Powering financial websites

Het aansturen van grote transactiesystemen
door middel van distributed objects in Java.

“The Holy Grail of computing is to be able to put applications together quickly and cheaply from reusable, maintainable code, preferably written by someone else.”

Samenvatting

Bij het ontwikkelen van backends die door derden gebouwde online transactie systemen zoals websites moeten voorzien van dynamische content, is het van uiterst groot belang dat deze in alle richtingen voldoende schaalbaar zijn en beheersbaar blijven.
Het is dan belangrijk dat niet alleen de verwerkingscapaciteit eindeloos uitgebreid kan worden, maar zeker zo belangrijk is het dat er in de loop der tijd gemakkelijk nieuwe functionaliteit toegevoegd kan worden met een zo laag mogelijke impact op de rest van het systeem.
Uit het verslag zal blijken dat we die aan die eisen konden voldoen door een zeer duidelijke scheiding te maken tussen data-, applicatie- en presentatielaag en gebruik te maken van distributed objects in Java in de applicatielaag.

Inhoudsopgave

SAMENVATTING 3

INHOUDSOPGAVE 4

1 INLEIDING 6

2 ALGEMENE INFORMATIE 7

2.1 OPDRACHTOMSCHRIJVING 7
2.1.1 Inleiding 7
2.1.2 Opdracht 7
2.1.3 Begeleiding 7
2.1.4 Tijdlijnen 8
2.2 BEDRIJFSPROFIEL 9
2.2.1 Algemeen 9
2.2.2 Strategie 9
2.2.3 Organisatieschema 10
2.3 PROBLEEMSTELLING 11
2.4 DOELSTELLING EN AANPAK 12

3 DISTRIBUTED OBJECTS 13

3.1 INLEIDING DISTRIBUTED OBJECTS 13
3.2 BESTAANDE TECHNOLOGIEËN 14
3.2.1 CORBA 14
3.2.2 COM / DCOM 17
3.2.2.1 COM 17
3.2.2.2 DCOM 18
3.2.3 RMI / EJB 19
3.2.3.1 RMI 20
3.2.3.2 EJB 20
3.2.3.3 Interfaces 22
3.2.3.4 Stateless session beans 22
3.2.3.5 Stateful session beans 23
3.2.3.6 Entity beans 24
3.2.4 SOAP 26
3.3 CONCLUSIE, COMPATIBILITY 27

4 EJB CONTAINERS 29

4.1 BESTAANDE CONTAINERS 29
4.1.1 Weblogic 29
4.1.2 Orion 30
4.1.3 Websphere 30
4.2 KEUZE VAN MARKETXS 31
4.2.1 Applicatieserver 31
4.2.2 Webserver 31

5 IMPLEMENTATIE MARKETXS 33

5.1 COMPONENTENVERDELING 33
5.1.1 com.marketxs.quote 34
5.1.1.1 QuoteBean 34
5.1.1.2 SymbolLookupBean 35
5.1.2 com.marketxs.csp 36
5.1.2.1 CSPPoolBean 37

5.1.3 com.marketxs.currency *37*
5.1.3.1 CurrencyBean 37
5.1.4 com.marketxs.profile *38*
5.1.4.1 ProfileBean 39
5.1.4.2 PortfolioBean 39
5.1.4.3 PortfolioContentBean 40
5.1.5 Overige componenten *41*
5.2 CHARTING 41
5.2.1 Functionaliteit *41*
5.2.2 Architectuur *42*

6 CONCLUSIE 45

BIJLAGE 1: VERKLARENDE WOORDENLIJST 46

BIJLAGE 2: MARKETXS HARDWARE ARCHITECTUUR 50

BIJLAGE 3: QUOTE SERVLET VOORBEELD 51

BIJLAGE 4: LITERATUURLIJST 52

1 Inleiding

Dit verslag is geschreven in het kader van mijn afstudeeropdracht bij MarketXS in Amsterdam ter afronding van mijn studie Hogere Informatica aan de Hogeschool Rotterdam.
Mijn werkzaamheden bij MarketXS betroffen het ontwerpen en implementeren van de complete backend engine die in een *business-to-business* opstelling de financiële websites en andere interactieve services van onze klanten moet voorzien van content zoals koersen, nieuws en bedrijfs- en beursinformatie.
In dit verslag zal ik ingaan op de relatief nieuwe en zeer interessante technieken die nodig waren om aan alle eisen te voldoen.
Het verslag is in principe geschreven voor een ieder die enige interesse heeft voor het ontwikkelen van zeer grote, *fail-safe* en bovenal schaalbare oplossingen voor *OLTP* (OnLine Transaction Processing) systemen.
Via deze weg wil ik ook mijn naaste collega's Floris Alkemade en Melchior van Wijlen bedanken voor de grote vrijheid die mij werd geboden tijdens de ontwikkeling.

Erik van Zijst
Student Hogere Informatica, Hogeschool Rotterdam.
29 mei 2000

2 Algemene informatie

2.1 Opdrachtomschrijving

2.1.1 Inleiding

MarketXS heeft zich ten doel gesteld een systeem te ontwikkelen dat een grote verscheidenheid aan financiële content ontvangt, verwerkt en gecontroleerd en gereguleerd beschikbaar stelt aan de websites van klanten.
Om zich te onderscheiden van soortgelijke, bestaande leveranciers, richt MarketXS zich op het aanbieden van een zo groot mogelijke financiële dekking (beginnend met heel Amerika en Europa) met unieke, zelfvergaarde data op een wijze die klanten een veel grotere verwerkings- en integratievrijheid geeft dan concurrenten momenteel kunnen aanbieden.
MarketXS stelt de data daarom beschikbaar via een presentatie-onafhankelijke API (Application Programming Interface) waarop klanten zelf kunnen programmeren. Zo zijn ze niet gebonden aan de toepassing van bijvoorbeeld het World Wide Web alleen, maar kan de data verder verwerkt en geïntegreerd worden in elke vorm van weergave.

2.1.2 Opdracht

De totale afstudeeropdracht bevat de volgende deelopdrachten:

- Het vaststellen van de eisen waaraan het MarketXS-systeem moet voldoen;
- Het scheiden van het systeem in duidelijke, onafhankelijke lagen;
- Het modulair opdelen van de logica in kleine, gespecialiseerde, makkelijk te koppelen software componentjes ten behoeve van de beheersbaarheid;
- Het zoeken naar een techniek om deze modulariteit te realiseren;
- Het vastleggen van een API voor de communicatie tussen deze componenten en de software van de klanten;
- Het kiezen van de juiste software pakketten voor iedere laag (database, applicatie- en webservers);
- Het kiezen van de juiste hardware;
- Het leiden van de bouw van het gehele systeem van database tot webserver inclusief het ontwerpen en implementeren van een complete online chart engine.

2.1.3 Begeleiding

De uitvoering van de omvangrijke opdracht zal worden begeleid door Floris Alkemade met ondersteuning van Melchior van Wijlen.

2.1.4 Tijdlijnen

De opdracht is per 1 december 1999 aangevangen, waarna de deeltaken zodanig ingepland werden, dat MarketXS begin tweede kwartaal 2000 een werkende website heeft waarop de belangrijkste zaken zoals het opvragen van koersen en het bijhouden van aandelenportefeuilles mogelijk is.
Deze website kan dan dienen als bewijs dat het onderliggende systeem voldoet aan de eisen (veilig, schaalbaar en met een gebruiksvriendelijke API) en zal uiteindelijk de showroom van MarketXS zijn tegenover klanten.
Hierna zal energie worden gestoken in het perfectioneren en versnellen van de software modules en implementeren van applicaties om een steeds breder assortiment van financiële gegevens aan te kunnen bieden.

2.2 Bedrijfsprofiel

2.2.1 Algemeen

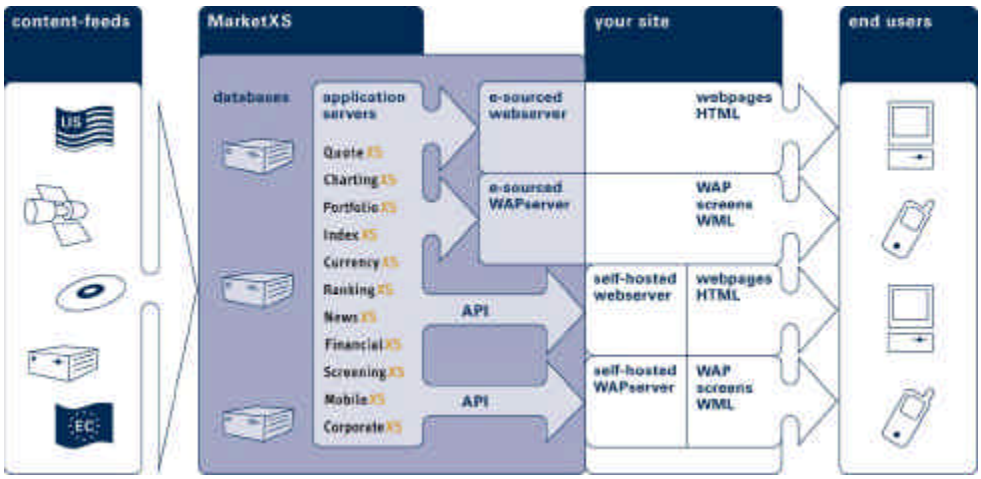
MarketXS is een begin september 1999 in Amsterdam opgericht business to business bedrijf, dat zich richt op het flexibel verstrekken van financiële content aan websites en andere online systemen van klanten.
Hiertoe levert MarketXS een groot aantal software modules die alle financiële markten van Europa en Amerika dekken en tezamen de bouwstenen vormen voor een complete financiële website.

2.2.2 Strategie

Financiële instellingen kunnen met de modules van MarketXS hun service eenvoudig uitbreiden via een eigen financiële website, terwijl internet ondernemingen met de modules hun dienstenpakket kunnen uitbreiden.

De implementatie tijd voor een klant is minimaal, omdat de modules bij MarketXS draaien en ook door MarketXS worden onderhouden, bijgewerkt en uitgebreid. De klant hoeft dus niets extra te hosten of te installeren en kan rechtstreeks vanuit een webserver communiceren met de modules.

Dit uitbesteden van modulaire internet toepassingen wordt in Amerika ook wel e-sourcing genoemd. De meeste Amerikaanse top 10 Internet brokers e-sourcen hun applicaties en bijbehorende content.



Figuur 1: e-souring via MarketXS. Duidelijk is de beperkte hoeveelheid werk voor een klant (reseller) om financiële gegevens weer te geven. De enige verantwoordelijkheid die voor een klant overblijft is het presenteren van de gegevens in HTML.

Omdat de modules alleen maar ruwe gegevens leveren die verder verwerkt worden in de applicatie van de klant, heeft de toepassing van e-sourcing geen gevolgen voor de vormgeving.

E-sourcing door MarketXS biedt belangrijke voordelen:

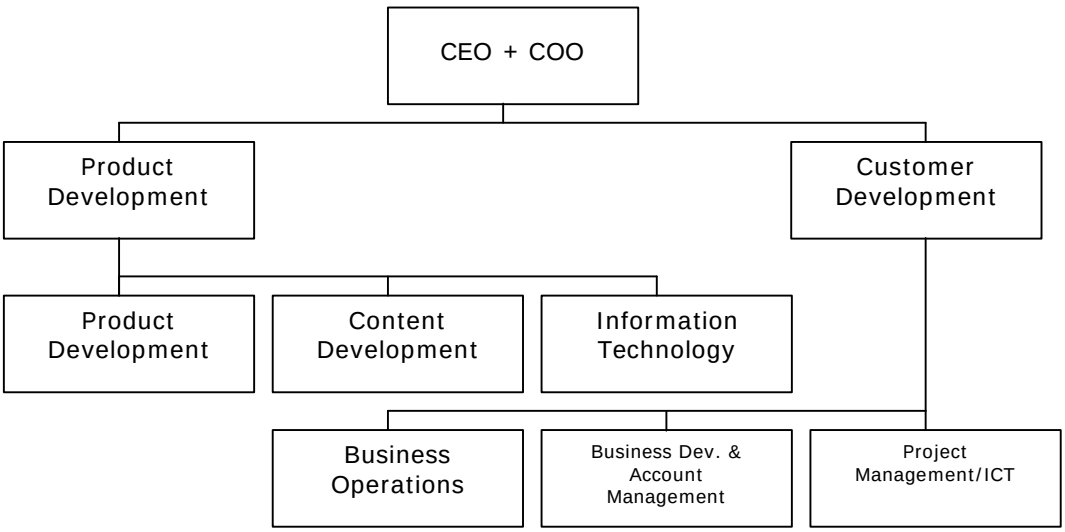
- Snelheid: nadat de modules voor de klant geconfigureerd zijn, zijn ze direct beschikbaar;
- Overzichtelijkheid: de modulaire plug-and-play aanpak levert een overzichtelijk systeem op;
- Schaalbaarheid: omdat de modules bij MarketXS draaien, hoeft de klant zich geen zorgen te maken over verwerkingskracht;
- Laatste technologie: MarketXS vernieuwt de modules voortdurend;
- Cost-efficiency: zowel de initiële als de operationele kosten beperkt.

Centraal in de e-sourcing aanpak van MarketXS staan de ontwikkelde modules. Deze bieden een uitgebreid aanbod van beurskoersen, nieuws en financiële informatie van ondernemingen.

MarketXS levert de volgende modules:

- QuoteXS koersinformatie van Europese en Amerikaanse effecten
- ChartingXS grafische mogelijkheden om het koersverloop weer te geven
- PortfolioXS aanmaken en volgen van persoonlijke effectenportefeuilles
- IndexXS weergave van internationale indices
- CurrencyXS valutaconversie
- RankingXS weergave van de meest actieve of fluctuerende effecten
- NewsXS internationaal bedrijfs- en marktnieuws
- FinancialXS financiële informatie van beursgenoteerde ondernemingen
- ScreeningXS selecteren van effecten op basis van eigen criteria
- MobileXS specifieke toepassingen voor WAP en SMS
- CorporateXS alle beschikbare informatie over één onderneming

2.2.3 Organisatieschema



Figuur 2: Organisatieschema van MarketXS.com.

2.3 Probleemstelling

MarketXS heeft zich ten doel gesteld een *business-to-business* backend op te bouwen dat financiële informatie in ruwe vorm op een eenduidige manier op applicatie niveau beschikbaar maakt voor online applicaties van klanten.

Dit wil zeggen dat applicaties van klanten, zoals de engine van een website, communiceren met de MarketXS backend (die ook op een locatie van MarketXS draait, in tegenstelling tot de website van de klant) en hier gegevens uit kunnen halen.

Concreet kan de probleemstelling worden omschreven in de volgende punten:

- Het zoeken naar een techniek om deze modulariteit te realiseren;
- Het scheiden van het systeem in duidelijke, onafhankelijke lagen;
- Het modulair opdelen van de logica in kleine, gespecialiseerde, makkelijk te koppelen software componentjes ten behoeve van de beheersbaarheid;
- Het vastleggen van een API voor de communicatie tussen deze componenten en de software van de klanten;

Hiernaast zijn enkele andere punten van belang om het geheel te kunnen uitvoeren:

- Het kiezen van de juiste software pakketten voor iedere laag (database, applicatie- en webservers);
- Het kiezen van de juiste hardware;
- Het leiden van de bouw van het gehele systeem van database tot webserver inclusief het ontwerpen en implementeren van een complete online chart engine.

2.4 Doelstelling en aanpak

De doelstelling van het project is een schaalbaar systeem te ontwikkelen dat financiële gegevens ruw (dus zonder enige wijze van formattering zoals bijvoorbeeld HTML) beschikbaar stelt en waarop meerdere *OLTP* (On-Line Transaction Processing) systemen zoals websites van verschillende klanten kunnen worden aangesloten.

Om dit te realiseren heb ik eerst het totale systeem verdeeld in meerdere lagen (tiers) zoals datalaag, applicatielaag en presentatielaag en gekeken naar een techniek die de middelste laag (met de eigenlijke logica) zeer modulair kan houden en erg schaalbaar is. Toen duidelijk werd dat een dergelijk probleem met distributed objects moet worden aangepakt, heb ik literatuuronderzoek gedaan naar de verschillende technologieën die hiervoor ontwikkeld zijn.

Na het maken van de keuze voor Enterprise Java Beans (EJB) heb ik alle modules die MarketXS zal aanbieden beschreven in één of meerdere beans en een API opgesteld waarop klanten gemakkelijk kunnen programmeren vanuit hun applicaties.
Het opstellen en vastleggen van de API is in nauw overleg gebeurd met mijn collega's

De laatste stap van de aanpak bestond uit het implementeren van de beans, het aansluiten ervan op alle heterogene datastromen (actieve satelliet verbindingen, passieve FTP bronnen, databases, etc) en het ontwikkelen van een eigen website die gebruik zal maken van de beans.

3 Distributed objects

Het ontwikkelen van software door middel van het aan elkaar koppelen van bestaande, veelal aangekochte software componenten is relatief nieuw.
Veel grote, stabiele en geaccepteerde systemen zoals database pakketten en besturingssystemen zijn als een monolithisch geheel gebouwd en ontwikkeld door een bedrijf. Nu is de hele computer- en softwaremarkt natuurlijk in zijn totaliteit behoorlijk nieuw en als we deze industrie vergelijken met andere, oudere takken, is het waarschijnlijk dat er een duidelijke verschuiving komt in de manier waarop software pakketten ontwikkeld gaan worden.
Waar software bedrijven nu vaak hun hele product zelf schrijven, zal dit steeds meer veranderen in het combineren van aangekochte componenten tot een commercieel, verkoopbaar eindproduct.

Deze ontwikkeling is te vergelijken met de veel oudere automobiel industrie die inmiddels meer dan 100 jaar evolutie heeft doorgemaakt.
Aanvankelijk was het zo dat auto producenten alle onderdelen zelf maakten. Dit veranderde langzaam door het gebruik van simpele, kleine, aangekochte onderdelen als standaard bouten en moeren. Hierbij is het belangrijk op te merken dat het succesvol gebruik van deze onderdelen mogelijk werd doordat ze gestandaardiseerd werden.

Het gebruik van gestandaardiseerde, vervangbare en combineerbare componenten kent verschillende duidelijke voordelen voor alle industrietakken.
In de eerste plaats betekent het op grote schaal gebruik van standaard componenten een beperking van grote monopolies.
Daarnaast zorgt de gespecialiseerde componentenmarkt voor betere kwaliteit en een lagere prijs. Component producenten zijn zeer gespecialiseerd en leveren zo een veel hogere kwaliteit af.

3.1 Inleiding distributed objects

Distributed object computing brengt de toepassing van software componenten naar een hoger niveau. Hierbij kunnen losse componenten draaien op verschillende machines in een heterogeen netwerk. Bij elkaar vormen deze delen dan een volledige applicatie.
In een dergelijk systeem is (gelijk de slogan van Sun) het netwerk de computer. Object oriëntatie kan deze vorm van ontwikkelen drastisch vereenvoudigen. De componenten worden geïmplementeerd in een object georiënteerde programmeertaal en draaien in hun eigen dynamische omgeving buiten een applicatie en worden over het netwerk aangeboden aan externe applicaties. Voor de applicatie programmeur lijken de componenten op normale wijze deel uit te maken van het gehele programma.
Dit is de essentie van zogenaamde plug-and-play software.

Deze vorm van programmeren kent enkele belangrijke voordelen die voornamelijk van pas komen bij de ontwikkeling van zeer grote systemen.

- Zo kunnen bepaalde taken ondergebracht worden in modules en geïnstalleerd worden op systemen die gespecialiseerd zijn in het uitvoeren van deze taken.

- De componentaanpak zorgt voor een hoge mate van overzichtelijkheid en modulariteit.
- Een ander voordeel is dat de componenten allemaal lokaal lijken te zijn en het op applicatieniveau helemaal niet duidelijk hoeft te zijn waar een component zich bevindt, op wat voor architectuur het draait of in welke programmeertaal het is geschreven. Dit maakt het geheel zeer flexibel.
- Tenslotte is deze manier van modulair programmeren uitermate schaalbaar. Een applicatie kan in eindeloos veel specifieke delen worden verdeeld, en op evenzoveel systemen draaien. Door het bijplaatsen van meerdere computers op het netwerk kan eenvoudig meer verwerkingskracht worden toegevoegd.

Het toepassen van distributed computing kent uiteraard ook nadelen die voortkomen uit het opdelen van een systeem in gescheiden componenten.

- Zo zal blijken dat componenten die onderling veel met elkaar communiceren dichter bij elkaar moeten staan om efficiënt te kunnen werken. Het werken over een netwerk introduceert immers vertraging.
- Sommige componenten kunnen alleen draaien op speciale machines op speciale locaties
- Kleinere, gespecialiseerde componenten komen de flexibiliteit van het geheel ten goede, maar veroorzaken in totaal ook meer netwerkverkeer.
- Grotere componenten verminderen netwerkbottlenecks, maar verlagen de flexibiliteit van het geheel.

3.2 Bestaande technologieën

3.2.1 CORBA

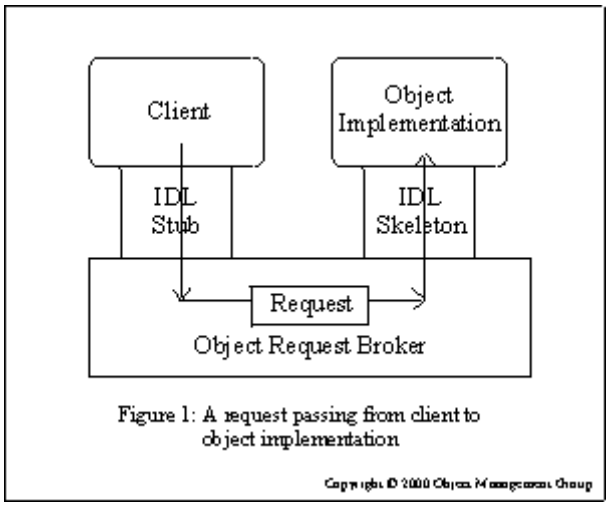
CORBA, Common Object Request Broker Architecture, is een initiatief van OMG (Object Management Group). OMG is een consortium van software producenten en gebruikersgroepen. Het doel is een standaard te ontwikkelen voor distributed programming, die vendor-, programmeertaal- en netwerkonafhankelijk is, wat de compatibiliteit tussen verschillende implementaties moet verbeteren en moet voorkomen dat de ontwikkeling van distributed programming uitkomt op een aantal verschillende en incompatible implementaties van verschillende fabrikanten.

Bij CORBA applicaties gaat het aanroepen van methods uit distributed objects via zogenaamde *stubs* en *skeletons*.

Om objecten in verschillende programmeertalen met elkaar te kunnen laten samenwerken, worden de interfaces van de objecten strikt beschreven *in OMG IDL* (Interface Definition Language).
De skeletons van de objecten worden geadverteerd bij zogenaamde *ORB*'s (Object Request Broker). Deze *ORB* weet dus wat voor objecten er zijn, waar ze zich bevinden en welke functionaliteit ze bevatten.
De eigenlijke implementatie van het object is ingekapseld in de *IDL skeleton*.

De client, het programma dat de functionaliteit van het remote object wil gebruiken, gebruikt de *IDL stub* om tegenaan te praten. Dit kan gewoon in

dezelfde programmeertaal. Het gedrag van de stub garandeert dat het remote object deel uitmaakt van het lokale client programma.
De programmeur hoeft dan ook niet op de hoogte te zijn van de locatie waar de objecten draaien of zelfs maar de programmeertaal waarin het geschreven is. De communicatie tussen client en remote object wordt transparant verzorgd door de ORB.
De IDL stub fungeert dus als lokaal proxy object.

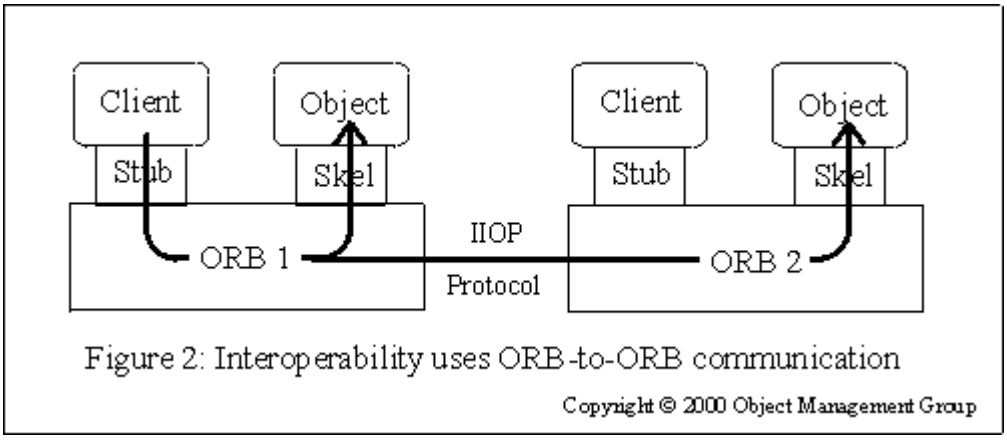


Figuur 3: Communicatie tussen client en server object via stubs en skeletons in CORBA.

Als client en object op verschillende servers draaien, vindt de communicatie plaats tussen de ORB's op beide machines. Het protocol dat hiervoor wordt gebruikt is in de meeste gevallen *IIOP* (Internet InterOrb Protocol).

Hoewel veel ORB implementaties ook andere protocollen ondersteunen, is de ondersteuning van IIOP verplicht om volledig aan de COBRA specificatie te voldoen. Hiermee wordt gegarandeerd dat producten van verschillende fabrikanten compatible blijven met elkaar.

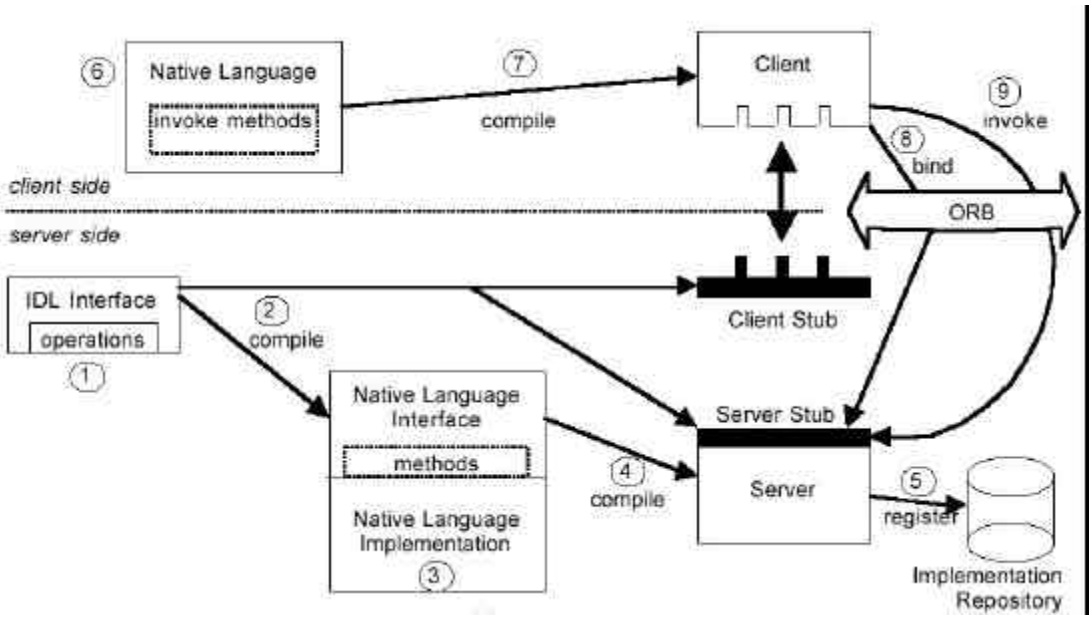
Het ontwikkelen van een CORBA applicatie begint bij het definiëren van de interface van het server object. Dit gebeurt in de programmeertaal onafhankelijke IDL taal. Met IDL wordt alleen de interface beschreven. Het uiteindelijke server



Figuur 4: Als client en server object gescheiden zijn door een netwerk, vindt de communicatie tussen de ORB's plaats over IIOP.

object wordt er niet in geprogrammeerd. Dit gebeurt met een bestaande taal. Bijvoorbeeld C++.

Daarna wordt de IDL interface compiled met een speciale IDL compiler.
Dit levert een server skeleton en een client stub op.
De server skeleton wordt vervolgens *meegelinked* in het server object en de client stub wordt al of niet dynamisch gelinkt in de client applicatie.



Figuur 5: Het hele CORBA proces in beeld gebracht, van implementatie tot toepassing ervan.

Het gehele proces verloopt volgens de volgende stappen.

1. Definieer de server interface in IDL.
2. Compile de IDL code met de IDL compiler. Dit levert een client en een server server stub op. Beide kunnen voor andere programmeertalen geschikt zijn. Optioneel kan de IDL compiler meteen een gecompilde interface beschrijving naar de implementation repository schrijven.
3. Implementeer de server.
4. Compile de server en link de server stub mee. Het resultaat is een uitvoerbaar server programma dat functie-aanroepen via CORBA kan aannemen.
5. Adverteer de server in de implementation repository met server naam, uitvoer commando en uitvoer en toegangs permissie. De server is nu beschikbaar.
6. Implementeer de client. De server methods worden gebruikt alsof ze lokaal beschikbaar en aanwezig zijn. Hiervoor wordt dus gewoon de syntaxis van de lokale programmeertaal gebruikt.
7. Compile de client en link deze met de client stub.
8. Als de client draait, wordt de ORB gebruikt om een server object te lokaliseren en een referentie ernaar te verkrijgen.
9. Via deze referentie kan de client de server methods uitvoeren.

Opgemerkt moet wel worden dat de werkwijze van OMG ook significante nadelen kent. Doordat de standaard zo open is, kunnen COBRA producten gebruik maken van eigen toevoegingen. Het is vaak zelfs noodzakelijk, omdat de de specificatie niet alle situaties dekt.
In de praktijk levert dit vaak incompatible producten op, waarbij communicatie via IIOP wel mogelijk is, maar niet alle geavanceerde functionaliteiten gebruikt kunnen worden. COBRA kan ook desondanks toch een goede oplossing zijn, als er overal gebruik wordt gemaakt van de producten van een producent.

3.2.2 COM / DCOM

Microsoft heeft zijn eigen componenten systemen om modulair programmeren mogelijk te maken. Voor het aan elkaar knopen van software componenten heeft Microsoft COM (Component Object Model).

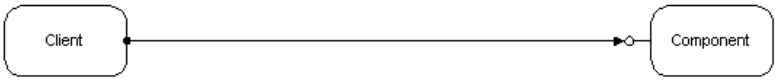
3.2.2.1 COM

Een COM object kan geschreven worden in vrijwel elke programmeertaal (C++, Delphi, Java, Visual Basic, VBScript etc). COM beschrijft vervolgens de specificatie waaraan de objecten moeten voldoen en levert een library (de COM library) die onderdeel is van het win32 besturingssysteem voor communicatie.
COM is volledig object georiënteerd

Voorbeelden waar COM voor gebruikt wordt in het win32 systeem zijn onder andere ActiveX controls, OLE (Object Linking and Embedding) en *drag-and-drop* onder Windows.

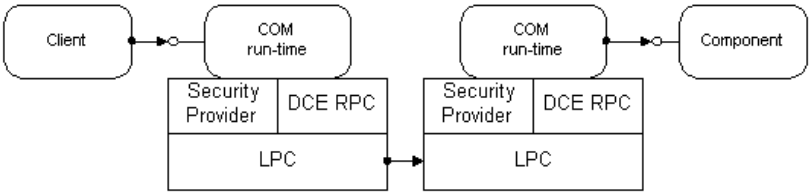
COM is dus een specificatie die de interface beschrijft waaraan objecten moeten voldoen, zodat ze elkaar onderling kunnen aanroepen en gebruiken. Windows zelf levert de library die de communicatie mogelijk maakt.

Communicatie tussen client applicatie en een object kan op verschillende niveaus plaatsvinden. De eenvoudigste manier is wanneer een client applicatie een COM rechtstreeks binnen hetzelfde proces gebruikt.
In dat geval kan de client het object direct, zonder overhead gebruiken.



Figuur 6: Tussen twee COM-componenten binnen hetzelfde proces vindt directe, in-memory, communicatie plaats.

Wanneer client en component zich in een ander proces bevinden, is er geen sprake meer van directe, *in-memory* communicatie. De COM interface zorgt dan voor een transparante LPC (Local Process Communication) laag.

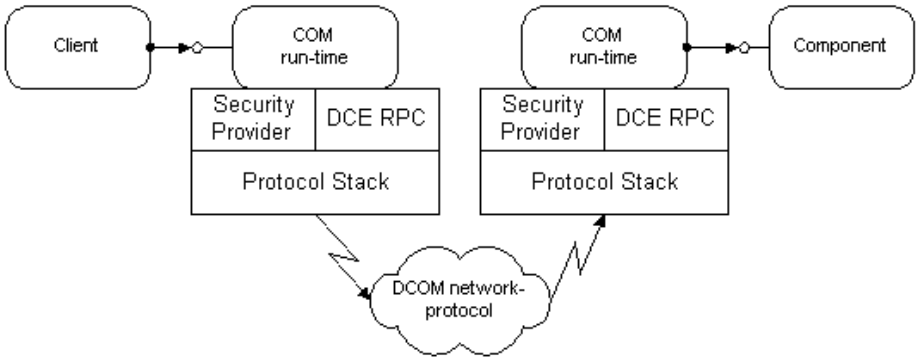


Figuur 7: Tussen twee COM-componenten in verschillende processen, verzorgt COM de communicatie transparant via LPC (Local Process Communication).

Deze brug heeft geen gevolgen voor de manier waarop client en component worden geïmplementeerd.

3.2.2.2 DCOM

DCOM (Distributed Component Object Model) voegt aan het bestaande COM systeem nog een laag toe, namelijk een netwerklaag. Zo wordt het mogelijk om client en component gescheiden te hebben over een netwerk.

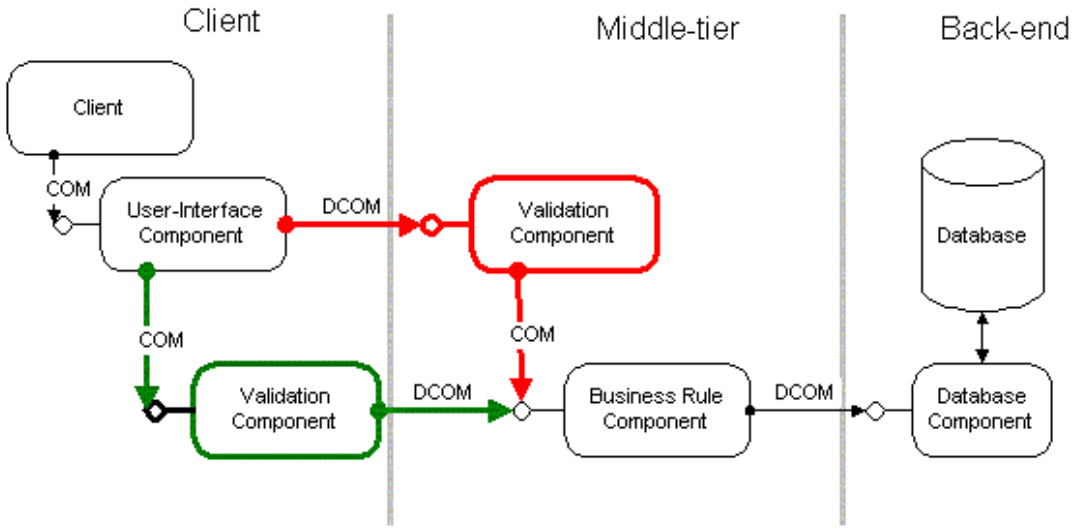


Figuur 8: DCOM zorgt voor transparante netwerkcommunicatie tussen COM-componenten die zich op verschillende machines bevinden.

Waar in het traditionele COM model de functie aanroepen via LPC (Local Process Communication) worden verstuurd, geeft COM ze nu door aan de DCOM laag. DCOM verstuurt ze vervolgens over het netwerk via het DCOM protocol naar de andere machine waar het gehele proces in omgekeerde volgorde plaatsvindt. Voor de implementatie van client en component is er geen verschil. Deze maken enkel gebruik van de COM interface die aan hun omgeving grenst. COM/DCOM verzorgt automatisch de transparant verbindingen. Of ze zich nu naast elkaar in het geheugen van hetzelfde proces bevinden, ieder in een eigen proces draaien, of zelfs in processen draaien op verschillende machines, op kilometers afstand.

De netwerktransparantie maakt het mogelijk de componenten zo te verdelen dat ze optimaal tot hun recht komen. Een voorbeeld is een situatie waarin een component dat autorisatie moet verrichten deployed kan worden op de machine

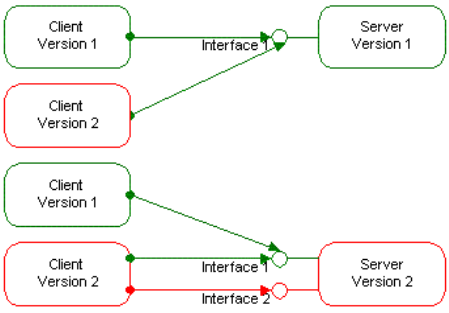
van de client in het geval dat de netwerk bandbreedte kritisch is, of verplaatst kan worden naar de server indien er genoeg bandbreedte is.



Figuur 9: Als bandbreedte een bottleneck vormt, kunnen componenten zo geplaatst worden, dat hier optimaal gebruik van kan worden gemaakt.

Versioning

Omdat gedistribueerde applicaties bestaan uit losse componenten waarvan vaak een deel zelfgemaakt (aanpasbaar) en een deel aangekocht (niet aanpasbaar) is, is het belangrijk dat de componenten onafhankelijk kunnen worden ge-upgrade. Hiervoor kent DCOM een flexibel versioning systeem waarbij een component een ander component kan queryen op beschikbare functies. Een component kan hierdoor een andere functionaliteit tonen aan clients met verschillende versie nummers. Terwijl een component uitgebreid is en dus een andere interface heeft met uitgebreidere functionaliteit, kan het zijn dat er in het netwerk nog clients draaien die de originele interface verwachten. Het server object zal aan deze clients dan de oude interface tonen (dit is mogelijk omdat de nieuwe interface enkel uitbreidingen bevat, maar ook nog de oorspronkelijke functies heeft).



Figuur 10: Versioning in DCOM maakt het mogelijk clients en servers onafhankelijk te upgraden.

Andersom kan ook, waarbij een nieuwe client gebruik maakt van een oud server object en hierbij genoeg heeft aan de oude interface.

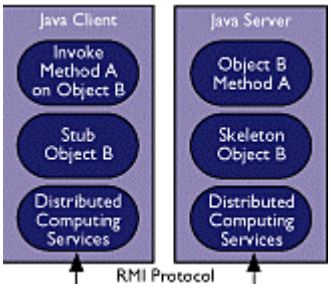
3.2.3 RMI / EJB

Sun Microsystems heeft voor Java een eigen systeem ontwikkeld om distributed computing mogelijk te maken: RMI (Remote Method Invocation).
Het RMI model heeft veel weg van de manier waarop voorgaande technieken (CORBA / DCOM) gedistribueerde uitvoering regelen.

EJB is een specificatie die het gebruik van componenten uitbreidt met enterprise services waarmee een volledige *middletier* gebouwd kan worden.

3.2.3.1 RMI

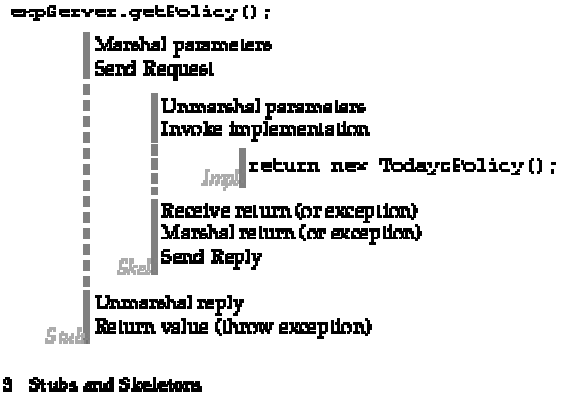
Het RMI systeem bestaat uit drie lagen: de stub/skeleton laag, de remote-reference laag en de transport laag.



Figuur 11: Het RMI model van boven naar beneden.

RMI is een laag bovenop de Java Virtual Machine en heeft automatisch de beschikking over de Java functies zoals *garbage collection*, security en class-loader mechanismen. De applicatielaag ligt op RMI. Wanneer een client object een funtie aanroept van een remote server object, reist deze functie-aanroep aan de client kant door de lagen naar beneden richting de transport laag. Vanaf hier stuurt Java het indien nodig over het netwerk naar de virtual machine aan de server kant. Hier vindt het proces in omgekeerde volgorde plaats.

Wanneer een client communiceert met een remote object, communiceert het in werkelijkheid met een stub of proxy object aan de client kant. Een proxy object is de implementatie van de remote interface van het server object die de functie-aanroepen doorstuurt naar de remote-reference laag.



Figuur 12: Het verloop van een functie aanroep via stubs en skeletons.

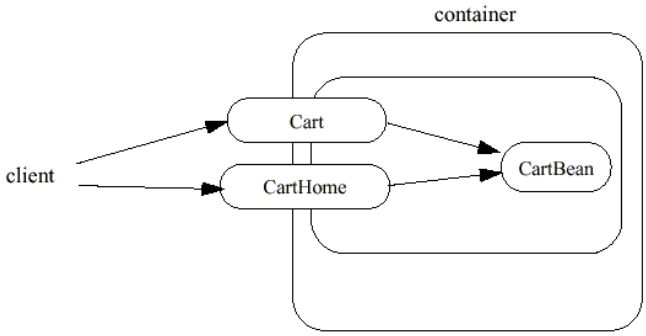
Omdat Java en dus ook RMI volledig object georiënteerd is en elk object dat de *serialization* interface implementeert verstuurd kan worden, is er in Java geen beperking wat betreft return type en argumenten. Beide kunnen complexe, zelfgemaakte objecten zijn.

3.2.3.2 EJB

EJB (Enterprise Java Beans) is een voortzetting op Javabeans in combinatie met RMI. Waar Javabeans het principe van software componenten introduceerde in Java, beschrijft EJB een specificatie om componenten over een netwerk met elkaar en clients te laten communiceren.
Bij EJB worden de componenten in Java geschreven en in een applicatie server (EJB container) geladen. De communicatie met externe clients gaat over het RMI protocol (Remote Method Invocation). Hiervoor worden interfaces gedefinieerd.

Deze beschrijven de functionaliteiten van de bean en de mogelijkheden om een instantie aan te kunnen maken of instanties op te zoeken.

Vervolgens worden de interfaces beschreven in Java.



Figuur 13: Een EJB (Enterprise Java Bean) wordt in een container geladen. De interfaces verzorgen de communicatie.

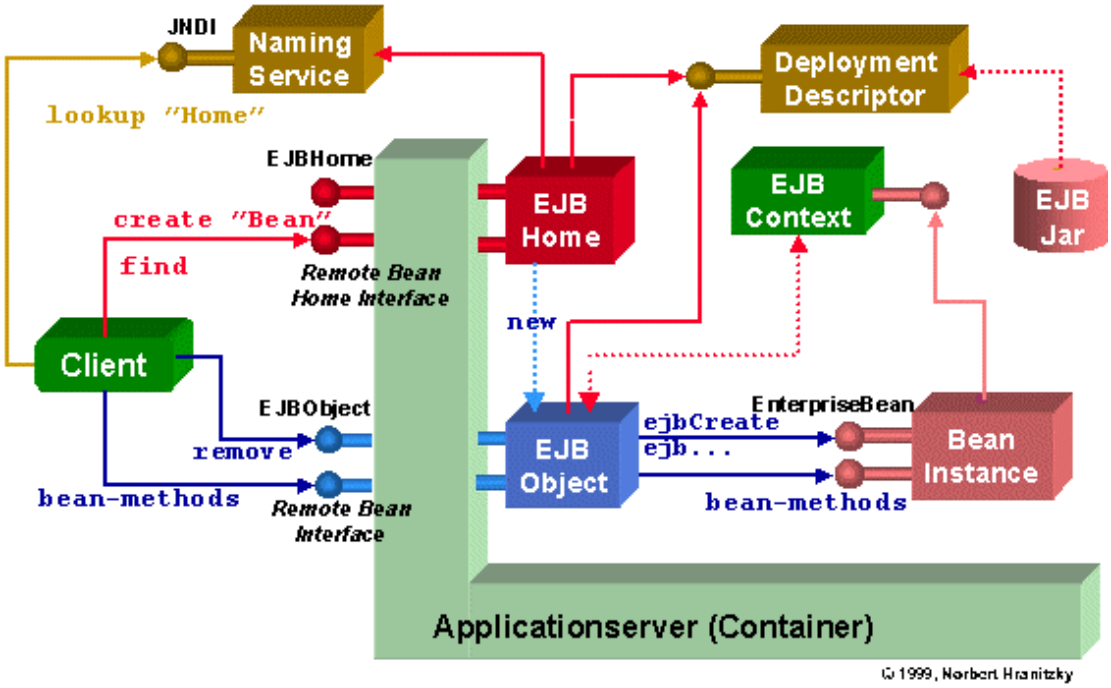
Hierin wordt onder andere genoemd welke functionaliteit de bean bevat en hoe het object geïnstantieerd kan worden.
Deze interfaces, die nodig zijn om de uiteindelijke communicatie over RMI (Remote Method Invocation) mogelijk te maken, worden bij de bean gevoegd en stellen de container (applicatie server) in staat de bean beschikbaar te stellen aan andere beans of externe applicaties.

De toegevoegde waarde die EJB levert over de normale toepassing van RMI, is dat EJB's geladen worden in een EJB container die voor allerlei extra services zorgt. Zo zal een EJB container gepoolde database connecties aanbieden, transacties services en vaak ook clustering van meerdere applicatie servers onderling tot een groter geheel.

Verder maakt EJB het gemakkelijker gedistribueerde componenten te bouwen. EJB's kunnen door de programmeur beschouwd worden als single threaded. De container zorgt voor het aanmaken van extra instanties die parallel door andere clients gebruikt kunnen worden. Threads lopen dus niet door dezelfde code, waardoor ingewikkelde synchronisatie van data niet nodig is terwijl het component wel in staat is in meerdere transacties tegelijk actief te zijn.

Verder is valt het instantiëren en vernietigen van instanties onder de verantwoordelijkheid van de container. Zo zal een container een instantie na gebruik bijvoorbeeld niet direct afbreken als er geen tekort aan geheugen is. Zo kan de container besluiten een instantie die niet gebonden is door een client, toch in het geheugen te houden. Wanneer er nu een client komt die om een instantie van hetzelfde object vraagt, hoeft de container enkel een reference ernaar terug te geven. Dit komt de performance erg ten goede.

EJB kent drie verschillende soorten enterprise beans. Beans zonder persistence, waarvan alle instanties gelijk zijn aan elkaar, beans met persistence, waarin gegevens tijdens transacties bewaard kunnen blijven en beans met persistence die een extern dataveld in bijvoorbeeld een database vertegenwoordigen.



Figuur 14: De hele EJB architectuur in beeld. Links de client, rechts alle delen van een EJB, deployed in de container.

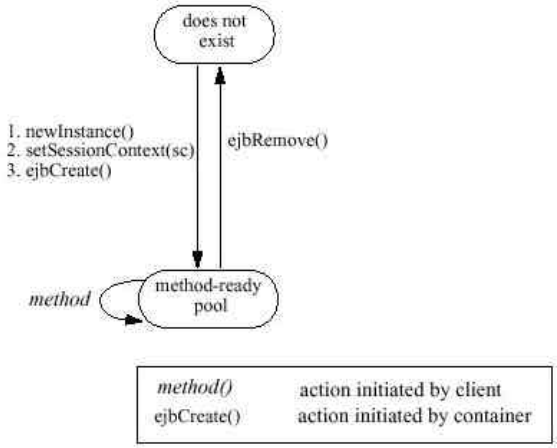
3.2.3.3 Interfaces

Een EJB heeft twee interfaces. Een zogenaamde *home interface* en een *remote interface*. Deze interfaces zijn de interfaces zoals elk RMI object ze heeft.
De *home interface* beschrijft functies voor het verkrijgen van een of meer references naar een bean. Een reference kan verkregen worden door bijvoorbeeld het instantiëren van een nieuwe instance of het opzoeken van een bestaande instance.
De *remote interface* beschrijft de functies die de bean heeft. Na het verkrijgen van een reference, kunnen hierop de methods uit de *remote interface* worden aangeroepen.
Het opzoeken van beans door een client gaat via de zogenaamde JNDI (Java Naming and Directory Interface), een bibliotheek waarin de de beans worden opgenomen. Een Java client kan direct vanuit de applicatie een verbinding maken met deze directory.

De communicatie tussen client en server regelt RMI via het JRMP (Java Remote Method Protocol) protocol.

3.2.3.4 Stateless session beans

De meest eenvoudige enterprise bean is de stateless session bean. Deze bean heeft enkel functionaliteit en is niet bedoeld voor het vasthouden van transactie gegevens tussen aanroepen.
De container behandelt deze beans dan ook als zodanig. Een bean is na gebruik hetzelfde als ervoor. Een container kan instanties dus elke keer aan een andere



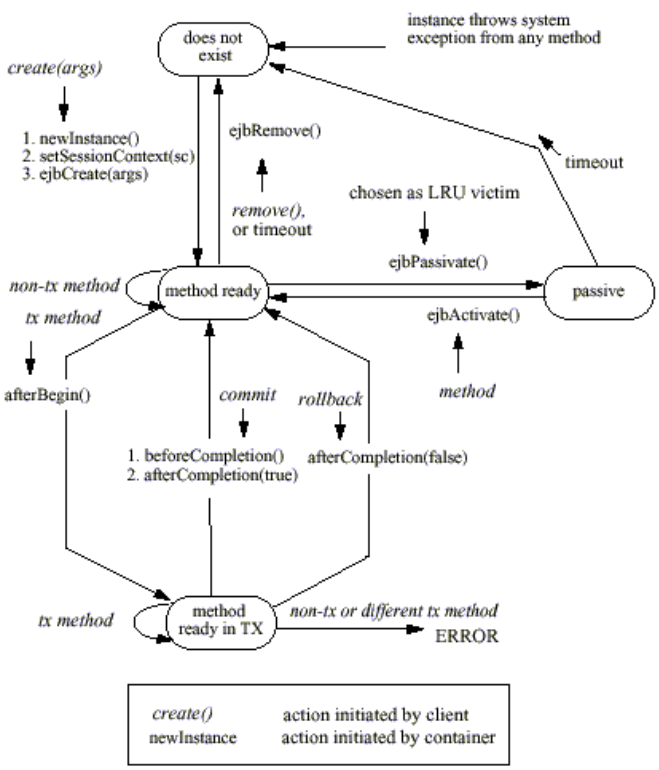
Figuur 15: Lifecycle van een stateless session bean.

client aanbieden zonder een verse instantie aan te maken. Tussen transacties in komen de instanties in een pool, klaar voor hergebruik.

Een client kan dus in verschillende transacties iedere keer een andere instantie krijgen. Er is geen manier voor de client een specifieke instantie terug te krijgen.

Dit maakt deze dat deze bean de minste resources vergt van het systeem en het snelst is in gebruik. Hierdoor wordt het totaal beter schaalbaar, maar introduceert ook meer complexiteit dan bij het gebruik van stateful beans.

3.2.3.5 Stateful session beans



Figuur 16: Lifecycle van een stateful session bean.

Session beans kunnen ook een *conversational state* bevatten. Dan zijn *het stateful session beans*.
Een instance van een stateful session bean kan tussen transacties elke keer gebonden worden door dezelfde client. Zelfs wanneer een client afgesloten wordt en op een later tijdstip op een andere computer weer wordt opgestart, kan de client de originele instance weer terug opvragen.
Dit gedrag maakt de stateful session bean geschikt voor gebruik als bijvoorbeeld een “shopping card” op een e-commerce site. Tussen de pageviews door kunnen de artikelen per klant opgeslagen worden in een stateful bean instance.

De container houdt deze instances dus in leven, maar moet er wel voor zorgen dat er op deze manier niet een hele grote hoeveelheid geheugen gealloceerd blijft aan instanties die vrijwel nooit actief zijn.
Om dit te voorkomen kent EJB *passivation*. Passivation stelt de container in staat een bean instance op te slaan op disk. Zo kunnen instances tussen transacties

door weggeschreven worden naar disk en blijft het geheugen vrij voor andere processen.
Op een drukke e-commerce site kunnen er gemakkelijk duizenden shopping cards tegelijk bestaan, zeker als een klant zijn shopping card een paar dagen kan laten staan voordat hij besluit de artikelen te kopen.
In dat geval kan de applicatie server overgaan tot het *passivaten* van instances die niet gebonden zijn met een client applicatie. Dit gebeurt over het algemeen door middel van een Less Recently Used algoritme. Hierbij worden de instances op volgorde van idle-time weggeschreven naar disk.

Bij het *passivaten* van een object gebeurt hetzelfde als wanneer het object over RMI over het netwerk verstuurd zou worden. Het object wordt via *serialization* beschreven als een binair stuk tekst. Dit kan gemakkelijk worden weggeschreven. Dit betekent uiteraard wel dat het object voordat het *ge-passivate* wordt alle connecties met de buitenwereld moet afsluiten. Als een object een database connectie open heeft, bijvoorbeeld als het naar disk wordt geschreven, zal de verbinding verbroken worden zodra het object door de *garbage-collector* wordt verwijderd uit het geheugen. Als het later dan weer *ge-activate* wordt, zal het object niet in de gaten hebben dat de reference naar de database opeens niet meer geldig is.

Om een en ander goed te laten verlopen, roept de container voordat het een object *passivate*, eerst de method `ejbPassivate()` aan. De programmeur dient er in de implementatie van deze method zelf zorg voor te dragen dat open resources gesloten worden.
Bij het *activaten*, het terug in memory brengen, roept de container eerst `ejbActivate()` op de bean aan. De programmeur dient er wederom zorg voor te dragen dat verbindingen met externe bronnen worden hersteld. Hierna kan de bean weer gebonden worden.

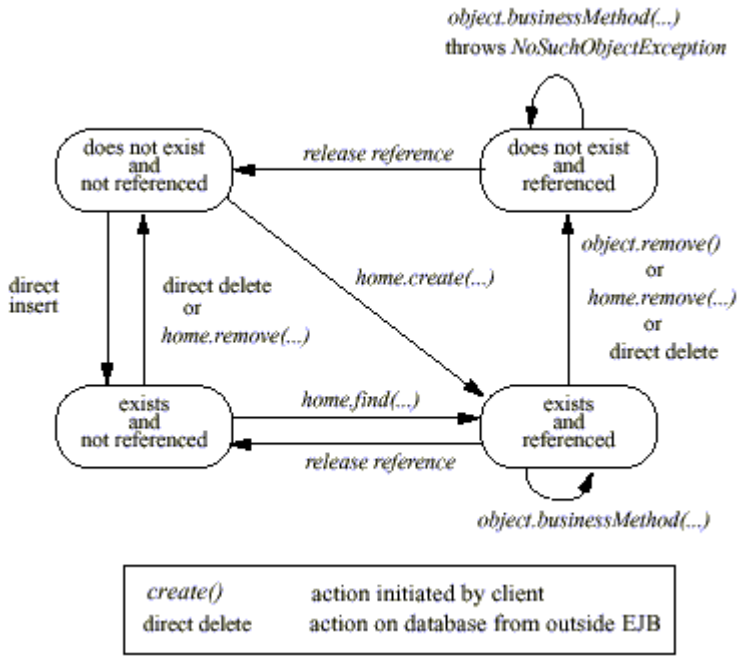
Wanneer een client applicatie een stateful instance later opnieuw nodig heeft, kan het de reference naar de bean opslaan (eventueel op disk). Hiermee kan op een later tijdstip direct de originele instance terug worden opgeroepen. Als de instance *ge-passivate* was, wordt hij automatisch eerst door de container *ge-activate*.

3.2.3.6 Entity beans

De EJB specificatie kent nog een derde enterprise bean, de entity bean.
Een entity bean instance is de representatie van externe persistent data. Een voorbeeld hiervan is een record uit een database tabel. Iedere rij wordt dan gerepresenteerd door een eigen bean instance.
Indien dit wordt toegepast over bijvoorbeeld een tabel met gebruikersprofielen, wordt ieder gebruikersprofiel weergegeven door een unieke instance. Entity beans hebben dus duidelijk een conversational state (iedere instance onderscheidt zich van de andere door unieke interne data) en persistence.

Entity beans leveren een volledige object georiënteerde interface over een relationele database.

Een entity bean (meestal dus een record uit een database) heeft de waarden van de database record in private variabelen die via get- en set-methods uitgelezen en aangepast kunnen worden. Wanneer een veld is aangepast, hoeft de client



Figuur 17: Lifecycle van een entity bean.

applicatie er geen zorg voor te dragen dat de onderliggende database nu ook wordt aangepast. Dit is iets dat volledig automatisch gebeurt door de container of eventueel door de entity bean zelf.

Container Managed Persistence

Voor het synchroniseren met de database kunnen twee tactieken gevolgd worden. Wanneer de container (applicatie server) hier voor zorgt spreken we van *Container Managed Persistence* (CMP). In dit geval dienen de relaties met de database beschreven worden in de *deployment descriptors* die bij de bean zitten. Hierin worden zaken als tabelnaam en velden vermeld die de bean moet representeren. Ook worden SQL queries beschreven waarmee de bean data moet uitlezen en zoeken.

Een voordeel van CMP is dat het ontwikkelen van een object georiënteerde laag over de database erg gemakkelijk is. Het creëren is gemakkelijk te automatiseren, omdat alleen gegevens over de database noodzakelijk zijn. Verder is het bewerken van een database via objecten aanzienlijk gemakkelijker dan expliciet via database queries.

Een nadeel kan performance zijn. Aangezien de synchronisatie in de handen ligt van een geautomatiseerd proces, kan er weinig aan *fine-tuning* worden gedaan. De bean zal voor en na elke transactie overgaan tot het *syncen* met de database. Ook als het read-only toegang betrof en er helemaal geen veranderingen zijn aangebracht. De container is hiervan immers niet op de hoogte.

Wanneer een CMP entity bean wordt gebonden in een transactie, zal de container de inhoud van de bean instance verversen door deze opnieuw uit de database op te halen. Dan wordt de record in de database *gelocked*. Deze zit nu in een transactie.
Wanneer de client de transactie beëindigt (COMMIT of ROLLBACK), zal de container de inhoud van de instance volgens de regels uit de deployment descriptors in de database updaten en de record weer *unlocken*.

Bean Managed Persistence

Wanneer er meer flexibiliteit nodig is in de relatie tot de database, of performance erg belangrijk is, kan het een optie zijn om de synchronisatie zelf te regelen in de bean. We spreken dan van *Bean Managed Persistence* (BMP).
Bij bean managed persistence roept de container voor een transactie nog steeds de method `ejbLoad()` aan op de bean en na de transactie `ejbStore()`, maar de

implementatie hiervan is aan de ontwikkelaar van de bean. Die kan dus zelf besluiten wel of niet iets naar de database te *syncen*.
Ook kunnen nu veel gecompliceerdere database relaties worden gelegd. De logica hiervoor is namelijk geheel aan de programmeur. Relaties over meerdere tabellen, meerdere databases zijn hiervan voorbeelden.
Als nadeel kan natuurlijk wel de grotere complexiteit worden aangerekend.

3.2.4 SOAP

Een nieuwe techniek op het gebied van distributed computing *is Simple Object Access Protocol* (SOAP).
SOAP is een ontwikkeling van Microsoft en past XML in combinatie met HTTP toe voor de overdracht van RPC (*Remote Procedure Call*) informatie. Hierbij worden functie calls en objecten beschreven in XML en als tekst over het HTTP protocol verstuurd.
De motivatie om deze technologieën te gebruiken is een protocol te hebben dat ook buiten het intranet betrouwbaar kan werken. Communicatie over het internet loopt vaak immers door firewalls die veel soorten verkeer tegenhouden. HTTP verkeer, het protocol waarop het gehele WWW is gebaseerd, wordt echter vrijwel altijd en overal doorgelaten. SOAP benut dit door HTTP te gebruiken en via *piggybacking* RPC informatie mee te sturen.
Hierdoor biedt SOAP de mogelijkheid distributed computing mogelijk te maken zonder aanpassingen te maken op de huidige infrastructuur of programmatuur.

SOAP maakt distributed computing mogelijk tussen alle computers op Internet die toegang hebben tot het WWW en komt waar andere protocollen als COBRA en DCOM niet kunnen komen.

SOAP brengt de communicatie tussen applicaties terug tot een erg laag niveau. Ontwikkelaars zijn dicht betrokken bij de definitie van objecten en het beschrijven ervan in XML.
Een voorbeeld van een functie aanroep in SOAP via HTTP kan geïllustreerd worden bij het opvragen van een koersprijs. Dit zou plaats kunnen vinden tussen een client applicatie die draait op de computer van de eindgebruiker, in of naast de webbrowser.

```
POST /StockQuote HTTP/1.1
Host: www.stockquoteserver.com
Content-Type: text/xml; charset="utf-8"
Content-Length: nnnn
SOAPAction: "Some-URI"

<SOAP-ENV:Envelope
  xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
  SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
      <SOAP-ENV:Body>
            <m:GetLastTradePrice xmlns:m="Some-URI">
                  <symbol>RHAT</symbol>
            </m:GetLastTradePrice>
      </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

Figuur 18: Voorbeeld van een functie-aanroep van client naar server via SOAP. De client beschrijft de functie en argumenten in XML.

Omdat de eindgebruiker in staat is een HTTP connectie op te bouwen naar de servers om te browsen, zal de client applicatie ook in staat zijn de functie aanroepen op de servers uit te voeren. Deze gaan immers via hetzelfde protocol.

```
HTTP/1.1 200 OK
Content-Type: text/xml; charset="utf-8"
Content-Length: nnnn

<SOAP-ENV:Envelope
  xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
  SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"/>
      <SOAP-ENV:Body>
            <m:GetLastTradePriceResponse xmlns:m="Some-URI">
                  <Price>34.5</Price>
            </m:GetLastTradePriceResponse>
      </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

Figuur 19: Voorbeeld van een functie return van server naar client via SOAP. De server beschrijft de return waardes XML.

Omdat SOAP alleen de manier van communicatie tussen uiteindelijke server en client beschrijft en niet gebonden is aan een bepaalde programmeertaal kan het gebruikt worden om componenten in verschillende talen aan elkaar te knopen. De enige vereiste wat betreft SOAP, is dat de client en server allebei HTTP begrijpen en de client een browser heeft en de server een webserver.

Of SOAP een echte concurrent gaat worden van bestaande technologieën als CORBA, DCOM en EJB is maar de vraag. De hoeveelheid overhead die de webserver en –client en de toepassing van XML introduceren, is significant.
Het is echter wel waarschijnlijk dat SOAP een belangrijke speler wordt in de communicatie tussen webbrowser en webserver en op deze manier steeds geavanceerdere client toepassingen mogelijk zal maken. Op dit punt kan het de huidige toepassing van Java applets beconcurreren. Client applicaties kunnen dan namelijk veel kleiner worden. Het leeuwendeel van de applicatie kan namelijk op de server blijven en via SOAP benaderd worden.
Nogmaals, een dergelijk gebruik van distributed computing is momenteel ook mogelijk via RMI in Java applets, of via het gebruik van DCOM, maar de protocollen die deze toepassingen gebruiken, werken zelden betrouwbaar over het Internet doordat firewalls roet in het eten gooien.

3.3 Conclusie, compatibility

Voor welke technologie ook gekozen wordt, belangrijk is natuurlijk om te weten hoe de verschillende systemen samen kunnen werken. Dit is voornamelijk belangrijk voor MarketXS, aangezien MarketXS niet alleen intern van distributed computing gebruik maakt, maar het zelfs als interface naar klanten toe aanbiedt. Hierbij is het uiteraard onwaarschijnlijk dat alle klanten gebruik zullen maken van een bepaalde toepassing.

CORBA is het meest platform-, producent- en programmeertaal-onafhankelijk, maar is waarschijnlijk wel de meest gecompliceerde toepassing. Bovendien doet het feit dat iedere producent van ORB's (Object Request Broker) zijn eigen

uitbreidingen en soms zelfs protocollen toevoegt, de compatibility binnen CORBA zelf weinig goeds.

De toepassing van DCOM betekent dat het gehele systeem op Microsoft besturingssystemen moet draaien. DCOM is namelijk wel programmeertaal onafhankelijk, maar niet binary compatible.

EJB werkt via RMI en is per definitie Java-only. Wel lost Java zelf al de problemen van systeem afhankelijkheid op (voor ieder systeem is namelijk wel een virtual machine), maar dit betekent wel dat beide kanten Java moet zijn. Het tegengestelde van de DCOM oplossing.

SOAP heeft niet te kampen met programmeertaal- of systeem-afhankelijkheden en kent geen of weinig protocolbeperkingen vanwege de toepassing van XML, maar wanneer performance ter sprake komt, blijft het duidelijk achter bij de rest.

Gelukkig zijn alle methodes wel aan elkaar te knopen. In het geval van EJB (RMI) met CORBA systemen kan dit door het standaard CORBA protocol IIOP te gebruiken aan beide kanten. RMI kan hiervan gebruik maken en verschillende ORB's zijn beschikbaar.

Java met DCOM is vrij gemakkelijk op te lossen door van een Java component een COM component te maken. Zo kan het aan de ene kant met de rest van het COM systeem communiceren en aan de andere kant vanuit Java RMI gebruiken.

Bij de koppeling tussen CORBA en DCOM kan dezelfde truc worden toegepast door van een CORBA applicatie een COM object te maken.

Mede gezien de wijd verbreide toepassing van Java op servers, de systeem- en fabrikant-onafhankelijkheid en snelle ontwikkeling, heeft MarketXS gekozen voor Enterprise Java Beans om de *middletier* mee te ontwikkelen.

4 EJB containers

Aangezien gekozen is voor de toepassing van Enterprise Java Beans, volgt de keuze van applicatie server (EJB container).

Een EJB container kan gezien worden als een omgeving waarin EJB's worden geladen (deployed).
De container regelt de toegang tot beans, waardoor de programmeur hier geen rekening mee hoeft te houden en zo ingewikkelde threading problemen kan ontlopen. Verder regelt een container de beveiliging van beans door het uitvoeren van zogenaamde *security-roles*. Via de deployment descriptors van een bean kan de persoon die de applicatie server inricht, bepaalde gebruikers (clients) de toegang tot beans, of zelfs specifieke bean-methods ontzeggen. De authenticatie van gebruikers gaat via de container.

De container biedt doorgaans ook toegang tot database resources via *gepoolde* database verbindingen. Hierbij opent de container van tevoren direct een (groot) aantal database verbindingen, zodat deze direct toegankelijk zijn voor EJB's. Na gebruik laat een EJB de verbinding weer los en stopt de container deze terug in de pool. Op deze manier hoeven database verbindingen niet ter plekke geopend te worden, zodat een hoop *overhead* vermeden kan worden.

Belangrijk bij de keuze voor een applicatieserver zijn de mogelijkheden tot clustering. Distributed computing door middel van componenten is erg geschikt voor clustering, dus is het belangrijk dat de applicatieserver dit voldoende ondersteunt.

4.1 Bestaande containers

Om tot een juiste beslissing te komen wat betreft applicatieserver, hebben we een aantal producten bekeken: BEA Weblogic, IBM Websphere en Evermind's Orion.
Alle producten ondersteunen bovengenoemde functies.

4.1.1 Weblogic

Weblogic (www.weblogic.com) is waarschijnlijk de bekendste en meest gebruikte EJB applicatieserver. BEA's oorspronkelijk aangekochte product gaat al een hele tijd mee en heeft lang voorop gelopen qua volledigheid. Een voorbeeld van *proven technology*.
De huidige versie van Weblogic (5.1) ondersteunt EJB, JSP (Java Server Pages), JMS (Java Messaging service), JDBC (Java DataBase Connectivity), XML (Extensible Markup Language) en WML (Wireless Markup Language).

De ondersteuning van JSP en servlets betekent dan ook dat Weblogic een volledige webserver meelevert. Het voordeel van een EJB container en een webserver binnen hetzelfde proces is dat servlets (draaiende JSP pages worden

gecompileerd tot servlets) de EJB's direct in het geheugen kunnen aanspreken en er zo geen extra protocol tussen hoeft te draaien.

MarketXS zal echter geen gebruik gaan maken van de webserver binnen een applicatieserver. Het gebruik hiervan betekent een te grote impact op de performance van de hardware. Liever gebruiken we een externe webserver op aparte machines. Als deze lagen ook wat betreft hardware gescheiden zijn, kan deze door middel van clustering ook onafhankelijk geschaald worden.

Een ander motief om geen gebruik te maken van de webserver ligt op het economische vlak. Weblogic rekent af op het gebruik per CPU (*Central Processing Unit*), waardoor dit erg duur uitkomt, terwijl er talloze losse servlet runners voor allerlei bestaande webservers te vinden zijn, die stukken goedkoper zijn of zelfs gratis.

BEA rekent af per CPU en vraagt verder voor talloze extra producten geld. De prijs per CPU voor de standaard editie bedraagt 14.940 Euro.
Voor een versie met cluster mogelijkheden moet 22.400 Euro per CPU betaald worden.
Verder rekent BEA geld voor support: 2.400 Euro en voor een database driver: 5.000 Euro.

4.1.2 Orion

Evermind's Orion (www.orionserver.com) is een relatief nieuw product. Het wordt gemaakt door een klein aantal mensen die het meer voor de lol lijken te doen dan om het geld. Dat blijkt wel aan de prijs van slechts 1500 dollar per commerciële toepassing. De heren zijn verder uitstekend individueel aanspreekbaar en altijd bereid vragen te beantwoorden van iedereen die ze aanspreekt op Internet.

Het motief achter Orion is het maken van het snelste product op de markt en dit lijkt gezien de benchmarks van de laatste versie (1.0) ook aardig gelukt.

Orion bevat net als Weblogic ook een volledige webserver met servlet en JSP support.
Orion wordt echter niet zoveel gebruikt als de andere producten en is vrij nieuw. Het moet zich dus nog bewijzen in de markt. Bovendien lijkt er weinig garantie te zijn wat betreft de stabiliteit van het kleine bedrijf.

4.1.3 Websphere

Websphere (http://www-4.ibm.com/software/webservers/appserv/) is de applicatie server van IBM.
Websphere ondersteunt net als de andere producten alle kritische services als pooled JDBC, EJB 1.1, JMS etc en levert ook weer een totale webserver erbij met servlet en JSP mogelijkheden.

IBM rekent voor het gebruik van Websphere 18.000 gulden per CPU.
Het gebruik van clustering kost niets extra's.

4.2 Keuze van MarketXS

4.2.1 Applicatieserver

Bij het verdiepen in EJB technologie, bleek dat vrijwel alle voorbeelden op het Internet voor Weblogic gemaakt waren. Nu is het in principe zo dat een Enterprise Java Bean niet specifiek voor een applicatieserver wordt geschreven en ook in een ander product gebruikt kan worden, maar de voorbeelden die ons aanvankelijk op weg hielpen beschreven uiteraard ook het inladen van beans in de applicatieserver en hiervoor was keer op keer gekozen voor Weblogic.
Dit maakte ons vertrouwd met het gebruik ervan.

Omdat EJB relatief nieuw is en alleen in zeer gespecialiseerde systemen wordt toegepast vanwege de grotere complexiteit, is er weinig documentatie en hulp te vinden. Het feit dat BEA een grote hoeveelheid publiekelijk toegankelijke documentatie op haar website heeft staan en zelfs een eigen newsserver heeft waar de ontwikkelaars van Weblogic zelf vragen beantwoorden, heeft ons tijdens het ontwikkelen erg geholpen.
Mede vanwege deze goede support hebben we uiteindelijk dan ook gekozen voor dit product.

Orion heeft een aantal duidelijke minpunten. Ten eerste het enorme gebrek aan documentatie. Er zit maar een klein groepje mensen achter de ontwikkeling en zij leggen meer nadruk op de implementatie van de server, dan op het schrijven van documentatie.
Een ander probleem vormt de kleine partij achter Orion. Er is weinig meer bekend over Evermind, dan dat het gevormd wordt door een klein groepje fanatieke ontwikkelaars. Het is duidelijk dat dit weinig garantie kan geven over de stabiliteit van het bedrijf. MarketXS kan niet het risico lopen dat het bedrijf opeens ophoudt te bestaan.

4.2.2 Webserver

Ondanks dat alle besproken applicatieservers een webserver meeleveren, zullen we daar geen gebruik van maken. Zoals eerder aangegeven heeft een webserver in hetzelfde proces als de beans als voordeel dat servlets deze beans direct kunnen adresseren en de communicatie niet via een netwerkprotocol hoeft te gaan. We zullen dit echter niet gebruiken, omdat we daarmee teveel vergen van de hardware van de applicatieserver. We willen alle verwerkingskracht van de applicatieservers inzetten voor de *business-logic*.

Voor de webservers hebben we twee aparte machines ingericht. Het voordeel van rechtstreeks adresseren zijn we dus kwijt.
Het is mogelijk op de webserver machine wederom Weblogic te installeren net als op de applicatieserver en hiervan dan alleen de webserver (*servletrunner*) te gebruiken. We zullen dit echter niet doen vanwege de prijs. Dit zou voor ons 14.940 Euro per *uni-processor* webserver machine betekenen en dat terwijl er op

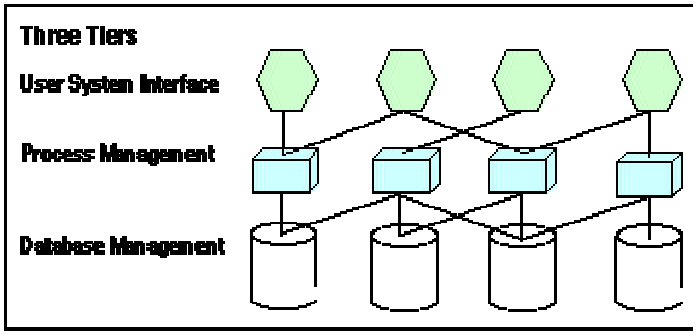
het gebied van *Java-enabled* webservers veel meer keuze is dan op het gebied van applicatieservers.

Zo is er een aantal *open source* oplossingen en een aantal relatief goedkope commerciële oplossingen.
Uiteindelijk hebben we gekozen om gebruik te maken van de interne webserver van Orion. Deze bleek in onze testen de snelste te zijn. Ook heeft Orion zoals eerder vermeld JSP support en is volledig *up-to-date* in tegenstelling tot een aantal open source projecten.
De twijfelachtige stabiliteit van het bedrijf achter Orion blijft bestaan natuurlijk, maar dit is op het gebied van *Java-enabled* webservers minder groot aangezien servlets niets webserver-specifieks bevatten en dus gemakkelijk over gezet kunnen worden naar een ander product.

Voor het totaal plaatje betekent dat dat we op de middelste laag Weblogic draaien en hiervan alleen de EJB container gebruiken en op de laag erboven, de presentatielaag, Orion toepassen waarvan we dan weer alleen de *servletrunner* gebruiken.

5 Implementatie MarketXS

We hebben het gehele systeem gescheiden in drie lagen: datalaag, logicalaag en presentatielaag en deze lagen verdeeld over aparte computers. Een dergelijk systeem wordt ***3-tier*** genoemd. Waar het vaak zo is dat bijvoorbeeld logica- en presentatielaag zich op dezelfde machines bevinden, zijn deze in ons systeem duidelijk gescheiden door ze op aparte hardware te plaatsen.
Zo kunnen we ook iedere laag onafhankelijk schalen.



Figuur 20: 3-Tier systeem architectuur. Zie ook bijlage 2.

Een ander voordeel van de scheiding in lagen is dat iedere laag op zijn eigen specifieke manier geclusterd kan worden.

De bovenste laag in ons 3-tier systeem bestaat uit twee webservers. Dit zijn Intel machines met Linux en Orion.
Clustering wordt hier geregeld via *round robin* DNS, waarbij aan de domain name meerdere IP adressen worden gekoppeld, die één voor één worden uitgegeven. Op deze manier krijgen webbrowsers die het adres van de server opvragen ieder een ander nummer, zodat ze dus ieder een verbinding maken met een andere webserver. Wanneer alle IP adressen een keer zijn uitgegeven, begint de DNS server weer van voor af aan.

De middelste laag, waar de logica in zit, wordt gevormd door 2 Sun 220R's met Solaris en Weblogic 5.1 waarin de modules als Enterprise Java Beans (EJB's) zijn deployed.
De webservers hebben ieder RMI verbindingen tussen de servlets en de de beans. Clustering wordt verzorgd door Weblogic zelf. Beide applicatieservers zijn met elkaar verbonden en vormen één geheel voor de buitenwereld. Weblogic zorgt zelf voor de synchronisatie van de beans.

De datalaag wordt gevormd door 2 Compaq DL380's met Linux en Oracle 8i Standard Edition. Hiervan wordt slechts 1 machine werkelijk gebruikt en benaderd door de applicatieservers. De andere is geconfigureerd als zogenaamde *hot-standby*. Deze server synchroniseert continue met de echte database en staat dus klaar indien de primaire database niet beschikbaar is vanwege calamiteiten.

5.1 Componentenverdeling

In dit hoofdstuk zullen de EJB's die geprogrammeerd zijn in kaart worden gebracht. Beschreven zal worden hun functionaliteit, implementatie en interactie met andere beans en buitenwereld. Ook zullen kleine stukjes voorbeeldcode getoond worden om de gegevens op te vragen vanuit een client.
Om deze code volledig te begrijpen kan de volledige MarketXS API documentatie, die in *Javadoc* formaat bijgevoegd is op CD, geraadpleegd worden.

Verder wordt in bijlage 3 een voorbeeld gegeven van een compleet servlet dat alle stappen uitvoert om een koers te verkrijgen. Onder deze stappen vallen het verbinden met en inloggen op de applicatieserver, het opzoeken van de juiste bean, creëren van een instantie, uitvoeren van de juiste functies en vervolgens het afbeelden van de teruggekregen data.

5.1.1 com.marketxs.quote

Het quote package bevat een aantal zaken die betrekking hebben op het opvragen van koersen, opzoeken van zogenaamde *ticker-symbols* (de afkorting die een bepaald fonds op een beurs heeft), indices en het opvragen van bedrijven die tot een bepaalde index behoren (zoals bijvoorbeeld de bedrijven op basis waarvan de koers van de AEX wordt bepaald).

Dit package bevat twee beans die door externe clients zijn aan te spreken en dus deel uitmaken van de MarketXS API. Dit zijn de QuoteBean en de SymbolLookupBean.

5.1.1.1 QuoteBean

De QuoteBean heeft een viertal functies voor het opvragen van koersen. Twee voor het opvragen van zogenaamde *delayed quotes* en twee voor *realtime quotes*.
Delayed quotes zijn koersen van 15 tot 20 minuten oud. Deze hebben voor beleggers uiteraard minder waarde dan realtime koersen, die niet vertraagd zijn. Beurzen vragen voor het vrijgeven van realtime koersen veel geld en rekenen meestal per *quote-request* af. Dit maakt het weergeven van koersen op grote schaal, zoals op een website een erg kostbare zaak. Daar komt nog eens bij, dat de meeste beurzen niet alleen precies willen weten wanneer er welke realtime gegevens zijn opgevraagd, maar ook op de hoogte willen zijn van de gegevens van de persoon die ze heeft opgevraagd. Denk hierbij aan NAW gegevens.
Een dergelijke eis betekent dus ook dat een gebruiker van het Internet zich eerst zal moeten registreren (zogenaamde *closed user groups*).

Om het vrijgeven van koersgegevens wat minder omslachtig te kunnen laten plaatsvinden, zijn delayed koersen geïntroduceerd Deze vertraagde informatie mag wel aan anonieme gebruikers getoond worden en kost veel minder. Bovendien rekent geen enkele beurs per opgevraagde delayed koers af.

De QuoteBean kan zowel gebruikt worden voor het opvragen van een losse quote, als voor het opvragen van een heleboel quotes tegelijk. Bij deze bulk verwerking is er minder *overhead* dan bij individueel opvragen.

Per quote kan een grote hoeveelheid gegevens worden opgevraagd. Hieronder vallen onder andere:

- Naam van het bedrijf;
- Naam van de beurs waaraan het genoteerd staat;
- Laatste koers;
- Tijdstip van de laatste transactie;

- Volume van de laatste transactie;
- Bied/laat prijzen;
- Prijsverandering in vergelijking met het slot van de vorige dag;
- Vertraging van de gegevens in minuten;
- Hoogste/laagste waarde van de dag;
- Etc..

Deze gegevens worden verpakt in een object en als zodanig teruggestuurd door de bean.
De programmeur kan zelf aangeven hoe uitgebreid de gegevens moeten zijn, waarbij in gedachten gehouden moet worden dat een uitgebreidere quote meer tijd kost dan een eenvoudige.
Om dit gemakkelijk te maken is een overervingstructuur gemaakt van quote objecten die steeds meer informatie bevatten.

Bij het verkrijgen van realtime koersen dient de functie-aanroep voorzien te worden met een object dat een gebruikersprofiel voorstelt. Dit is noodzakelijk, aangezien MarketXS deze gegevens moet loggen.

```
try
{
  long tickerId = 57575;
  Context ctx = getInitialContext();
  QuoteHome qh = (QuoteHome)ctx.lookup("com.marketxs.quote.Quote");

  Quote quote = qh.create();
  ShortQuote shortquote = quote.getQuote(tickerId, new ShortQuote());

  out.println("Last quote for tickerId "+tickerId+" = "+shortquote.getLast());
}
catch (QuoteException e){out.println("Fout in de EJB.");}
catch (NamingException e){out.println("Fout bij het opzoeken van de EJB.”);}
catch (CreateException e){out.println("Fout bij het verkrijgen van een EJB
       instantie");}
catch (RemoteException e){out.println("Fout in de RMI communicatie.”);}
```

Figuur 21: Client voorbeeld dat de huidige, vertraagde koers van een aandeel weergeeft. Het tickerId is voor dit voorbeeld gehardcode.

5.1.1.2 SymbolLookupBean

In het voorgaande code voorbeeld was te zien dat een fonds aangegeven wordt door een zogenaamd *tickerId*. Dit is het nummer dat het fonds in de database van MarketXS heeft.
Uiteraard wordt een bedrijf op de beurs geïdentificeerd door middel van een *ticker-symbol*, dus beschrijft de code ook een EJB waarmee aan de hand van symbols en bedrijfsnamen de tickerId's gevonden kunnen worden: de SymbolLookupBean.

De reden dat MarketXS intern niet de symbols gebruikt die de beurs ook hanteert, is omdat dezelfde lettercombinaties op verschillende beurzen kunnen voorkomen, zonder dat dit direct hetzelfde bedrijf betreft.

Deze bean bevat functies voor het zoeken van bedrijven aan de hand van een symbol of (onvolledige) bedrijfsnaam.
Ook kan de bean een lijst leveren met alle markten die beschikbaar zijn (als een klant alleen betaalt voor de AEX, zal dit de enige markt zijn die zichtbaar is).
Voor het achterhalen van alle bedrijven die tot de samenstelling van een bepaalde index (bijvoorbeeld AEX-index, CAC40, FTSE100) horen, heeft de EJB ook functies.

Het resultaat van de functies in de SymbolLookupBean betreft altijd één of meer objecten die een ticker (fonds) voorstellen. Hierin staan geen koersgegevens.
Voor het afbeelden van een lijst met koersen van alle bedrijven die tot de FTSE100 index behoren, dienen de tickerId's uit de lijst met ticker objecten door de QuoteBean gehaald worden. Dit levert dan weer een lijst met 100 koers objecten op.

```
try
{
  Context ctx = getInitialContext();
  SymbolLookupHome slh =
        (SymbolLookupHome)ctx.lookup("com.marketxs.quote.SymbolLookup");

  SymbolLookup slookup = slh.create();
  Collection col = slookup.getTickersBySymbol(“MSFT”, 1);
  if(col.size() > 0) {
    Iterator it = col.iterator();
    while(it.hasNext()) {
      TickerData buffer = (TickerData)it.next();
      out.println(buffer.getName() +” (“+buffer.getSymbol()+”) Market: ” +
         buffer.getMarketName());
  } else
    out.println(“No matches.”);
}
catch (Exception e){out.println("Fout opgetreden.");}
```

Figuur 22: Client voorbeeld dat bedrijven zoekt met het symbol 'MSFT'.

5.1.2 com.marketxs.csp

In dit package zit de logica die de applicatieserver koppelt aan de externe koers feed.
MarketXS krijgt via satelliet schotels een continue stroom van koers updates. Iedere keer als er ergens op een beurs een transactie is geweest en dus de koers van een bedrijf is veranderd, wordt dit verzonden via een satelliet netwerk en ontvangen onze schotels dat onmiddellijk

Deze koersgegevens worden geleverd door Standard & Poors en worden opgevangen door speciale machines van S&P zelf die aangesloten zijn op de schotels. Deze slaan van ieder fonds de laatste koers op in geheugen. Deze machines heten CSP's (Client Site Processor).

Per seconde komen er momenteel zo'n 300 koers updates binnen.

Gezien de grote hoeveel updates per seconde, slaan we deze data niet op in een database, maar benadert de com.marketxs.csp.CSPPool EJB deze machines wanneer er een koers opgevraagd wordt.

5.1.2.1 CSPPoolBean

Deze bean heeft één duidelijk functie, namelijk het open houden van netwerkverbindingen naar de *Client Site Processors*. Hierbij maken we gebruik van het feit dat een EJB instance na gebruik niet direct vernietigd wordt. Dit stelt ons in staat bij het creëren van een nieuwe instance van de CSPPool een TCP/IP verbinding te openen naar de CSP machine en deze niet meer te sluiten.
Nieuwe clients krijgen dan een bestaande instantie terug van de CSPPool bean en er wordt geen *overhead* gegenereerd bij het aanmaken van netwerkverbindingen. We hebben hiermee een *pool* met beschikbare netwerkverbindingen gecreëerd

De bean wordt alleen intern gebruikt door de QuoteBean en is niet benaderbaar voor externe clients omdat deze in een andere *security-realm* zit dan de beans die wel benaderd mogen worden.

5.1.3 com.marketxs.currency

Bij een volledig scala van financiële gegevens horen ook valuta. MarketXS heeft toegang tot valuta via de API zo geregeld, dat koersen omgerekend kunnen worden naar elke andere valuta. Dit is noodzakelijk voor het weergeven van fondsen uit verschillende landen binnen één portfolio.
Valuta kunnen uiteraard ook gewoon met hun huidige waarde opgevraagd worden.

5.1.3.1 CurrencyBean

De CurrencyBean is verantwoordelijk voor het ophalen van de juiste waarden van de valuta. Het verkrijgt deze gegevens via de CSPPoolBean uit de CSP's. Valuta veranderingen worden namelijk ook doorgegeven via de satellietverbinding.

Om te voorkomen dat valuta elke keer dat ze nodig zijn opnieuw opgevraagd gaan worden via de CSP, terwijl deze waardes maar weinig fluctueren, passen we opnieuw het gegeven toe dat de container instanties niet direct vernietigt.
Wanneer een instantie van de CurrencyBean wordt gemaakt, haalt deze alle valuta binnen en houdt deze vervolgens in het geheugen. Zo hoeven valuta nooit opnieuw worden opgevraagd en zijn ze altijd direct vanuit het geheugen te gebruiken.
Om te voorkomen dat de bean na verloop van tijd verouderde gegevens geeft, ververst de bean om de zoveel tijd de waarden.

```
try {
  Context ctx = getInitialContext();
  double bedrag = 100;
  CurrencyHome ch =    (CurrencyHome)ctx.lookup("com.marketxs.quote.Currency");

  Currency curr = ch.create();
  CurrencyData curr1 = curr.getCurrency(1);
  CurrencyData curr2 = curr.getCurrency(2);

  out.println(bedrag +" "+ curr1.getName() +" = "+ curr.convertCurrency(bedrag,
  curr1.getCurrencyId(), curr2.getCurrencyId()) +" "+curr2.getName());

}
catch(Exception e) {out.println("Fout opgetreden.");}
```

Figuur 23: Client voorbeeld dat een bedrag van de ene naar de andere valuta omrekent.

5.1.4 com.marketxs.profile

PortfolioXS is de module van MarketXS die onze klanten in staat stelt gebruikersgegevens en portfolio's (aandelenportefeuilles) op te slaan in het systeem.
Portfolio's op een financiële site zijn lijstjes met aandelen die een belegger wil volgen. Het is te vergelijken met een map waar de belegger aandelen in stopt en zo een duidelijk overzicht heeft van de status ervan.

MarketXS biedt een dergelijk portfolio systeem aan via de API door middel van een aantal beans.
Ten eerste de ProfileBean, deze wordt gebruikt voor het beheren van gebruikersprofielen. Profielen kunnen worden toegevoegd, gewijzigd en verwijderd.
De tweede bean is de PortfolioBean waarmee portfolio's bijgehouden kunnen worden.
Een derde bean maakt het systeem compleet. Namelijk de PortfolioContentBean. Deze bean beheert de records per portfolio. Een portfolio bevat namelijk een aantal PortfolioContent objecten. Deze bevatten ieder informatie over één aandeel, plus gegevens over aantal aandelen die de belegger ervan heeft alsmede prijs die de belegger ervoor betaalde.

Your portfolio — edit portfolio | delete portfolio

Symbol	Name	Last	Change (%)	Volume	Current Value	Purchase Price	Profit
CORL	COREL CORP	4.03	-0.38 (-8.51 %)	1,000	4,031.25	9.10	-5,068.75
PHI	KON. PHILIPS ELECTRONICS	52.80	-0.10 (0.00 %)	500	26,400.00	44.00	4,400.00
MSFT	MICROSOFT CORP	78.94	0.12 (0.16 %)	1,500	118,406.25	70.42	12,776.25
RHAT	RED HAT INC	27.94	0.31 (1.13 %)	1,100	30,731.25	30.00	-2,268.75
Calculated in Euro					**188,775.44**		**10,165.66**

Mijn tweede portfolio — edit portfolio | delete portfolio

Symbol	Name	Last	Change (%)	Volume	Current Value	Purchase Price	Profit
NKE	NIKE INC. B - US6541061031	42.25	0.25 (0.00 %)	900	38,025.00	44.90	-2,385.00
MCD	MCDONALD'S CORP	32.19	-0.50 (-1.53 %)	1,240	39,912.50	31.80	480.50
KO	COCA-COLA CO	60.03	2.16 (3.73 %)	1,000	60,031.25	48.00	12,031.25
Calculated in Euro					**143,976.18**		**10,878.81**

Figuur 14: Het portefeuille systeem van MarketXS.

Tezamen leveren deze componenten alle zaken die nodig zijn om een gepersonifieerd portfoliosysteem aan te bieden op een website.

5.1.4.1 ProfileBean

Zoals gemeld wordt deze bean aangesproken door externe client programma's om gebruikersprofielen te beheren. De interface bevat hiervoor een aantal eenvoudig te begrijpen en te gebruiken functies.

Intern gebruikt deze bean een entity bean die de gebruikersprofielen op een objectgeoriënteerde manier beschikbaar stelt.

```
try
{
  ProfileHome ph = (ProfileHome)ctx.lookup("com.marketxs.profile.Profile");
  Profile profileManager = ph.create();
  ProfileData userProfile = profileManager.getProfile("jdoe");

  userProfile.setFirstname("John");
  userProfile.setLastname("Doe");

  profileManager.update(userProfile);
}
catch(Exception e) { out.println("Fout opgetreden.");}
```

Figuur 25: Client code voorbeeld dat het profiel van gebruiker "jdoe" oproept en zijn voor- en achternaam wijzigt.

Een uitgebreid voorbeeld voor het aanmaken en beheren van gebruikersprofielen is te vinden in de handleiding voor de API die als apart document is bijgevoegd.

5.1.4.2 PortfolioBean

Voor het aanbieden van portfolio's per gebruiker wordt de PortfolioBean gebruikt. De communicatie met de bean gaat steeds via zogenaamde PortfolioData objecten Dit is een klein object waarin in één keer alle eigenschappen van een portfolio staan, zodat communicatie efficiënt kan plaatsvinden.

Elke portfolio heeft een eigenaar, namelijk het gebruikersprofiel van de eindgebruiker die hem heeft aangemaakt.

Intern gebruikt de PortfolioBean een entity bean die een object georiënteerde koppeling met de database verzorgt.

```
try {
  PortfolioHome ph =
    (PortfolioHome)ctx.lookup(“com.marketxs.portfolio.PortfolioHome”);
  Portfolio pmanager = ph.create();

  PortfolioData portfolio = new PortfolioData(“My Portfolio”);
  portfolio.setOwner(userProfile);

  portfolio = pmanager.add(portfolio);
}
catch(Exception e) { out.println(“Fout opgetreden.”);}
```

Figuur 26: Client code voorbeeld dat een nieuwe portfolio met de naam "My Portfolio" aanmaakt voor gebruiker John Doe.

N.B. Het afgebeelde voorbeeld werkt verder met het profiel van gebruiker John Doe die in het vorige voorbeeld verkregen was.

5.1.4.3 PortfolioContentBean

Eerder is al genoemd dat het hiërarchische portfolio model verder gaat met het implementeren van afzonderlijke PortfolioContent object binnen een portfolio. Deze PortfolioContent object bevatten ieder gegevens over een enkel fonds, inclusief gegevens die voor de eigenaar van belang zijn zoals aantal aandelen en de aankoopprijs ervan.

De PortfolioContentBean biedt de functionaliteit deze objecten per portfolio te beheren met functies voor onder andere het toevoegen van nieuwe fondsen aan een bestaande portfolio, het verwijderen ervan en het aanpassen van de eigenschappen zoals aantal aandelen.

Intern wordt weer gebruik gemaakt van een entity bean die de synchronisatie met de database op een object georiënteerde manier mogelijk maakt.

```
try {
  PortfolioContentHome pch =
    (PortfolioContentHome)ctx.lookup(“com.marketxs.profile.PortfolioContentHome”);
  PortfolioContent pcmanager = pch.create();

  PortfolioContentData pcd = new PortfolioContentData(57575);
  pcd.setShares(1500);
  pcd.setPurchaseprise(15.2);
  pcd.addTo(portfolio);

  pcd = pcmanager.add(pcd);
}
catch(Exception e) { out.println(“Fout opgetreden.”);}
```

Figuur 27: Client code voorbeeld dat een nieuw fonds (tickerId 57575) toevoegt aan een bestaand portfolio.

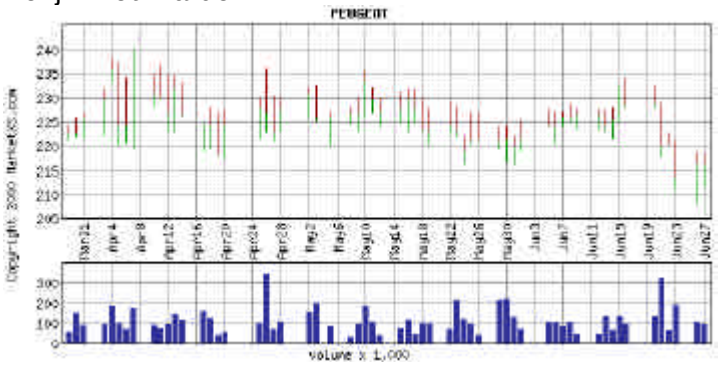
N.B. Het afgebeelde voorbeeld werkt verder met de nieuwe portfolio die aangemaakt is in het vorige voorbeeld.

5.1.5 Overige componenten

Ten tijde van dit schrijven is een aantal andere componenten nog niet volledig gereed en in de laatste fasen van ontwikkeling.
Deze componenten vormen de basis achter NewsXS en FinancialXS.

5.2 Charting

Bij een volledige set financiële gegevens voor een website horen ook grafieken van koersverloop.
Het afbeelden van grafieken gaat op een andere manier dan die van de hiervoor besproken componenten, namelijk niet via de API.
De output van de chart module is binaire, grafische data, namelijk een plaatje in GIF formaat. Het is duidelijk dat dit plaatje niet meer aan te passen is door de klant. Daarom hebben we hier gekozen voor de oplossing waarbij de chart-engine rechtstreeks bereikbaar is vanaf het Internet en rechtstreeks aangeroepen wordt door de webbrowser van de eindgebruiker. Het enige dat de klant van ons doet om een koersgrafiek weer te geven is het URL naar onze chart-engine op te nemen in de pagina. In het URL van de chart worden dan de eigenschappen opgenomen die uiteindelijk als argumenten aan de chart-engine worden meegegeven.



Figuur 28: 3-maanden grafiek van Peugeot op Paris bourse met high, low, close gegevens en volume, zoals deze door de chart-engine wordt gegenereerd.

Deze argumenten zijn onder andere tickerId, waarmee aangegeven wordt van welk bedrijf of welke index het koersverloop weergegeven moet worden. Ook kunnen meerdere tickerId's meegegeven worden. In dat geval zal een zogenaamde *benchmark* afgebeeld worden waarbij het relatieve koersverloop van alle meegegeven tickers worden afgebeeld. Dergelijke *benchmark* grafieken geven snel een indicatie over hoe goed een bedrijf het doet in vergelijking met een ander bedrijf.

5.2.1 Functionaliteit

De chart module zal aanvankelijk de volgende functionaliteit aanbieden:

- Periode van koersverloop: *intraday*, week, maand, 3 maanden, 1 jaar, 2 jaar en 5 jaar.
  Bij *intraday*- en weekgrafieken zal elke beurstransactie van die dag een punt op de grafiek vertegenwoordigen.
  Bij 1 en 3 maand grafieken zullen per dag de *high*, *low* en *close* waarden weergegeven worden in verticale streepjes.
  Bij grotere tijdsperioden zal van iedere dag enkel de close waarde als punt voor de lijngrafiek gelden.
- Mogelijkheid tot het afbeelden van een staafgrafiek met de verhandelde volumes.
- Grootte in hoogte en breedte van het plaatje kunnen meegegeven worden.
- Kleur van de lijnen, achtergrond, etc kunnen aangepast worden.

Deze eigenschappen worden via URL parameters aangegeven. Een voorbeeld van een URL zou zijn:
http://www.marketxs.com/servlet/Charts?tickerId=57575&x=400&y=300&type=y&bgcolor=040404

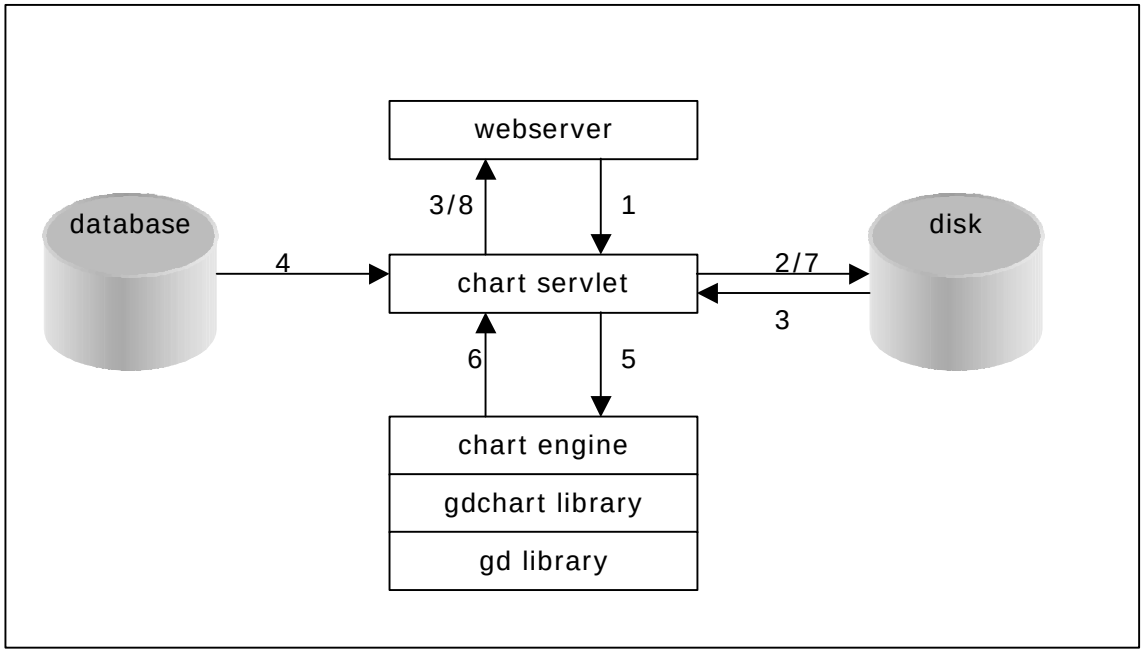
5.2.2 Architectuur

Figuur 29: 5-jaars grafiek van Peugeot op Paris bourse.

De chart applicatie ontwikkelen we zelf. Hiervoor stond performance voorop. De applicatie zal uiteindelijk door alle klanten van ons worden aangeroepen en dus het van groot belang dat het geen bottleneck gaat vormen.
Daarom hebben we besloten de engine in C te ontwikkelen. Om dit netjes in het voor de rest Java gebaseerde systeem te koppelen, zal er bovenop de C applicatie een Java laag worden gebouwd die vanuit een Java *servlet* aangeroepen kan worden.
Zo hebben we de flexibiliteit van Java en de performance van C.

Een ander belangrijk onderdeel is *caching*. Het genereren van een plaatje blijft een heleboel werk en het is zinloos een grafiek bij elke direct op elkaar volgende pageview opnieuw te maken. We gaan de plaatjes dan ook opslaan op schijf na het genereren.

Wanneer hetzelfde plaatje nu direct weer opgevraagd wordt, wordt eerst gekeken of het plaatje al op schijf bestaat. Als dat zo is hoeven er geen historische koersgegevens uit de database opgevraagd te worden en de chart engine niet te draaien, maar de GIF als statische content direct worden ingelezen.

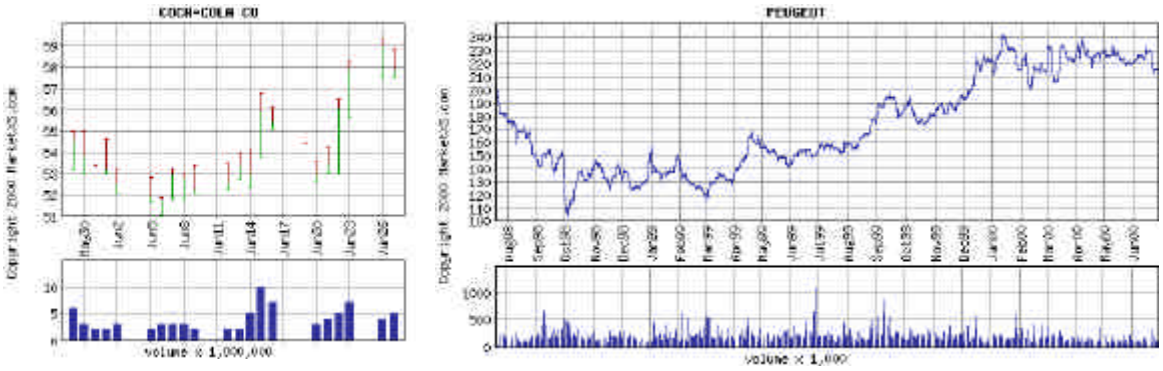


Figuur 30: De volledige werking van de MarketXS chart-engine stap voor stap in beeld gebracht.

De logica die met de database communiceert bij het opvragen van historisch koersgegevens en gegenereerde bestanden naar schijf schrijft en leest brengen we onder in de Java laag die boven de in C geschreven engine ligt.

Het gehele proces is als volgt samen te vatten:

1. Het chart *servlet* ontvangt een URL voor een grafiek;
2. Het *servlet* combineert de parameters tot een bestandsnaam en checkt op disk of deze file al bestaat;
3. Als deze bestaat, opent het *servlet* de file, leest de inhoud ervan en geeft deze data direct door naar de webserver die het naar de webbrowser verstuurt.
4. Als de file niet op schijf wordt gevonden, pakt het *servlet* een database connectie en vraagt de koersgegevens voor de gevraagde periode van het gegeven bedrijf of index op.
5. De ingelezen koersdata wordt samen met de overige parameters als pixel grootte doorgegeven aan de in C geschreven engine.
6. De engine tekent het plaatje in het geheugen en geeft deze binaire data terug aan het *servlet*.
7. Het *servlet* opent een file met de eerder gegenereerde naam en schrijft de binaire data hier in weg.
8. Tegelijkertijd wordt de binaire data ook doorgegeven via de webserver naar de webbrowser.



Figuur 31: Links een 1-maand grafiek van Coca-Cola op de New-York Stock Exchange. Rechts een 2-jaars grafiek van Peugeot.

De chart engine van MarketXS is zelf ontwikkeld. Om dit ontwikkelproces niet te omslachtig te maken zijn we op zoek gegaan naar een library of bestaande applicatie die op een zo rudimentair mogelijke manier afbeeldingen kan genereren en waar we de rest van ons systeem flexibel overheen konden programmeren.
We hebben een dergelijke library gevonden in de gdchart library van Bruce Verderaime. Dit is een C library over de bekende, gratis gd-library heen.
De gd-library, ontwikkeld door Thomas Boutell, is een *open-source* library voor het genereren van afbeeldingen aan de hand van een eenvoudige tekenset.
De gdchart library, die ook volledig *open-source* is, bouwt hierop verder door een gemakkelijke library aan te bieden die is toegespitst op het genereren van eenvoudige grafieken.
Bovenop de gdchart library ligt de MarketXS chart-engine.

Opgemerkt moet worden dat we enige aanpassingen aan de gdchart library hebben aangebracht met betrekking tot het weergeven van de X-as. Hierover hebben we niet alleen contact, maar ook uitstekende medewerking en hulp gehad van de auteur zelf.

6 Conclusie

Bij de uitvoering van de afstudeeropdracht is gebleken dat bij het ontwikkelen van grootschalige online systemen de oplossing gezocht moet worden in het toepassen van de zogenaamde 3-tier architectuur, waarbij het systeem verdeeld wordt in 3 duidelijk verschillende lagen. Hierdoor blijft het systeem overzichtelijk en onderhoudbaar.

Om het systeem qua hardware schaalbaar en redundant te houden, is iedere laag dubbel en geclusterd uitgevoerd.

Dit verslag concentreert zich op de middelste laag van de 3-tier architectuur: de logicalaag. Hierin zit de code die alle systemen erboven voorziet van data.
Om deze laag overzichtelijk, maar vooral ook schaalbaar te houden, hebben we gekozen voor de toepassing van distributed computing en wel in het bijzonder voor de Java implementatie ervan: Enterprise Java Beans.

Door een duidelijke API te definiëren voor de beans, kunnen deze componentjes aan elkaar gekoppeld worden en door software van derden gebruikt worden.
Op deze manier komt de *Holy Grail* van software ontwikkelen behoorlijk dicht in zicht.

Een aantal zaken is in dit verslag niet duidelijk naar voren gekomen.
Zo beschrijft de opdracht onder meer het leiden van de bouw en ontwikkeling van het hele MarketXS systeem en het kiezen van de juiste hardware. De reden dat deze zaken minder duidelijk beschreven staan, is omdat de nadruk van het verslag ligt op de hoofdtaken die van HBO niveau zijn.

Bijlage 1: Verklarende woordenlijst

API	Application Programming Interface: beschrijving van de manier waarop met een stuk software gecommuniceerd moet worden.
COM	Techniek van Microsoft die programmeren met componenten mogelijk maakt. Een individueel object heet COM-object en is programmeertaal onafhankelijk. http://www.microsoft.com/com/default.asp
CORBA	Common Object Request Broker Architecture: specificatie opgesteld door OMG (Object Management Group) voor distributed computing. CORBA is programmeertaal en systeem onafhankelijk en gebruikt IIOP als netwerk protocol. http://www.corba.com
DCOM	Distributed Component Object Model: extra laag onder *COM* die het mogelijk maakt COM-objecten ook over een netwerk te laten communiceren.
client site processor	Zie *CSP*.
CSP	Machine van S&P Comstock die koers updates opvangt van een satellietschotel en ze beschikbaar stelt.
deployment descriptors	De bestanden van een Enterprise Java Bean (zie *EJB*) waarin configuratie opties staan die bij het inladen door de EJB-container worden geïnterpreteerd.
distributed computing	Het aan elkaar koppelen van programmacomponenten die verspreid zijn over een netwerk.
container	De applicatie server waar Enterprise Java Beans (zie *EJB*) in worden geladen en beheerd. Containers bieden vaak extra services zoals JDBC database pools.
conversational state	Term waarmee aangeduid wordt, dat een Enterprise Java Bean (zie *EJB*) gegevens vast houdt die betrekking hebben op een specifieke client.
EJB	Java Enterprise Beans: de techniek die het mogelijk maakt gemakkelijk gedistribueerde applicaties in Java te ontwikkelen. http://java.sun.com/products/ejb
EJB-container	Zie *container*.
fine-tuning	Het afstellen van software voor optimale performance.
firewall	Computer die twee netwerken verbindt en bepaalde soorten verkeer tegenhoudt, om zo het achterliggende netwerk veilig te houden.

garbage collection	Geautomatiseerd proces in een Java Virtual Machine dat gealloceerd geheugen waarnaar geen references meer zijn vrijgeeft.
home interface	De interface van een Java component bij toepassing van RMI, waarin functies gedefinieerd zijn voor het verkrijgen van instanties van het object.
hot-standy	Bij databases de machine die volledig gesynchroniseerd is met de primaire database en klaar is om in geval van calamiteiten de primaire database over te nemen.
HTTP	HyperText Transfer Protocol: het protocol waarover webservers en webbrowsers gegevens uitwisselen.
IDL	Interface Definition Language: de taal waarin bij een CORBA applicatie de interfaces van het server component worden beschreven. Deze IDL file wordt later met een IDL compiler gecompileerd tot een binaire file die aan de server kan worden gelinkt.
IIOP	Internet InterOrb Protocol: het standaard protocol van CORBA.
implementation repository	In een CORBA applicatie de database waarin de beschrijvingen van de interfaces van server component staan.
intraday	Bij koersgrafieken een grafiek die het koersverloop van 1 dag afbeeldt, waarbij iedere beurstransactie doorgaans een apart punt op de lijn is.
Java applicatie server	Zie *container*.
javadoc	Java programma dat uit programmacode volledige programmadocumentatie genereert in HTML formaat. Hiertoe dient de programmacode met speciale tags gedocumenteerd te zijn.
JDBC	Java DataBase Connectivity: class library die Java applicaties de mogelijkheid geeft door middel van een JDBC driver een database te benaderen. http://java.sun.com/products/jdbc
JMS	Java Message Service: class library die Java de beschikking geeft over een asynchroon message systeem. http://java.sun.com/products/jms/
JNDI	Java Naming and Directory Interface: class library die Java applicaties toegang biedt tot directory georiënteerde bronnen zoals LDAP. http://java.sun.com/products/jndi/
JSP	Java Server Pages: toepassing die het ontwikkelen van dynamische webpagina's met Java het gemak geeft van

	een scripttaal. http://java.sun.com/products/jsp/
linken	Het samen voegen van meerdere losse programma onderdelen tot één programmabestand. Afzonderlijk kunnen deze onderdelen niets doen.
LPC	Local Process Communication: het protocol waarmee COM objecten in aparte processen met elkaar communiceren.
multi-tier	Ook wel n-tier genoemd. Het scheiden van een online systeem in meerdere lagen. Typische voorbeelden zijn datalaag, applicatielaag en presentatielaag.
OMG	Object Management Group: een consortium van software producenten en gebruikersgroepen. http://www.omg.org
open source	Software die gratis en inclusief broncode publiekelijk beschikbaar wordt gesteld en anderen toestaat veranderingen eraan aan te brengen.
ORB	Object Request Broker: bij een CORBA applicatie zorgen de ORB's voor de communicatie tussen client en server. Ze maken ook de netwerktransparantie mogelijk.
overhead	Overbodige belasting die ontstaat door toepassing van een bepaalde methode.
passivaten	Zie ook *passivation*. De Nederlandse verbuiging van het Engelse werkwoord waarmee het proces bedoeld wordt waarbij *EJB* instanties naar disk worden geschreven.
passivation	Het proces waarin een *container* instanties van Enterprise Java Beans (zie *EJB*) naar schijf kan schrijven om geheugen te sparen.
proxy object	Bij een gedistribueerde applicatie een kopie van het server component aan de client-kant. Dit object bevat niet de implementatie en geeft functie aanroepen transparant door naar de onderliggende lagen.
remote interface	De interface van een Java component bij toepassing van RMI, waarin functies gedefinieerd zijn die het object ondersteunt.
servlet	Servlets zijn Java programma's die op een webserver draaien en dynamische HTML pagina's genereren. http://java.sun.com/products/servlet/
servletrunner	Webserver software die dynamische pagina's laat genereren door Java *servlets*.
SOAP	Simple Object Access Protocol: wijze van gedistribueerd programmeren, waarbij *HTTP* als protocol en *XML* als codering wordt gebruikt voor de communicatie. http://msdn.microsoft.com/workshop/

	xml/general/SOAP_White_Paper.asp
three-tier	Het scheiden van een online systeem in drie lagen: datalaag, applicatielaag en presentatielaag.
ticker-symbols	De afkorting waarmee een bedrijf op de beurs wordt aangeduid. Voorbeelden zijn MSFT (Microsoft), AAB (ABN Amro Bank) en CORL (Corel Corporation).
RMI	Remote Method Invocation: Java bibliotheek die gedistribueerde applicaties in Java mogelijk maakt. Hierbij verstuurt RMI de functie aanroepen over het netwerk. http://java.sun.com/products/jdk/rmi/
RPC	Remote Procedure Call: oudste vorm van het uitvoeren van functies, waarvan de implementatie zich in een ander proces bevindt.
serialization	De techniek die Java toepast om instanties van object te beschrijven als statische data. Serialization wordt door RMI gebruikt om objecten over het netwerk te kunnen versturen.
skeletons	De laag tussen server software component en ORB in een CORBA applicatie.
stubs	De laag tussen client en ORB in een CORBA applicatie. Dankzij de stub lijken de functies van de server lokaal in het client programma beschikbaar te zijn.
XML	eXtensible Markup Language: taal waarin datastructuren beschreven kunnen worden door het gebruik van eigen attributen (tags). http://www.xml.org/xmlorg_resources/whitepapers.shtml

Bijlage 3: Quote servlet voorbeeld

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;
```

Required for the servlet

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

Required for EJB, RMI and JNDI.

```
import java.util.*;
```

```
import com.marketxs.quote.*;
```

Required for the stubs

```
public class TestServlet extends HttpServlet
{
        public void service(HttpServletRequest req, HttpServletResponse res) throws
ServletException, IOException
        {
         PrintWriter out;
         long tickerId;
         Double last;
         String mask = "1";
         Quote q;

         res.setContentType("text/html");
         out = res.getWriter();
         tickerId=req.getParameter("tickerid") == null ? 57257 : new
Long(req.getParameter("tickerid")).longValue();

         try
         {
          Context ctx = getInitialContext();
          QuoteHome qh = (QuoteHome)ctx.lookup("com.marketxs.quote.Quote");
          q = qh.create();
          QuoteData qd = getQuote(q, tickerId, mask);
```

Connect to the application server
Look up the QuoteXS EJB in the Naming Directory
Create a Quote object
Create a QuoteData object

```
          out.println("Last quote for tickerId "+tickerId+" = "+qd.getLast());
         }
         catch (NamingException e){out.println("Ooops: NamingException occured!");}
         catch (CreateException e){out.println("Ooops: CreateException occured!");}

        } // service

        public Context getInitialContext() throws NamingException
        {
         String user     = "foo";
         String password = "bar";
         String url      = "t3://path.to.appserver.com:7001";
```

Define the username, password and specify the URL where the servlet may find the application server.

```
         Hashtable hash = new Hashtable();
         hash.put(Context.INITIAL_CONTEXT_FACTORY, "weblogic.jndi.WLInitialContextFactory");
         hash.put(Context.PROVIDER_URL, url);
         hash.put(Context.SECURITY_PRINCIPAL, user);
         hash.put(Context.SECURITY_CREDENTIALS, password);
```

Set context properties to log on to the application server

```
         return new InitialContext(hash);
        }

       }
        public QuoteData getQuote(Quote q, long tickerId, String mask)
        {
         QuoteData quoteData = new QuoteData();
         try
         {
          quoteData = q.getQuote(tickerId, mask);
         }
         catch(RemoteException e){}
         catch(QuoteException e) {}
         return quoteData;
        }
}
```

Bijlage 4: Literatuurlijst

Literatuurbronnen

Tricks Of The Java Programming Gurus – Sams Net
Glenn L. Vanderburg
ISBN: 1-57521-102-5

Programmeren met Java! – Academic Service
Tim Ritchie
ISBN: 90-395-0390-7

Java 1.1 in 21 days, second edition – Sams Net
Laura Lemay and Charles L. Perkins
ISBN: 1-57521-142-4

Online bronnen

http://www.cetus-links.org/oo_distributed_objects.html
Verzameling met links naar documenten over distributed computing in het algemeen.

http://www.corba.com
Homepage voor CORBA

http://www.omg.org
Homepage van Object Management Group, het consortium achter CORBA.

http://www.omg.org/corba/cichpter.html
CORBA/IIOP 2.3.1 specificatie.

http://www.cetus-links.org/oo_object_request_brokers.html
Pagina met vrijwel alle ORB implementaties die er zijn.

http://java.sun.com/products/ejb
Sun's Enterprise Java Bean homepage.

ftp://ftp.java.sun.com/pub/ejb/11final-129822/ejb1_1-spec.pdf
Enterprise Java Beans 1.1 specificatie.

http://java.sun.com/products/ejb/javadoc-1.1/
Enterprise Java Bean 1.1 API.

http://java.sun.com/products/servlet/index.html
Sun's homepage over servlets.

ftp://ftp.java.sun.com/pub/servlet/22final-182874/servlet2_2-spec.pdf
Servlet 2.2 specificatie.

http://java.sun.com/products/servlet/2.2/javadoc/
Servlet 2.2 API.

http://java.sun.com/products/jsp

Sun's homepage over Java Servlet Pages.

http://216.32.244.50/pub/jsp/11final-87721/jsp1_1-spec.pdf
Java Server Pages 1.1 specificatie.

http://www.cetus-links.org/oo_ole.html
COM/DCOM/COM+/MTS/MSMQ/OLE/ActiveX links.

http://www.comdeveloper.com/articals/COMIdea.asp
Intro in COM.

http://www.dbmsmag.com/9704d13.html
Diepgaande bespreking van DCOM.

http://www.fred.net/brv/chart
Homepage van de gdchart library.

http://www.boutell.com/gd
Homepage van de gd library.