svhoecke";
connectionPool = new ConnectionPool(jdbcDriver, dbURL, user, passwd);
connectionPool.setInitialConnections(5);
connectionPool.setIncrementalConnections(5);
connectionPool.setMaxConnections(100);
connectionPool.createPool();
```

Het eigenlijke connecteren gebeurt dan in de doGet functie op de volgende manier:

```
Connection dbConn = null;
dbConn = connectionPool.getConnection();
```

3.4 Caching

Dat connection pooling de performantie verbetert, is nu wel duidelijk. Er bestaan echter nog manieren om een verhoogde performantie te verkrijgen, de belangrijkste hieronder is caching. Caching kan in het drielagenmodel op meerdere niveaus gebeuren.

3.4.1 Back-end caching

Op de EIS treedt caching op in de databank zelf. Wanneer eenzelfde query meerdere malen opgevraagd wordt, zal de databank via caching maken dat de antwoordtijden voor het

[3]De broncode van class ConnectionPool is te vinden in bijlage A

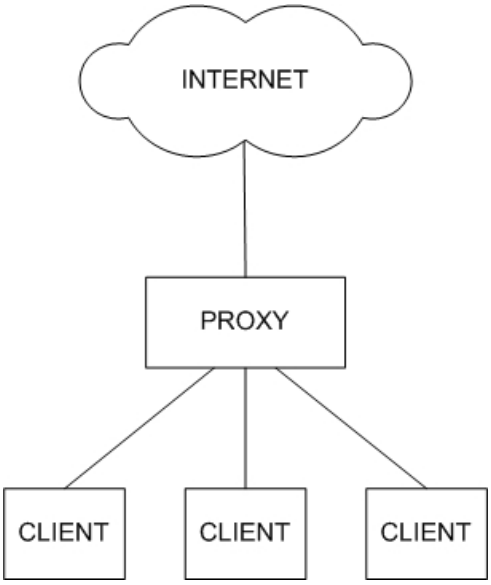
leveren van dit queryresultaat afnemen. Dit heeft natuurlijk invloed op de testresultaten aangezien daar telkens dezelfde query opgevraagd wordt. Als poging dit effect tegen te gaan, is er een vergelijkende test gedaan met een variabele query. Door middel van een case-select werd er één van de vijf query's gekozen. Dit bracht echter niet echt een oplossing aangezien deze query's nog steeds te cachen waren. Ook in de praktijk zal het ook zo zijn dat gelijkaardige query's opgevraagd zullen worden. Dus ook daar zal de back-end caching zijn nut bewijzen.

3.4.2 Front-end caching

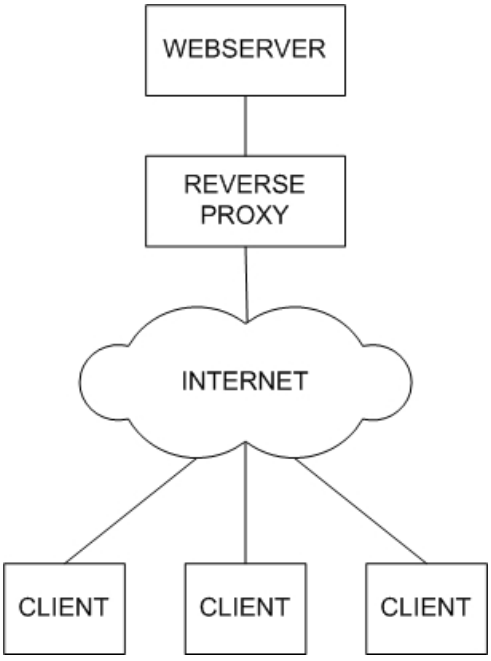
Een andere manier om de performantie te verbeteren van webapplicaties bestaat erin gebruik te maken van de caching mogelijkheid binnen het HTTP-protocol. De client die een pagina opvraagt wordt voorzien van extra informatie, de zogenaamde expiration date, die vertelt hoelang deze pagina accurate informatie bevat. Zo weet de client, dat zolang de expiration date niet overschreden is, hij de webserver niet meer hoeft te contacteren om dezelfde informatie op te vragen. Hierdoor worden zowel de webserver als eventueel achterliggende applicatielaag of databases niet meer belast voor de gevraagde informatie.

Deze techniek geeft pas performantiewinst indien de gecachte informatie door dezelfde client meerdere malen opgevraagd wordt binnen het cache-interval. Door het plaatsen van caching servers in het netwerk kunnen de verschillende client requests gecached worden en treedt de cache op als één client voor de webapplicatie. De cache server kan alle request naar het intra-, extra- en internet cachen, de zogenaamde proxy mode (zie figuur 3.4). Of als doel hebben een welbepaalde webapplicatie te cachen door de cache als reverse proxy voor de webapplicatie te plaatsen (zie figuur 3.5). Hierbij worden dynamisch opgebouwde pagina's gecached voor de webserver zelf, waardoor de antwoordtijden van de applicatie een stuk korter kunnen worden aangezien ze onmiddelijk als gecachte objecten teruggestuurd worden.

De laatste evolutie van de technologie bestaat erin een soort van reverse proxy in de webserver zelf te integreren, zoals de nieuwste IIS 6.0 webserver in windows 2003. Dit



Figuur 3.4: Proxy mode



Figuur 3.5: Reverse proxy mode

heeft als voordeel dat men geen extra cache server hoeft te voorzien.

In de HTTP-headers is er een mogelijkheid om een expiration date te voorzien. Deze headers kunnen algemeen worden meegegeven door de webserver te configureren, of kun-

nen mee verwerkt in de JSP of ASP code. Een voorbeeld van dergelijke header vindt men hieronder.

```
HTTP/1.0 200 OK
Date: Mon, 26 May 2003 21:05:35 GMT
Content-Length: 182
Content-Type: text/html
Expires: Mon, 26 May 2003 21:20:35 GMT
Cache-Control: public
Server: Microsoft-IIS/5.0
```

3.4.3 Conclusie

De diversiteit en intensiteit aan informatie van de webapplicatie heeft een belangrijke invloed op de performantie bij het cachen. Zo is de kans dat dezelfde informatie korte tijd na elkaar opgevraagd wordt, kleiner wanneer de hoeveelheid aan data zeer groot is. Toch kan bepaalde informatie zeer frequent opgevraagd worden (bijv. de introductiepagina van een bepaalde webapplicatie) waardoor caching dan weer wel interessant wordt. Dit geldt zowel voor front- als back-end caching. Front-end caching heeft tevens het bijkomende voordeel dat ook de webserver zowel als eventuele applicatielaag ontlast worden.

Hoofdstuk 4

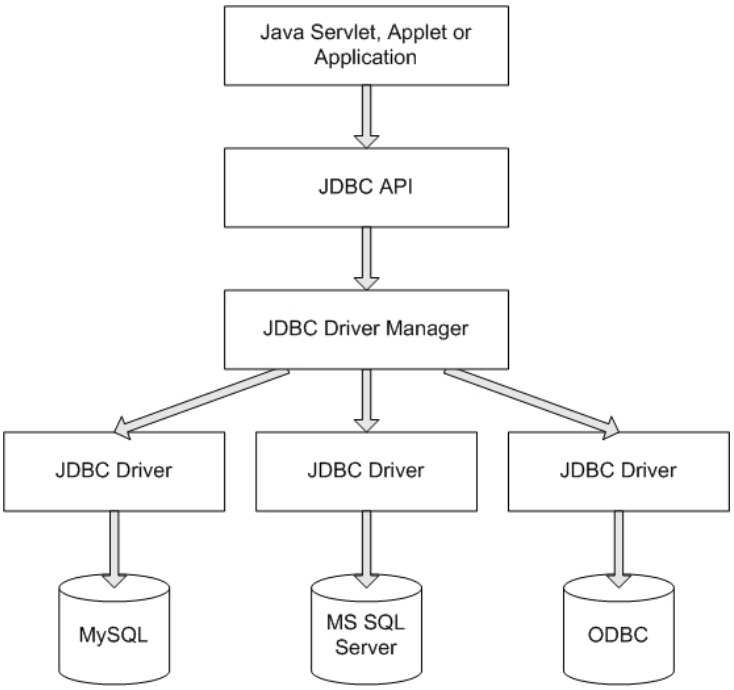
J2EE-applicaties

Bij J2EE zijn er verschillende manieren om gegevens uit onze databank te halen. Zo heeft men aan de ene kant de servlets en aan de andere kant de enterprise javabeans. De servlets gebruiken JDBC. Ook de enterprise javabeans kunnen deze manier van werken gebruiken, maar zij hebben nog het alternatief om met entity beans te werken.

Entity beans vereenvoudigen databankinteractie vanuit het oogpunt van applicatie-ontwikkeling bekeken. Bij de servlets en session beans zal men echter connecteren via JDBC wat iets complexer te implementeren is dan entity beans, maar het brengt wel een betere performantie met zich mee. Omdat het werken met entity beans vooral bedoeld is om eenvoudige applicatie-ontwikkeling te verkrijgen maar men daardoor wel inboet aan performantie, gebruikt men vaak session beans om zo toegang te verkrijgen tot de entity beans. Aangezien de session en de entity bean beiden op dezelfde EJB container lopen, kan men de entity bean benaderen via de local interface in plaats van via de remote interface. Dit vermindert het aantal remote calls en brengt een betere performantie met zich mee. Het werken met session beans via JDBC daarentegen brengt echter nog een betere performantie met zich mee. Omdat het vooral over de performantie te doen is en niet over de eenvoud van applicatie-ontwikkeling, wegen de voordelen van de entity bean dus niet op tegen de performantie-inbinding. Bijgevolg zullen er geen entity beans gebruikt worden en wordt hier enkel JDBC in detail besproken.

4.1 JDBC

JDBC is een API specificatie ontwikkeld door Sun Microsystems die een uniforme interface definieert voor databanktoegang naar verschillende relationele databanken. JDBC is een deel van het Java platform en bijgevolg ingesloten in de standaard JDK distributie. De JDBC API gebruikt een driver manager en databankspecifieke drivers om transparante connectiviteit met verschillende databanken te creëren. De JDBC driver manager zorgt ervoor dat de juiste driver gebruikt wordt bij datatoegang. Deze driver manager kan meerdere drivers tegelijk ondersteunen zodat met meerdere databanken kan geconnecteerd worden.



Figuur 4.1: Lagen van de JDBC architectuur

Een JDBC driver vertaalt standaard JDBC calls naar een netwerk- of databankprotocol of naar een API call van de databankbibliotheek waardoor communicatie met de databank vlotter verloopt. Deze vertalingslaag maakt dat JDBC applicaties geschreven kunnen worden zonder databank afhankelijkheid. Wanneer immers de back-end databank verandert, moet enkel de JDBC driver vervangen worden met enkele minieme aanpassingen in

de code.
Nu rest er de keuze om enerzijds via een rechtstreekse JDBC driver te werken of een tweede manier bestaat er in de meer algemene JDBC-ODBC bridge te gebruiken.

Deze JDBC-ODBC bridge is een JDBC driver die JDBC operaties zal implementeren door deze te vertalen naar ODBC operaties. Voor ODBC lijkt dit als een normale applicatie. Deze bridge implementeert JDBC voor elke databank waarvoor een ODBC driver beschikbaar is. Aangezien deze laatste manier voor grote overhead zorgt, is het aan te raden deze JDBC-ODBC driver enkel te gebruiken voor prototyping of wanneer er geen directe JDBC driver bestaat. Aangezien dit zowel voor MySQL als MS SQL Server niet het geval is, wordt J2EE enkel met een directe JDBC driver getest.

Databank	Gebruikte JDBC driver
MySQL	MySQL Connector/J
MS SQL Server	jTDS
MS SQL Server	Microsoft JDBC

4.2 Servlets

4.2.1 Algemeen

De Java servlet technologie biedt aan web ontwikkelaars eenvoudige methodes aan om de services van de HTTP server uit te breiden en te verbeteren. Servlets accepteren HTTP requests als input en antwoorden met een HTTP response als output.

Een servlet opereert binnen een servlet container (de servlet engine). Deze container zorgt voor de echte details om een netwerk te connecteren, het juist formuleren van antwoorden en het cachen van vragen. De servlet container is ook verantwoordelijk voor het verwerken van client-vragen, deze vragen doorsturen in de vorm van objecten naar de servlet en het antwoord gegenereerd door de servlet terug sturen naar de client.

Er zijn verschillende verkopers die servlet containers aanbieden, zoals onder andere WebLogic, Tomcat en jBoss. Alhoewel de actuele implementatie van de servlet container zal variëren van verkoper tot verkoper, hangt iedere container vast aan protocols gedefinieerd door de Servlet API, die de interface tussen de container en de servlets beschrijft.
Naast requests verwerken en protocols vertalen, is de container ook verantwoordelijk voor het beheren van de levenscyclus van de individuele instanties van de servlets. Wanneer naar een servlet voor de eerste keer verwezen wordt, laadt de container de servlet in zijn geheugen, ofwel gebeurt dit direct bij server startup ofwel indirect door een client request. Van zodra de servlet ingeladen is, roept de container de init() methode op die de acties nodig bij initialisatie uitvoert. Voor iedere request naar de servlet, is de container verantwoordelijk om de juiste service methode aan te roepen, zoals bijvoorbeeld doGet() bij een GET request en doPost() bij een POST request. Uiteindelijk wanneer de servlet verwijderd wordt van de container, roept de engine de destroy() methode aan die gebruikt kan worden om alle door de servlet gebruikte resources af te sluiten. Men kan de levenscyclus van een servlet dus als volgt samenvatten:

- de servlet container creëert een instantie van de servlet
- de container roept de init() methode van de servlet aan
- voor elk request naar de servlet, roept de container de juiste methode aan
- alvorens de servlet instantie verwijderd wordt, roept de container de destroy() methode aan

Als servlet container wordt de Apache Jakarta Tomcat 4. 0 gebruikt.

4.2.2 Source servlet

Omdat het volledig in detail bespreken van de broncode te ver zou leiden, worden hier enkel de relevante zaken met betrekking tot databanktoegang besproken.
Alvorens een JDBC driver gebruikt kan worden om een databankconnectie te creëren, moet de driver eerst geregistreerd worden bij de driver manager. De driver manager houdt

een referentie bij naar alle driver objecten die beschikbaar zijn voor de JDBC clients. Een JDBC driver registreert zichzelf automatisch bij de driver manager wanneer hij ingeladen wordt. Om een driver in te laden gebruikt men de Class.forname().newInstance() methode zoals in de code hieronder te zien is:

```
driver = (Driver) Class.forName("com.mysql.jdbc.Driver").newInstance();
```

Eenmaal de driver ingeladen is, kan men een connectie met de databank tot stand brengen. Een JDBC connectie wordt geïdentificeerd door een databank URL die de driver manager meedeelt welke driver en welke databron te gebruiken. De syntax van de databank URL is als volgt:

```
jdbc:SUBPROTOCOL:SUBNAME
```

Het eerste deel in deze URL toont aan dat JDBC gebruikt wordt om de connectie te leggen. Het SUBPROTOCOL is de naam van een geldige JDBC driver. De SUBNAME is meestal een logische naam, of een alias, naar de fysische databank. Alhoewel de meeste databank URL's nauwgezet de standaard syntax volgen, zijn JDBC databank URL conventies redelijk soepel en laten ze elke driver toe de gewenste informatie in de URL te includeren.
Bij de servlet die een query opvraagt uit de MySQL databank maakt men op volgende manier de connectie:

```
Connection con = null;
String sHost  = "10.10.10.86:3306";
String sConnect = "jdbc:mysql://"+sHost+"/svhoeckeDB?user=sofie"
sConnect += "&password=svhoecke&autoReconnect=true";
con = driver.connect(sConnect, new Properties());
```

Connecteert men echter met de MS SQL Server, dan gebruikt men een andere JDBC driver en krijgt men bijgevolg ook een licht gewijzigde registratie en connectiestring:

```
driver = (Driver)
    Class.forName("net.sourceforge.jtds.jdbc.Driver").newInstance();
```

```
Connection con = null;
String sHost = "192.168.0.4";
String sConnect = "jdbc:jtds:sqlserver://"+sHost+"/svhoeckeDB;user=sa;"
sConnect += "password=svhoecke";
con = driver.connect(sConnect, new Properties());
```

Wanneer men echter gebruik wil maken van connection pooling[1], moet men getConnection() in plaats van connect() gebruiken. Connection pooling vereist echter ook nog configuratiewijzigingen bij de applicatieserver. Het is dus niet zo dat het gebruik van getConnection() automatisch connection pooling implementeert.

```
Context ctx = new InitialContext();
DataSource ds = (DataSource)ctx.lookup("java:comp/env/jdbc/TestDB");
conn = ds.getConnection();
```

Eenmaal de connectie gemaakt is, kan men een SQL query uitvoeren op de databank. Om een query uit te voeren, moet een Statement object gecreëerd worden met behulp van de createStatement() methode van het Connection object.

```
Statement s = con.createStatement();
```

Met de executeQuery() methode van het Statement object kan men via de SQL SELECT informatie uit de databank halen:

```
ResultSet res = s.executeQuery( "SELECT * FROM tabelnaam");
```

Metadata wordt gedefinieerd als data over de data. JDBC voorziet specifieke data over de databank of de ResultSet via metadata objecten. Zo'n metadata object kan opgevraagd worden via de getMetaData() methode. In de servlet wordt op die manier het aantal kolommen in de ResultSet opgevraagd met de getColumnCount() methode van het getMetaData() object. Op die manier kan op een dynamische manier de HTML output gegenereerd worden. Deze methode is erg flexibel. Immers, wanneer er meer informatie gewenst is, moet enkel het statement die de ResultSet creëert, geupdated worden om een

[1] Zie 3.2 voor meer informatie omtrent connection pooling.

extra kolom bij te krijgen. De HTML output zal automatisch mee geupdated worden als gevolg van de nieuwe ResultSet.

```
ResultSetMetaData dbMeta = res.getMetaData();
int nColumns = dbMeta.getColumnCount();
```

Om de resultaten van de query te bekijken, kan men gebruik maken van de next() methode van het ResultSet object. Met behulp van getString() kan de waarde van een bepaald veld opgevraagd worden.

```
ResultSetMetaData dbMeta = res.getMetaData();
int nColumns = dbMeta.getColumnCount();
while (res.next()) {
    String sRow = "";
    for (int i = 0; i<nColumns; i++) {
         sRow += res.getString(i+1)+"   ";
    }
out.println("Row "+ (nRows+1) + " : " + sRow + "<br>");
nRows++;
```

En natuurlijk eindigt men met het sluiten van de databankconnectie. Alvorens dat echter te doen, moeten ook de ResultSet en Statement gesloten worden.

```
res.close();
s.close();
con.close();
```

4.3 Enterprise JavaBean

4.3.1 Algemeen

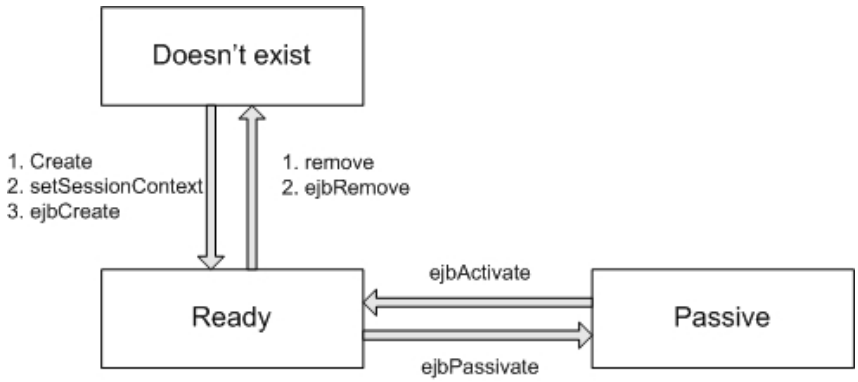
Enterprise beans zijn de J2EE componenten die de Enterprise JavaBean (EJB) technologie implementeren. Enterprise javabeans lopen in de EJB container, een EJB runtime

omgeving binnen de J2EE server. Een EJB is dus eigenlijk een server-side component die de bedrijfslogica van een applicatie bevat.
Zoals reeds gezegd zijn er verschillende soorten EJBs:

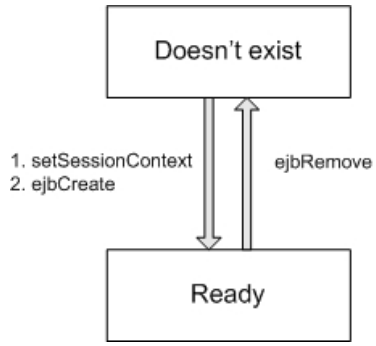
- Session beans: voeren een taak voor de client uit
  - Stateless Session Beans
  - Stateful Session Beans
- Entity Beans: stellen persistente data voor
  - Container-Managed Entity Beans
  - Bean-Managed Entity Beans
- Message-Driven Beans: verwerken berichten asynchroon

Aangezien er bij de testen met session beans gewerkt wordt, wordt er op deze beans wat dieper ingegaan.
Een session bean stelt één enkele client binnen de J2EE server voor. Om toegang te krijgen tot een applicatie ontwikkeld op de server, roept de client de methodes van de session bean op. Deze session bean voert dan werk uit voor de client zodat de client niet geconfronteerd wordt met de complexiteit van de taken op de server. Wanneer de client stopt met uitvoeren, verdwijnt de session bean en zijn data.
Bij een stateful session bean wordt er eigenlijk een unieke status van de client bijgehouden. Deze status blijft behouden gedurende de volledige client-bean sessie. Wanneer de client de bean verwijdert of stopt, eindigt de sessie en verdwijnt ook de status. Een stateless session bean daarentegen houdt geen status bij voor een bepaalde client. Bijgevolg kunnen stateless session beans meerdere clients tegelijk ondersteunen. Hierdoor creëren ze ook een betere schaalbaarheid voor applicaties die een groot aantal clients vereisen. Stateless beans bieden dan ook een betere performantie dan stateful beans.

Een enterprise javabean bestaat uit meerdere files. Als eerste heeft men de deployment descriptor. Dit is een XML file die informatie bevat over de bean op zich zoals bijvoorbeeld de transactie attributen. Er zijn ook de interfaces, een remote en een home interface



Figuur 4.2: Levenscyclus stateful session bean



Figuur 4.3: Levenscyclus stateless session bean

vereist voor remote toegang. Daarnaast is er nog de enterprise bean class die de methoden uit de interfaces implementeert.

Als applicatieserver wordt jBoss 3.0 gebruikt aangezien bij de reference-implementatie van Sun het aantal simultane clients beperkt is. Ook de antwoordtijden liggen bij Sun een factor 6 tot 10 lager dan bij jBoss.

4.3.2 Source EJB

De Home interface van de EJB bevat enkel een create() methode die een interface retourneert van het type Query. Deze Home interface kan geraadpleegd worden door de client zodat deze toegang kan krijgen tot de onderliggende bean interfaces en methoden.

```
public interface QueryHome extends EJBHome {
   Query create() throws RemoteException, CreateException;
}
```

De Query interface op zich geeft de client de mogelijkheid een JDBC driver in te laden, alsook een query uit te voeren op de databank. Bij de tweede functie geeft de interface een string met daarin de resultaten terug.

```
public interface Query extends EJBObject{
   public String accessDB(String queryString) throws RemoteException;
   public boolean loadJDBCDriver() throws RemoteException;
}
```

De client is nu in staat de EJB methoden bevat in deze interfaces op te roepen. De EJB interface zal dan deze opgeroepen methodes en de betreffende argumenten doorgeven aan de onderliggende EJB implementatie en het resultaat teruggeven.

De Java client zal eerst de lokatie van de Home interface opzoeken. Vervolgens vraagt de client aan deze interface een instantie van de EJB interface te creëren.

```
InitialContext ic=new InitialContext();
Object objRef=ic.lookup("java:comp/env/ejb/TheQuery");
QueryHome home=
     (QueryHome)PortableRemoteObject.narrow(objRef,QueryHome.class);
varQuery=home.create();
```

Eenmaal dit gebeurd is, roept de client de loadJDBCDriver() methode op van de Query interface.

```
varQuery.loadJDBCDriver();
```

Wanneer men echter deze loadJDBCDriver() oproept binnen de ejbCreate() methode, wordt de JDBC driver automatisch ingeladen de eerste keer dat een client toegang tot de EJB zoekt. Deze manier van werken zorgt er voor dat de driver niet telkens opnieuw moet ingeladen worden, maar ingeladen blijft gedurende de levensduur van de EJB.

```
public void ejbCreate()  {
   loadJDBCDriver();
}
```

Om de eigenlijke query uit te voeren, roept de client de methode accessDB(String queryString) op om erna de resultaten te verwerken.

```
String queryString= "SELECT * FROM tabelnaam";
String results=varQuery.accessDB(queryString);
```

De actuele bedrijfsimplementatie van de public accessDB(String queryString) methode is niet echt verschillend van de servlet versie. De bean op zich is een stateless session bean waardoor geen status bijgehouden wordt en er meerdere simultane clients toegelaten worden.

```
public QueryBean() {}
public void ejbCreate() {}
public void setSessionContext(SessionContext sc) {}
```

Wanneer men de driver automatisch wil inladen wanneer de EJB voor de eerste keer benaderd wordt, krijgt men zoals hierboven vermeld een aangepaste versie van ejbCreate(). Bij een stateful session bean zijn er nog twee extra functies die voor passivatie en activatie van de session bean zorgen:

```
public void ejbActivate() {}
public void ejbPassivate() {}
```

Wanneer de client stopt met de uitvoering, verdwijnt de session bean doordat de ejbRemove() methode opgeroepen wordt.

```
public void ejbRemove() {}
```

Hoofdstuk 5

.NET-applicaties

5.1 ADO

Voordat .NET op het toneel verscheen, gebruikte men ActiveX Data Objecten (ADO). ADO is een high level object programming interface die de functionaliteit van datatoegang op zicht neemt. Deze technologie maakt het mogelijk om op een eenduidige manier verschillende soorten data-objecten aan te spreken. Bij ADO wordt de data opgeslagen in een RecordSet object. Dit RecordSet object fungeert als het ware als een tijdelijke opslagplaats voor de data binnen het systeemgeheugen. Operaties zoals Insert, Update en Delete worden dan op de RecordSet uitgevoerd, die dan als gevolg hiervan de originele databron bijwerkt.

Ondanks het feit dat de RecordSet breedvoerig gebruikt wordt voor datatoegang en het ook erg populair is, zijn er toch enkele nadelen en beperkingen aan. Het grootste nadeel van de RecordSet is dat het een continue connectie met de databron vereist. Continue connecties naar een databank zijn echter zeer kostelijk om meerdere redenen. Ten eerste zijn databankconnecties gelimiteerd. Continue connectie van één client betekent dan dus automatisch een connectie minder voor een andere client. Ten tweede vereisen continue connecties ook een groot deel van de server zijn geheugenbronnen.

Eén van de meest gebruikte connectiemethoden onder ADO is Open Database Connectivity (ODBC).

5.2 ADO.NET

Met de invoering van het .NET Framework zijn er vele zaken veranderd, zo ook heeft Microsoft een nieuw model ingevoerd om datatoegang te verkijgen. In dit .NET Framework wordt datatoegang verkregen door een verzameling van classes, ADO.NET genaamd, die eigenlijk een uitbreiding zijn van de bestaande ADO. Er zijn echter een aantal veranderingen gebeurd ten opzichte van ADO, zowel intern als extern. De meest duidelijke interne verandering is het feit dat ADO.NET volledig gebaseerd is op XML. Als interne verandering merken we direct op dat er geen RecordSet object meer is [15].

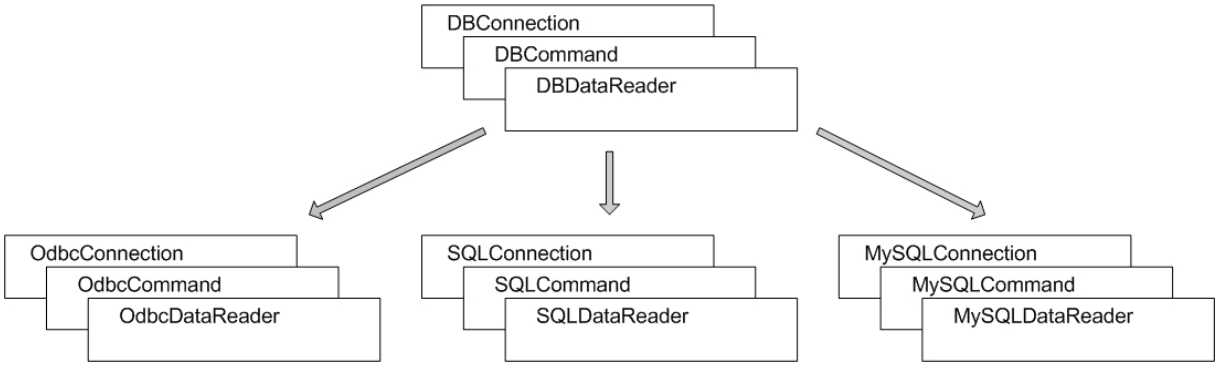
Bij ADO.NET heeft men de functionaliteit van de RecordSet opgesplitst in drie groepen:

- Het DataReader object biedt op zo'n efficiënt mogelijke manier een eenvoudige toegang tot een verzameling van records.
- Met behulp van de DataSet en DataSetCommand objecten kan men aan clientzijde een cache creëren van één of meerdere gerelateerde RecordSets om deze dan offline te bewerken. Op die manier krijgt men het concept van connectieloze datatoegang.
- Als derde zijn er nog de functionaliteiten van ADO die ADO.NET niet aanbiedt. Dit houdt onder andere in data-updates doen gebruik makend van locking door een connectie-geörienteerde aanpak.

Aangezien ADO.NET zich vooral richt tot n-laagse webvriendelijke applicaties, biedt het slechts de eerste twee van deze drie functionaliteiten van de RecordSet aan.

In ADO.NET heeft men drie classes die vertrouwd zullen aanvoelen door ADO-gebruikers, namelijk DBConnection, DBCommand en DBDatareader. Echter, men zal deze classes eigenlijk niet echt gebruiken in een .NET applicatie. In de plaats daarvan zal men gebruik maken van een verzameling classes die erven van deze drie classes. Enkele voorbeelden hiervan zijn te zien op figuur 5.1.

Nu is de vraag of men in ASP.NET gebruik moet maken van het DataReader of het



Figuur 5.1: Afgeleide klasses in ADO.NET

DataSet object.

Wanneer de data altijd up to date moet zijn en bijgevolg elke keer zo nodig is uit de databank moet gehaald worden, gebruikt men hiervoor beter de DataReader aangezien het creëren, van een DataReader object minder overhead met zich meebrengt dan het creëren van een DataSet object.

De DataSet daarentegen heeft meer mogelijkheden dan het DataReader object. Zo kan men slechts eenmaal doorheen de DataReader lopen. Wanneer men dus meerdere iteraties doorheen de data van eenzelfde request nodig heeft, gebruikt men hiervoor beter de DataSet want bij de DataReader zou de data tweemaal van de databank opgevraagd moeten worden.

Ook kan, in tegenstelling tot de DataReader, de DataSet gebruikt worden wanneer de databron geen databank is, maar bijvoorbeeld een XML-file.

Ook wanneer er bijvoorbeeld tijdrovende bewerkingen op de resultaten van de query moeten uitgevoerd worden, biedt de DataSet een beter alternatief dan de DataReader. Immers bij de DataReader zal de connectie pas vrijgegeven worden na het expliciet sluiten van de connectie en dus na het uitvoeren van de bewerkingen op de data. Bij de DataSet daarentegen wordt de data opgeslagen in een cache. Hierdoor is de DataSet in staat de connectie onmiddelijk te sluiten en terug vrij te geven [16].

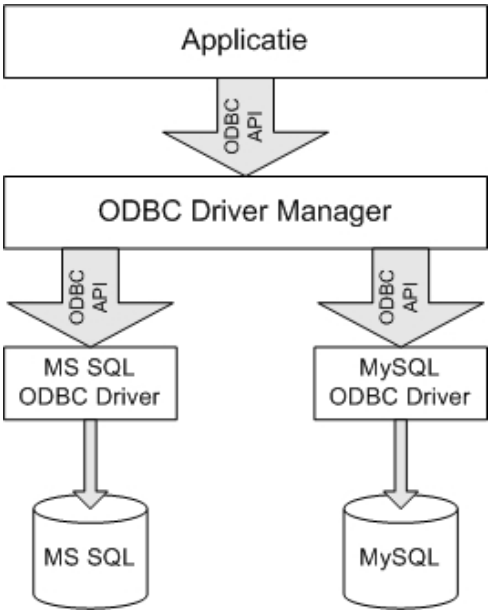
5.3 .NET Data Providers

5.3.1 Algemeen

Een .NET Data Provider biedt functionaliteit aan om te connecteren met een databron, commando's uit te voeren en resultaten te verwerven. Deze resultaten kunnen direct verwerkt worden met de ADO.NET DataReader of geplaatst worden in de DataSet voor verdere verwerking terwijl men niet langer geconnecteerd blijft.
Alle .NET Data Providers zijn ontworpen om zo licht mogelijk te zijn. Ze bevatten een minimale laag tussen de databron en de code. Dit bevordert functionaliteit zonder performantie op te offeren.
Er zijn drie types .NET Data providers. Enerzijds hebben we de native .NET Data Provider zoals bijvoorbeeld de SQL Server .NET Data Provider en de MySQL .NET Data Provider. Daarnaast hebben we ook de OLEDB .NET Data Provider en de ODBC .NET Data Provider.

5.3.2 ODBC .NET Data Provider

ODBC is geen nieuw concept. Het is een specificatie voor een databank API. Deze API is onafhankelijk van de DBMS of het besturingssysteem. Ondanks het feit dat deze API C gebruikt, is hij toch taalonafhankelijk.
De functies in de ODBC API zijn geïmplementeerd door ontwikkelaars van DBMS-specifieke drivers. Applicaties roepen dan de functies in deze drivers op om datatoegang te verkrijgen op een DBMS-onafhankelijke manier. Een ODBC Driver Manager regelt de communicatie tussen de applicaties en de drivers. In .NET is dat de ODBC .NET Data Provider die bevat zit in de 'Microsoft.Data.Odbc' namespace. De ODBC .NET Data Provider is een add-on component bij de Microsoft .NET Framework Software Development Kit (SDK).
Men ziet deze ODBC architectuur op figuur 5.2. De ODBC .NET Data Provider ondersteunt meerdere ODBC drivers en databronnen waardoor een applicatie data simultaan kan opvragen uit meerdere databronnen. Men ziet op de figuur ook dat de ODBC API op twee plaatsen gebruikt wordt, namelijk tussen de applicatie en de ODBC Driver Manager



Figuur 5.2: ODBC architectuur

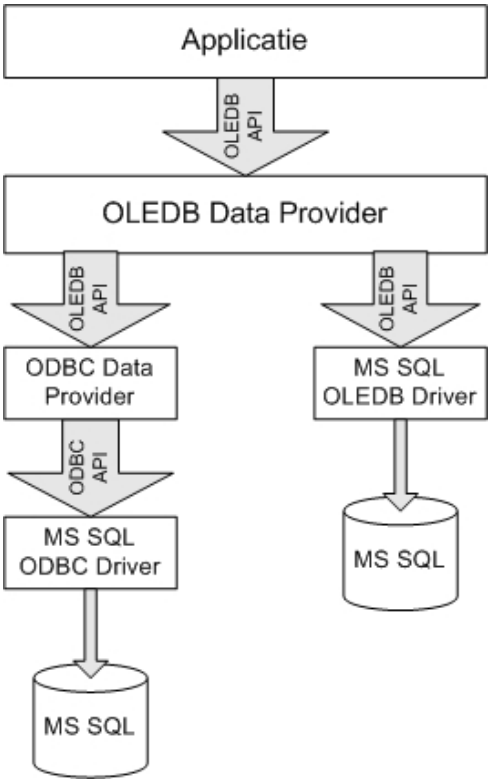
en ook tussen de ODBC Driver Manager en elke ODBC Driver. De interface tussen de ODBC Driver Manager en de ODBC Drivers wordt soms ook de Service Provider Interface (SPI) genoemd. Bij ODBC zijn echter de API en de SPI gelijk aan elkaar.

ODBC heeft bijgevolg als groot voordeel dat er maximum interoperabiliteit gegarandeerd wordt. Een enkele applicatie kan toegang krijgen tot verschillende databanken met één en dezelfde code. Dit maakt het gebruik van ODBC bijgevolg erg eenvoudig.
Performantie is echter een negatieve factor bij ODBC aangezien het tamelijk veel lagen bevat. Bij ODBC maken we ook gebruik van een System DSN waardoor we telkens een opzoeking in de registers moeten doen wat tijd kost, zeker bij groot aantal simultane connecties.

5.3.3 OLEDB .NET Data Provider

De OLEDB .NET Data Provider biedt toegang tot native OLEDB Providers. OLEDB is een COM-gebaseerd object voor datatoegang tot alle types van data. Het biedt eveneens toegang tot gedisconnecteerde databronnen (bijv. databank-snapshot op een laptop van tijdens de laatste connectie).

OLEDB bevindt zich tussen de ODBC laag en de applicatie. Bijvoorbeeld bij ASP.NET is ADO.NET de applicatie die zich boven OLEDB bevindt. Bijgevolg worden de ADO.NET calls eerst naar OLEDB gezonden, die ze dan doorstuurt naar de ODBC laag.
Bij native OLEDB providers kan je echter rechtstreeks via OLEDB werken, in plaats van via ODBC om. Dit brengt natuurlijk een performantiestijging met zich mee.
De OLEDB architectuur is te zien in figuur 6.3. MySQL ondersteunt officieel het gebruik van OLEDB niet.

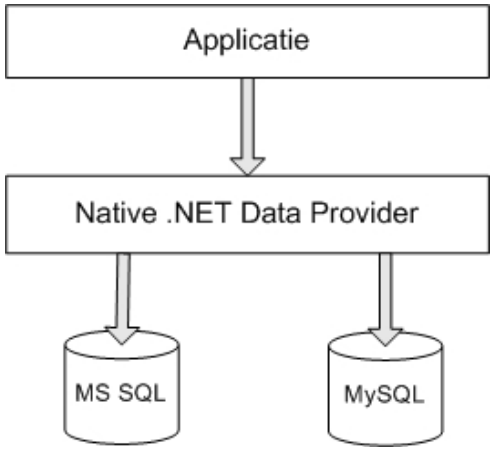


Figuur 5.3: OLEDB architectuur

5.3.4 Native .NET Provider

De native .NET Provider is een directe implementatie die geen OLEDB of ODBC bridge vereist. Het is de enige laag tussen de applicatie en de databank en neemt de databankinteractie op zich. Deze .NET Data Providers zijn ontwikkeld om op een snelle en efficiënte manier datatoegang te verkrijgen.
SQL Server .NET Data Provider is de meest efficiënte manier om met MS SQL Server te

connecteren. Het maakt gebruik van een eigen protocol voor databankconnectie wat resulteert in een geoptimaliseerd gebruik van de SQL Server en snellere datatransacties. De 'System.Data.Sqlclient' namespace bevat classes voor de SQL Server .NET Data Provider. Voor de MySQL databank hebben we de MySQL .NET Data Provider. Momenteel zijn er twee MySQL .NET Data Providers beschikbaar. Enerzijds is dit dbProvider [17] van eInfoDesigns en anderzijds hebben we MySQLNet [18], de freeware MySQL .NET Data Provider van SourceForge. Aangezien deze laatste nog in ontwikkeling is, ondersteunt deze, in tegenstelling tot dbProvider, nog geen connection pooling.
Het principe van de native .NET Data Providers is geïllustreerd in figuur 5.4.



Figuur 5.4: Architectuur native .NET Data Provider

5.4 Source

5.4.1 ASP - ADO

Bij de ASP-pagina's wordt ADO gebruikt om een databankconnectie te creëren via ODBC.

```
dim adoConn
dim adoRS
set adoConn = Server.CreateObject("ADODB.Connection")
set adoRS = Server.CreateObject("ADODB.Recordset")
adoConn.Open "DSN=mssql_dsn; UID=sa; PWD=svhoecke;"
```

```
set adoRS = adoConn.Execute("SELECT * FROM tabelnaam")
```

In bovenstaande code wordt de DSN voor de MS SQL Server gebruikt. Om een databankconnectie met MySQL te creëren, volstaat het de DSN aan te passen.

```
adoConn.Open "DSN=mysql_dsn"
```

De DataReader biedt nu een eenvoudige en efficiënte toegang tot een verzameling van records. Bij ADO wordt de RecordSet op volgende manier doorlopen:

```
while not adoRS.EOF
response.write adoRS.fields(0).value & adoRS.fields(1).value
adoRS.MoveNext
wend
```

Na het weergeven van de data-output, moeten de RecordSet en de databankconnectie gesloten worden.

```
adoRS.Close
adoConnn.Close
```

Hieruit wordt nog eens bevestigd dat ADO connectie-geöriënteerd werkt. De databankconnectie wordt gedurende de volledige werktijd open gehouden.

5.4.2 ASP.NET - ADO.NET

Binnen ASP.NET kan men gebruik maken van ASP.NET server controls om webpagina's te programmeren. Server controls worden binnen een .aspx file gedeclareerd door middel van custom tags of intrinsieke HTML tags die een runat="server" attribuut bevatten. Intrinsieke HTML tags worden behandeld door één van de controls in de System.Web.UI.HtmlControls namespace. Om het even welke tag die niet expliciet past met deze controls, krijgt het type System.Web.UI.HtmlControls.HtmlGenericControl.
Bij de ASP.NET pagina's creëert men de databankconnecties binnen de Page_Load() methode. Wanneer een client de ASP-pagina oproept, is de code die zich binnen Page_Load bevindt, het eerste wat uigevoerd wordt.

```
<%@ Page Language="C#" Debug="true" %>
<%@ Import Namespace="Microsoft.Data.Odbc" %>
<script language=C# runat="server">
  void Page_Load(Object Src,EventArgs E) {
  }
```

Import Namespace=”Microsoft.Data.Odbc” is vereist wanneer men de databankconnectie via ODBC wil creëren. Wil men echter OLEDB gebruiken, dan is een andere namespace vereist.

```
<%@Import Namespace="System.Data.OleDb"%>
```

Connecteren via de MS SQL native .NET Data Provider vereist de System.Data.SqlClient namespace,

```
<%@ Import Namespace="System.Data.SqlClient" %>
```

terwijl voor MySQL volgende namespace geïmporteerd moet worden:

```
<%@ Import Namespace="ByteFX.Data.MySQLClient" %>
```

Alvorens bovenstaande import voor MySQL werkt, zijn nog extra aanpassingen vereist. Zo moet allereerst de ByteFX library (ByteFX.Data.dll) ingeladen worden in het .NET Framework. Dit gebeurt met behulp van de .NET Framework Configuration tool waar men de dll in de Assembly Cache kan toevoegen. Ten tweede moeten we de dll ook registreren in de web.config file die zich in de rootdirectory van de webapplicaties bevindt. Dit gebeurt aan de hand van volgende XML-string:

```
<add assembly="ByteFX.Data, Version=0.6.8.21263, Culture=neutral,
       PublicKeyToken=f2fef6fed1732fc1" />
```

Eenmaal de nodige namespace geïmporteerd is, wordt binnen de Page_Load() methode de databankconnectie gemaakt en de query uitgevoerd.
Voor ODBC gebeurt dit op volgende manier:

```
string strConn = "DSN=mssql_dsn;host;UID=sa;PWD=svhoecke;";
string strSQL = "SELECT * FROM tabelnaam";
OdbcConnection conn = new OdbcConnection(strConn);
OdbcCommand cmd = new OdbcCommand(strSQL, conn);
conn.Open();
OdbcDataReader objDR = cmd.ExecuteReader();
```

Bovenstaande code geldt voor de MS SQL Server. Om met MySQL te werken is analoog als bij ASP en ADO ook hier enkel een aanpassing van de gebruikte DSN nodig.

```
string strConn = "DSN=mysql_dsn;host;UID=sofie;PWD=svhoecke;";
```

Met de MS SQL native .NET driver gebeurt het maken van de connectie en het opvragen van de query op de volgende manier:

```
SqlConnection objConn = new SqlConnection("Server=157.193.184.107;
      Database=svhoeckeDB; UId=sa; Pwd=svhoecke;");
objConn.Open();
string strQuery = "SELECT * FROM tabelnaam";
SqlCommand objCmd = new SqlCommand(strQuery, objConn);
objCmd.CommandType = CommandType.Text;
SqlDataReader objDR = objCmd.ExecuteReader();
```

Bij de MySQL native .NET Data Provider krijgt men een gelijkaardige manier van connecteren en opvragen van de query. In plaats van SqlConnection gebruikt men hier MySQLConnection en SqlCommand wordt vervangen door de MySQLCommand methode.

```
MySQLConnection objConn = new MySQLConnection("data
      source=10.10.10.86;database=svhoeckeDB;UID=sofie;pwd=svhoecke;");
objConn.Open();
string strQuery = "SELECT * FROM tabelnaam";
MySQLCommand objCmd = new MySQLCommand(strQuery, objConn);
objCmd.CommandType = CommandType.Text;
MySQLDataReader objDR = objCmd.ExecuteReader();
```

Het creëren van de databankconnectie, alsook het opvragen van de query, gebeurt via OLEDB op een gelijkaardige manier.

```
OleDbConnection objConn = new OleDbConnection("Provider=SQLOLEDB;
      server=157.193.184.107;Integrated Security=SSPI;
      Initial Catalog=svhoeckeDB");
objConn.Open();
string strQuery = "SELECT * FROM tabelnaam";
OLEDBCommand objCmd = new OLEDBCommand(strQuery, objConn);
objCmd.CommandType = CommandType.Text;
OleDbDataReader objDR = objCmd.ExecuteReader();
```

Zoals reeds vroeger gezegd, biedt de DataReader een eenvoudige en efficiënte toegang tot een verzameling van records. Bij ADO moesten we onze RecordSet doorlopen op volgende manier:

```
while not adoRS.EOF
response.write adoRS.fields(0).value & adoRS.fields(1).value
adoRS.MoveNext
wend
```

Bij ADO.NET is het gebruik van de MoveNext() methode overbodig. Men maakt immers gebruik van de Read() methode. Deze Read() verplaatst automatisch zijn pointer en geeft false wanneer er geen data meer beschikbaar is.

```
while(objDR.Read())
{
 Response.Write(objDR[0]);
 Response.Write(objDR[1]);
}
```

Na het weergeven van de data-output, moeten de DataReader en de databankconnectie gesloten worden.

```
objDR.Close();
objConn.Close();
```

Het werken met de DataReader maakt echter geen gebruik van de mogelijkheid binnen ADO.NET om connectieloos te werken. Daarvoor moet men gebruik maken van de DataSet in plaats van de DataReader.
Omdat het werken met een DataSet voor alle connectiemogelijkheden op een gelijkaardige manier verloopt, wordt enkel het geval van OLEDB hier besproken.

```
<Script Language=VB Runat=Server>
   Sub Page_Load(Sender As Object, e As EventArgs)
     Dim objConn As OleDbConnection
     Dim MyCommand As OleDbDataAdapter
     Dim MyDataset As DataSet
     Dim MyTable As DataTable
     Dim index, numrows As Integer
     Dim strQuery As String
     strQuery = "SELECT * FROM tabelnaam"
```

Het creëren van de databankconnectie en opvragen van de query gebeurt op gelijkaardige manier als bij de DataReader.

```
objConn = New OleDbConnection("Provider=SQLOLEDB;" _
      & "server=157.193.184.107;Integrated Security=SSPI;" _
      & "Initial Catalog=svhoeckeDB")
MyCommand = New OleDbDataAdapter(strQuery, objConn)
```

Vervolgens moet de DataSet gevuld worden met de data die verkregen werd uit de databank.

```
MyDataset = New DataSet
MyCommand.Fill(MyDataset)
```

Nu moet er een nieuw DataTable object gecreëerd worden en hieraan moet de nieuwe tabel in de Tables collectie toegevoegd worden.

```
MyTable = New DataTable
MyTable = MyDataset.Tables(0)
```

Met behulp van de Count methode komt men te weten hoeveel rijen er in de Rows collectie van het nieuwe DataTable object zijn.
Eenmaal dit allemaal gebeurd is, kan men de resultaten uitschrijven.

```
numrows = MyTable.Rows.Count
For index = 0 To numrows - 1
   Response.Write(MyTable.Rows(index).Item("fnaam") & "<br>")
Next index
```

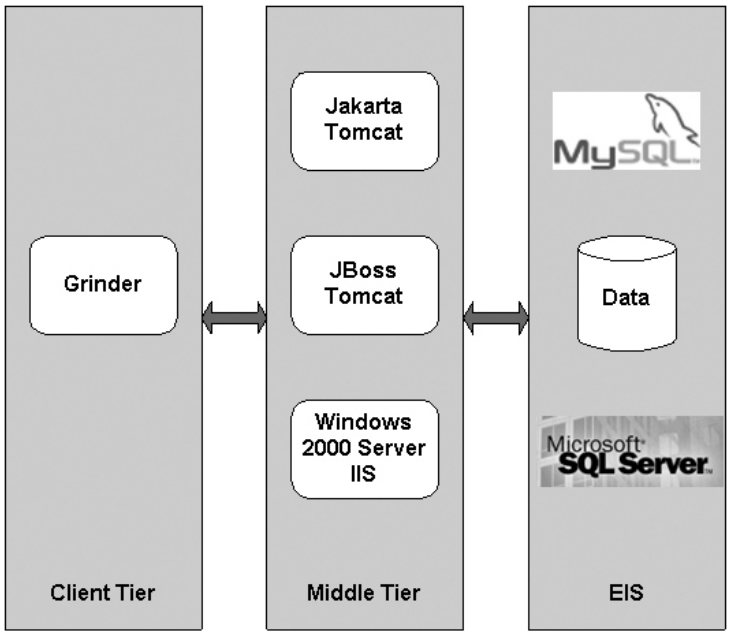
Deze manier van werken heeft vooral als groot voordeel dat men connectieloos of offline werkt. Na het raadplegen van de databank wordt een cache gecreëerd in het DataSet object en wordt de connectie met de databank gesloten. Alle bewerkingen op deze cache gebeuren dan offline. Voor het opvragen van data die up to date moet zijn, gebruikt men echter beter de DataReader omdat het werken via een DataSet voor overhead zorgt. Het gebruik van de DataSet komt pas tot zijn recht bij uitgebreide bewerkingen op de verkregen data, of wanneer meerdere keren doorlopen van de data vereist is.

Hoofdstuk 6

Testresultaten

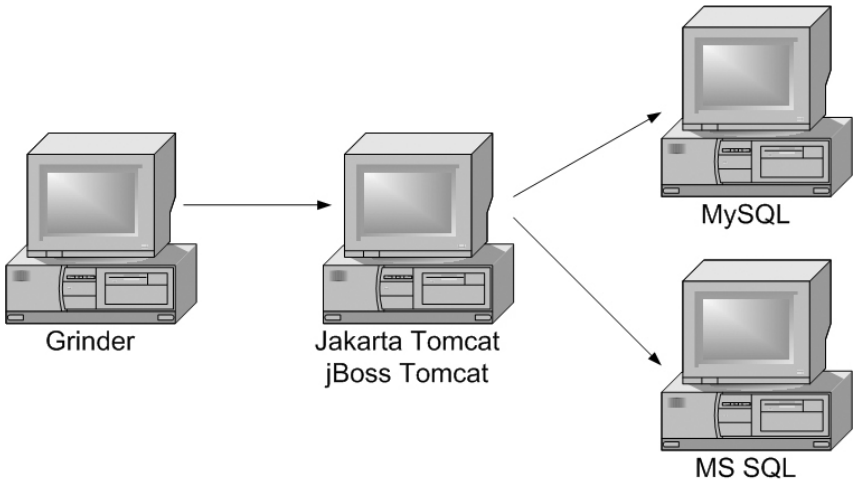
6.1 Testopstelling

Bij de testen worden er telkens twee databanken bekeken, namelijk MySQL en MS SQL Server. Als applicatieserver wordt de Jakarta Tomcat voor de servlet gebruikt, de jBoss Tomcat voor de session bean en Windows 2000 server met IIS voor de ASP en ASP.NET pagina's. Dit leidt tot het meerlagenmodel zoals weergegeven in figuur 6.1.

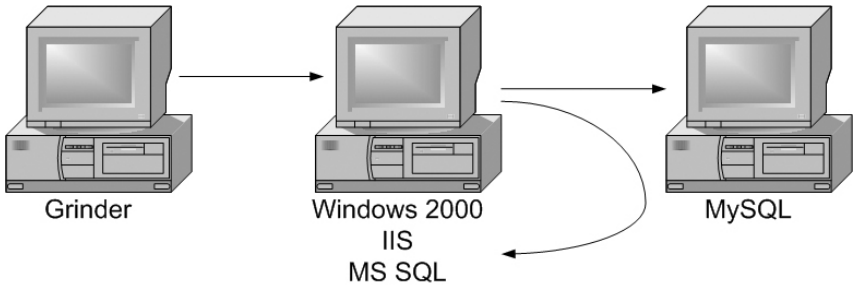


Figuur 6.1: Lagenstructuur testopstelling

Figuur 6.2 en figuur 6.3 geven de gebruikte testopstelling in de praktijk weer.



Figuur 6.2: Testopstelling J2EE



Figuur 6.3: Testopstelling .NET

De gebruikte pc's waren allen een AMD Athlon 1800+.

Het feit dat zowel IIS als MS SQL server op eenzelfde pc draaien, zorgt niet echt voor afwijkende resultaten. Er zal hierdoor slechts een minieme winst gemeten worden bij de .NET-testen met MS SQL omdat er geen netwerktransfer nodig is tussen applicatieserver en databank. Aangezien de datatransfer telkens onder de beperkingen van het netwerk blijft, m.a.w. onder de 100 Mbit, fungeert de netwerktransfer niet als beperkende factor, maar enkel de query en het aantal simultane clients. Wanneer er bijvoorbeeld grotere bestanden uit de databank zouden gehaald worden, zoals videobestanden (avi), dan zal de netwerktransfer wel als beperking optreden en zal er een grotere winst optreden wanneer zowel applicatieserver en databank zich op eenzelfde computer bevinden. Ook het CPU-

gebruik blijft telkens onder de 100% wat een aanduiding is dat het systeem niet beperkt wordt door het draaien van zowel IIS als de MS SQL server.

6.2 Testscript Grinder

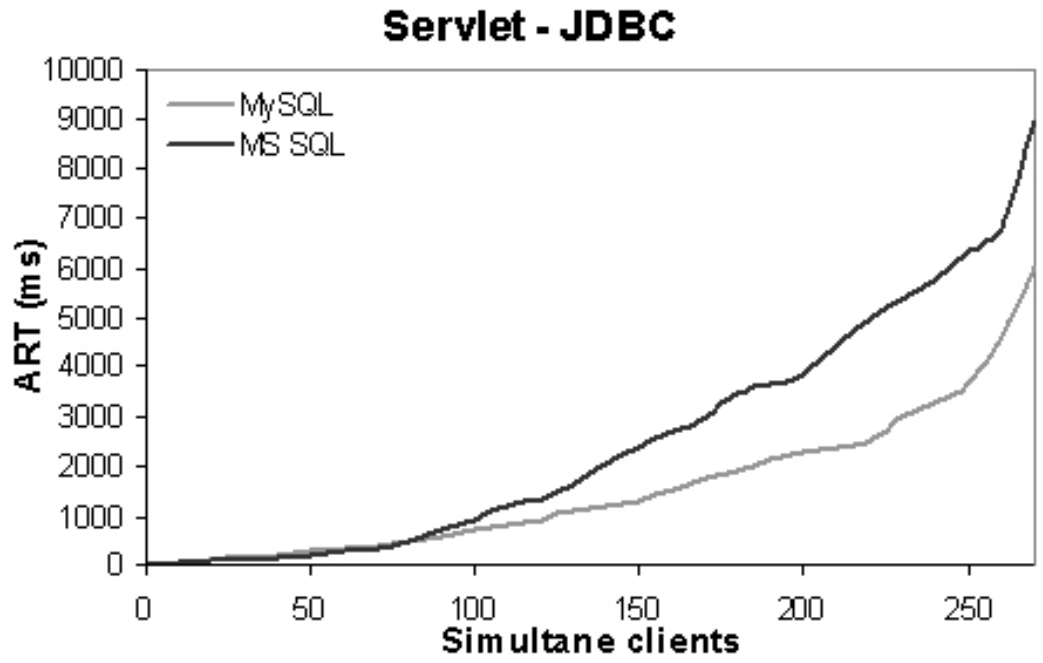
```
grinder.processes=1
grinder.threads=10
grinder.cycles=100

grinder.receiveConsoleSignals=false
grinder.reportToConsole=false

grinder.plugin.parameter.useCookies=true
grinder.plugin=net.grinder.plugin.http.HttpPlugin

grinder.test0.parameter.url=http://10.10.10.85:8080/query
```

Wanneer men de instellingen van het testscript wat nader bekijkt, dan merkt men op dat er slecht één process is, dit stemt overeen met het aantal virtuele machines, de JVM of CLR. Het aantal threads komt overeen met het aantal simultane clients. Het aantal cycles is vrij te kiezen, maar kiest men best zo hoog mogelijk om een gemiddelde antwoordtijd als resultaat te verkrijgen die niet te afhankelijk is van schommelingen. Als URL geeft men de URL naar de te testen applicatie mee. Aangezien men via HTTP-URL's werkt, is deze performantietool zowel geschikt voor de J2EE als de .NET applicaties. De Grinder biedt de mogelijkheid om met een grafische console te werken. Receive en report met deze console staat echter beiden op false omdat het werken met de console voor overhead zorgt en de resultaten bijgevolg zou beïnvloeden.



Figuur 6.4: Servlet met JDBC

6.3 J2EE

6.3.1 Servlet

Wanneer men de basisconfiguratie van de Jakarta Tomcat gebruikt, kan men uit figuur 6.4 afleiden dat het werken met MySQL bij een kleine query tot betere resultaten leidt dan bij MS SQL.
In server.xml, de configuratiefile van de Jakarta Tomcat server, kan men de instellingen van deze server wijzigingen.

```
<!-- Define a non-SSL Coyote HTTP/1.1 Connector on port 8080 -->
<Connector className="org.apache.coyote.tomcat4.CoyoteConnector"
           port="8080" minProcessors="5" maxProcessors="75"
           enableLookups="true" redirectPort="8443"
           acceptCount="300" debug="5" connectionTimeout="20000"
           useURIValidationHack="false" disableUploadTimeout="true" />
```

Een Connector stelt de interface voor tussen de externe clients die requests versturen naar (en antwoorden ontvangen van) een bepaalde Service.
Het HTTP/1.1 Connector element stelt een Connector component voor die het HTTP/1.1 protocol ondersteunt. Het maakt dat Catalina fungeert als een stand-alone web server, bovenop zijn taak om JSP pagina's en servlets uit te voeren. Een bepaalde instantie van deze component luistert naar connecties op een specifieke TCP poort op de server.
Een aantal attributen van deze HTTP/1.1 Connector zijn:

className De Java class naam van de implementatie die gebruikt moet worden. Deze class moet de org.apache.catalina.Connector interface implementeren.

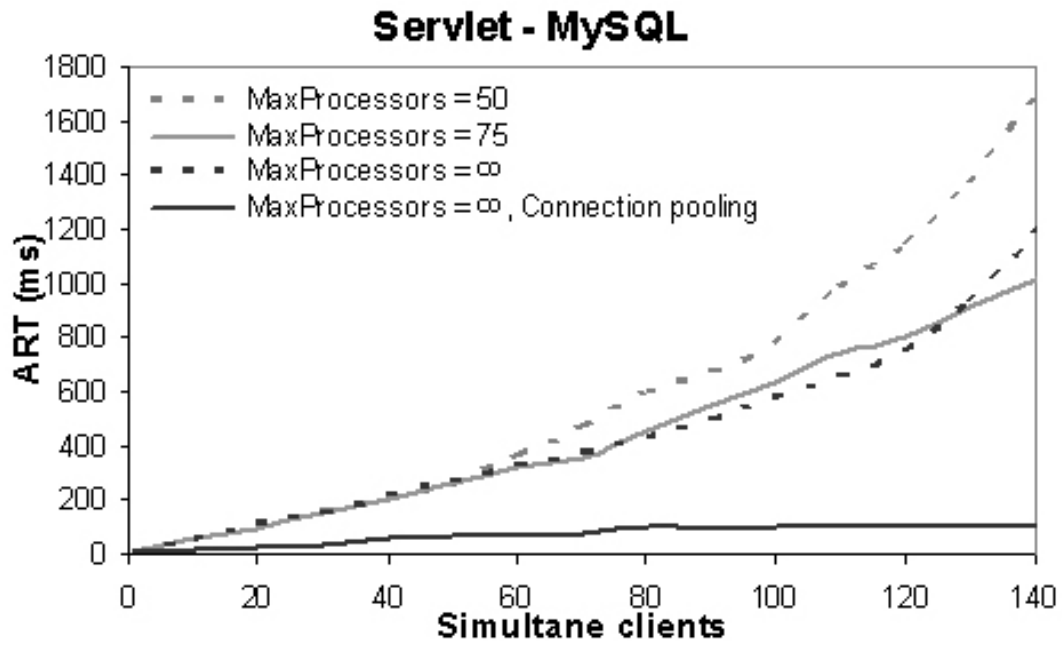
port Het TCP poortnummer waarop de Connector zijn server socket zal creëren en wachten op inkomende connecties.

minProcessors Het aantal request verwerkende threads dat gecreëerd zal worden wanneer de Connector voor het eerst gestart wordt. Deze waarde moet kleiner zijn dan maxProcessors. De default waarde is 5.

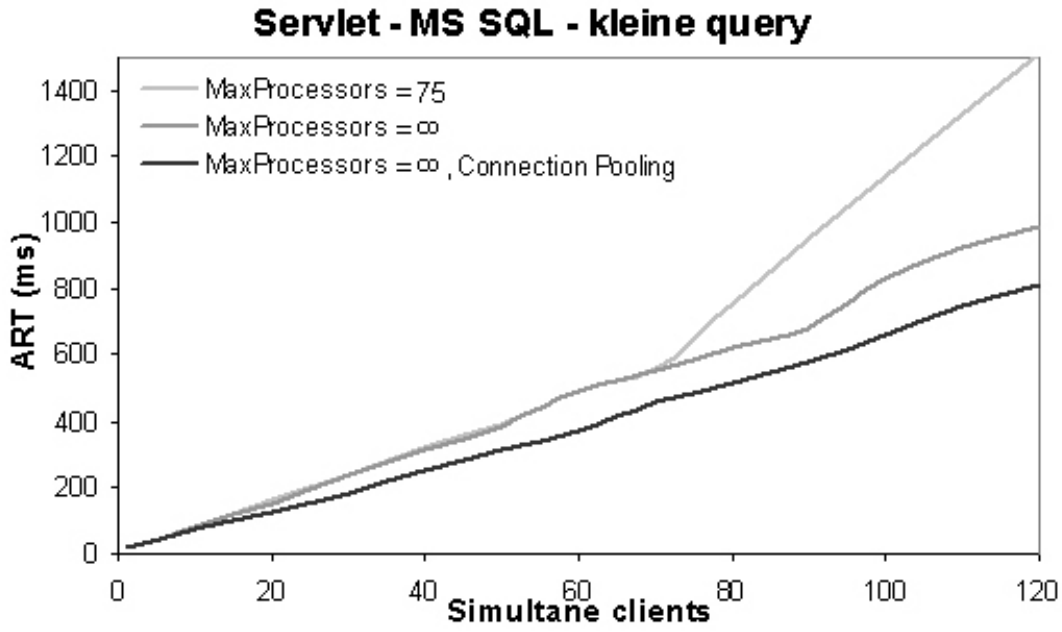
maxProcessors Het maximum aantal request verwerkende threads dat door de Connector gecreëerd kan worden. Deze waarde determineert het maximum aantal simultane clients dat behandeld kan worden. Wanneer maxProcessors op -1 staat, is het aantal request verwerkende threads oneindig groot.

acceptCount De maximum queue lengte voor inkomende connectie-aanvragen wanneer alle mogelijke request verwerkende threads in gebruik zijn. Request die ontvangen worden wanneer deze queue vol is, worden geweigerd.

Uit figuur 6.5 en figuur 6.6 kan men afleiden dat het variëren van maxProcessors rechtstreeks invloed heeft op de performantie. Wanneer maxProcessors bereikt wordt, treedt er plotseling een grotere richtingscoëfficiënt op in de grafiek. Een stijgende waarde voor maxProcessors brengt bijgevolg een betere performantie met zich mee. Deze vaststelling kan echter niet tot in het oneindige doorgetrokken worden, zoals ook te zien is op de figuur. Op een bepaald moment zal voor een te groot aantal simultane clients, de threadbehandeling belastender zijn dan de clientapplicatie op zich. Het extra inschakelen van

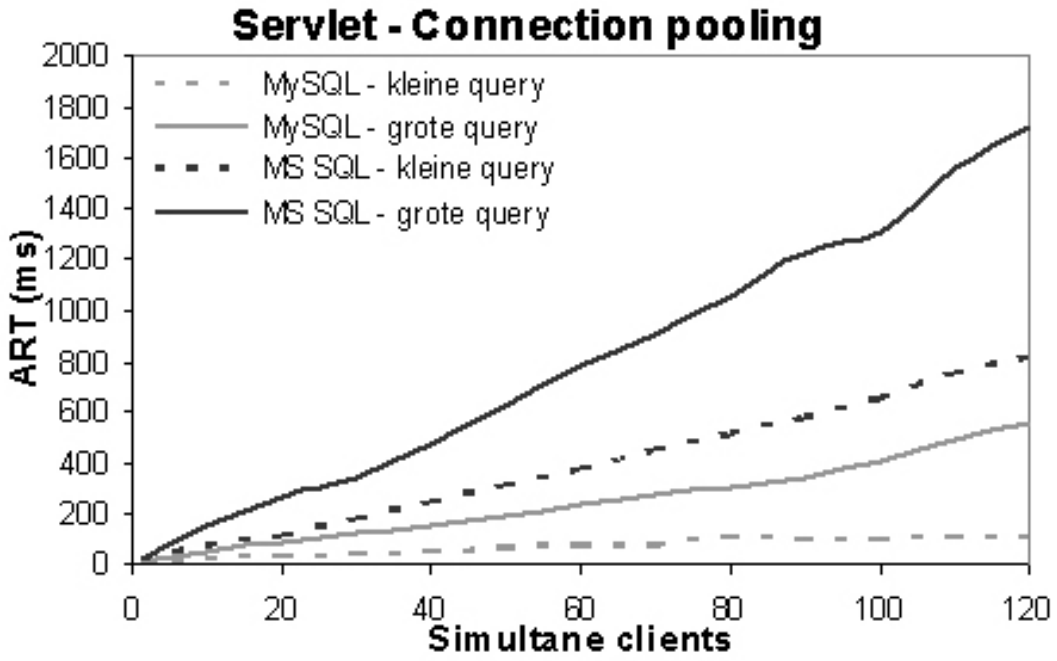


Figuur 6.5: Optimalisatie van de serverconfiguratie bij MySQL



Figuur 6.6: Optimalisatie van de serverconfiguratie bij MS SQL

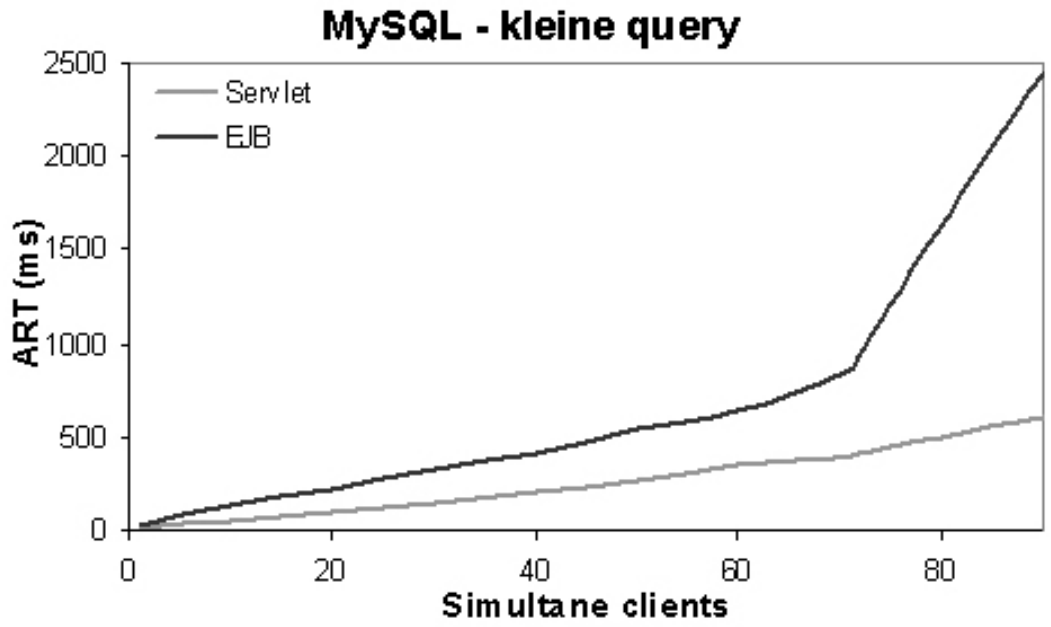
Connection Pooling zorgt voor een supplementaire daling van de antwoordtijd. Wanneer men voor zowel MySQL als MS SQL de resultaten vergelijkt met de meest optimale server-instelling, namelijk maxProcessors op oneindig en Connection Pooling (zie figuur 6.7), dan ziet men dat het effect van Connection Pooling minder invloed heeft in het geval van MS SQL. Dit is te verklaren door de gebruikte JDBC drivers. Bij MS SQL is het verschil op vlak van performantie tussen commerciële en freeware JDBC drivers enorm groot [19]. Het gebruik van een commerciële JDBC driver voor MS SQL zou hier tot betere resultaten leiden.



Figuur 6.7: Optimale serverconfiguratie plus connection pooling

6.3.2 Session Bean

Wanneer men dezelfde databankquery via een session bean uitvoert in plaats van via een servlet, krijgt men het resultaat dat te zien is op figuur 6.8. Men merkt direct op dat het werken met session beans minder performant is dan het werken via servlets. Dit is eenvoudig te verklaren aangezien bij een servlet de presentatielogica en bedrijfslogica alle-

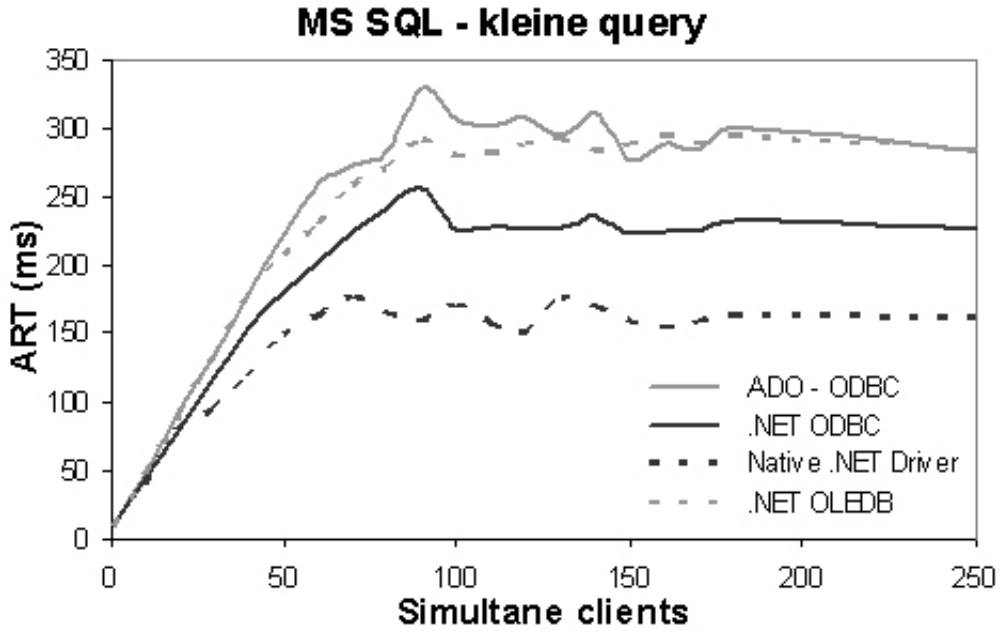


Figuur 6.8: Session EJB vs servlet

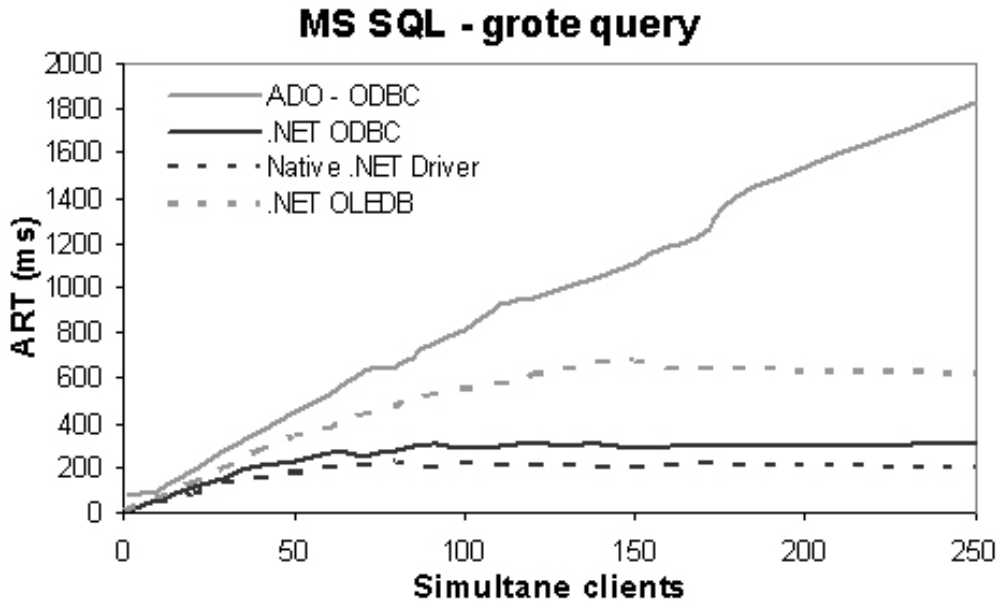
maal in één en dezelfde class geïmplementeerd zijn. Bij een session bean maken we echter gebruik van meerdere classes. Presentatielogica en bedrijfslogica zijn daar gescheiden van elkaar. Hierdoor is de antwoordtijd bij session beans minder performant. Een voordeel van session beans ten opzichte van servlets is echter dat deze scheiding van presentatie en bedrijfslogica herbruikbaarheid van de code veel eenvoudiger maakt dan bij servlets. Bij de session bean merkt men ook op dat de stijging van de richtingscoëfficiënt wanneer maxProccessors bereikt wordt, veel groter is dan de stijging bij de servlet.

6.4 .NET

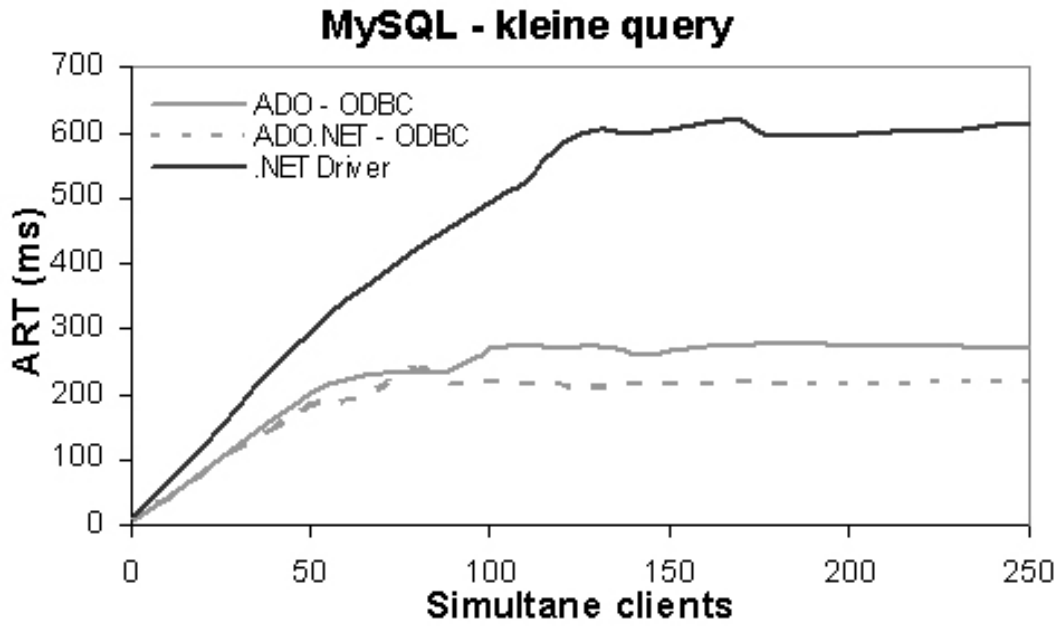
Bij .NET is een configuratie van de IIS niet nodig. Deze IIS staat standaard geconfigureerd zodat een oneindig aantal simultane clients toegang krijgen, alsook connection pooling staat automatisch aan indien de gebruikte drivers pooling ondersteunen.
Wat de MS SQL server betreft, zijn er vier verschillende connectiemogelijkheden met elkaar vergeleken. Enerzijds ADO met ODBC en anderzijds ADO.NET met ODBC, OLEDB en de native .NET driver.



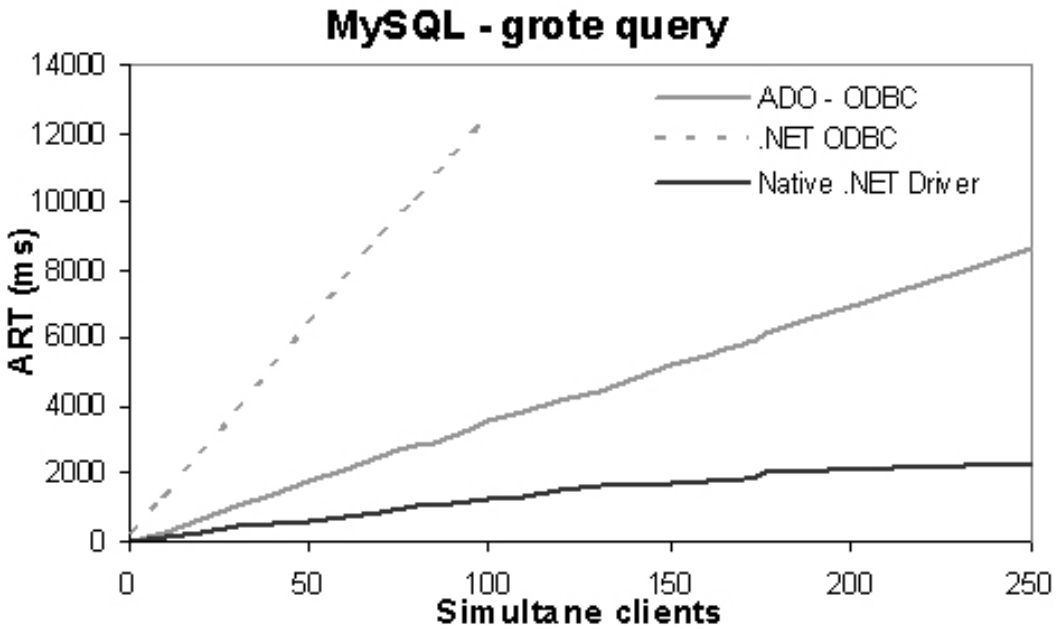
Figuur 6.9: .NET resultaten bij kleine query met MS SQL



Figuur 6.10: .NET resultaten bij grote query met MS SQL



Figuur 6.11: .NET resultaten bij kleine query met MySQL

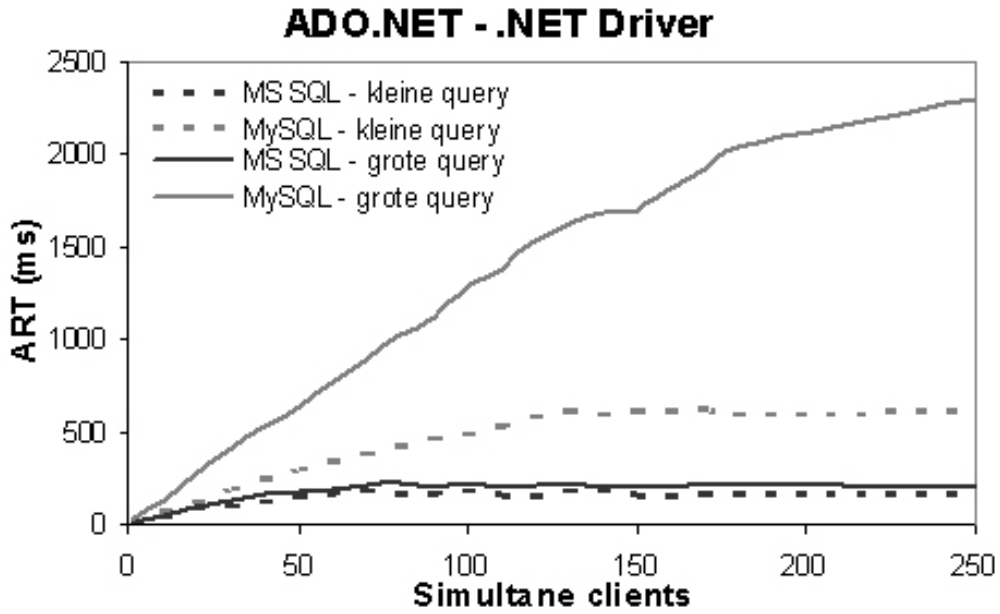


Figuur 6.12: .NET resultaten bij grote query met MySQL

Uit de figuren 6.9 en 6.10 kan men afleiden dat zowel voor kleine als grote query's het gebruik van de native .NET Data Provider de meest performante oplossing biedt. Dit was te verwachten aangezien deze native .NET driver speciaal ontworpen is om op een efficiënte en snelle manier databankconnecties te creëren zonder gebruik van een ODBC of OLEDB brug.

Aangezien MySQL officieel OLEDB niet ondersteunt, zijn er slechts drie mogelijkheden voor MySQL getest, namelijk ADO met ODBC en ADO.NET met enerzijds ODBC en anderzijds de native .NET driver.

Voor kleine query's ziet men op figuur 6.11 dat ADO.NET met ODBC de performantste oplossing biedt. Voor grote query's daarentegen biedt de native .NET driver de beste oplossing (zie figuur 6.12). Op de figuur zijn ook opmerkelijk minder performante resultaten te zien bij ADO.NET met ODBC. Een mogelijke verklaring hiervoor is dat .NET standaard geen ODBC meer ondersteunt. Deze extra functionaliteit heeft als doel compabiliteit te garanderen met vroegere applicaties. Bij MS SQL waren hierdoor de resultaten voor ODBC.NET niet zo extreem aangezien Microsoft voor zijn eigen databank deze ODBC connectie wel geoptimaliseerd heeft.

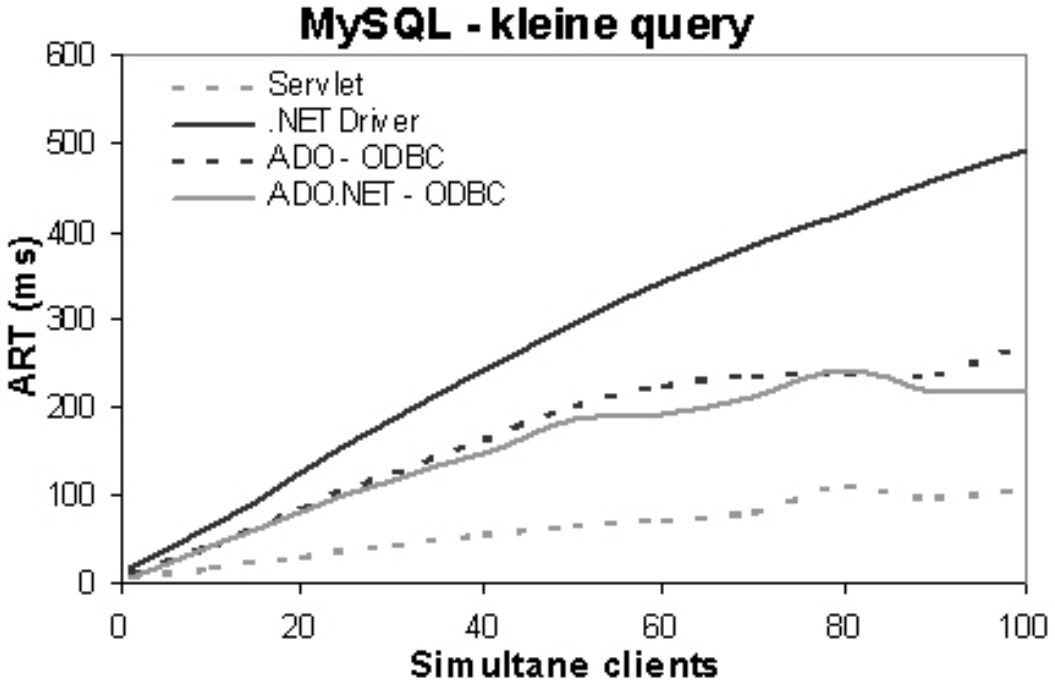


Figuur 6.13: Native .NET Data Provider

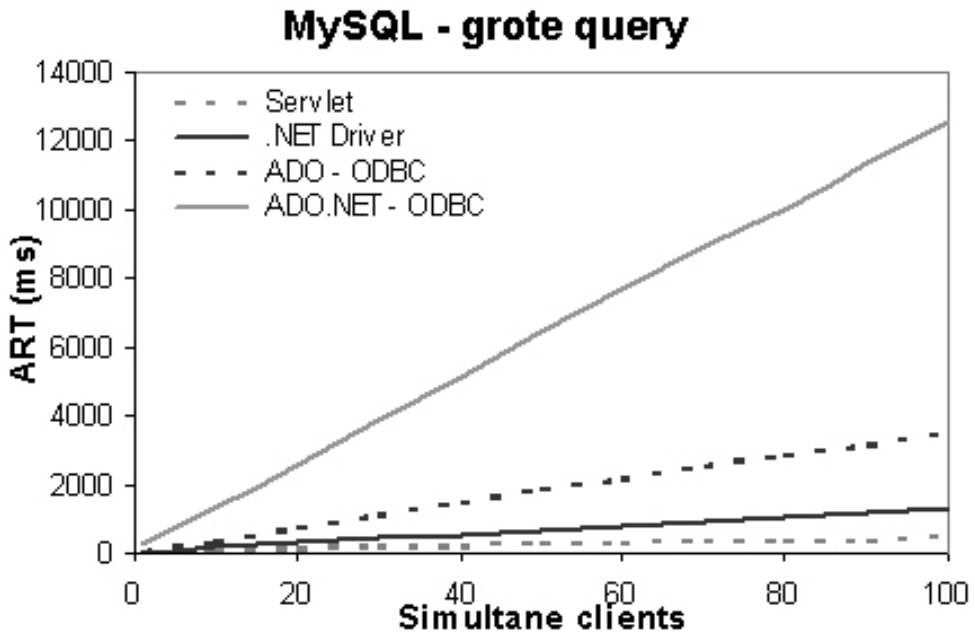
Voor grote query's biedt de native .NET Data Provider zowel voor MS SQL als MySQL de meest performante connectiemethode. Men merkt echter een groot verschil op tussen de antwoordtijden bij MS SQL en MySQL voor deze native .NET driver (zie figuur 6.13). Dit is te verklaren door het feit dat de MySQL native .NET driver nog geen connection pooling ondersteunt aangezien deze driver nog in ontwikkeling is, terwijl deze voor MS SQL dit wel ondersteunt. Eenmaal connection pooling wel geïmplementeerd wordt, kan men hier betere resultaten voor MySQL verwachten.

6.5 Vergelijking J2EE vs .NET

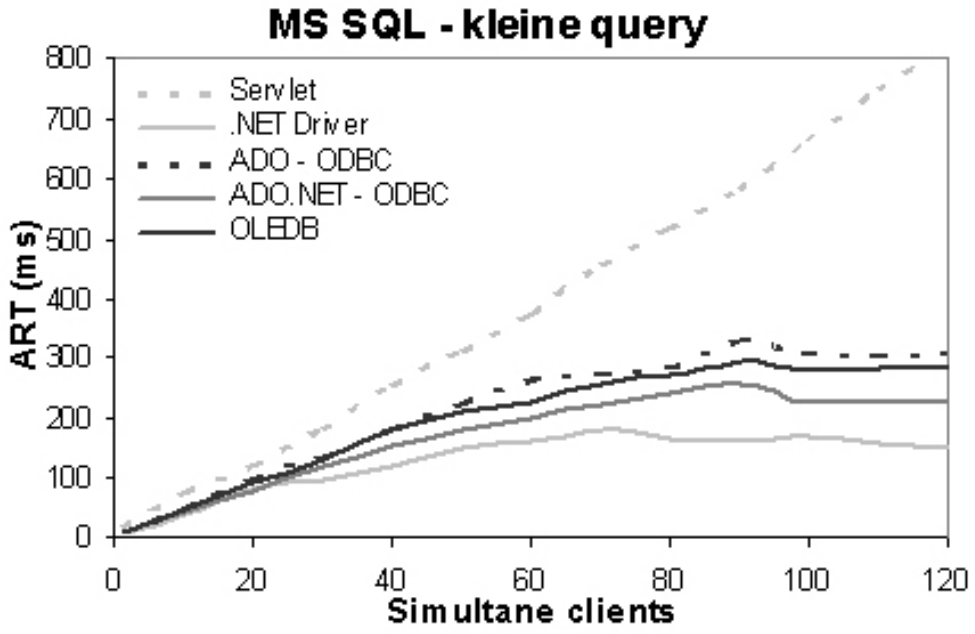
Wanneer men nu zowel de J2EE servletresultaten als de .NET resultaten op eenzelfde grafiek brengt, dan merkt men uit figuren 6.14 en 6.15 op dat zowel voor kleine als grote databankquery's bij MySQL de servlet de kleinste antwoordtijd heeft.



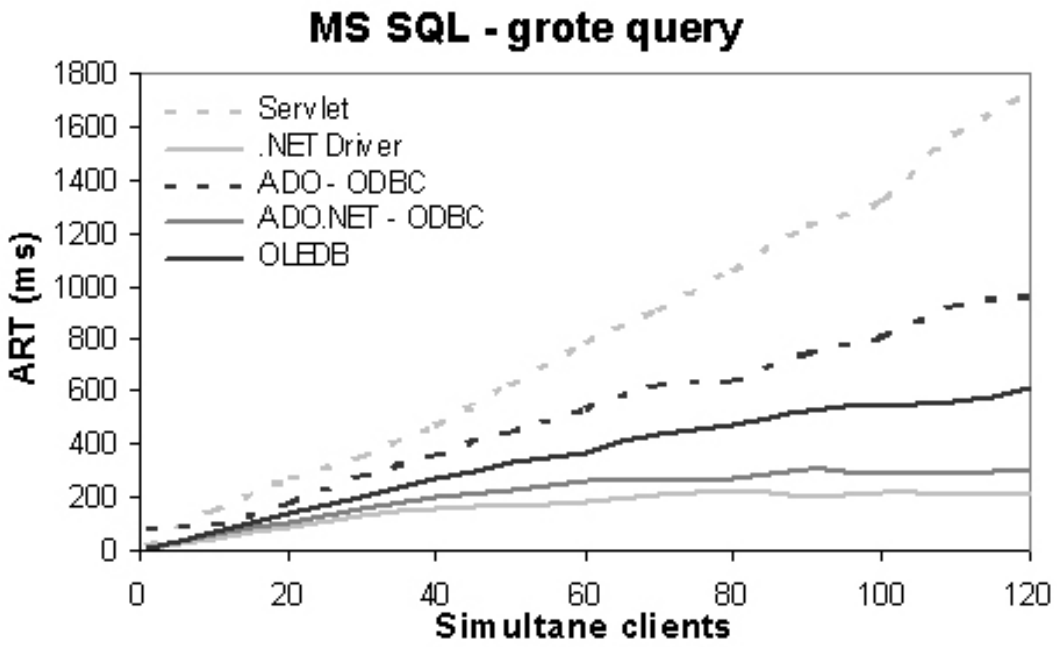
Figuur 6.14: Vergelijking J2EE vs .NET voor kleine query's met MySQL



Figuur 6.15: Vergelijking J2EE vs .NET voor grote query's met MySQL



Figuur 6.16: Vergelijking J2EE vs .NET voor kleine query's met MS SQL



Figuur 6.17: Vergelijking J2EE vs .NET voor grote query's met MS SQL

Bij MS SQL daarentegen heeft de native .NET driver de kleinste antwoordtijd (zie figuren 6.16 en 6.17).

Men moet echter deze resultaten met een korreltje zout nemen. Zoals reeds gezegd, ondersteunt de gebruikte freeware MySQL native .NET driver nog geen connection pooling aangezien deze driver nog in ontwikkeling is. Wanneer dit wel het geval zal zijn, bestaat de kans dat voor MySQL niet langer de servlet maar de native .NET driver de beste resultaten oplevert.
Voor MS SQL kan het gebruik van een commerciële JDBC driver voor betere resultaten zorgen wat de servlet betreft. Of deze resultaten ook beter zullen worden dan deze met de native .NET driver is dan de vraag.
Maar dit is nog niet alles. Naast driverafhankelijkheid van de resultaten, is er ook nog serverafhankelijkheid. De J2EE testen zijn gebeurd met de freeware jBoss applicatieserver. Het gebruik van commerciële applicatieservers zoals WebSphere of WebLogic kan ook nog eens extra invloed hebben op de resultaten.

Dit in het achterhoofd houdende, kan men dus niet eenduidig besluiten wat het beste resultaat geeft voor een bepaalde databank. Kijkt men echter naar de geteste omstandigheden, dan komt J2EE's servlet als performantste methode naar voren voor MySQL, terwijl voor MS SQL de beste resultaten geboekt worden met de native .NET Data Provider.

Hoofdstuk 7

Besluit

7.1 Resultaten

Tijdens dit afstudeerwerk is duidelijk geworden dat het niet eenvoudig is een eenduidige performantievergelijking te maken tussen J2EE en .NET aangezien de performantie door meerdere factoren beïnvloed wordt:

Configuratie Uit hoofdstuk 3 en 6 blijkt dat connection pooling een belangrijke performantiefactor is. Zo is connection pooling voor beide technologieën geïmplementeerd. Bij .NET wordt automatisch gebruik gemaakt van deze connection pooling, terwijl er bij J2EE extra serverconfiguraties vereist zijn alvorens connection pooling optreedt. Deze serverconfiguratie verdient zeker extra aandacht bij J2EE-implementaties. In de toekomst zal er waarschijnlijk bij J2EE ook automatisch gebruik gemaakt worden van connection pooling, zeker bij commerciële pakketten (als dit al niet zo is).

Databasedriver Zelfs wanneer men beide technologieën test met connection pooling, is de performantievergelijking steeds afhankelijk van de gebruikte drivers. Er zijn vandaag meerdere JDBC drivers en native .NET drivers op de markt. De keuze voor één bepaalde driver heeft een rechtstreekse invloed op de performantieresultaten.
Wanneer men de .NET resultaten bekijkt, komt men tot de conclusie dat het connecteren via de native .NET Data Provider de meest performante oplossing brengt in vergelijking met het werken via ODBC of OLEDB. Dit was te verwachten aangezien deze drivers speciaal ontwikkeld zijn voor snelle en efficiënte databanktoegang. Of

men beter voor .NET met de native .NET drivers kiest of voor J2EE's servlet kan niet eenduidig gezegd worden door de afhankelijkheid van drivers en applicatieservers. Maar onder de geteste omstandigheden bleek voor MySQL de servlet het performantst, terwijl .NET met zijn native driver als winnaar uit de bus kwam voor MS SQL.

Applicatieserver Ook de gebruikte applicatieserver zal invloed hebben. Bij .NET heeft men slechts één serverprovider, namelijk IIS van Microsoft. Maar bij J2EE is de verscheidenheid aan applicatieservers groter, gaande van Sun's applicatieserver, over jBoss, naar WepSphere en WebLogic. Aangezien Sun's applicatieserver slechts een reference implementatie is, is deze functioneel wel compleet maar allesbehalve snel en dus niet performant. Het gebruik van jBoss bracht hier een performantere oplossing. Antwoordtijden werden een factor 6 tot 10 sneller. Ook is het aantal simultane clients bij de Sun implementatie beperkt. Dit is een duidelijke aanwijzing dat de performantie van een applicatie ook afhangt van de gebruikte server. Het gebruik van commerciële applicatieservers kan hier nog een performantieverbetering teweegbrengen.

Learning curve Kijkt men naast performantieresultaten ook naar de learning curve, dan komt .NET naar voren als winnaar. Beide technologieën vereisen ongeveer eenzelfde programmeercapaciteit, maar bij J2EE drijft het vereiste configureren van de server de complexiteit de hoogte in.

Natuurlijk is het ook nog bepalend dat J2EE slechts één programmeertaal ondersteunt, terwijl bij .NET meerdere programmeertalen mogelijk zijn. .NET is ontwikkeld voor het Windows platform, terwijl J2EE platformonafhankelijk is.

De vraag of een ontwikkelaar met het oog op de toekomst het beste voor Microsoft .NET of voor Java kan kiezen, kan dus niet eenduidig beantwoord worden, maar zal een afwegen worden van de verschillende factoren. Er wordt verwacht dat het grootste deel van Microsofts marktaandeel gedomineerd zal worden door de SMB's (small and midsize businesses), aangezien deze bedrijven hun infrastructuur geconcentreerd is rond één enkel platform. Hierbij lijkt .NET aantrekkelijker. Grote bedrijven zullen echter de eerste jaren

zowel Microsoft als Java technologieën blijven ondersteunen. Doordat de infrastructuur bij grote bedrijven bestaat uit een variëteit aan hardware en software, zullen deze bedrijven echter meer gaan investeren in J2EE. Maar ondanks dat zullen ook Microsoft technologieën blijven gebruikt worden. Daarom is het nodig dat applicatie integratie strategieën ontwikkeld worden die interoperabiliteit tussen de beide technologieën klaarspelen [20]. Een mogelijke oplossing om deze interoperabiliteit te creëren bestaat in het gebruik van Web Services waarbij data uitgewisseld wordt in de vorm van een XML-bestand.

7.2 Verder onderzoek

In volgende domeinen is verder onderzoek echter noodzakelijk :

- In hoofdstuk 3 werd, naast het bespreken van de invloed van connection pooling op de performantie, ook front- en back-end caching vermeld als mogelijke manier om de performantie te verhogen. Aangezien zowel de diversiteit als de intensiteit aan informatie van de webapplicatie invloed heeft op de performantie bij het cachen, is verder onderzoek hiervan noodzakelijk.

- Net zoals een vergelijking gemaakt is tussen webapplicaties bij J2EE en .NET voor databanktoegang, kan een analoog onderzoek gedaan worden op het vlak van de componenten met de EJB's voor J2EE en COM+ voor .NET.

- Voor J2EE heeft het onderzoek zich beperkt tot freeware applicatieservers. Er werd een groot verschil gemerkt tussen SUN's reference applicatieserver en jBoss. Dat het gebruik van commerciële applicatieservers nog een betere performantie met zich mee kan brengen is niet onrealistisch en vormt bijgevolg nog stof tot verder onderzoek.

- Beide technologieën werden getest op een Intel-architectuur[1]. J2EE ondersteunt echter meerdere platformen zoals bijv. het Sun-platform wat eventueel een invloed op de performantie zou kunnen hebben.

- Ook de invloed van verschillende bandbreedtes bij de clients vormt een onderwerp tot verder onderzoek. Bij de testen was het immers zo dat alle clients eenzelfde

[1] AMD behoort tot de Intelarchitectuur.

bandbreedte gebruikten. In de praktijk zal dit echter niet het geval zijn. Aangezien zowel clients met smallband als met breedband een databankquery zullen opvragen, zal dit voor extra belasting van de applicatieserver zorgen.

- Maar ook buiten het domein van databanktoegang is er nog veel te onderzoeken: Web Services, XML support, E-Commerce,...

Bijlage A

ConnectionPool.java

```
import java.sql.*;
import java.util.*;
/**
 * ConnectionPool creates a pool of connections of the specified
 * size to the specified database. The connection pool object
 * allows the client to specify the JDBC driver, database,
 * username, and password. In addition, the client can also
 * specify the number of connections to create when this class is
 * instantiated, the number of additional connections to create
 * if all connections are exhausted, and the absolute maximum
 * number of connections.
 * @author Dustin R. Callaway
 */
public class ConnectionPool {
private String jdbcDriver = "";
private String dbUrl = "";
private String dbUsername = "";
private String dbPassword = "";
private String testTable = "";
private int initialConnections = 10;
```

```
private int incrementalConnections = 10;
private int maxConnections = 150;
private Vector connections = null;
/**
 * Constructor stores the parameters passed by the calling
 * object.
 *
 * @param jdbcDriver String containing the fully qualified name
 * of the jdbc driver (class name and full package info)
 * @param dbUrl String containing the database URL
 * @param dbUsername String containing the username to use when
 * logging into the database
 * @param dbPassword String containing the password to use when
 * logging into the database
 */
public ConnectionPool(String jdbcDriver, String dbUrl, String
dbUsername, String dbPassword) {
   this.jdbcDriver = jdbcDriver;
   this.dbUrl = dbUrl;
   this.dbUsername = dbUsername;
   this.dbPassword = dbPassword;
}
/**
 * Returns the initial number of connections to create.
 *
 * @return Initial number of connections to create.
 */
public int getInitialConnections() {
   return initialConnections;
}
```

```
/**
 * Sets the initial number of connections to create.
 *
 * @param initialConnections Initial number of connections to
 * create
 */
public void setInitialConnections(int initialConnections) {
   this.initialConnections = initialConnections;
}
/**
 * Returns the number of incremental connections to create if
 * the initial connections are all in use.
 *
 * @return Number of incremental connections to create.
 */
public int getIncrementalConnections() {
   return incrementalConnections;
}
/**
 * Sets the number of incremental connections to create if
 * the initial connections are all in use.
 *
 *@param incrementalConnections Number of incremental
 * connections to create.
 */
public void setIncrementalConnections( int incrementalConnections){
   this.incrementalConnections = incrementalConnections;
}
/**
 * Returns the absolute maximum number of connections to
```

```
 * create. If all connections are in use, the getConnection()
 * method will block until one becomes free.
 *
 * @return Maximum number of connections to create.
 */
public int getMaxConnections() {
   return maxConnections;
}
/**
 * Sets the absolute maximum number of connections to create.
 * If all connections are in use, the getConnection() method
 * will block until one becomes free.
 *
 * @param maxConnections Maximum number of connections to
 * create.
 */
public void setMaxConnections(int maxConnections) {
   this.maxConnections = maxConnections;
}
/**
 * Returns the name of the table that should be tested to
 * insure that the database connection is still open.
 *
 * @return Name of the database table used to test the
 * connection.
 */
public String getTestTable() {
   return testTable;
}
/**
```

```
 * Sets the name of the table that should be tested to insure
 * that the database connection is still open.
 *
 * @param testTable Name of the database table used to test the
 * connection.
 */
public void setTestTable(String testTable) {
   this.testTable = testTable;
}
/**
 * Creates a pool of connections. Number of connections is
 * determined by the value of the initialConnections property.
 */
public synchronized void createPool() throws Exception {
   //make sure that createPool hasn't already been called
   if (connections != null) {
      return; //the pool has already been created, return
   }
   //instantiate JDBC driver object from init param jdbcDriver
   Driver driver = (Driver)(Class.forName(jdbcDriver).newInstance());
   DriverManager.registerDriver(driver); //register JDBC driver
   connections = new Vector();
   //creates the proper number of initial connections
   createConnections(initialConnections);
}
/**
 * Creates the specified number of connections, places them in
 * a PooledConnection object, and adds the PooledConnection to
 * the connections vector.
 *
```

```
 * @param numConnections Number of connections to create.
 */
private void createConnections(int numConnections)
throws SQLException {
   //create the specified number of connections
   for (int x=0; x < numConnections; x++) {
       //have the maximum number of connections been created?
       //a maxConnections value of zero indicates no limit
       if (maxConnections > 0 && connections.size() >= maxConnections) {
           break; //break out of loop because we're at the maximum
       }
       //add a new PooledConnection object to connections vector
       connections.addElement(new PooledConnection(newConnection()));
       System.out.println("Database connection created...");
   }
}
/**
 * Creates a new database connection and returns it.
 *
 * @return New database connection.
 */
private Connection newConnection() throws SQLException {
   //create a new database connection
   String sConnect = "jdbc:microsoft:sqlserver://157.193.184.107:1433;";
   sConnect += "DatabaseName=svhoeckeDB;user=sa;password=svhoecke;";
   sConnect += "SelectMethod=cursor";
   Connection conn = DriverManager.getConnection (sConnect);
   //if this is the first connection, check the maximum number
   //of connections supported by this database/driver
   if (connections.size()== 0) {
```

```
        DatabaseMetaData metaData = conn.getMetaData();
        int driverMaxConnections = metaData.getMaxConnections();
        //driverMaxConnections value of zero indicates no maximum
        //or unknown maximum
        if (driverMaxConnections > 0 && maxConnections > driverMaxConnections)
        {   maxConnections = driverMaxConnections;
        }
    }
    return conn; //return the new connection
}
/**
 * Attempts to retrieve a connection from the connections
 * vector by calling getFreeConnection(). If no connection is
 * currently free, and more can not be created, getConnection()
 * waits for a short amount of time and tries again.
 *
 * @return Connection object
 */
public synchronized Connection getConnection()throws SQLException
{
    //make sure that createPool has been called
    if (connections == null) {
        System.out.println("the pool has not been created");
        return null; //the pool has not been created
    }
    Connection conn = getFreeConnection();//get free connection
    System.out.println("Connection_connectionpool.java");
    while (conn == null) //no connection was currently free {
        //sleep for a quarter of a second and then check to see if
        //a connection is free
```

```
            wait(250);
            conn = getFreeConnection(); //try again to get connection
        }
        return conn;
    }
    /**
     * Returns a free connection from the connections vector. If no
     * connection is available, a new batch of connections is
     * created according to the value of the incrementalConnections
     * variable. If all connections are still busy after creating
     * incremental connections, the method will return null.
     *
     * @return Database connection object
     */
    private Connection getFreeConnection() throws SQLException {
        //look for a free connection in the pool
        Connection conn = findFreeConnection();
        if (conn == null) {
            //no connection is free, create additional connections
            createConnections(incrementalConnections);
            //try again to find a free connection
            conn = findFreeConnection();
            if (conn == null) {
                //there are still no free connections, return null
                return null;
            }
        }
        return conn;
    }
    /**
```

```
 * Searches through all of the pooled connections looking for
 * a free connection. If a free connection is found, its
 * integrity is verified and it is returned. If no free
 * connection is found, null is returned.
 *
 * @return Database connection object.
 */
private Connection findFreeConnection()throws SQLException {
   Connection conn = null;
   PooledConnection pConn = null;
   Enumeration enum = connections.elements();
   //iterate through the pooled connections looking for free one
   while (enum.hasMoreElements()) {
       pConn =(PooledConnection)enum.nextElement();
       if (!pConn.isBusy()) {
          //this connection is not busy, get a handle to it
          conn = pConn.getConnection();
          pConn.setBusy(true); //set connection to busy
          //test the connection to make sure it is still valid
          if  (!testConnection(conn)) {
              //connection is no longer valid, create a new one
              conn = newConnection();
              //replace invalid connection with new connection
              pConn.setConnection(conn);
          }
          break; //we found a free connection, stop looping
       }
   }
   return conn;
}
```

```
/**
 * Test the connection to make sure it is still valid. If not,
 * close it and return FALSE.
 *
 * @param conn Database connection object to test.
 * @return True indicates connection object is valid.
 */
private boolean testConnection(Connection conn) {
   try {
       //determine if a test table has been designated
       if (testTable.equals("")) {
          //There is no table to test the database connection so
          //try setting the auto commit property. This verifies
          //a valid connection on some databases. However, the
          //test table method is much more reliable.
          conn.setAutoCommit(true);
       }
       else {
          //check if this connection is valid
          Statement stmt = conn.createStatement();
          stmt.execute("select count(*) from " + testTable);
       }
   }
   catch(SQLException e) {
       //connection is no longer valid, attempt to close it
       closeConnection(conn);
       return false;
   }
   return true;
}
```

```
/**
 * Turns off the busy flag for the current pooled connection.
 * All ConnectionPool clients should call returnConnection() as
 * soon as possible following any database activity (within a
 * finally block).
 *
 * @param conn Connection object
 */
public void returnConnection(Connection conn) {
   //make sure that createPool has been called
   if (connections== null) {
       return; //the pool has not been created
   }
   PooledConnection pConn = null;
   Enumeration enum = connections.elements();
   //iterate through the pooled connections looking for the
   //returned connection
   while (enum.hasMoreElements()) {
       pConn =(PooledConnection)enum.nextElement();
       //determine if this pooled connection contains the returned
       //connection
       if (conn == pConn.getConnection()) {
          //the connection has been returned, turn off busy flag
          pConn.setBusy(false);
          break;
       }
   }
}
/**
 * Refreshes all of the connections in the connection pool.
```

```
 */
public synchronized void refreshConnections()throws SQLException {
   //make sure that createPool has been called
   if (connections == null) {
       return; //the pool has not been created
   }
   PooledConnection pConn = null;
   Enumeration enum = connections.elements();
   while (enum.hasMoreElements()) {
       pConn =(PooledConnection)enum.nextElement();
       if (!pConn.isBusy()) {
           wait(1000); //wait 5 seconds
       }
       closeConnection(pConn.getConnection());
       pConn.setConnection(newConnection());
       pConn.setBusy(false);
   }
}
/**
 * Closes all of the connections and empties the connection
 * pool. Once this method has been called, the createPool()
 * method can again be called.
 */
public synchronized void closeConnections() throws SQLException {
   //make sure that createPool has been called
   if (connections == null) {
      return; //the pool has not been created
   }
   PooledConnection pConn = null;
   Enumeration enum = connections.elements();
```

```
    while (enum.hasMoreElements()) {
        pConn = (PooledConnection)enum.nextElement();
        if (!pConn.isBusy()) {
            wait(5000); //wait 5 seconds
        }
        closeConnection(pConn.getConnection());
        connections.removeElement(pConn);
    }
    connections = null;
}
/**
 * Closes a database connection.
 *
 * @param conn Database connection to close.
 */
private void closeConnection(Connection conn) {
    try {
        conn.close();
    }
    catch (SQLException e) {
    }
}
/**
 * Sleeps for a specified number of milliseconds.
 *
 *@param mSeconds Number of seconds to sleep.
 */
private void wait(int mSeconds) {
    try {
        Thread.sleep(mSeconds);
```

```
   }
   catch (InterruptedException e) {
   }
}
/**
 * Inner class encapsulating the properties of a pooled
 * connection object. These properties include a JDBC database
 * connection object and a flag indicating whether or not the
 * database object is currently in use (busy).
 */
class PooledConnection {
   Connection connection = null;
   boolean busy = false;
   public PooledConnection(Connection connection) {
      this.connection = connection;
   }
   public Connection getConnection() {
      return connection;
   }
   public void setConnection(Connection connection){
      this.connection = connection;
   }
   public boolean isBusy() {
      return busy;
   }
   public void setBusy(boolean busy) {
      this.busy = busy;
   }
}
}
```

Bibliografie

[1] M. Pickavet, B. Dhoedt, Algoritmisch denken en programmeren, Universiteit Gent, INTEC, 2001.

[2] M. Campione, K. Walrath, A. Huml, The Java Tutorial, Third Edition, A Short Course on the Basics, Addison-Wesley, 2001.

[3] S. Bodoff, D. Green, K. Haase, E. Jendrock, M. Pawlan, B. Stearns, The J2EE Tutorial, Addison-Wesley, 2002.

[4] The Source for Java Technology
http://java.sun.com

[5] T. Thai, H. Lam, .NET Framework Essentials, O'Reilly & Associates, 2002.

[6] J. Farley, Microsoft .NET vs. J2EE: How Do They Stack Up? O'Reilly & Associates, 2000.

[7] P. Zadrozny, P. Aston, T. Osborne, J2EE Performance Tuning With BEA WebLogic Server, Expert Press, 2002.

[8] Scalability and Performance evaluation of Microsoft SQL Server access through the use of Distributed Java Technologies, Universiteit Gent, INTEC, 2002.

[9] J.M. Genender, Enterprise Java Servlets, Addison-Wesley, 2002.

[10] G. Barish, Building Scalable and High-Performance Java Web Applications Using J2EE Technology, Addison-Wesley, 2002.

[11] Dive into connection pooling with J2EE
http://www.javaworld.com/javaworld/jw-10-2000/jw-1027-pool.html

[12] The Tomcat 4 Servlet/JSP Container
http://jakarta.apache.org/tomcat/tomcat-4.1-doc/jndi-datasource-examples-howto.html

[13] D.R. Callaway, Database Access with JDBC, Addison-Wesley, 2001.

[14] jBoss :: Professsional Open Source
http://www.jboss.org/index.html

[15] R. Macdonald, Essentials - ADO.NET, DNJ Online, 2001.

[16] J. Goodyear, Use DataReader or DataSet? Visual Studio Magazine, 2003.

[17] eInfoDesigns - dbProvider
http://www.einfodesigns.com/products.aspx

[18] SourceForge - Managed Provider for MySql
http://sourceforge.net/projects/mysqlnet

[19] JSQLConnect Compared to Microsoft JDBC
http://www.j-netdirect.com/JSQLConnect/JSQLvsMicrosoftJDBC.html

[20] M. Driver, .NET vs. Java - Competition or Cooperation? Gartner Symposium IT-Expo, Florence, Italy, 8 - 10 April 2002.

Lijst van figuren