Inhoudsopgave

Deel I

Webtechnologie

Hoofdstuk 1

Dynamische webapplicaties

Het Internet bestaat al lang niet meer uit enkel statische HTML-pagina's. Zowel client-side als server-side is er dynamiek toegevoegd.

Client-side scripting maakt het mogelijk om de layout van een webpagina te veranderen zonder dat er een HTTP-bericht heen en weer gestuurd wordt naar de webserver.

Met server-side scripting kan de functionaliteit van de webserver uitgebreid worden zodat hij niet enkel statische HTML-bestanden kan doorsturen naar de client, maar ook, afhankelijk van de aanvraag van de client, dynamisch een webpagina kan genereren.

1.1 Client-side scripting

Client-side scripts worden toegevoegd in de HTML-code van de webpagina en uitgevoerd door de browser. De belangrijkste taken van client-side scripts is het controleren van de invoer van de gebruiker en het genereren van dynamische webpagina's zonder dat hiervoor HTTP-berichten naar de server gestuurd moeten worden ([Seb08, WEB]). Voorbeelden van het gebruik van client-scripts zijn

- Nagaan of alle gegevens van een HTML-formulier ingevuld zijn voordat het formulier wordt doorgestuurd naar de server. Dit voorkomt nodeloos netwerkverkeer indien niet alle gevraagde gegevens ingevuld zijn.
- Het realiseren van een dynamische layout: uitklapbare menu's, wisselende figuren, de stijl (kleur, ...) van HTML-elementen veranderen, ...

De meest gebruikte taal voor client-side scripts is JavaScript ([ISO, Cha01, Fla06]), maar ook VB-Script en JScript kunnen gebruikt worden als scripttaal. JScript is een aangepaste versie van JavaScript die ontwikkeld werd door Microsoft. VBScript kan enkel gebruikt worden in combinatie met Internet Explorer.

Client-side scripts worden toegevoegd aan HTML-pagina's m.b.v. <script>-tags. Om de webpagina te kunnen manipuleren maken de scripts gebruik van het Document Object Model (DOM, [DOM]). De HTML-pagina wordt voorgesteld als een *Document* uit de DOM-API.

De webpagina in het volgend voorbeeld bestaat uit een HTML-formulier met een tekstveld en een submit-knop. De gegevens van het formulier worden doorgestuurd naar de server als het tekstveld ingevuld is. In het andere geval wordt er een popup-venster met een waarschuwing getoond, maar wordt het formulier niet doorgestuurd.

```
<html>
    <head>
        <script type="text/javascript">
            function controleer_invoer() {
                var verder = false;
                if (document.invoerGegevens.naam.value == "") {
                    window.alert("Naam invoeren!");
                } else {
                    verder = true;
                }
                return verder;
            }
        </script>
    </head>
    <body>
        <form name="invoerGegevens" action="..." onsubmit="return controleer_invoer();">
            Naam: <input type="text" name="naam" size="20"><br>
            <input type="submit" value="Log in" >
        </form>
    </body>
</html>
```

Listing 1.1: voorbeeld javascript controle invoer

De impliciete variabele *document* stelt de webpagina voor. De verschillende componenten van de webpagina vormen een volgens het DOM-model een boomstructuur waarvan de knopen op naam opgevraagd kunnen worden, bv. *document.invoerGegevens.naam*.

De HTML 4 standaard ([HTM]) ondersteunt een aantal gebeurtenissen (*events*) die afgehandeld kunnen worden door Javascript. Voor elke gebeurtenis heeft de HTML-component een bijhorend attribuut (bv. *onsubmit*). Dit attribuut wordt gebruikt om te verwijzen naar de opdrachten die uitgevoerd moeten worden als de gebeurtenis optreedt.

Het volgend stukje code illustreert hoe Javascript in combinatie met CSS ([CSS]) en DOM gebruikt kan worden om de layout van een webpagina te veranderen. In dit geval wordt de kleur van een titel veranderd bij het selecteren van een keuzerondje (*radio button*).

```
<html>
    <head>
        <script type="text/javascript">
            function veranderKleur(kleur) {
                document.getElementById('titel').style.color = kleur;
            }
        </script>
    </head>
    <body>
        <h1 id="titel" style="color:yellow">
            Titel in kleur
        </h1>
        <form action="...">
```

```
            Kies kleur:
            <input type="radio" name="kleur" onclick="veranderKleur('red')"> rood
            <input type="radio" name="kleur" onclick="veranderKleur('blue')"> blauw
            <input type="radio" name="kleur" onclick="veranderKleur('green')"> groen
        </form>
    </body>
</html>
```

Listing 1.2: voorbeeld javascript layout

Merk op dat de methode *getElementById* behoort tot de DOM API.

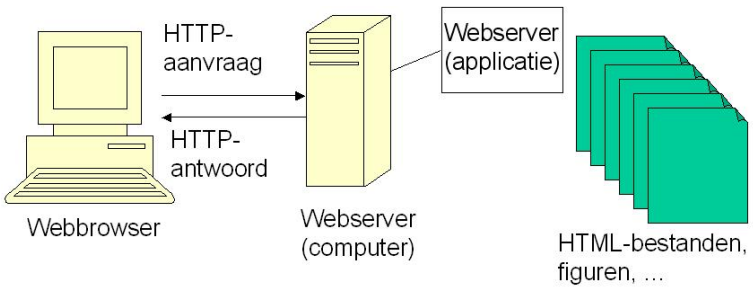
DHTML, Adobe Flash

De combinatie van HTML, CSS en Javascript om dynamische, interactieve en geanimeerde webpagina's te maken wordt vaak DHTML, Dynamic HTML, ([DYNa, DYNb]) genoemd.

Om webpagina's op te fleuren met animaties, filmpjes en spelletjes kan je Adobe Flash gebruiken.

1.2 Server-side scripting

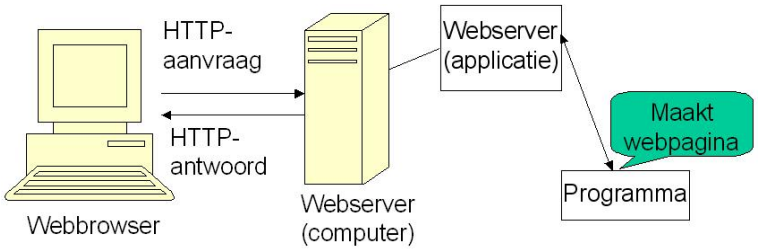
Webserverapplicaties zoals Apache ([APA]) of Microsofts Internet Information Services ([IIS]), voorzien een directory structuur voor elke webapplicatie die ze huisvesten. In die mappen bevinden zich HTML-bestanden, PDF-bestanden, figuren, geluidsbestanden, ... Deze bestanden kunnen opgevraagd worden door de webclients (bv. browsers) door het opgeven van de juiste URL. De taak van de webserverapplicatie is om aan de hand van de URL het juiste bestand te vinden en het door te sturen naar de client verpakt in een HTTP-antwoordbericht (zie figuur 1.1). Merk op de we het woord webserver zowel gebruiken voor de webserverapplicatie als voor de computer waarop deze applicatie draait.



Figuur 1.1: Statische webapplicatie

Indien webpagina's dynamische elementen bevatten, waarvan de waarde niet op voorhand kan opgenomen worden in de HTML-code omdat de inhoud variabel kan zijn (bv. de huidige datum), dan moet een deel van de HTML-pagina bepaald worden op het ogenblik van de aanvraag. Hiervoor doet de webserver beroep op een andere applicatie of programma, zonder dat de client hiervan iets merkt (zie

figuur 1.2). De webserver geeft de parameters van de HTTP-aanvraag door aan het hulpprogramma. Dit programma gebruikt de parameters om dynamisch de HTML-pagina te genereren die dan door de webserver verpakt wordt in een HTTP-antwoord en doorgestuurd wordt naar de client. De parameters van de HTTP-aanvraag kunnen door het hulpprogramma ook gebruikt worden om gegevens te bewaren op de server, bv. in een gegevensbank.



Figuur 1.2: Dynamische webapplicatie

Er bestaat een waaier aan mogelijkheden om deze hulpprogramma's te realiseren. Ze worden ontwikkeld in zowel geïnterpreteerde programmeertalen (bv. ASP, PHP, ...) als in gecompileerde (bv. CGI, servlets, ASP.NET, ...).

1.2.1 De Common Gateway Interface

Er bestaan verschillende manieren om de functionaliteit van bestaande webserversoftware naar eigen wensen uit te breiden. Zo kan je de meeste webservers zodanig configureren dat ze kleine programma's die je zelf hebt geschreven (vaak *scripts* genoemd) automatisch oproepen wanneer een gepaste URL wordt opgevraagd. Deze scripts verwerken dan de clientgegevens en zorgen op deze manier voor dynamische HTML-pagina's. Het schrijven van een dergelijk script is vanzelfsprekend een stuk eenvoudiger dan zelf het volledige HTTP-protocol te implementeren.

Opdat deze scripts goed zouden samenwerken met de webserver, moeten ze een aantal conventies volgen die de server begrijpt, en dan nog liefst conventies die onafhankelijk zijn van de gekozen serversoftware. Gelukkig zijn de softwareproducenten het hier al vrij vroeg in de geschiedenis van het web over een standaard eens geworden: de *Common Gateway Interface* (*CGI*). Programma's die deze conventies volgen, noemt men *CGI-scripts*.

Omdat veel CGI-scripts op het Internet in de programmeertaal PERL zijn gecodeerd, denken vele programmeurs dat dit noodzakelijk is. Je kan CGI-scripts echter even goed in een andere programmeertaal schrijven of zelfs rechtstreeks als UNIX shell-script implementeren.

In principe kan een CGI-script dus ook in Java zijn geschreven. In de praktijk zal men in een Java-omgeving echter streven naar een meer rechtstreekse communicatie met de webserver (of meer precies, met een web*container*) en gaat de voorkeur uit naar servlets (zie hoofdstuk **??**).

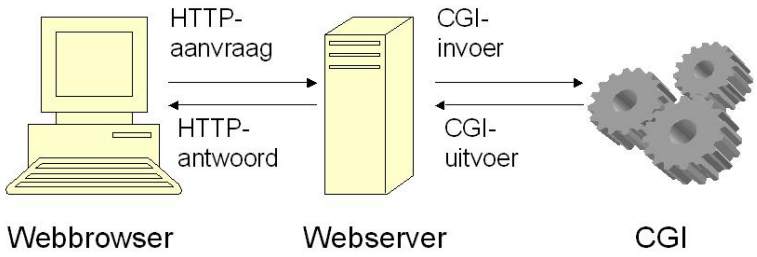
Hoewel het concept van een CGI-script betrekkelijk eenvoudig is, duiken er bij grotere toepassingen

al snel een aantal complicaties op. CGI is eigenlijk enkel geschikt voor haastklussen en prototypetoepassingen en niet voor professioneel werk.

Basiswerking

Een typische CGI-sessie verloopt op de volgende manier:

1. De gebruiker geeft aan zijn browser een URL op, klikt op een link of drukt op de submitknop van een HTML-formulier.
2. Als gevolg hiervan lanceert de HTTP-client een HTTP-aanvraag met methode GET of POST en de gegeven URL. De server verwerkt deze aanvraag en houdt een kopie bij van het corpus en van een aantal gegevens uit de headerlijnen.
3. De server bepaalt welk CGI-script er met de gevraagde URL overeenkomt en start het op in een nieuw proces. De gegevens uit de headerlijnen en het corpus worden aan het script doorgegeven.
4. Het script verwerkt deze gegevens en geeft een resultaat terug aan de server.
5. De server stuurt de gegevens die hij van het script heeft gekregen terug naar de client in de vorm van een HTTP-antwoord.
6. De browser toont dit resultaat op het scherm.



Figuur 1.3: CGI

Hoewel in de praktijk de meeste URLs die met CGI-scripts worden geassocieerd een naam hebben waarin ergens de letters `cgi` voorkomen, is dit absoluut niet noodzakelijk. Je kan aan een URL in principe niet zien of er een CGI-script achter steekt. Het is uiteindelijk de webserversoftware die bepaalt of er een script wordt opgeroepen of niet.

Het CGI-*protocol* bepaalt welke gegevens er naar het CGI-script worden doorgegeven en in welke vorm, en omgekeerd, hoe het CGI-script aan de server duidelijk maakt wat hij naar de client moet terugsturen.

1.2.2 PHP

Een populair en veel gebruikt alternatief voor CGI-scripts is PHP ([PHPb]). PHP is een server-side scripttaal ontworpen om dynamisch webpagina's te creëren. Oorsponkelijk was PHP een procedurele programmeertaal, maar ondertussen is het ook mogelijk om meer objectgeoriënteerd te programmeren.

Een PHP-bestand (met extensie .php) bestaat uit een combinatie van PHP-code en stukken (X)HTML. Als er op de server een HTTP-aanvraag toekomt die gelinkt wordt met dit bestand, dan zal de webserver het PHP-bestand parsen en uitvoeren. Het resultaat van deze actie is een (X)HTML-pagina die met het HTTP-antwoord naar de client wordt verstuurd. De PHP-code wordt dus geïnterpreteerd op de server. Het volgende voorbeeld uit de PHP-tutorial ([PHPa]) toont een eenvoudig voorbeeld van een PHP-bestand.

```
<?php
if (strpos($_SERVER['HTTP_USER_AGENT'], 'MSIE') !== FALSE) {
?>
<p>You are using Internet Explorer</p>
<?php
} else {
?>
<p>You are not using Internet Explorer</p>
<?php
}
?>
```

Listing 1.3: voorbeeld PHP

Het PHP-bestand bestaat uit (X)HTML gecombineerd met PHP-code tussen *<?php* en *?>*. Het resultaat van het parsen en uitvoeren van dit PHP-bestand is een stukje HTML dat aangeeft of je al dan niet Internet Explorer gebruikt als browser. Merk op dat de PHP-syntax gelijkenissen vertoont met zowel C als Perl.

PHP is een van de meest gebruikte talen voor het ontwikkelen van webapplicaties. Het is beschikbaar op de meeste webservers en wordt vaak gebruikt in combinatie met Linux, Apache ([APA]) en MySQL ([MYS]), afgekort als LAMP.

1.2.3 Ruby on Rails

Ruby on Rails ([RAI]) is een framework om webapplicaties te ontwikkelen in de programmeertaal Ruby ([RUB]) op een eenvoudige, snelle en efficiënte wijze. Dit framework is gebaseerd op de Model-View-Controller-architectuur (zie Hoofdstuk 6) voor webapplicaties. Twee belangrijke fundamenten van Ruby on Rails zijn *Conventie boven Configuratie, Convention over Configuration (CoC)* en *Herhaal jezelf niet, Don't repeat yourself (DRY)*.

CoC betekent dat de ontwikkelaar voor de hand liggende configuratie niet hoeft uit te schrijven. Het framework koppelt bijvoorbeeld automatisch een klasse *Book* aan een tabel *books*. Zoals het voorbeeld illustreert zijn webapplicaties ontwikkeld met Ruby on Rails, nauw verbonden met hun gegevensbank.

DRY betekent dat definities slechts op één plaats gespecifieerd moeten worden en dat ze dan overal

beschikbaar zijn. Bovendien is de programmeertaal Ruby een compacte taal. Dit zorgt ervoor dat je in Ruby minder code nodig hebt om een applicatie te schrijven.

1.3 Ajax

Volgens het klassiek patroon van een webapplicatie stuurt de webbrowser een bericht naar de server door op een link te klikken, door op een formulier in te dienen, ... Vervolgens wacht de webbrowser op een antwoord van de server met een nieuw document. Het volledig browservenster wordt dan vervangen door een nieuw document. Deze werkwijze wijkt sterk af van het principe van desktopprogramma's en maakt dat webapplicaties eerder eenvoudig en minder interactief zijn.

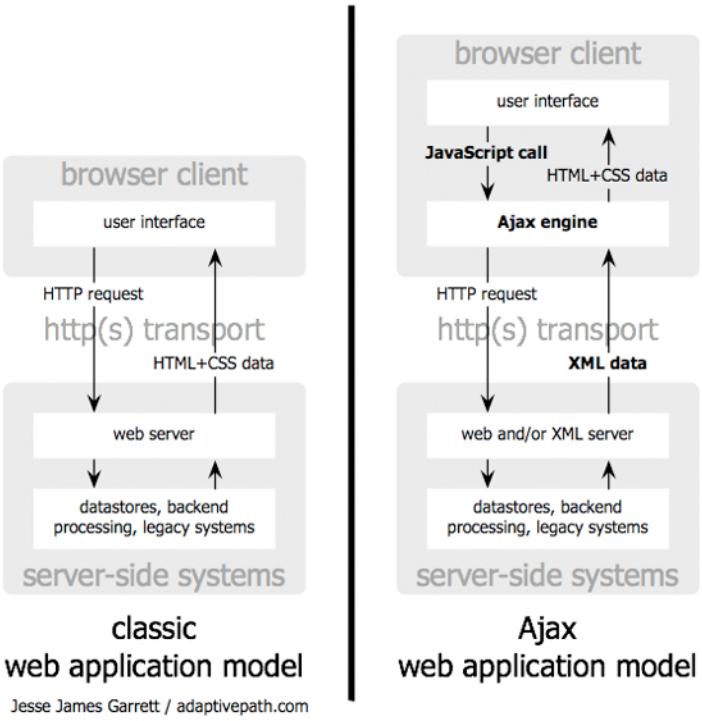
Ajax, een afkorting van Asynchronous Javascript + XML ([Gar05]), biedt hierover een oplossing. Ajax is een combinatie van bestaande technologieën die het mogelijk maken om interactievere en complexere webapplicaties te ontwikkelen, zoals bijvoorbeeld Google Suggest ([GOOb]) en Google Maps ([GOOa]). Ajax maakt gebruik van de volgende technologieën:

- XHTML ([XHT]) en CSS ([CSS]) voor de presentatie van de documenten
- DOM ([DOM]) om een dynamische en interactieve layout te realiseren
- XML ([XMLa]) en XSLT ([XSL]) om gegevens uit te wisselen en te manipuleren
- XmlHttpRequest ([XMLb]) om gegevens asynchroon op te halen
- Javascript als programmeertaal en bindmiddel voor de andere technologieën

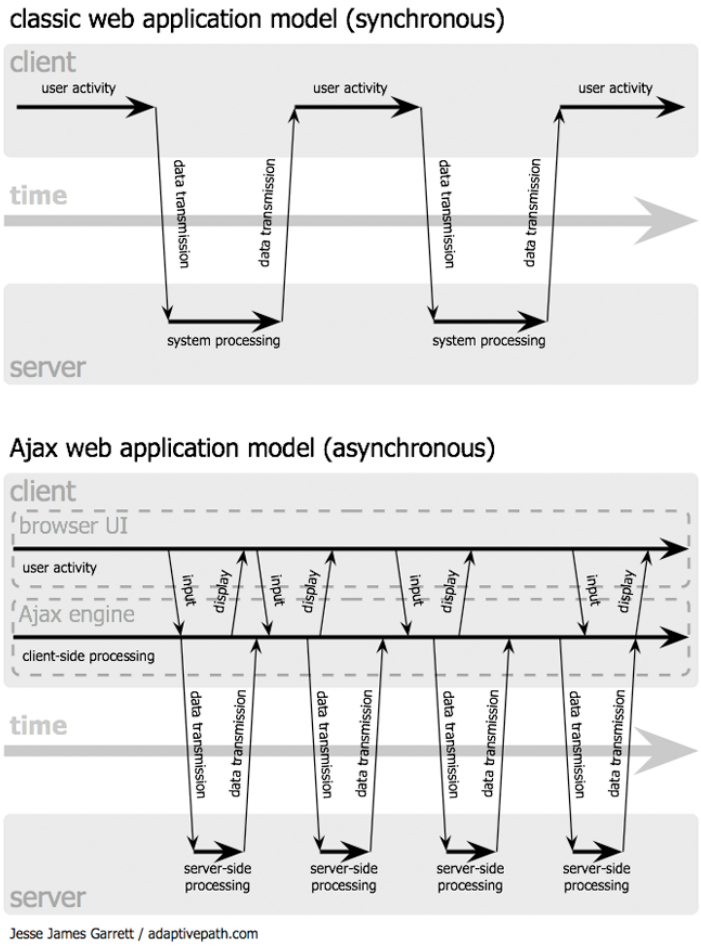
Een Ajax-webapplicatie verschilt op twee manieren van een klassieke webapplicatie. Ten eerste is de communicatie met de webserver asynchroon. Dat betekent dat de browser niet wacht op het antwoord van de server en de gebruiker dus verder kan werken terwijl de server de aanvraag van de browser verwerkt en een nieuw document doorstuurt. Een tweede aanpassing is dat de server slechts een klein stuk van de getoonde webpagina bezorgt, wat minder tijd vraagt om te transporteren en om te tonen. Beide aanpassingen zorgen voor een snellere interactie tussen de browser en de server.

Om deze aanpassingen te realiseren voegt een Ajax-webapplicatie een laag toe tussen de client en de server, een Ajax-engine. Bij de start van een sessie laat de browser, i.p.v. een webpagina, een Ajax-engine. Deze engine is geschreven in Javascript en meestal bewaard in een verborgen frame. Het is de taak van deze engine om de gebruikersinterface te genereren (*render*) en om de communicatie met de server te realiseren. (Zie figuur 1.4 uit [Gar05])

Elke gebruikeractie die normaal een HTTP-aanvraag veroorzaakt wordt nu een Javascript-aanroep van de Ajax-engine. Indien mogelijk (bv. bij validatie of navigatie) handelt de Ajax-engine de aanroep zelf af. In het andere geval stuurt hij asynchroon, zonder dat de gebruiker hiervan hinder ondervindt, een aanvraag naar de server. Figuur 1.5 uit [Gar05] illustreert dat de gebruiker verder kan werken terwijl de Ajax-engine communiceert met de server.



Figuur 1.4: Een klassiek webapplicatie versus een Ajax-webapplicatie



Figuur 1.5: Synchrone versus asynchrone interactie

Hoofdstuk 2

Webtoepassingen

Met de opkomst van computernetwerken en het wereldwijde web ontstond een nieuw soort computersoftware: niet langer enkelvoudige computer*programma's* die op één enkele computer draaien (hetzij micro, mini of mainframe) maar gedistribueerde computer*toepassingen* (engels: *applications*) die bestaan uit verschillende programma's die met elkaar samenwerken en gegevens uitwisselen over een computernetwerk.

In deze tekst bespreken we de softwarearchitectuur van zogenaamde *webtoepassingen*. Zij maken gebruik van dezelfde technologie die aan de basis ligt van het wereldwijde web, in het bijzonder, van de alomtegenwoordige webbrowsers en webservers en het HTTP-protocol waarmee deze programma's gegevens uitwisselen met elkaar.

Om de gedachten te vestigen, behandelen we hier in hoofdzaak slechts twee raamwerken waarmee professionele webtoepassingen kunnen worden ontwikkeld: de veelgebruikte, op Java gebaseerde, *J2EE*-standaard (Java 2 Enterprise Edition) en het *.NET*-raamwerk. Frameworks van andere producenten hanteren vergelijkbare principes en bieden gelijkaardige mogelijkheden. Merk op dat we ook van J2EE en .NET slechts een gedeelte bespreken, in het bijzonder komen de zogenaamde *Enterprise JavaBeans* (*EJBs*) niet aan bod, een technologie die nochtans onmisbaar is bij meer complexe toepassingen.

2.1 Het meerlagenmodel

2.1.1 Algemeen

Bij de discussie van professionele bedrijfssoftware kan men niet voorbijgaan aan het concept van de *meerlagenarchitectuur* (Engels: *multi-tier architecture*). Een voorloper van de meerlagenarchitectuur is de client/server-toepassing.

Client/server-toepassing

Een client/server-toepassing is een voorbeeld van een architectuur met twee lagen: de clientlaag houdt zich hoofdzakelijk bezig met *presentatie* (het afbeelden van de gegevens en de invoer van nieuwe opdrachten) en de serverlaag is verantwoordelijk voor de opslag en de verwerking van de gegevens.

Een typisch voorbeeld van een client/server-toepassing is een centrale gegevensbankserver en een applicatie die toelaat om gegevens te bekijken, aan te passen of te veranderen. Deze applicatie is geïnstalleerd op de computers van de verschillende gebruikers.

Een nadeel van deze configuratie is dat de performantie afhankelijk is van de PC van de gebruiker. Bovendien veroorzaakt ze ook veel netwerkverkeer tussen de verschillende gebruikers en de centrale server. Tot slot impliceren kleine veranderingen aan de applicatie dat deze ook op elke lokale computer doorgevoerd moeten worden of dat verschillende gebruikers andere versies van de applicatie hebben.

3-lagenarchitectuur

De 3-lagenarchitectuur splitst de clientlaag van de vorige architectuur op in twee delen: een presentatielaag en een applicatielaag. Verder wordt ook de serverlaag veralgemeend tot een gegevenslaag die niet langer bestaat uit één gegevensbankserver, maar uit verschillende gegevensbronnen: gegevensbanken, andere informatiesystemen, bestanden, ...

De presentatielaag bestaat uit de grafische gebruikersinterface en de applicatielaag bevat de eigenlijke applicatielogica. Deze laag haalt de gewenste gegevens op vraag van de presentatielaag, controleert data van de gebruiker en verwerkt gegevens.

Een voordeel van deze architectuur is dat verschillende gebruikersinterfaces gebruik kunnen maken van dezelfde applicatielaag, bovendien kunnen de gegevens nu uit verschillende bronnen opgehaald worden.

Meerlagenarchitectuur

De meerlagenarchitectuur is een verfijning van de 3-lagenarchitectuur. De applicatielaag wordt verder opgedeeld in drie stukken: de presentatielogica, de eigenlijke applicatielogica en extra infrastructuur. De presentatielogica is dat stuk van de applicatie dat communiceert met de gebruikersinterface, dat bepaalt hoe een vraag van de gebruiker behandeld moet worden, dat eenvoudige validaties uitvoert enz. De interactie met de datalaag maakt deel uit van de applicatielogica. Extra infrastructuur voorziet in bijkomende functionaliteit voor de toepassing bv. mailing.

Vaak komen de verschillende lagen van een dergelijke architectuur ook overeen met verschillende programma's die op verschillende computers worden uitgevoerd en is de verbinding tussen beide lagen een fysische netwerkverbinding. Anderzijds kan je dit lagenmodel ook meer in abstracte zin beschouwen, en is elke laag een pakket van klassen of componenten die nauw met elkaar vervlochten zijn, terwijl de communicatie tussen beide pakketten gebeurt volgens een welgedefinieerde interface.

Een *interface* is een contract waarin staat welke publieke methoden een object heeft, welke parameters deze methoden aanvaarden en welke gegevens ze weergeven. Bij het ontwerp van een toepassing is het belangrijk de interfaces zo te definiëren dat bij veranderende bedrijfregels die interfaces niet aangepast moeten worden, maar enkele de onderliggende componenten.

2.1.2 Het meerlagenmodel in webtoepassingen

Een webtoepassing kan je op de volgende manier in lagen verdelen:

- Een *clientlaag* die zich vooral bezighoudt met presentatie van de gegevens en opvangen van de invoer van de gebruiker (bijvoorbeeld via toetsenbord en muis). Meestal is het programma dat instaat voor deze laag niets anders dan een standaardwebbrowser.
- Een *weblaag* die ervoor zorgt dat de clientlaag de juiste gegevens te zien krijgt en die reageert op opdrachten van de client. De weblaag zorgt voornamelijk voor de verpakking van de gegevens die het uitwisselt met de volgende laag en beheert ook een aantal statische gegevens (webpagina's). Deze taak wordt uitgevoerd door één of meerdere webservers.
- Een laag met de logica (de *business logic*) van de toepassing. Hier worden de gegevens feitelijk verwerkt en opgeslagen (zonder dat er met de presentatie hoeft worden rekening gehouden). Databankservers horen thuis in deze laag en ook de Enterprise JavaBeans.

Niet alle auteurs gebruiken dezelfde onderverdeling in lagen. Soms wordt de business logic nog opgesplitst in diverse subdomeinen of maakt men onderscheid tussen het verwerken van de gegevens en het opslaan ervan. Deze laatste laag noemt men ook de *EIS*-laag, een afkorting voor de term '*Enterprise Information Systems*' (databanken, oudere informatieverwerkende systemen, en andere bronnen).

De grens tussen client- en weblaag is tegelijk een *fysische* grens: meestal bevindt de clientbrowser zich op een andere computer dan de webserver. Voor kleine en middelgrote toepassingen is de grens tussen de andere twee lagen niet zo scherp: webserver en databank draaien vaak op dezelfde machine, of in sommige toepassingen wordt er geen echte databank gebruikt maar worden gegevens opgeslagen en verwerkt door de programmatuur van de webserver zelf. Dit laatste eigenlijk echter enkel aan te raden voor kleinere toepassingen.

Hier is niet de plaats om verder in te gaan op de voor- en nadelen van een dergelijke meerlagenarchitectuur, voor meer details verwijzen we naar een cursus softwareontwikkeling.

2.2 Webtechnologie in een Java-omgeving

De J2EE-standaard voorziet een raamwerk voor dit meerlagenmodel, gebaseerd op de programmeertaal Java. Merk op dat ook andere producenten dan Sun Microsystems deze J2EE-specificaties volgen in hun producten.

Centraal voor J2EE zijn de begrippen *component* en *container*. *Componenten* zijn kleine programmaonderdelen (Java-klassen, of exacter, objecten die tot deze klassen behoren) die bepaalde functionaliteiten van de webtoepassing voor hun rekening nemen. Ideale componenten zijn 'herbruikbaar': ze kunnen ook dienst doen als onderdeel van andere toepassingen. Vaak hebben componenten parameters die je kan instellen zonder aan de programmacode te raken en zonder de klassen te moeten hercompileren.

Een zogenaamd *applet* is een typisch voorbeeld van een dergelijke component. Een applet is een klein Java-programmaatje dat wordt uitgevoerd binnenin een webbrowser en waarvan de grafische uitvoer als onderdeel van de webpagina wordt afgebeeld. Met behulp van een applet kan je een browser Java-code laten uitvoeren zonder dat je daarom het browserprogramma zelf moet aanpassen.

Containers zijn *run time*-omgevingen waarin componenten worden uitgevoerd. Zij zorgen voor de noodzakelijke 'lijm' voor de interactie tussen deze componenten en bieden bepaalde *diensten* aan deze componenten aan. Dit gaat van levenscyclusbeheer van componenten tot netwerkondersteuning voor hun onderlinge communicatie.

Een webbrowser speelt bijvoorbeeld de rol van appletcontainer. Hij zorgt dat het appletobject wordt gecreëerd wanneer een webpagina wordt opgeroepen die naar dit applet verwijst en geeft parameters door aan het applet die in de HTML-pagina zijn verwerkt.

Opdat containers en componenten naadloos zouden kunnen samenwerken, moeten ze aan een aantal afspraken voldoen. In de praktijk betekent dit dat de corresponderende Java-klassen een bepaalde interface moeten implementeren of een bepaalde klasse uitbreiden. De J2EE-standaard is in wezen niet veel meer dan een specificatie van deze interfaces en van het gedrag dat componenten en containers moeten vertonen.

De taak van de webprogrammeur bestaat erin om componentenklassen te ontwikkelen die de functionaliteit van de webtoepassing implementeren en om de containers zodanig te configureren dat de juiste componenten op het juiste moment worden ingezet. (De containersoftware zelf is een bestaand softwareproduct en kan tegelijkertijd voor verschillende webtoepassingen worden ingezet.)

We geven een kort overzicht van de verschillende Java-producten die voor ons van nut zijn en die verder in deze tekst aan bod komen.

2.2.1 De clientlaag

Omdat de hoofdrolspeler in de clientlaag de webbrowser is van de gebruiker, met andere woorden, omdat je voor dit gedeelte van de toepassing weinig effectief hoeft te programmeren, is Java meestal niet zo sterk aanwezig in de clientlaag. Als ontwikkelaar moet je hoofdzakelijk op de hoogte te zijn van HTML en eventueel een client-scripttaal zoals Javascript (die behalve de naam trouwens niets met Java heeft te maken). Andere technologieën die hier een rol spelen zijn *style sheets* (CSS en XSLT) en in toenemende mate ook XML.

Als clientcomponenten biedt Java hier applets aan, maar in bedrijfsstoepassingen spelen deze meestal een mindere rol (ze zijn vooral populair omwille van hun grafische mogelijkheden). Zoals reeds vermeld, doet de webbrowser hier dienst als clientcontainer.

In sommige gevallen gebruikt men als client een afzonderlijke losstaande Java-toepassing. Die communiceert dan rechtstreeks met de weblaag (of met een dieper gelegen laag, hoewel je dan nog moeilijk van een webtoepassing kan spreken). Dergelijke clienttoepassingen bieden heel wat flexibiliteit, maar vragen meestal een langere ontwikkelingstijd en bovendien hebben ze het nadeel dat ze bij elke gebruiker afzonderlijk moeten worden geïnstalleerd (terwijl iedereen wel een browser bezit).

Tot slot vermelden we nog dat de client ook in een andere taal kan geschreven worden dan Java. Communicatie met de andere toepassingslagen gebeurt immers via welbekende netwerkprotocollen (HTTP in het bijzonder) die door vele platformen en programmeeromgevingen worden ondersteund.

2.2.2 De weblaag

De weblaag wordt bevolkt door Java *servlets* en *JSP*-pagina's, componenten die draaien onder de hoede van een webcontainer. Daarnaast gebruikt een webtoepassing ook vaak statische webpagina's die eventueel door een traditionele webserver worden beheerd. Moderne webserversoftware doet meestal tegelijk dienst als webcontainer (of omgekeerd), of biedt een eenvoudige manier om met een alleenstaande webcontainer gegevens uit te wisselen.

Communicatie tussen weblaag en clientlaag gebeurt hoofdzakelijk via het HTTP-protocol, het standaardprotocol voor het wereldwijde web. Aangezien het de webcontainer is die zelf deze communicatie beheert, blijven de meeste protocoldetails gelukkig verborgen voor de servlet- of JSP-programmeur.

2.2.3 Business logic

Bij een echte J2EE-toepassing wordt de *business logic* verzorgd door de EJBs. Bij vele webtoepassingen is dit 'geschut' echter te 'zwaar'.

In deze tekst beperken we ons tot toepassingen waarbij de logica kan worden verzorgd door Java klassen (en 'gewone' JavaBeans). Dit zijn eigenlijk geen componenten in de betekenis die we er hierboven aan gegeven hebben en ze hoeven ook niet door een afzonderlijke container te worden beheerd (in tegenstelling tot EJBs waarvoor je een EJB-container nodig hebt).

Communicatie tussen klassen en databank gebeurt met behulp van *JDBC*, een Java-API die onder andere op SQL is gebaseerd, en die onafhankelijk is van de gekozen databanksoftware.

2.2.4 Java API's

In deze paragraaf overlopen we kort de beschikbare API in J2EE. De JDBC-, Servlet- en JSP-API worden verder in deze nota's besproken.

Entreprise Java Beans (EJB)

EJB's zijn componenten die een stuk applicatielogica beschrijven en die bij voorkeur herbruikbaar zijn. De EJB-technologie maakt het mogelijk om veilige, schaalbare en persistente componenten te maken die deel kunnen uitmaken van transacties.

Een EJB bestaat uit een Java-klasse en een XML-bestand. EJB's kunnen gebruikt worden vanuit andere JVM's.

Java Mail

Deze API voorziet interface naar een mailserver.

Java Message Service (JMS)

JMS wordt gebruikt om op een asynchrone wijze boodschappen te versturen en te ontvangen. Het versturen van boodschappen kan volgens twee modellen verlopen: *één naar veel* of *punt naar punt*. In het eerste geval worden boodschappen naar topics gestuurd en kunnen clients aangeven dat ze in bepaalde topics geïnteresseerd zijn. Bij 'punt naar punt' communicatie worden boodschappen verstuurd via een wachtrij.

Java Naming and Directory Interface (JNDI)

JNDI maakt communicatie met hiërarchische gegevensbanken (naming/directory-service) eenvoudiger. Hiërarchische gegevensbanken zijn geoptimaliseerd voor snelle leestoegang. Enkele voorbeelden zijn IP-adressen, bestandssysteem, DNS, enz.

Java Transaction API (JTA)

Met JTA kan men transacties naar verschillende systemen zoals gegevensbanken, SAP, enz. implementeren.

Remote Method Invocation (RMI)

Met RMI kan men communicatie tussen objecten in verschillende JVM's realiseren. De objecten gedragen zich alsof ze lokaal zijn voor de applicatie, ze kunnen aangesproken worden zoals de lokale objecten. Deze technologie maakt gebruik van sockets over TCP/IP. RMI-IIOP (Inter-ORB Protocol) is een uitbreiding van de RMI-API die compatibel is met CORBA (Common Object Request Broker Architecture).

2.3 Webtoepassingen in het .NET-framework

Het .NET-framework is een nieuwe infrastructuur van Microsoft om het ontwikkelen en beheren van applicaties in een internetomgeving te vereenvoudigen. Net zoals bij J2EE is het de bedoeling dat de onwikkelaar zich kan concentreren op het maken van de eigenlijk applicatie en dat het raamwerk voor de nodige infrastructuur en diensten zorgt waarvan de applicatie gebruik kan maken. De *Common Language Runtime* en de *.NET Framework Class Library* vormen samen de kern van het .NET-platform.

2.3.1 Common Language Runtime

Deze runtime is verantwoordelijk voor het uitvoeren en beheren van stukjes programmacode. Dit omvat onder andere geheugen- en threadsbeheer, veiligheid, levensloopbeheer enz.

Eén van de kernmerken van de Common Language Runtime is dat zij bruikbaar is voor verschillende programmeertalen. De broncode in een programmeertaal wordt gecompileerd naar een bestand in IL(Intermediate Language)-code. De eerste keer dat die code uitgevoerd moet worden compileert de JIT(Just in Time)-compiler de IL-code naar machinecode. Deze machinecode blijft in het geheugen voor eventuele volgende aanvragen.

2.3.2 .NET Framework Class Library

Het .NET-framework voorziet ook een bibliotheek herbruikbare klassen, interfaces, enz. die in verschillende programmeertalen (lees JScript, VB en C#) gebruikt kan worden. Deze bibliotheek bevat pakketten om webapplicaties te ontwikkelen (ASP.NET), om toegang tot relationele gegevensbanken te voorzien (ADO.NET), om windowstoepassingen te maken (Windows Forms). Uiteraard bevat de bibliotheek ook pakketten met basisfuncionaliteiten zoals invoer/uitvoer, strings, enz.

Hoofdstuk 3

ASP.NET vervolg

ASP.NET: vervolg

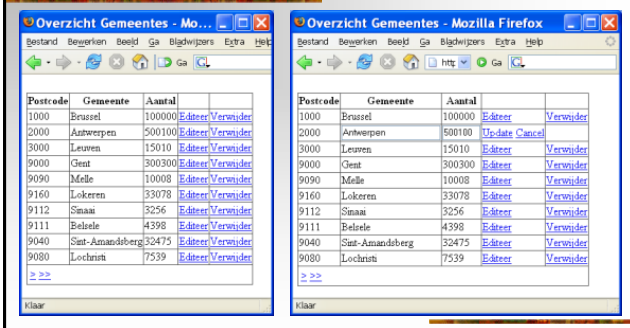
Veerle Ongenae

Overzicht

- Wat is ASP.NET?
- Gebeurtenissen
- Servercomponenten
- Statusinformatie bewaren
- Editeerbare tabellen in ASP-pagina's
- Levensloop
- Beveiliging
- Sjablonen en Navigatie

Voorbeeld Editeerbare tabellen

- Webapplicatie Gemeente
  - overzicht.aspx



Editeerbare tabellen

- Sinds Visual Studio 2005
- Zonder code te schrijven
- Combinatie
  - GridView
  - DataSourceControl
- Moeilijkere scheiding verschillende lagen in applicatie

Aanmaak Editeerbare GridView

- GridView-component
  - Attributen/eigenschappen
    - AutoGenerateColumns
      - False: zelf kolommen en bijhorende eigenschappen bepalen
    - DataSourceID
      - Id van de databron (type: DataSourceControl)
    - DataKeyNames
      - Kolomnamen van de primaire sleutel(s)
  - Kindelement/eigenschap Columns
    - Beschrijving van de kolommen

Beschrijving kolommen

- Kindelement/eigenschap Columns
  - BoundField toevoegen
    - HeaderText: Tekst hoofding
    - DataField
      - Corresponderende kolom in de tabel
    - Read only?
  - CommandField toevoegen
    - Opschrift knop wijzigen
    - "Knop" toevoegen

DataSourceControl

- Component
  - Gebruikt om te binden aan servercomponenten
  - Toegang tot gegevensbron
- Drie types in ASP.NET
  - SqlDataSource
    - Beschikbaar voor gegevensbank waarvoor een .NET-provider bestaat
  - AccessDataSource
    - Access-bestanden
  - XmlDataSource
    - XML-gegevens

AccessDataSource

- Attributen/eigenschappen
  - DataFile
    - Pad naar bestand
  - SelectCommand
    - SQL-string om gegevens te selecteren
  - UpdateCommand
    - SQL-string om gegevens aan te passen
  - DeleteCommand
    - SQL-string om gegevens te verwijderen

Opmerking Access

- De ASP.NET gebruiker moet schrijf- en wijzig-rechten hebben op het access-bestand en op de map waarin het bestand staat

GridView: Paging

- Eigenschap/attribuut AllowPaging
  - True
- Eigenschap PagerSettings
  - Eigenschap Mode
    - Volgende-Vorige of Met nummers
  - Eigenschap NextPageText
    - >
  - Eigenschap PrevPageText
    - <

Overzicht

- Wat is ASP.NET?
- Gebeurtenissen
- Servercomponenten
- Statusinformatie bewaren
- Editeerbare tabellen in ASP-pagina's
- Levensloop
- Beveiliging
- Sjablonen en Navigatie

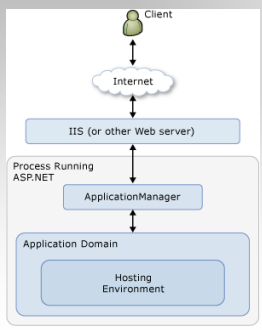
Levensloop

- Levensloop webapplicatie
- Levensloop ASP.NET Page (webformulier)
- Levensloop component

Levensloop webapplicatie

- Starten applicatie-omgeving
- Voor elke HTTP-aanvraag
  - Aanmaak HTTP-objecten
  - Toewijzen HttpApplication-object
  - Aanvraag afhandelen
- Afsluiten applicatie

Starten applicatie-omgeving

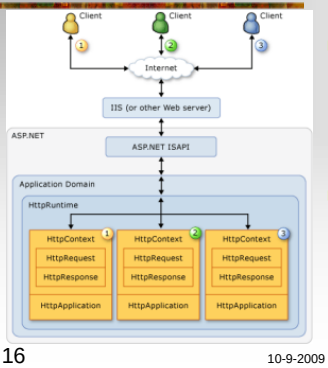


- ApplicationManager
  - Aanmaken en initialiseren webapplicaties
  - Beheer levensloop
    - Webapplicatie
    - Objecten in de webapplicatie
- Per webapplicatie één Application Domain
  - HostingEnvironment
    - Informatie over de webapplicatie
      - Bv. pad

Aanmaak HTTP-objecten

- Aanmaak objecten die de HTTP-aanvraag voorstellen
  - HttpContext
    - Alle HTTP-informatie van de huidige aanvraag
    - bevat
      - HttpRequest
      - HttpResponse

Toewijzen HttpApplication



- Aanmaak HttpApplication-object
  - Instantie van Global.asax, indien beschikbaar
- HttpApplication-object
  - Handelt één aanvraag af
  - Kan hergebruikt worden

Aanvraag afhandelen

- Een hele reeks gebeurtenissen worden opgeroepen en afgehandeld
- Gebeurtenissen die je zelf kan afhandelen
  - Maken deel uit van Global.asax
    - Application_Start
      - Wordt maar één keer uitgevoerd: bij het opstarten van de applicatie
    - Application_End
      - Wordt maar één keer uitgevoerd: bij het afsluiten van de applicatie
    - …

Levensloop

- Levensloop webapplicatie
- Levensloop ASP.NET Page (webformulier)
- Levensloop component

Levensloop Page-object (webformulier)

- Bij elke aanvraag wordt een Page-object aangemaakt
- Het Page-object bevat de verschillende componenten (controls)
- Na het doorsturen van het antwoord wordt het Page-object verwijderd
- Wat zijn de verschillende stappen in de levensloop van dit Page-object?

Levensloop webformulier

- Start
  - aanmaak Page-object, instellen eigenschap IsPostBack
- Initialisatie
  - Toevoegen componenten aan webformulier
- Laden
  - Herstellen eigenschappen componenten (uit verborgen formulierelementen en HTTP-parameters)
- Validatiefaze
  - Controles van de verschillende componenten uitvoeren (methode Validate)

Levensloop webformulier (vervolg)

- Afhandelen events
  - Oproepen eventhandlers van de verschillende componenten
- Renderen
  - Statusinformatie verschillende componenten bewaren
  - Uitvoer genereren
- Opruimen pagina

Bijhorende events (van Page)

- PreInit
- Init
- Load
  - Gebruik: instellen eigenschappen componenten
- Events van de componenten
- PreRender
- UnLoad

Koppelen methodes

- Hoe eventhandlers koppelen aan de events van het webformulier (Page-object)?
- Afhankelijk van het attribuut AutoEventWireup (@Page-richtlijn)
  - true (standaard)
    - Methode met naam Page_*Event* toevoegen aan Code Behind
  - false
    - Methode schrijven in Code Behind
    - Methode zelf toevoegen aan event

Levensloop

- Levensloop webapplicatie
- Levensloop ASP.NET Page (webformulier)
- Levensloop component

Levensloop component

- Initialisatie
- Ophalen statusinformatie
- Verwerken veranderingen data
- Laden
- Veranderingen melden
- Afhandelen gebeurtenissen
- Aanpassingen uitvoeren
- Statusinformatie bewaren
- Uitvoer genereren
- Opruimen
- Verwijderen

Overzicht

- Wat is ASP.NET?
- Gebeurtenissen
- Servercomponenten
- Statusinformatie bewaren
- Editeerbare tabellen in ASP-pagina's
- Levensloop
- Beveiliging
- Sjablonen en Navigatie

Beveiliging

- Afschermen delen van een website
- Twee stappen
  - Authenticatie
    - Gebruiker "herkennen"
    - De gebruiker is wie hij zegt te zijn
    - Inloggen
  - Authorisatie
    - Bepaalde rechten (niet) toekennen aan bepaalde gebruikers
      - Leden kunnen meer pagina's bekijken dan niet-leden
      - Administrators kunnen nog andere pagina's raadplegen

Authenticatie

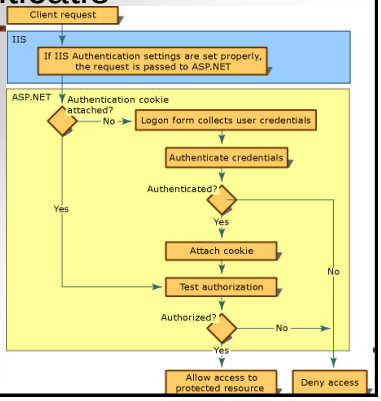
- Verkrijgen identificatiegegevens gebruiker
  - Bv. Login en wachtwoord
- Valideren van deze gegevens
  - Gegevensbank, XML-bestand, Windows

Authenticatie in ASP.NET

- Verschillende mogelijkheden
  - Windows
    - Windowsgebruiker
  - Forms
    - Met een loginformulier
  - Passport
    - Microsoft paspoort
  - None
    - Geen authenticatie
- Instellen in Web.config

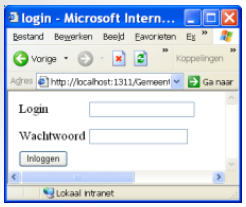
```
<authentication mode="Forms"/>
```


Forms Authenticatie



Voorbeeld inloggen

- Webapplicatie Gemeente
  - login.aspx
  - Web.config
- Loginformulier wordt getoond bij een aanvraag van een willekeurige pagina van de applicatie



Forms Authenticatie

- Niet geauthenticeerde aanvragen
  - Naar inlogformulier
- Inloggen gelukt
  - Cookie (al dan niet persistent)
  - Volgende aanvragen niet meer inloggen

Forms Authenticatie

- Web.config

```
<authentication mode="Forms">
   <forms loginUrl="login.aspx"/>
   </authentication>
```

- login.aspx
  - Controle OK

```
FormsAuthentication.RedirectFromLoginPage(L
  oginNaam.Text,False);
```

Geen persistente cookie

Ingebouwde Forms Authenticatie

- Web.config

```
<authentication mode="Forms">
   <forms loginUrl="LoginPagina.aspx"/>
</authentication>
```

- LoginPagina.aspx
  - Login-component
    - Loginnaam, wachtwoord, …
    - Configuratie via eigenschappen/attributen
  - ValidationSummary
    - Toon foutboodschappen

Aanmaken gebruikers

- Via Web Site Administration Tool
  - Onderliggende gegevensbank in SQL Server Express
  - DB-bestand automatisch aangemaakt in de map App_Data
- Maakt gebruik van de AspNetSqlMembershipProvider
  - Aanmaken en beheren gebruikers
  - …

Windows Authenticatie

- Gebruiker moet ook gebruiker van de servercomputer zijn
- Toegang wordt dan bepaald door de rechten die de gebruiker op de servercomputer heeft

Authorisatie

- Bepalen of een gebruiker al dan niet toegang heeft tot een bepaalde pagina
- Verschillende mogelijkheden
  - Bestandsniveau
    - Windows authenticatie nodig
  - URL-authorisatie
    - In Web.config
      - authorization-element
        - allow
        - deny

URL-authorisatie

- <[element] [users] [roles] [verbs]/>
  - Element
    - allow
    - deny
  - Users
    - Gebruikersnamen gescheiden door ,
    - * : alle gebruikers
    - ? : anonieme gebruikers
  - Roles
    - Gebruikersgroepen
  - Verbs
    - GET, HEAD, POST, … (soorten HTTP-aanvragen)

URL-authorisatie: Web.config

```
<authorization>
    <deny users="?" />
</authorization>
```

- Wie ingelogd is mag de pagina's bekijken
- De eerste regel die toegepast kan worden, wordt gebruikt.
  - Volgende regels worden dan genegeerd

Authorisatie

- ASP.NET
  - NTFS file permissions
  - XML-bestand
    - Gebruikers
    - Rollen
    - HTTP types
- IIS
  - Hostname/IP-adres

Authorisatie voor verschillende rollen

- Web.config bestanden in verschillende map
  - Neemt instellingen bovenliggende map over
  - Voegt eigen instellingen toe

Meer invloed op loginproces

- Events van de Login-component afhandelen
  - LoggingIn: voor controle door component
    - bv. nakijken of gebruikersnaam een email is
  - LoggedIn: na controle door component
  - LoginError: fout bij controle door component
    - Bv. extra links voor hulp tonen
  - Authenticate: zelf controle uitvoeren

Andere Security-componenten

- LoginStatus
  - Toont een link login (of logout) afhankelijk van de inlogstatus van de gebruiker
- LoginView
  - Biedt de mogelijkheid om ingelogde gebruikers andere informatie te tonen dan niet-ingelogde gebruikers
- PasswordRecovery
  - Gebruiker is wachtwoord vergeten
    - Stuurt email met (nieuw?) wachtwoord indien juist antwoord op wachtwoordvraag

Voorbeeld afhandelen Authenticate-event

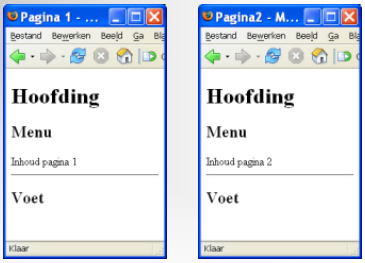
```
protected void
   LoginComponent_Authenticate(object
   sender, AuthenticateEventArgs e) {
       e.Authenticated = … // true of false
}
```


Overzicht

- Wat is ASP.NET?
- Gebeurtenissen
- Servercomponenten
- Statusinformatie bewaren
- Editeerbare tabellen in ASP-pagina's
- Levensloop
- Beveiliging
- Sjablonen en Navigatie

Sjablonen

- Uniforme layout voor alle pagina's van de webapplicatie
- Webapplicatie VbSjabloon
  - MasterPage.master
  - pagina1.aspx
  - pagina2.aspx



Master pages (sjabloon)

- Uniforme layout voor alle pagina's van de webapplicatie
- Layout is bepaald door een master page
  - .master-bestand
  - Ipv een page-richtlijn is er een master-richtlijn

```
<%@ Master Language="C#"
   CodeFile="MasterPage.master.cs"
   Inherits="MasterPage" %>
```


Inhoud master page

- HTML
- Servercomponenten
- Eén of meerdere ContentPlaceHolder-componenten
  - Definiëren een gebied waar variabele inhoud komt
  - Die "variabele inhoud" wordt beschreven in gewone ASP.NET-pagina's

ContentPlaceHolder

- In master page

<asp:contentplaceholder id="Inhoud" runat="server" />

- In ASP.NET-pagina's

<asp:Content ID="Content1" ContentPlaceHolderID="Inhoud" Runat="Server">Inhoud pagina 1 </asp:Content>

- De uitvoer van de master page en de ASP.NET-pagina worden gecombineerd tot één antwoord-pagina

Link master page en inhoud

- Per ASP.NET-pagina

<% @ Page Language="C#" MasterPageFile="~/Master.master" Title="Content Page 1" %>

- Globaal voor de hele applicatie
  - In Web.config

<pages masterPageFile="MySite.Master" />

  - Sjabloon wordt toegepast voor alle webformulieren met Content-componenten

Toegang tot leden van de master page

- Koppeling tussen webformulier en master

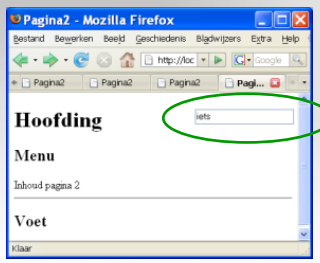
<%@ Page masterPageFile="~/MasterPage.master"%>

- Type van de master declareren

<%@ MasterType virtualPath="~/MasterPage.master"%>

- Eigenschap Master van Page
  - Referentie naar de master page
  - Oproepen leden en componenten van de master page is dan mogelijk

Voorbeeld

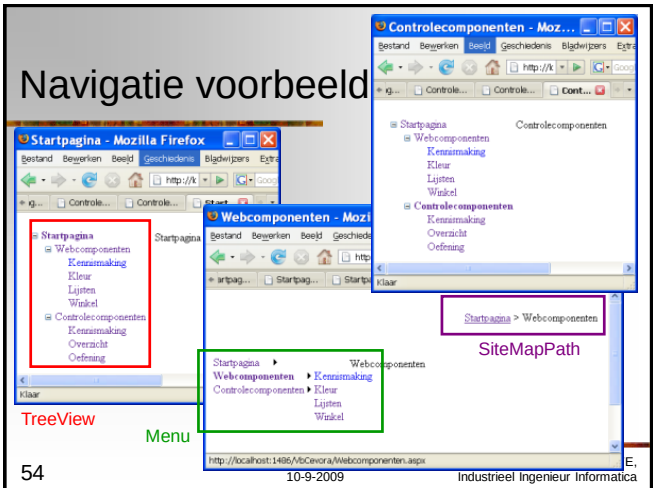


Ingevuld door pagina2.aspx
Deel masterpage

Navigatie

- Consequent en overzichtelijk
- Ingebouwd in ASP.NET
- Drie stappen
  - Structuur applicatie
    - Bestand Web.sitemap
  - Structuur inlezen in een data-object
    - SiteMapDataSource-object definiëren
  - Data-object tonen in een navigatiecomponent
    - TreeView
    - Menu

Navigatie voorbeeld



Structuur vastleggen

```
<?xml version="1.0" encoding="utf-8" ?>
<siteMap xmlns="http://schemas.microsoft.com/AspNet/SiteMap-File-1.0" >
    <siteMapNode url="~/Start.aspx" title="Startpagina“
      description="Startpagina">
        <siteMapNode url="~/Webcomponenten.aspx" … >
            <siteMapNode url="Hallo.aspx" title="Kennismaking" … />
            <siteMapNode url="Kleur.aspx" title="Kleur" … />
            …
        </siteMapNode>
        <siteMapNode url="~/Controlecomponenten.aspx" … >
            <siteMapNode url="KeuzeMetControle.aspx" … />
            …
        </siteMapNode>
    </siteMapNode>
</siteMap>
```


Structuur vastleggen

- XML-bestand **Web.sitemap**
- Basiselement (root element) **siteMap**
  - Eén kindelement: **siteMapNode** – element
    - Kan opnieuw (meerdere) **siteMapNode**-elementen bevatten
    - Boomstructuur
    - Stellen een pagina van de webapplicatie voor

Structuur → data-object

- Door XmlSiteMapProvider
- Resultaat
  - SiteMapDataSource-object
  - Kan gebonden worden aan navigatiecomponenten
- Hoe?
  - Grafisch SiteMapDataSource-component (uit ToolBox → Data) op aspx-pagina plaatsen

Navigatiecomponenten

- Navigatiecomponenten
  - **TreeView**
  - **Menu**
  - **SiteMapPath**
    - Pad naar pagina
- ToolBox → Navigation
- Attribuut/Eigenschap **DataSourceID**
  - Verwijst naar het **ID**-attribuut van het **SiteMapDataSource**-object
- Geselecteerd menu in **TreeView** kan in vetjes
  - Aanpassen eigenschap **SelectedNodeStyle**

Opmaak **Menu**

- Zichtbare menu-items: statisch
- Onzichtbare “uitklapbare” menu-items: dynamisch
- Verschillende stijlen editeerbaar o.a. via properties-venster
  - **StaticMenuItemStyle, StaticMenuStyle, StaticSelectedStyle**
  - **DynamicMenuItemStyle, DynamicMenuStyle, DynamicSelectedStyle**

Installatie van een ASP.NET-applicatie

- Menu Website
  - Copy Website
- Applicatie aanmaken in IIS

Structuur webapplicatie

- aspx-bestanden, aspx.cs-bestanden, ascx-bestanden, ascx.cs-bestanden, asax-bestanden, Web.config
  - App_Code
    - Alle andere codebestanden
  - App_Data
    - Bv. Lokale gegevensbanken
  - …

Informatie over ASP.NET

- ASP.NET Web Applications in the .NET Framework (http://msdn.microsoft.com/en-us/library/655cec97.aspx)
- Microsoft ASP.NET QuickStarts Tutorial (http://quickstarts.asp.net/QuickStartv20/aspnet/Default.aspx)
- The Official Microsoft ASP.NET Site (http://www.asp.net/)

Informatie over ASP.NET

- ASP.NET syntax (http://msdn.microsoft.com/en-us/library/fy30at8h.aspx)
- MSDN Code Gallery (http://code.msdn.microsoft.com/)

Hoofdstuk 4

Object Relational Mapping

ORM (OBJECT RELATIONAL MAPPING)

Veerle Ongenae

Inhoud

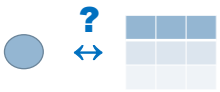
2

- Wat is ORM?
- Objecten afbeelden
  - Eigenschappen
  - Overerving
  - Relaties
- Nhibernate
  - Session en SessionFactory
  - Objecten ophalen en bewaren

ORM (Object Relational Mapping)

3

- Applicatie
  - Software
    - Object technologie (bv. Java, C#, …)
  - Data
    - Relationele gegevensbanken
- Afbeelding (mapping) tussen
  - Data in objecten
  - Data in tabellen
- Hoe en hoe realiseren?



Waarom ORM?

4

- JDBC – ADO.NET
  - tijdsintensief
- Gebruik DataSets
  - Beperkte controle bij compileren
- Data uit gegevensbank rechtstreeks binden aan componenten (.NET)
  - GridView …
  - Dichte koppeling
    - Front end
    - Gegevensbank
  - Groeiende applicatie
    - Onderhoud = nachtmerrie
- Scheiding datalaag en eigenlijke applicatie
  - Losse koppeling
  - Beter onderhoudbaar
  - Opsplitsbaar → werken in verschillende teams

Mogelijke implementaties

5

- Java ↔ DB
  - Hibernate (www.hibernate.org)
  - Java Persistence API (http://java.sun.com/javaee/5/docs/tutorial/doc/bnbpy.html en http://java.sun.com/javaee/technologies/persistence.jsp)
  - …
- .NET ↔ DB
  - NHibernate (C#) (http://www.nhibernate.org)
  - LINQ
    - Deel van het .NET-platform
  - …

Gebruik NHiberate in .NET

6

- DLL's downloaden en toevoegen aan project (in bin-map)
- web.config aanpassen
  - Informatie gegevensbank
    - Provider
    - Connectiestring
    - …
  - NHibernate parameters
  - (Eventueel) loging: log4net

Inhoud

7

- Wat is ORM?
- Objecten afbeelden
  - Eigenschappen
  - Overerving
  - Relaties
- Nhibernate
  - Session en SessionFactory
  - Objecten ophalen en bewaren

Object ↔ Tabel ??

8

- Eigenschap v.e. object → nul of meerdere kolommen
  - Nul?
    - Sommige informatie moet niet bewaard worden
    - Vb. Gemiddelde
      - Berekend op basis van informatie uit DB
  - Meer dan één?
    - Attribuut = object
      - Heeft op zijn beurt attributen
    - Vb. Persoon-object heeft als attribuut een Adres-object

Schaduwinformatie

9

- Shadow Information
- Data
  - Nodig om informatie te bewaren (in DB)
    - Bv. Primaire sleutel
  - Geen deel van business model

Afbeelding bepalen

10

- NHibernate
  - XML
    - Één bestand per object (of per verzameling van objecten)
    - .hbm.xml-bestand
  - Attributen

Eigenschap object ↔ één kolom

11

```
public partial class ProductOrder {
    private int? _productOrderID;
    private int? _customerID;
    private string _productName;
    private int _quantity;
    private decimal _totalCost;
    // Property Definitions...
}
```

ProductOrder
ProductOrderID
CustomerID
ProductName
Quantity
TotalCost

DDL

```
<?xml version="1.0" encoding="utf-8" ?>
<hibernate-mapping xmlns="urn:nhibernate-mapping-2.2">
  <class name="ProductOrder, __Code" table="ProductOrder">
    <id name="ProductOrderID" column="ProductOrderID"
        type="Int32" unsaved-value="null">
      <generator class="native" />
    </id>
    <property name="CustomerID" column="CustomerID"
     type="Int32" />
    <property name="ProductName" column="ProductName"
          type="String" />
    <property name="Quantity" column="Quantity" type="Int32" />
    <property name="TotalCost" column="TotalCost" type="Decimal"
    />
  </class>
</hibernate-mapping>
```


ID

12

```
<id name="ProductOrderID" column="ProductOrderID"
      type="Int32" unsaved-value="null">
    <generator class="native" />
</id>
```

- <id> Vereist in NHibernate
  - NHibernate ondersteunt samengestelde sleutels
    - Gebruik enkel voor legacy databases
- <generator>
  - Hoe id aangemaakt bij nieuwe objecten?
  - native
    - Standaard generator huidige database

Inhoud

13

- Wat is ORM?
- Objecten afbeelden
  - Eigenschappen
  - Overerving
  - Relaties
- Nhibernate
  - Session en SessionFactory
  - Objecten ophalen en bewaren

Object ↔ Tabel ??

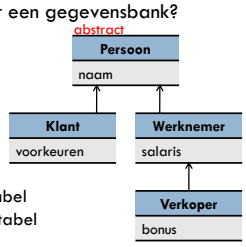
14

- Klasse → tabel
  - Vaak, maar niet altijd!
    - Soms anders o.w.v. performantie

Overerving

15

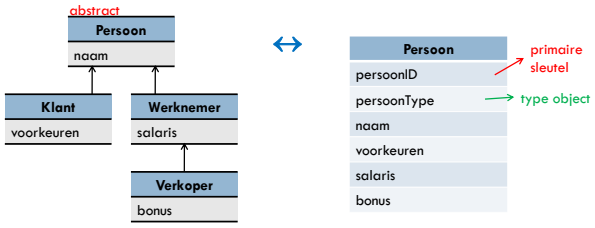
- Hoe vertaal je overerving naar een gegevensbank?



- Mogelijkheden
  - Volledige hiërarchie naar één tabel
  - Elke concrete klasse naar eigen tabel
  - Elke klasse naar een tabel
  - Alle klassen naar een generieke tabel

Volledige hiërarchie naar één tabel

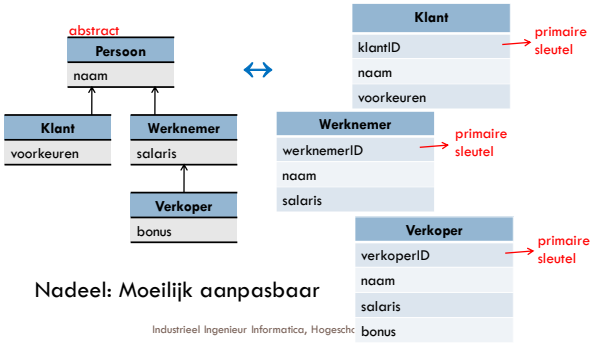
16



Nadeel: Moeilijk uitbreidbaar

Elke concrete klasse naar eigen tabel

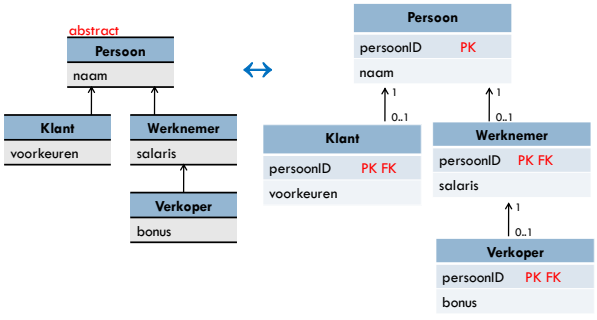
17



Nadeel: Moeilijk aanpasbaar

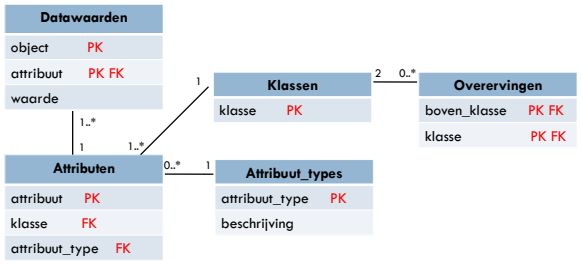
Elke klasse naar een tabel

18



Alle klassen naar een generieke tabel

19



Gebruik: complexe applicaties met weinig data

Alle klassen naar een generieke tabel

20

abstract

Persoon
naam

Klant
voorkeuren

Objecten

object	attribuut	waarde
OBJ_1	naam	Peeters
OBJ_1	voorkeuren	…

Klassen

klasse
Persoon
Klant

Attributen

attribuut	klasse	attribuut_type
naam	Persoon	String
voorkeuren	Klant	VoorkeurType

Overervingen

boven_klasse	klasse
Persoon	Klant

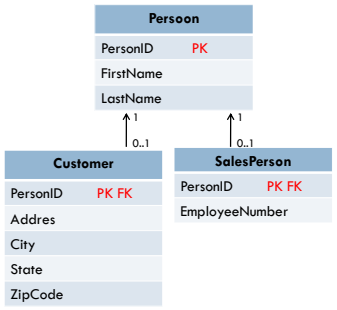
Attribuut_types

attribuut_type	beschrijving
String	…
VoorkeurType	…

Voorbeeld overerving

21

```
public partial class Person {
  private int? _personID;
  private string _firstName;
  private string _lastName;
  // Property Definitions...
}
public partial class Customer : Person {
  private string _address;
  private string _city;
  private string _state;
  private string _zipCode;
 // Property Definitions...
}
public partial class SalesPerson : Person {
  private string _employeeNumber;
  // Property Definitions...
}
```



Voorbeeld overerving

22

```
<?xml version="1.0" encoding="utf-8" ?>
<hibernate-mapping xmlns="urn:nhibernate-mapping-2.2">
 <class name="Person, __Code" table="Person">
  <id name="PersonID" column="PersonID" type="Int32"  unsaved-value="null">
   <generator class="native" />
  </id>
  <property name="FirstName" column="FirstName" type="String"/>
  <property name="LastName"  column="LastName" type="String"/>

  <joined-subclass name="Customer, __Code" table="Customer">
   <key column="PersonID"/>
   <property name="Address" column="Address" type="String"/>
   <property name="City"    column="City"    type="String"/>
   <property name="State"   column="State"   type="String"/>
   <property name="ZipCode" column="ZipCode" type="String"/>
  </joined-subclass>

  <joined-subclass name="SalesPerson, __Code" table="SalesPerson">
   <key column="PersonID"/>
   <property name="EmployeeNumber" column="EmployeeNumber"  type="String"/>
  </joined-subclass>
 </class>
</hibernate-mapping>
```


Inhoud

23

- Wat is ORM?
- Objecten afbeelden
  - Eigenschappen
  - Overerving
  - Relaties
- Nhibernate
  - Session en SessionFactory
  - Objecten ophalen en bewaren

Relaties

24

- Object relaties
  - Associatie
  - Aggregatie
  - Compositie
- Indeling volgens multipliciteit
  - 1-op-1
  - 1-op-veel
  - Veel-op-veel
- Indeling volgens richting
  - Eénrichtingsrelatie ← Onbestaand in DB
  - Tweerichtingsrelatie

Relaties gerealiseerd

25

- In Objecten
  - Referentie naar object (enkelvoudige relatie)
    - getter en setter
  - Referentie naar een verzameling van objecten (meervoudige relatie)
    - getter en setter
    - add- en remove-methodes
  - In één object (éénrichtingsrelatie)
  - In beide objecten (tweerichtingsrelatie)
- In DB
  - Verwijssleutels (foreign keys)
  - 1-1 of 1-veel
    - Tabel met PK
    - Tabel met FK
  - Veel-veel
    - Associatietabel (FK, FK)
    - Twee tabellen met PK
  - Altijd tweerichtingsrelatie

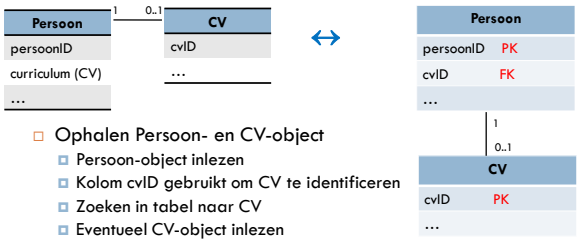
Vertalen relaties

26

- Behoud de multipliciteit
  - 1-1
  - 1-veel
  - veel-veel

1-1 relatie ophalen

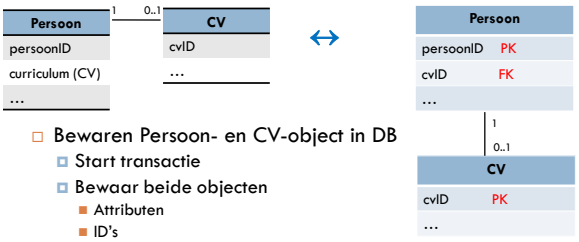
27



- Ophalen Persoon- en CV-object
  - Persoon-object inlezen
  - Kolom cvID gebruikt om CV te identificeren
  - Zoeken in tabel naar CV
  - Eventueel CV-object inlezen
  - (Indien tweerichtingsrelatie: omgekeerde referentie instellen)

1-1 relatie bewaren

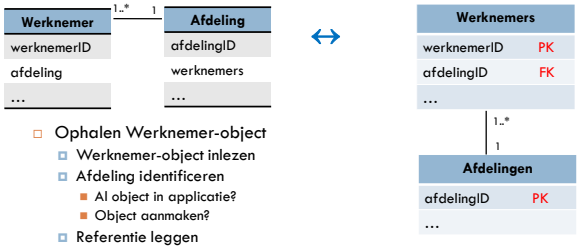
28



- Bewaren Persoon- en CV-object in DB
  - Start transactie
  - Bewaar beide objecten
    - Attributen
    - ID's
  - Commit
  - Beëindig transactie

1-veel relatie ophalen

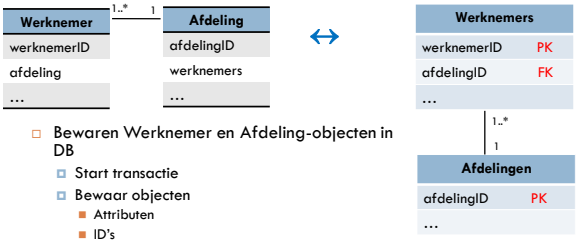
29



- Ophalen Werknemer-object
  - Werknemer-object inlezen
  - Afdeling identificeren
    - Al object in applicatie?
    - Object aanmaken?
  - Referentie leggen
  - Werknemer-object toevoegen aan Afdeling-object

1-veel relatie bewaren

30

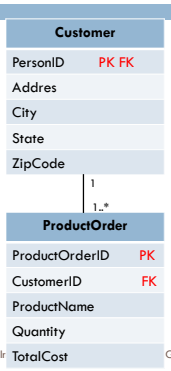


- Bewaren Werknemer en Afdeling-objecten in DB
  - Start transactie
  - Bewaar objecten
    - Attributen
    - ID's
  - Commit
  - Beëindig transactie

Voorbeeld 1-veel relatie

31

```
public partial class Customer : Person {
  private string _address;
  private string _city;
  private string _state;
  private string _zipCode;
  private ISet _orders;
 // Property Definitions...
}

public partial class ProductOrder {
  private int? _productOrderID;
  private int? _customerID;
  private string _productName;
  private int _quantity;
  private decimal _totalCost;
  // Property Definitions...
}
```

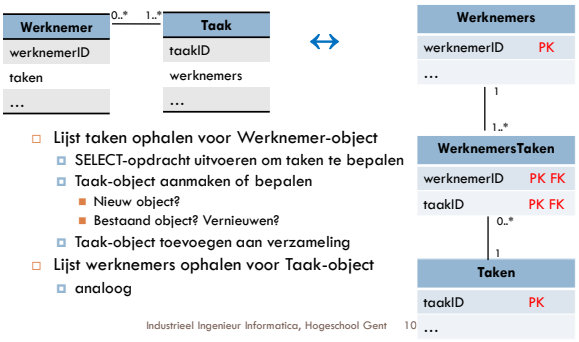



```
<joined-subclass name="Customer,
    __Code" table="Customer">
<set name="Orders">
    <key column="CustomerID"/>
    <one-to-many class="ProductOrder,
    __Code"/>
</set>
</joined-subclass>
```

- Beschrijving vertaling ProductOrder
  - Eigen bestand ProductOrder.hbm.xml

Veel-veel relatie ophalen

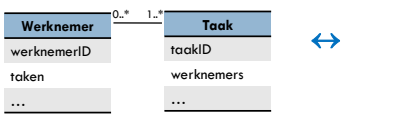
32

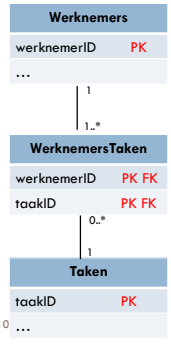


- Lijst taken ophalen voor Werknemer-object
  - SELECT-opdracht uitvoeren om taken te bepalen
  - Taak-object aanmaken of bepalen
    - Nieuw object?
    - Bestaand object? Vernieuwen?
  - Taak-object toevoegen aan verzameling
- Lijst werknemers ophalen voor Taak-object
  - analoog

Veel-veel relatie bewaren

33



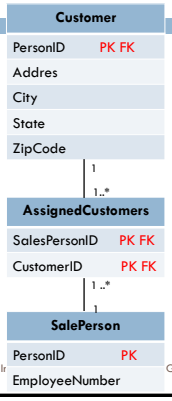


- Taken van Werknemer-object bewaren
  - Start transactie
  - Update voor alle veranderde Taak-objecten
  - Insert voor alle nieuwe Taak-objecten
  - Delete voor alle verwijderde Taak-objecten (indien nodig)
  - Delete voor alle "WerknemersTaken" (indien nodig)
  - Delete voor alle taken die niet meer toegekend zijn aan de werknemer
  - Commit
  - Einde transactie

Voorbeeld veel-veel relatie

34

```
public partial class Customer : Person {
  private string _address;
  private string _city;
  private string _state;
  private string _zipCode;
  private ISet _orders;
 // Property Definitions...
}

public partial class SalesPerson : Person
{
  private string _employeeNumber;
  private ISet _assignedCustomers;

  // Property Definitions...
}
```




```
<joined-subclass name="SalesPerson,
    __Code" table="SalesPerson">
…
<set name="AssignedCustomers"
    table="AssignedCustomers"
    cascade="save-update">
  <key column="SalesPersonID"/>
  <many-to-many class="Customer,
    __Code" column="CustomerID"/>
</set>
</joined-subclass>
```

Bewaart ook Customer-objecten die toegevoegd zijn aan de Iset.

Standaard: enkel properties bewaard

Inhoud

35

- Wat is ORM?
- Objecten afbeelden
  - Eigenschappen
  - Overerving
  - Relaties
- Nhibernate
  - Session en SessionFactory
  - Objecten ophalen en bewaren

Objecten uitwisselen

36

- NHibernate
  - Vertaling vastleggen: xml-bestanden
  - Objecten ophalen en bewaren
    - Session Factory
    - Session

Session

37

- NHibernate.ISession
- Voert opdrachten uit
  - Ophalen, bewaren, aanpassen, verwijderen objecten in gegevensbank
- Bevat functionaliteit gegevensbank
- Bevat connectie met gegevensbank
- Light weight
  - Telkens aanmaken en vernietigen
  - Niet thread safe

Session Factory

38

- NHibernate.ISessionFactory
- Beheren Session-objecten
  - Aanmaken
  - …
- Één per applicatie
- Inlezen
  - Configuratie
  - Vertalingen (.hbm.xml-bestanden)
  - …
- Heavy weight
- Thread Safe

Voorbeeld aanmaak sessies

39

- Webapplicatie
  - Session-object
    - Aangemaakt bij start verwerken HTTP-aanvraag
    - Verwijderd bij doorsturen HTTP-antwoord
- NHibernateHelper.cs
  - Statische constructor
    - Aanmaken Session Factory
      - Configuratie inlezen
      - Object aanmaken
  - Metodes
    - Aanmaken Session-object
    - Sluiten Session-object
    - Afsluiten Session Factory

Inhoud

40

- Wat is ORM?
- Objecten afbeelden
  - Eigenschappen
  - Overerving
  - Relaties
- Nhibernate
  - Session en SessionFactory
  - Objecten ophalen en bewaren

Object ophalen (m.b.v. ID)

41

```
public static Customer getCustomer(int? customerID) {
  // Open the session
  ISession session = NHibernateHelper.GetCurrentSession();

  // Load the order from the database
  Customer customer = (Customer)session.Load(typeof(Customer), customerID);

  return customer;
}
```

- Afsluiten sessie
  - Application_EndRequest (Global.asax)

```
void Application_EndRequest(object sender, EventArgs e)  {
      NHibernateHelper.CloseSession();
}
```


Object ophalen (a.d.v. criteria)

42

```
public static List<ProductOrder> getProductOrdersForCustomer(int? customerID) {
  List<ProductOrder> orders = new List<ProductOrder>();

  // Get the session
  ISession session = NHibernateHelper.GetCurrentSession();

  // Load the order from the database
  ICriteria criteria = session.CreateCriteria(typeof(ProductOrder));
  criteria.Add(Expression.Eq("CustomerID", customerID));
  criteria.List(orders);

  return orders;
}
```

- Opbouwen WHERE-clausule
- Andere opdrachten: <, >, tussen, …

Bewaren of aanpassen object

43

```
public static void saveOrUpdateCustomer(Customer customer) {
  ISession session = NHibernateHelper.GetCurrentSession();
  ITransaction transaction = null;

  try   {
    // Start transaction
    transaction = session.BeginTransaction();

    // Save or update (saves if id is null; otherwise, updates)
    session.SaveOrUpdate(customer);

    // Commit transaction
    transaction.Commit();
  }   catch (Exception ex)   {
    transaction.Rollback();
    throw ex;
  }
}
```

- SaveOrUpdate
  - Programmeur hoeft operatie niet te kiezen
  - Save, Update kan ook
- Delete
- Gebruik steeds transacties
  - Achter schermen vaak meerdere opdrachten

Lazy Loading

44

- Attribuut: verzameling
  - Wanneer geladen?
- Aanmaken Customer-object
  - Enkel informatie uit Customer-tabel
    - address, city, state, zipCode
  - Proxy-object voor orders
    - Geladen als het opgevraagd wordt
- Kan enkel als de sessie nog actief is
- Kan uitgeschakeld worden in de .hbm.xml-bestanden

```
public partial class Customer : Person {
  private string _address;
  private string _city;
  private string _state;
  private string _zipCode;
  private ISet _orders;
 // Property Definitions...
}

public partial class SalesPerson : Person
{
  private string _employeeNumber;
  private ISet _assignedCustomers;
  // Property Definitions...
}
```


Meer informatie

45

- Mapping Objects to Relational Databases: O/R Mapping In Detail (http://www.agiledata.org/essays/mappingObjects.html)
- Using NHibernate as an ORM Solution for .NET (http://www.developer.com/net/asp/article.php/10917_3709346_6)
- Nhibernate (http://www.theserverside.net/tt/articles/showarticle.tss?id=NHibernate)
- Nhibernate – Part 2 (http://www.theserverside.net/tt/articles/showarticle.tss?id=NHibernateP2)

Meer informatie

46

- Your first NHibernate based application (http://blogs.hibernatingrhinos.com/nhibernate/archive/2008/04/01/your-first-nhibernate-based-application.aspx)
- Nhiberate als ORM oplossing (http://mark.mymonster.nl/wp-content/uploads/2007/07/nhibernate.pdf)
- NHibernate Quick Start Guide (http://www.hibernate.org/362.html)

Hoofdstuk 5

EL en JSTL

EL (Expression Language) en JSTL (JSP Standard Tag Library)

Veerle Ongenae

Waarom?

- JSP zonder scripts!!
  - Eenvoudiger voor niet Javaprogrammeurs
- EL
  - Expression Language
  - Toegang tot Java-objecten
- JSTL
  - JSP Standard Tag Library
  - Eenvoudige opdrachten: lus, url, …

EL

- Principe lijkt op XPath en Javascript
  - Typisch webdesign
- JSP's dienen om de inhoud van Java-objecten (JavaBeans) te tonen
  - GEEN logica
- EL
  - Eenvoudigere toegang tot deze javabeans

Attributen

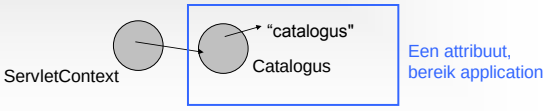
- Vier objecten die attributen kunnen hebben
  - application
  - session
  - request
  - pageContext
- Attribuut
  - Naam – id – sleutel
  - Waarde – object - javabean

Javabean-eigenschappen tonen

- Met <jsp:getProperty …>
- Wat als de eigenschap geen String of een primitief type is?
- Geen geneste eigenschappen (een eigenschap van een eigenschap) mogelijk met <jsp:getProperty …>
- Oplossing? EL

Voorbeeld

toonBoek.jsp
• parameter: isbn



Een attribuut, bereik application

De interface Catalogus

be.hogent.iii.boeken.bo.Catalogus
<<interface>>

+Boek getBoek(String isbn) throws BoekNietBeschikbaarException;
+Boek[] getBoeken(); → Eigenschap boeken
+Map<String,Boek> getBoekMap();

ISBN-nummer

Eigenschap boekMap

Code voorbeeld

- toonBoekZonderEL.jsp
  - Bean catalogus → boek
  - Boek → pageContext
  - Eigenschappen tonen van boek
- toonBoek.jsp
  - Met EL

EL uitdrukking

- Vorm
  - ${ … }
  - Operatoren: . en [ ]
- ${eersteding.tweededing}

Impliciet EL-object of Naam v/e attribuut (bereik: page, request, session, application)

Type? java.util.Map of javabean

. operator

- ${eersteding.tweededing}

Sleutel van een Map-object of Eigenschap van een javabean

Correcte Java-naam

Voorbeeld .operator

- ${catalogus.boeken}

Attribuut scope application

Eigenschap boeken Methode getBoeken()

- Map landen
  - “NL”, Nederland
  - “BE”, België
  - …
- ${landen.NL} → Nederland

[] operator

- ${eersteding[tweededing]}

Impliciet EL-object of Naam v/e attribuut (bereik: page, request, session, application)

Type? java.util.Map java.util.List javabean array (tabel)

[] operator

- ${eersteding[tweededing]}

Sleutel van een Map-object
Eigenschap van een javabean
Index van een List- of arrayobject

Stukje tekst
Geheel getal (bij array of List)
EL-uitdrukking

Resultaat → waarde (tekst of int) wordt gebruikt

Voorbeeld []-operator

- ${boek["isbn"]}
  - boek, attribuut scope request, type Boek
- ${boeken[0]}
- ${boeken["0"]}
  - boeken, attribuut scope application, type Boek[]
- ${landen["NL"]}
  - landen, attribuut, type Map

Een aantal impliciete EL-objecten

- pageScope
- requestScope
- sessionScope
- applicationScope

Map met attributen in een bepaalde "scope"

→ Handig indien meerdere attributen met zelfde naam

- param → Map met parameters HTTP-aanvraag
- initParam → Map met initialisatieparameters van de webapplicatie

Voorbeelden

Impliciet EL-object
Sleutel van een Map-object (als String)

- ${param.isbn}
  - Resultaat
    - String (tekst)
    - Waarde van de parameter "isbn"
- ${catalogus.boekMap}
  - Resultaat
    - Map<String,Boek>

Naam attribuut op application
Eigenschap van een bean

Voorbeelden

Map-object
String → sleutel

- ${catalogus.boekMap[param.isbn]}
  - Resultaat
    - Boek-object
- ${catalogus.boekMap[param.isbn].titel}

Boek-object (javabean)
Eigenschap van een bean

  - Resultaat
    - String
    - Titel van het boek

Voorbeelden

Attribuut application, javabean

- ${catalogus.boeken]}
  - Resultaat
    - Array van Boek-objecten

Eigenschap bean

- ${catalogus.boeken[0]}

Array Boek-objecten
int, index array

  - Resultaat
    - Boek-object
- ${catalogus.boeken["0"]}
  - Resultaat
    - Boek-object

String → index array

Aan- en uitschaken EL

- Aan
  - <%@page isELIgnored="false"%>
- Uit
  - <%@page isELIgnored="true"%>
- Standaard?
  - Afhankelijk van de webcontainer
    - Oudere → uit
    - Recentere → aan

Overzicht

- EL
  - Expression Language
  - Toegang tot Java-objecten
- JSTL
  - JSP Standard Tag Library
  - Eenvoudige opdrachten: lus, url, …
    - Bv. inhoud van een array of een List-object tonen

JSTL

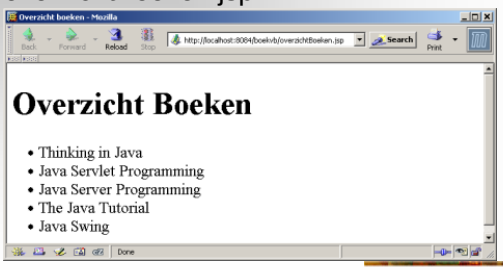
- Geen deel van de JSP-specificatie
  - jstl.jar toevoegen aan WEB-INF/lib
    - In Netbeans
      - Klik rechts op “Libraries” in het projectvenster
      - Kies “Add Library”
      - Selecteer JSTL
  - taglib-richtlijn gebruiken

<%@taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c"%>

    - Afspraak: steeds prefix “c”

Voorbeeld

- overzichtBoeken.jsp



JSTL

- Extra tags
  - Eenvoudige opdrachten
- Gecombineerd met EL
  - Toegang java-objecten

<c:forEach>

- Overlopen van arrays (tabellen) en collections
- Voorbeeld

```
<c:forEach var="boek" items="${catalogus.boeken}">
   <li>${boek.titel}</li>
</c:forEach>
```

array, Collection, Map

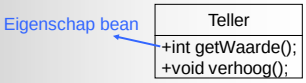
Variabele
Enkel gekend in tag
Bevat beurtelings elementen uit array, Collection of Map

Voorbeeld

- tellerZonderScript.jsp

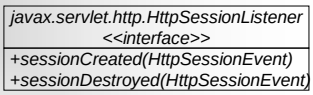
Principe Teller

- Klasse Teller
- Bij opstarten sessie
  - Nieuw Teller-object op sessie
    - Luisteraar: AanmaakTeller
    - HttpSessionListener
- Bij elke aanvraag voor de applicatie
  - Teller-object verhogen
    - Luisteraar: AanpassenTeller
    - ServletRequestListener

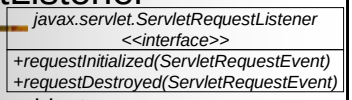


HttpSessionListener

- Luistert naar
  - Aanmaken sessie
  - Vernietigen sessie
- AanmaakTeller
  - Event → sessie
  - Teller op sessie



ServletRequestListener



- Luistert naar
  - Initialisatie aanvraagobject
  - Vernietigen aanvraagobject
- AanpassenTeller
  - Event → request → HTTP-request
  - HTTP-request → sessie
  - Sessie → teller
  - Teller aanpassen

Tonen teller + boodschap

```
<c:choose>
<c:when test="${teller.waarde == 1}">
Dit is een nieuwe sessie.
</c:when>
<c:otherwise>
Welkom terug. Dit is uw ${teller.waarde}e bezoek.
</c:otherwise>
</c:choose>
```

bean

Eigenschap bean

Meerdere when's mogelijk

Min of meer equivalent met switch-opdracht

URL's en sessies via JSTL

- <c:url value='tellerZonderScript.jsp'/>
  - Voegt automatisch JSESSIONID toe aan url als er geen cookies ondersteund worden

```
<c:url value='eenUrl'>
   <c:param name='param' value='Mijn parameter'/>
</c:url>
```

Codeert automatisch de parameter

JSTL

- Bibliotheek met tags
  - JSP zonder script
- Vier bibliotheken
  - core
  - formatting
  - SQL
  - XML

Informatie over EL en JSTL

- Head First Servlets & JSP (O'Reilly)
- Java Tutorial (EL) (http://java.sun.com/j2ee/1.4/docs/tutorial-update2/doc/JSPIntro7.html)
- JSTL Tutorial (http://java.sun.com/j2ee/1.4/docs/tutorial/doc/JSTL.html)

Hoofdstuk 6

Model-View-Controller

TE VERWERKEN: The alias identifies the web component that should handle a request. The alias path must start with a forward slash (/) and end with a string or a wildcard expression with an extension (for example, *.jsp).

6.1 Algemeen

In een meerlagensysteem is het mogelijk om het MVC-patroon te gebruiken. Dit betekent dat de gegevens (*model*) gescheiden worden van de voorstelling van deze informatie (*view*). De *controller* bepaalt hoe de informatiestroom in de applicatie verloopt.

Het MVC-patroon is een model dat afkomstig is uit de GUI-ontwikkeling. Een grafische gebruikersinterface bestaat uit verschillende componenten. De voorstelling van de component (*view*) toont de gebruiker de informatie (*model*) van die component (bv. een tekstveld). De gebruiker kan deze informatie aanpassen door gebruik te maken van de *controller* (in het voorbeeld: gebruik makend van het toetsenbord, de muis, enz.). De controller zal dan de informatie wijzigen en het model zorgt ervoor dat presentatie aangepast wordt.

Dezelfde principes kan men nu ook toepassen op applicatieniveau. Het model, de view en de controller zijn alle drie stukken van de applicatie.

De controller bepaalt het gedrag van de applicatie. Voor de gebruiker is hij het ingangspunt van de applicatie. Op basis van een gebruikersactie past hij de status van de applicatie (model) aan en selecteer de juiste presentatie (view) voor de gebruiker.

Het model bevat de status van de applicatie en bepaalt de toegang tot de gegevens. De functionaliteit van de applicatie, de structuur van de gegevens en de bewerkingen op gegevens maken deel uit van het model.

De view of presentatie toont de inhoud van het model.

6.2 MVC in J2EE

Het MVC-patroon kan in J2EE op de volgende manier gerealiseerd worden. De controller bestaat uit een servlet (en eventueel bijhorende controleklassen) die de HTTP-aanvraag aanvaardt. Deze servlet zorgt ervoor dat op basis van de aanvraag gegevens in JavaBeans en EJB's (model) aangepast worden. Vervolgens selecteert de controller de juiste JSP's (en eventueel HTML-pagina's) die de presentatie vormen en zorgt ervoor dat ze het HTTP-antwoord aanmaken.

6.2.1 Een voorbeeldapplicatie

Om het MVC-patroon te illustreren werken we een deel van een broodjesapplicatie uit. We maken een webtoepassing met onder andere de volgende functionaliteiten:

- De toepassing is bedoeld voor werknemers van één bedrijf.
- Elke morgen kunnen de werknemers voor 10u00 via deze applicatie broodjes bestellen.
- Elke werknemer heeft een krediet waarmee hij broodjes kan bestellen.

Deze toepassing bestaat uit een aantal JSP's (*View*), een pakket *be.wolterslaan.broodjes.bo* dat alle business-objecten bevat (*Model*) en een pakket *be.wolterslaan.broodjes.web* dat de controleservlet en bijhorende controleklassen bevat (*Controller*).

6.2.2 Model

Het pakket *be.wolterslaan.broodjes.bo* bevat de eigenlijke business-objecten. Dit omvat de klassen die communiceren met de gegevensbank, klassen die het beheer van gebruikers en broodjes realiseren, interfaces die broodjes, gebruikers, enz. voorstellen. Twee van die klassen zijn een 'Singleton Pattern'. Dit wil zeggen dat er maar één object van die klasse aangemaakt mag worden. De objecten van die klassen verzorgen respectievelijk het beheer van de gebruikers en van de broodjes. De structuur van de klasse *GebruikersBeheerder* is de volgende.

```
package be.wolterslaan.broodjes.bo;

public class GebruikersBeheerder {

  private static GebruikersBeheerder deBeheerder;

  public static synchronized GebruikersBeheerder getInstance(){
    if (deBeheerder==null){
      deBeheerder = new GebruikersBeheerder();
    }
    return deBeheerder;
  }

  protected GebruikersBeheerder(){
  }
```

```
  public IGebruiker getGebruiker(String gebruikersNaam){
     ...
  }
}
```

De interface *IGebruiker* zou er bijvoorbeeld als volgt uit kunnen zien.

```
package be.wolterslaan.broodjes.bo;
import be.wolterslaan.util.Bedrag;

public interface IGebruiker {

  String getGebruikersNaam();
  String getVoornaam();
  String getNaam();
  Bedrag getKrediet();

  void setVoornaam(String voornaam);
  void setNaam(String naam);

  void vulKredietAan(Bedrag bedrag);
  void betaal(Bedrag bedrag) throws OnvoldoendeKredietException;
}
```

6.2.3 Controller

Het pakket *be.wolterslaan.broodjes.web* bevat een interface *BroodjesConstanten* en een klasse *ControllerServlet*. De interface bestaat uit alle constanten die gebruikt worden in het JSP's en het pakket *be.wolterslaan.broodjes.web*.

```
package be.wolterslaan.broodjes.web;

public interface BroodjesConstanten {

  String CFG_GEBRUIKERSBEHEER = "gebruikersbeheer";
  String CFG_BROODJESBEHEER = "broodjesbeheer";

  String SESS_GEBRUIKER = "gebruiker";
  String SESS_BROODJE = "broodje";

  String REQ_EXCEPTION = "exception";
  String REQ_VIEW = "scherm";

  String PARAM_ACTIE = "actie";
  String PARAM_BELEG = "beleg";

  String ACTIE_TOON_BELEGLIJST = "toon-beleglijst";
  String ACTIE_BESTEL_BROODJE = "bestel-broodje";
  ...

  String JSP_TEMPLATE = "/template.jsp";
  String JSP_HOME = "/home.jsp";
  String JSP_NOLOGIN = "/nologin.jsp";
  String JSP_BELEGFORMULIER = "/belegformulier.jsp";
  String JSP_TOON_FOUT = "/toon-fout.jsp";
```

```
  ...
}
```

De *ControllerServlet* krijgt de HTTP-aanvraag van elke pagina van de applicatie binnen. Aan de hand van de URL bepaalt hij welke actie uitgevoerd moet worden en zorgt ervoor dat die door de juiste business-objecten wordt uitgevoerd. Vervolgens wordt de HTTP-aanvraag doorgestuurd naar de juiste JSP.

```
public void doGet(HttpServletRequest req,
  HttpServletResponse res)
  throws ServletException,IOException
{
  doPost(req,res);
}

public void doPost(HttpServletRequest req,
  HttpServletResponse res)
  throws ServletException,IOException
{
  String view = doActie(req);
  req.setAttribute(REQ_VIEW,view);
  RequestDispatcher rd = req.getRequestDispatcher(JSP_TEMPLATE);
  if (rd==null) view=JSP_HOME;
  rd.forward(req,res);
}
```

De methode *doActie* bepaalt welke actie de gebruiker wenst, zorgt ervoor dat de benodigde status-informatie bewaard wordt in de business-objecten en bepaalt de presentatie van HTTP-antwoord. De naam van deze presentatie is een constante, die als naam/waarde-paar toegevoegd wordt aan het HTTP-antwoord.

Bij het opstarten van de *ControllerServlet* worden de 'Singleton Patterns' van het type *GebruikersBeheerder* en *BroodjesBeheerder* toegevoegd aan de servletcontext.

```
public void init() throws ServletException {
  m_gebruikersBeheer = GebruikersBeheerder.getInstance();
  m_broodjesBeheer =  BroodjesBeheerder.getInstance();
  ServletContext ctx = config.getServletContext();
  ctx.setAttribute(CFG_GEBRUIKERSBEHEER,m_gebruikersBeheer);
  ctx.setAttribute(CFG_BROODJESBEHEER,m_broodjesBeheer);
}
```

Het bepalen van de actie, het aanpassen van de statusgegevens en het selecteren van de presentatie vindt plaats in de methode *doActie*.

```
protected String doActie(HttpServletRequest req){
  String jsp=JSP_HOME;
  try {
    if (!controleerGebruiker(req)){
      jsp=JSP_NOLOGIN;
    } else {
      String actie = req.getPathInfo();
```

```
      if (actie == null) {
        jsp=JSP_HOME;
      } else {
        actie=actie.substring(1);
        if (actie.startsWith(ACTIE_TOON_BELEGLIJST)) {
          jsp=JSP_BELEGFORMULIER;
        } else if (actie.startsWith(
          ACTIE_TOON_BROODJESFORMULIER)){
          maakBestelling(req);
          jsp=JSP_BROODJESFORMULIER;
        } else if (actie.startsWith(ACTIE_BESTEL_BROODJE)){
          if (verifieerEnBestel(req)) {
            jsp=JSP_BROODJE_BESTELD;
          } else {
            jsp=JSP_BROODJESFORMULIER;
          }
        } else if (actie.startsWith(ACTIE_TOON_BESTELLING)) {
          jsp=JSP_TOON_BESTELLING;
        } else if (actie.startsWith(ACTIE_SLUIT_AF)){
          sluitBestellingAf(req);
          jsp=JSP_BESTELLING_AFGESLOTEN;
        }
      }
    }
  } catch (Exception e) {
    req.setAttribute(REQ_EXCEPTION,e);
    jsp=JSP_TOON_FOUT;
  }
  return jsp;
}
```

Deze methode controleert of de gebruiker toegang heeft tot de applicatie, vervolgens wordt het stuk van het pad na het pad van de toepassing opgehaald. Is de URL van de vorm 'http://host/broodjes/toon-beleglijst' dan geeft de methode *getPathInfo* de string '/toon-beleglijst' weer. Vervolgens laat men de '/' vallen. Op basis van deze actiestring worden de nodige acties uitgevoerd. De uitwerking van de methode *controleerGebruiker* zou bijvoorbeeld de volgende kunnen zijn.

```
protected boolean controleerGebruiker(HttpServletRequest req){
  HttpSession sessie = req.getSession();
  IGebruiker gebruiker =
    (IGebruiker)sessie.getAttribute(SESS_GEBRUIKER);
  if (gebruiker==null) {
    String gebruikersnaam = req.getRemoteUser();
    if (gebruikersnaam!=null) {
      gebruiker = m_gebruikersBeheer.getGebruiker(
          gebruikersnaam);
      sessie.setAttribute(SESS_GEBRUIKER,
        m_gebruikersBeheer.getGebruiker(gebruikersnaam));
    }
  }
  return gebruiker != null;
}
```

De methode *getRemoteUser* van *HttpServletRequest* geeft de loginnaam van de gebruiker of *null* als de gebruiker niet ingelogd is. Merk op dat bij een grote applicatie de code onderhoudbaarder wordt als de methodes voor de verschillende acties deel zijn van aparte klassen uit het pakket *be.wolterslaan.broodjes.web*.

6.2.4 View

Alle webpagina's hebben dezelfde structuur die bepaald wordt door de JSP 'template.jsp'. Alle pagina's hebben dezelfde voettekst, de eigenlijke pagina en de titel van het browservenster wordt bepaald door het naam/waardepaar dat toegevoegd werd aan de HTTP-aanvraag in de *ControllerServlet*.

```
<%@ page import="be.wolterslaan.broodjes.web.*" %>
<html>
  <head>
  <title>
    <%=(String)request.getAttribute(BroodjesConstanten.REQ_VIEW)%>
  </title>
  </head>
  <body>
     <jsp:include
       page="<%=(String)request.getAttribute(
       BroodjesConstanten.REQ_VIEW)%>"
       flush="true"%>
     <jsp:include
       page="<%=BroodjesConstanten.JSP_FOOTER%>"
       flush="true"%>
  </body>
</html>
```

Hoofdstuk 7

JSF

Java Server Faces (JSF)

Veerle Ongenae

JSF

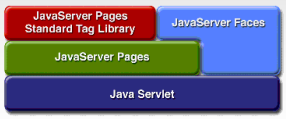
- Inleiding
- Voorbeeld
- Componenten en beans
- Levensloop
- Events
- UIData
- Taal-afhankelijkheid
- Navigatie
- Validatie
- Conversie
- Componenten

JSF

- Technologie
  - Webapplicaties in Java
  - Componenten voor gebruikersinterface
  - Server side
- API
  - UI-componenten
- Tag-bibliotheken

JSF

- Verder uitwerken scheiding presentatie ↔ functionaliteit (logica)
- Meestal ingebed in JSP's, hoeft niet
  - Gebouwd op servlet-technologie

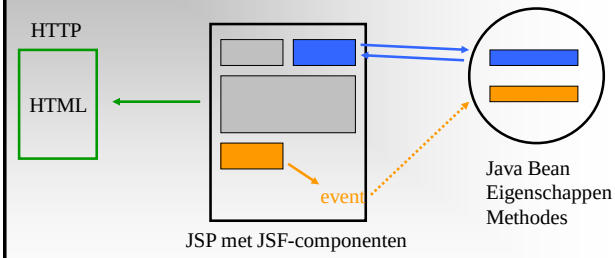


(java.sun.com)

Functionaliteit

- Opmaken UI met herbruikbare en uitbreidbare componenten
- Koppeling UI-componenten aan data (server)
- Bewaren van de status van UI-componenten tussen verschillende HTTP-aanvragen
- Koppeling events (client) aan functionaliteit (server)

Principe



JSP met JSF-componenten

JSF

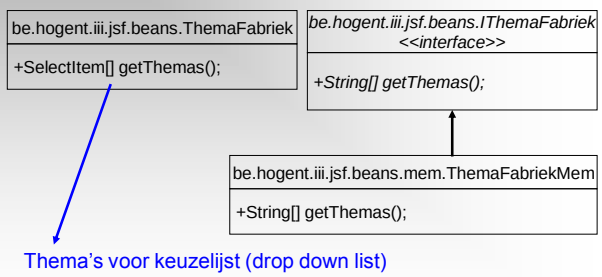
- Inleiding
- Voorbeeld
- Componenten en beans
- Levensloop
- Events
- UIData
- Taal-afhankelijkheid
- Navigatie
- Validatie
- Conversie
- Componenten

Voorbeeld

Voorbeeld

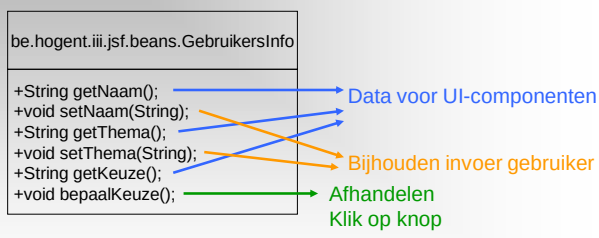
- http://localhost:8084/jsf/index.jsp
  - Doorsturen naar keuzeJSF.jsp
    <jsp:forward page="faces/keuzeJSF.jsp"/>
- http://localhost:8084/jsf/faces/keuzeJSF.jsp
  maakt gebruik van 2 beans
  - GebruikersInfo
  - ThemaFabriek

Gebruikte beans: ThemaFabriek



Thema's voor keuzelijst (drop down list)

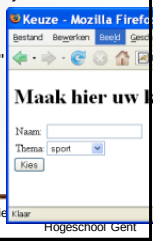
Gebruikte beans: GebruikersInfo



keuzeJSF.jsp

```
<%@page contentType="text/html"%>
<%@page pageEncoding="UTF-8"%>
<%@taglib prefix="f" uri="http://java.sun.com/jsf/core"%>
<%@taglib prefix="h" uri="http://java.sun.com/jsf/html"%>

<html>
   <head>
      <meta http-equiv="Content-Type" content="text/html;
    charset=UTF-8">
      <title>Keuze</title>
   </head>
   <body> … </body>
</html>
```

```
<body><f:view>
  <h1><h:outputText value="Maak hier uw keuze" /></h1>
   <h:form><table>
    <tr><td>Naam:</td>
        <td><h:inputText id="invoerNaam"
                value="#{GebruikersInfo.naam}" /></td></tr>
    <tr><td>Thema:</td>
        <td><h:selectOneMenu id="invoerThema"
                value="#{GebruikersInfo.thema}">
              <f:selectItems value="#{ThemaFabriek.themas}"/>
            </h:selectOneMenu>
        </td></tr></table>
    <h:commandButton value="Kies" action="keuzeJSF.jsp"
       actionListener="#{GebruikersInfo.bepaalKeuze}" />
    <br>
    <h:outputText value="#{GebruikersInfo.keuze}"/>
 </h:form></f:view></body>
```



JSF

- Inleiding
- Voorbeeld
- Componenten en beans
- Levensloop
- Events
- UIData
- Taal-afhankelijkheid
- Navigatie
- Validatie
- Conversie
- Componenten

JSP met JSF

- f:view
  - Omvat alle andere componenten
  - boomstructuur
- h:form
  - Data sturen naar server
  - Genereert attributen form-tag
  - Bevat de andere componenten
- h:outputText
  - Label

Gebruikte componenten

- h:inputText
  - Invoer tekst
- h:selectOneMenu
  - Eén selecteren uit lijst
  - f:selectItems
    - Items waaruit de lijst bestaat
    - value-atttibuut
      - Unified EL
        - Wijst naar eigenschap
        - List SelectItem
        - Array SelectItem

Gebruikte componenten

- h:commandButton
  - action
    - volgende pagina
    - String
      - Letterlijk
      - Resultaat methode (Unified EL)
  - actionListener
    - Methode die event afhandelt
    - Unified EL

value-attribuut

```
<h:outputText value="Maak hier uw keuze" />
<h:inputText id="invoerNaam"
   value="#{GebruikersInfo.naam}" />
<h:selectOneMenu id="invoerThema"
   value="#{GebruikersInfo.thema}"> …
```

- Tekst
  - Data voor component
- unified EL – uitdrukking
  - #{…}
  - Data voor component
  - Bijhouden invoer gebruiker
  - Eigenschap bean

Unified EL

- #{…}
- Gelijkaardig EL
  - value-binding
    - Eigenschappen beans

```
<h:inputText id="invoerNaam"  value="#{GebruikersInfo.naam}" />
```

  - method-binding
    - Methodes beans

```
<h:commandButton value="Kies" action="keuzeJSF.jsp"
        actionListener="#{GebruikersInfo.bepaalKeuze}" />
```

  - Andere EL-elementen

Componenten

- Tags ↔ UI-componenten (server)
  - Genereren uitvoer
    - Eventueel ophalen data uit bean
  - Bij indienen pagina
    - Data opslaan in bean
    - Eventueel informatie opslaan in verborgen veld formulier

UI-componten

- Registreren
  - Event listeners
  - Validators
  - Convertors

Overzicht UI-componenten

- UIColumn: kolom in UIData
- UICommand: actie uitvoeren
- UIData: tonen groep data
- UIForm: groep componenten, ~HTML-form
- UIGraphic: figuur
- UIInput: invoer, afgeleid van UIOutput
- UIMessage: bericht tonen
- UIMessages: berichten tonen
- UIOutput: uitvoer op pagina
- UIPanel: zorgt voor layout kindcomponenten
- UIParameter: parameter
- UISelectBoolean: invoer logische waarde, afgeleid van UIInput
- UISelectItem: een item
- UISelectItems: meerdere items
- UISelectMany: invoer meerdere items uit keuze, afgeleid UIInput
- UISelectOne: invoer item uit keuze, afgeleid UIInput
- UIViewRoot: basis componentboom

Renderen UI-componenten

- UI-component bevat NIET hoe hij gerenderd moet worden
- Aparte renderer
- UICommand
  - 2 renderers
    - Link
    - Knop
  - 2 corresponderende tags
    - commandButton
    - commandLink

Overzicht tags

- column: kolom in tabel
- commandButton: submit-knop
- commandLink: link
- dataTable: tabel
- form: formulier
- graphicImage: figuur
- inputHidden: verborgen veld
- inputSecret: wachtwoordveld
- inputText: tekstveld
- inputTextarea: invoer meerdere lijnen
- message: bericht
- messages: berichten
- outputLabel: label
- outputLink: link (zonder actie)
- outputFormat: bericht
- outputText: één lijn tekst
- panelGrid: groepeert componenten in tabelvorm
- selectBoolean: checkbox
- selectItem: één item in lijst
- selectItems: meerdere items in lijst
- selectManyCheckbox: lijst checkboxen
- selectManyListbox: lijst, meerdere items selecteren mogelijk
- selectManyMenu: keuzelijst, meerdere items selecteren mogelijk
- selectOneListbox:lijst, één item selecteren
- selectOneMenu: keuzelijst, één item selecteren
- selectOneRadio: lijst radiobuttons

Managed Beans

- Java Beans beschikbaar voor JSF
- Eigenschappen
  - Data UI-componenten
  - Invoer Gebruiker
- Methodes
  - Afhandelen events
  - …
- Hoe gekend in JSP-pagina?
  - faces-config.xml
  - In WEB_INF/

Configuratie Managed Beans

```
<managed-bean>
    <description>
      Bevat de gegevens die de gebruiker ingaf.
    </description>
    <managed-bean-name>GebruikersInfo</managed-bean-name>
    <managed-bean-class>
      be.hogent.iii.jsf.beans.GebruikersInfo
    </managed-bean-class>
    <managed-bean-scope>session</managed-bean-scope>
</managed-bean>
```

Naam, gebruikt in JSP

klassenaam

scope

Eigenschappen bean initialiseren

```
<managed-bean>
    <managed-bean-name>customer</managed-bean-name>
    <managed-bean-class>…</managed-bean-class>
    <managed-bean-scope> request </managed-bean-scope>
    <managed-property>
        <property-name>mailingAddress</property-name>
        <value>#{addressBean}</value>
    </managed-property>
</managed-bean>
<managed-bean>
    <managed-bean-name>addressBean</managed-bean-name>
    <managed-bean-class> … </managed-bean-class>
    <managed-bean-scope> none </managed-bean-scope>
</managed-bean>
```


Eigenschappen bean initialiseren

- In configuratie
  - <managed-property>
    - Naam
    - Waarde
      - Constante
      - Unified EL
        - Object in zelfde of ruimere scope
        - Nieuw aan te maken (scope = none)

Scope bean	Scope verwijzing
none	none
application	none, application
session	none, application, session
request	none, application, session, request

JSF

- Inleiding
- Voorbeeld
- Componenten en beans
- Levensloop
- Events
- UIData
- Taal-afhankelijkheid
- Navigatie
- Validatie
- Conversie
- Componenten

web.xml

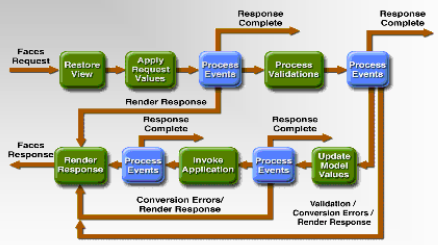
- Contextparameters
  - JSF-configuratie
  - Andere configuratie voor applicatie
    - Vb. implementatieklasse ThemaFabriek
- Alle aanvragen doorsturen naar één servlet (FacesServlet)

ThemaFabriek

- Constructor
  - Ophalen contextparameter
    - Via FacesContext
      - Informatie voor één aanvraag
    - Via ExternalContext
      - Toegang tot applicatieomgeving: sessie, ServletContext, aanvraag, antwoord, …

```
FacesContext fctx = FacesContext.getCurrentInstance();
ExternalContext ectx = fctx.getExternalContext();
String contextParam = ectx.getInitParameter("klasseThemaFabriek");
```


Levensloop JSF-pagina



(java.sun.com)

Postback ↔ eerste aanvraag

- Eerste aanvraag
  - Restore View
  - Render Response
- Postback
  - Formulier indienen naar zelfde url

Restore View

- Opbouwen view (indien postback)
  - Boomstructuur van componenten
    - Informatie ophalen (verborgen veld, server, …)
  - Binden van luisteraars, validators, … aan componenten
- Aanmaken nieuwe lege view
- Bewaren view in FacesContext
  - Informatie ivm één aanvraag

Apply Request Values

- HTTP-parameters → nieuwe waarde voor sommige componenten
  - Eventueel foutboodschap op FacesContext (bv. String → Integer mislukt)
- Events genereren
  - Doorsturen naar luisteraars
- Indien (methodes FacesContext)
  - renderResponse → Render Response fase
  - responseComplete → externe redirect

Process validations

- Validatie componenten
  - Indien gespecifieerd
- Validatie mislukt
  - Foutboodschap op FacesContext
  - Naar Render Response fase
- Events genereren
  - Doorsturen naar luisteraars
- Indien
  - renderResponse → Render Response fase
  - responseComplete → externe redirect

Update Model Values

- Data component (~value-attribuut) → data Java Beans
  - Mislukt? → Render Response faze
- Events genereren
  - Doorsturen naar luisteraars
- Indien
  - renderResponse → Render Response fase
  - responseComplete → externe redirect

Invoke Application

- Application level events
  - Bv.
    - Indienen formulier
    - Link naar andere pagina
  - Luisteraar voor bijhorende actie oproepen
  - Te genereren pagina bepalen
- Events gekoppeld aan componenten afhandelen

Render Response

- Opbouwen view (indien eerste aanvraag)
  - Boomstructuur van componenten
- JSP-container
  - Overloopt componenten
    - Genereren uitvoer
- Fouten in vorige fazen
  - Originele pagina + foutberichten
- Bewaren statusinformatie

FacesServlet

- Registratie FacesServlet in web.xml
  - Handelt alle aanvragen af (zie levensloop)

```
<servlet>
   <servlet-name>Faces Servlet</servlet-name>
   <servlet-class>javax.faces.webapp.FacesServlet</servlet-class>
   <load-on-startup>1</load-on-startup>
</servlet>
<servlet-mapping>
   <servlet-name>Faces Servlet</servlet-name>
   <url-pattern>/faces/*</url-pattern>
</servlet-mapping>
```


Bibliotheken

- jsf-api.jar
- jsf-impl.jar
- jstl.jar
- standard.jar
- commons-beanutils.jar
- commons-collections.jar
- commons-digester.jar
- commans-loggins.jar

JSF

- Inleiding
- Voorbeeld
- Componenten en beans
- Levensloop
- Events
- UIData
- Taal-afhankelijkheid
- Navigatie
- Validatie
- Conversie
- Componenten

Events

- Event-object
  - Gegenereerd door component
  - Informatie gebeurtenis
- Luisteraar registreren
  - Verwittigd bij optreden event
- Drie types
  - Value-change events: veranderen waarde component
  - Action events: klikken op knoppen, links, …
  - Data-model events: selecteren rij, …

Luisteraar

- Twee mogelijke methodes
  - Methode managed bean
  - Luisterklasse schrijven

Luisteraar

- Methode managed bean
  - Attribuut actionlistener van component
    - Met unified EL verwijzen naar methode

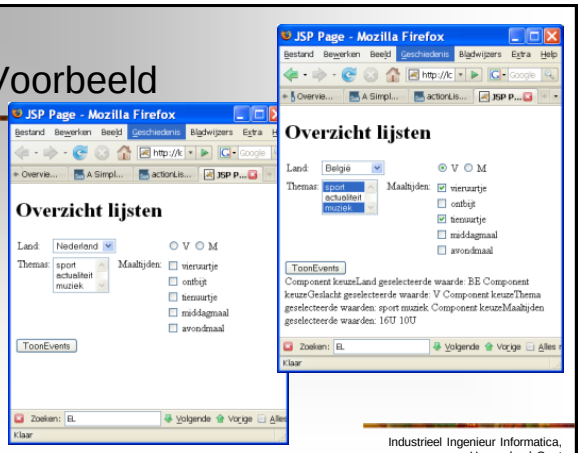
```
<h:commandButton  actionListener="#{GebruikersInfo.bepaalKeuze}" />
```

  - Signatuur methode in bean

```
public void bepaalKeuze(ActionEvent e) {
      keuze = "Hallo " + naam + ", u koos " + thema + ".";
}
```

  - javax.faces.event.ActionEvent

Voorbeeld



Voorbeeld

- http://localhost:8084/jsf/faces/lijsten.jsp

```
<h:selectOneRadio layout="lineDirection" id="keuzeGeslacht"
  valueChangeListener="#{InfoLijsten.itemGeselecteerd}">
    <f:selectItems value="#{ThemaFabriek.geslachten}" />
</h:selectOneRadio>
```

- be.hogent.iii.jsf.beans.InfoLijsten
  - Reageren op acties + bericht tonen

```
public void itemGeselecteerd(ValueChangeEvent e)
   throws AbortProcessingException  { … }
public void itemsGeselecteerd(ValueChangeEvent e)
   throws AbortProcessingException  { … }
```

- be.hogent.iii.jsf.beans.ThemaFabriek
  - Opvullen lijsten

Luisteraar

- Luisterklasse schrijven
  - Implementatie van de interface javax.faces.event.ValueChangeListener of javax.faces.event.ActionListener

```
public void processValueChange(ValueChangeEvent event) throws
   AbortProcessingException
public void processAction(ActionEvent event) throws
   AbortProcessingException
```

  - Registreren
    - Tag valueChangeListener of actionListener
      - Attribuut type: klasse luisteraar

```
<h:inputText  …>
   <f:valueChangeListener type="listeners.NameChanged" />
</h:inputText>
```

Luisteraar

- Methode managed bean
  - Attribuut actionlistener van component
    - Met unified EL verwijzen naar methode
- Luisterklasse schrijven
  - Implementatie van de interface ValueChangeListener of ActionListener
  - Registreren
    - Tag valueChangeListener of actionListener
      - Attribuut type: klasse luisteraar

JSF

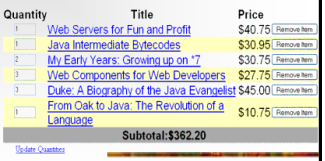
- Inleiding
- Voorbeeld
- Componenten en beans
- Levensloop
- Events
- UIData
- Taal-afhankelijkheid
- Navigatie
- Validatie
- Conversie
- Componenten

UIData component

- Voorbeeld
  - toonboeken.jsp
- Corresponderende tag
  - h:dataTable
- Toont verzameling (=value-attribuut) van data-objecten in tabelvorm

UIData component

- h:column
  - Beschrijft layout kolom
  - Kan ook knop, link, … zijn
- f:facet
  - Headers beschrijven
    - In <h:column>
  - Footers beschrijven
    - Na <h:column>-tags



JSF

- Inleiding
- Voorbeeld
- Componenten en beans
- Levensloop
- Events
- UIData
- Taal-afhankelijkheid
- Navigatie
- Validatie
- Conversie
- Componenten

Taal-afhankelijke informatie in bundels

- Labels per taal in .properties bestanden
  - Properties-bestand
    - Tekstbestand
    - Per lijn een sleutel-waarde-paar
      sleutel=waarde
    - Commentaar na '#'
- Referentie naar bundel

```
<f:loadBundle basename="be.hogent.iii.jsf.beans.labels" var="labels"/>
```

- Label ophalen uit bundel

```
<f:facet name="header">
  <h:outputText value="#{labels.titel}"/>
</f:facet>
```

JSF

- Inleiding
- Voorbeeld
- Componenten en beans
- Levensloop
- Events
- UIData
- Taal-afhankelijkheid
- Navigatie
- Validatie
- Conversie
- Componenten

Navigatie

- Opeenvolging pagina's
- Regels
  - Wat na klikken op knop of link?
  - In configuratie (faces-config.xml)

```
<navigation-rule>
  <from-view-id>/greeting.jsp</from-view-id>
  <navigation-case>
    <from-outcome>success</from-outcome>
    <to-view-id>/response.jsp</to-view-id>
  </navigation-case>
</navigation-rule>
```


Navigatie

- Welke regel?
  - action-attribuut
    - String
    - Resultaat methode

```
<h:commandButton id="submit" action="success" value="Submit" />
<h:commandButton value="Submit" action="#{cashier.submit}" />
```


Voorbeeld

Navigatie in faces-config.xml

```
<navigation-rule>
    <from-view-id>/gokken.jsp</from-view-id>
    <navigation-case>
      <from-action>#{Casino.speel}</from-action>
      <from-outcome>pech</from-outcome>
      <to-view-id>/speelOpnieuw.jsp</to-view-id>
    </navigation-case>
    <navigation-case>
      <from-action>#{Casino.speel}</from-action>
      <from-outcome>gewonnen</from-outcome>
      <to-view-id>/gewonnen.jsp</to-view-id>
    </navigation-case>
</navigation-rule>
```


Casino

- faces-config.xml

```
<managed-bean>
    <managed-bean-name>Casino</managed-bean-name>
    <managed-bean-class>be.hogent.iii.jsf.beans.Casino</managed-bean-class>
    <managed-bean-scope>application</managed-bean-scope>
</managed-bean>
```

- Haalt gokken van de sessie
- Gewonnen?

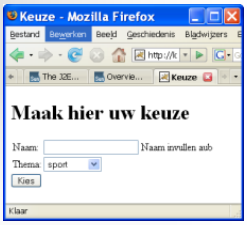
JSF

- Inleiding
- Voorbeeld
- Componenten en beans
- Levensloop
- Events
- UIData
- Taal-afhankelijkheid
- Navigatie
- Validatie
- Conversie
- Componenten

Validatie

- Controle gebruikersinvoer
- keuzeJSF.jsp

Geen invoer



Enkel spaties

Validatie

- Attribuut required = true
  - Automatisch foutboodschap indien niet opgegeven
- Attribuut validator
  - Methode

```
public void controleerNaam(FacesContext context,
    UIComponent toValidate, Object value) { … }
```

  - Validatie mislukt instellen
  - Foutboodschap

Ingebouwde validatie

- validateDoubleRange -tag
  - Interval (reëel)
- validateLength-tag
  - Lengte (string)
- validateLongRange-tag
  - Interval (geheel)
- Attributen
  - Minimum
  - Maximum

```
<h:inputText id="quantity" size="4" value=
    "#{item.quantity} " >
    <f:validateLongRange minimum="1"/>
</h:inputText>
<h:message for="quantity"/>
```


Andere methode eigen validatie

- Implementeren Validator-interface

```
public void validate(FacesContext context, UIComponent component,
    Object toValidate)
```

- Koppelen aan component

```
<h:inputText ... >
  <f:validator validatorId="customValidator" />  ...
</h:inputText>
```

- Registeren bij applicatie (faces-config.xml)

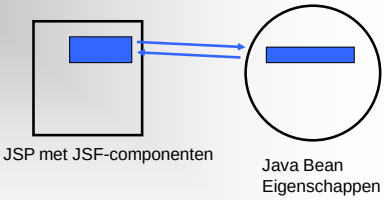
```
<validator>
  ...
  <validator-id>customValidator</validator-id>
  <validator-class>
    validators.CustomValidator
  </validator-class>
</validator>
```


JSF

- Inleiding
- Voorbeeld
- Componenten en beans
- Levensloop
- Events
- UIData
- Taal-afhankelijkheid
- Navigatie
- Validatie
- Conversie
- Componenten

Convertor

- Data component
  - Model view
    - Bv. int, long
  - Presentatie view
    - Bv. datum in bepaald string-formaat
- Automatische conversie



Automatische conversie

- Bean

```
Integer age = 0;
public Integer getAge(){ return age;}
public void setAge(Integer age) {this.age = age;}
```

- Component

```
<h:inputText value="#{MijnBean.age}" />
```

- Achter de scherm wordt de IntegerConverter gebruikt

Expliciete conversie

- Gebruik converter-attribuut

```
<h:inputText converter="javax.faces.convert.IntegerConverter“ … />
```

- Gebruik converter-tag

```
<h:inputText value="#{LoginBean.age}" />
  <f:converter converterId="Integer" />
</h:inputText>
```


Conversie datums

- convertDateTime-tag

```
<h:outputText value="#{cashier.shipDate}">
  <f:convertDateTime pattern="EEEEEEEE, MMM dd, yyyy" />
</h:outputText>
```


Conversie getallen

- convertNumber-tag

```
<h:outputText value="#{cart.total}" >
  <f:convertNumber type="currency“/>
</h:outputText>
```

- Bv. aantal cijfers voor of na komma, …

Eigen converter

- Koppelen aan component

```
<h:inputText converter="CreditCardConverter“ …>
  ...
</h:inputText>
```

- Registreren in applicatie (faces-config.xml)

```
<converter>
  <description> … </description>
  <converter-id>CreditCardConverter</converter-id>
  <converter-class>
    converters.CreditCardConverter
  </converter-class>
</converter>
```

Eigen converter - klasse

- Implementatie Converter-interface
- Twee methodes
- Presentatie → Model

Object getAsObject(FacesContext context, UIComponent component, String value)

- Model → Presentatie

String getAsString(FacesContext context, UIComponent component, Object value)

JSF

- Inleiding
- Voorbeeld
- Componenten en beans
- Levensloop
- Events
- UIData
- Taal-afhankelijkheid
- Navigatie
- Validatie
- Conversie
- Componenten

rendered-attribuut

- Logische uitdrukking
  - Bepaalt of de component uitvoer genereert

```
<h:commandLink id="check"
    ...
    rendered="#{cart.numberOfItems > 0}">
  <h:outputText
    value="#{bundle.CartCheck}"/>
</h:commandLink>
```


immediate attribuut

- In welke faze events, validaties, … uitvoeren
- Indien true
  - Apply request values faze
  - Vervroegd uitvoeren
- Gebruikt bij commandLink
  - Voorkomt bv. dat formuliergegevens verwerkt worden (Logout, Annuleren, …)

Componenten binden

- Normaal binden waarde component aan eigenschap Java Bean
- Soms component zelf binden aan eigenschap Java Bean
  - type = type component
  - Component kan gemanipuleerd worden vanuit Java Bean

```
<inputText binding="#{UserNumberBean.userNoComponent}" />
```

```
UIInput userNoComponent = null;
public void setUserNoComponent(UIInput userNoComponent) {
    this.userNoComponent = userNoComponent; }
public UIInput getUserNoComponent() {
    return userNoComponent; }
```


Informatie

- The Java EE 5 Tutorial (http://java.sun.com/javaee/5/docs/tutorial/doc/bnaph.html)
  - Hoofdstuk 10 – 15
- JSF Tag Library (http://java.sun.com/javaee/javaserverfaces/1.2_MR1/docs/tlddocs/index.html)

Hoofdstuk 8

Struts

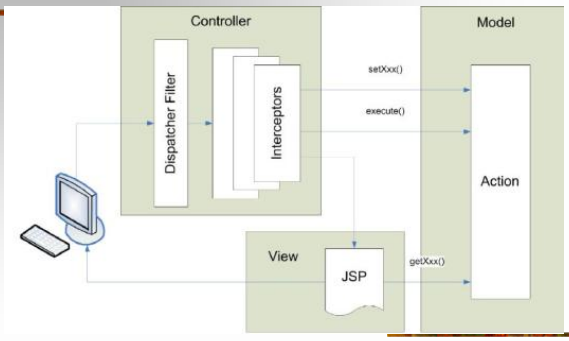
Struts 2

Veerle Ongenae

Struts

- Implementatie MVC/Model 2
  - Eigenlijk MVC2 of pull-MVC
    - Acties maken deel uit van het model
- Struts 2: samengaan
  - Struts
  - WebWork

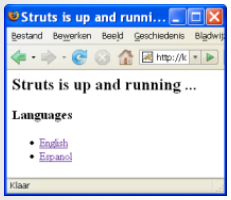
Principe Struts 2



Principe Struts 2

- Controller
  - Filter
  - Interceptors
    - Instellen eigenschappen actieklassen
    - Oproepen actie
- Model
  - Actieklassen
    - Uitvoeren actie
    - Data: instellen en opvragen
- View
  - Tonen model
  - Bv. JSP

Voorbeeld



- http://localhost:8084/struts/HelloWorld.action
- Actie
  - example.HelloWorld.java
- View
  - HelloWorld.jsp

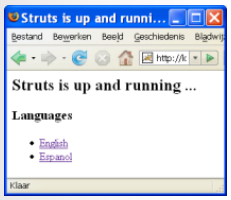
Controller

- Filter
  - Verschil met servlets
    - Genereren GEEN antwoord
  - Request bekijken en manipuleren
    - Headers, parameters, attributen
  - Response: headers en body aanpassen
  - Ketting van filters
- Configuratie in web.xml

Configuratie Controller

```
<filter>
      <filter-name>struts2</filter-name>
      <filter-class>
            org.apache.struts2.dispatcher.FilterDispatcher
      </filter-class>
  </filter>

  <filter-mapping>
      <filter-name>struts2</filter-name>
      <url-pattern>/*</url-pattern>
  </filter-mapping>
```

Actie



- HelloWorld.java
- Java-klasse
  - Afgeleid van
    com.opensymphony.xwork2.ActionSupport
  - Methode execute
    - Eigenlijke actie
    - Resultaat: string → type resultaat actie
      - Bv. SUCCESS
  - Eigenschappen
    - Data

Actie: ingebouwde methodes, …

- getText()
  - Opvragen informatie uit properties-bestand (zelfde map als klasse)
    - package.properties
    - package_es.properties
- SUCCESS
  - Resultaat uitvoeren actie
  - Beschikbare resultaatcodes (via ActionSupport)

```
String SUCCESS = "success";
String NONE    = "none";
String ERROR   = "error";
String INPUT   = "input";
String LOGIN   = "login";
```

Samenvoegen

- URL verbinden met actie
- Resultaat actie koppelen aan view
- struts.xml
  - In WEB-INF/classes
  - Opnemen in basis van Java-map

struts.xml

- Configuratie voor Struts
  - Overschrijven standaardconfiguratie
  - Beschrijving acties
  - Kan opgesplitst worden in verschillende xml-bestanden

```
<struts>

  <constant name="struts.enable.DynamicMethodInvocation"
   value="false" />
  <constant name="struts.devMode" value="false" />

  <include file="example.xml"/>
</struts>
```

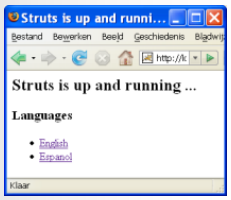
struts.xml

- example.xml

```
<struts>

  <package name="example" extends="struts-default">

    <action name="HelloWorld" class="example.HelloWorld">
      <result>/HelloWorld.jsp</result>
    </action>
  </package>
</struts>
```

Beschrijving actie

View



- HelloWorld.jsp
- Gebruik taglib struts-tags
  - property-tag
    - Eigenschap van de bijhorende actieklasse tonen
  - url-tag
    - Maakt URL op basis van action-attribuut
    - Toevoegen parameters met param-tag
    - Gebruikt in a-tag
  - a-tag
    - Genereert een link
  - text-tag
    - Haalt tekst uit properties-bestand

View

- Toegang tot data via OGNL
  - Object Graph Navigation Language (OGNL)
  - Maakt gebruik van een context map (Map-object)
    - value-stack
      - Een stack met objecten
        - Tijdelijke objecten (bv. van een foreach, …)
        - Action-object
    - Named objects (~scope): #application, #session, #request, #parameters, #attr (page, request, session, application)
  - %{ …}
    - Te evalueren uitdrukking
  - value-attribuut: OGNL-uitdrukking

OGNL value-stack

- Objecten op value-stack
  - Eigenschappen onmiddellijk opvragen
    - Zonder verwijzing naar object
  - Bv. 2 objecten op value-stack
    - Type Stad
    - Type Foto
  - naam
  - omschrijving
  - url

Foto
•String getOmschrijving() •String getUrl()

Stad
•String getNaam()

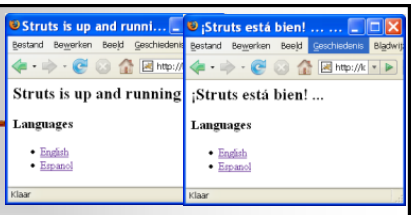
OGNL named objects

- #parameters[‘thema’] of #parameters.thema (~getParameter)
- #request[‘view] of # request.view (~getAttribute)
- #session[‘naam’] of #session.naam
- #application[‘catalogus’] of #application.catalogus
- #attr[‘view’] of #attr.view

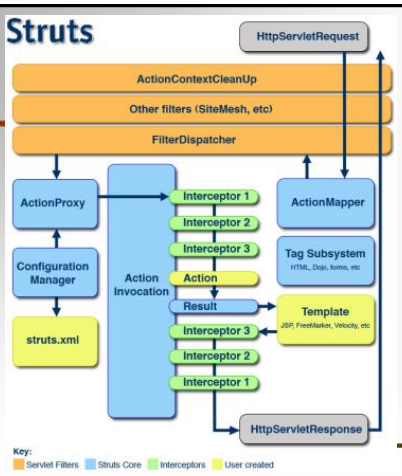
Achter de schermen

- HTTP-request voor HelloWorld.action
- web.xml → doorspelen aan FilterDispatcher (ingang applicatie)
- struts.xml → bijhorende Actie-object bepalen
- Aanmaken Actie-object; oproepen execute-methode
- Resultaat execute → view bepalen (JSP)
- JSP parsen
  - property-tag → oproepen methode actie-object
- HTTP-antwoord naar client

Localization



- Pagina’s taalonafhankelijk maken
- Labels per taal in properties-bestanden
  - package.properties (voor het hele package)
  - NaamClass.properties (per klasse)
- Taal instellen via parameter
  - request_locale
  - Opgevangen door een interceptor (I18n Interceptor)
    - Plaatst locale op ActionContext



Interceptors

- Localization
- Oproepen actie
- Validatie

Voorbeeld 2

- http://localhost:8084/struts/Welcome.jsp
- http://localhost:8084/struts/Welcome.action

url samengevat

- Link naar bron

```
<link href="<s:url value="/css/examplecss"/>" …. />
```

- Link naar actie

```
<a href="<s:url action="Login_input"/>">Sign On</a>
```

- Link met parameter

```
<s:url id="url" action="HelloWorld">
    <s:param name="request_locale">en</s:param>
</s:url>
<s:a href="%{url}">English</s:a>
```

Koppeling naar struts

- Welcome.action en Welcome.jsp
  - Geen actie
    - Geen class-attribuut
    - Default
      - ActionSupport
  - Wel struts-tags
- struts.xml (example.xml)

```
<action name="Welcome" >
   <result>/Welcome.jsp
   </result>
</action>
```

- Alternatieven

```
<action name="*" >
  <result>/tutorial/{1}.jsp</result>
</action>
```

- of

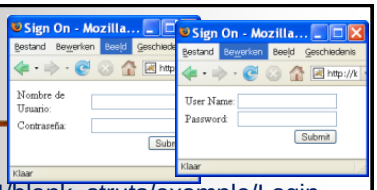
Lege actieklasse

```
<action name="*"
    class="example.ExampleSupport">
    <result>/{1}.jsp</result>
</action>
```

struts.xml

- Kan meerdere action-elementen bevatten
- Het eerste element dat toegepast kan worden, wordt toegepast
  - <action name="*" …> steeds als laatste

Inlogformulier



- http://localhost:8084/blank_struts/example/Login_input.action
- Maakt gebruik van de struts-tags s:form, s:textfield, s:password en s:submit
- Actie formulier: Login.action
- key-attribuut (s:textfield en s:password)
  - Sleutel om data bijhorend label uit properties-bestand
  - Naam parameter van bijhorende input-tag

Verwerken loginformulier

- Actie: Login.java
- Eigenschappen actie-klassen worden automatisch ingevuld met formulierparameters
- Actie
  - Methode execute
    - Resultaat
      - INPUT → opnieuw naar login-formulier
      - SUCCESS → verder naar applicatie

Meerdere resultaten mogelijk

- In struts.xml
- Meerdere result-tags in de action-tag
  - name-attribuut = resultaat actie
    - Standaard: success
  - type-attribuut = type resultaat
    - Doorsturen naar JSP, naar andere actie, …

```
<action name="Login" class="example.Login">
    <result name="input">/Login.jsp</result>
    <result type="redirect-action">Menu</result>
</action>
```

Validatie

- Controle invoer gebruiker
- XML-document
  - Naamactie-validation.xml
  - Bv. Login-validation.xml
    - Naam te valideren veld, type validatie, foutboodschap (kan uit .properties-bestand → key)
- Eerste keer tonen loginformulier
  - Geen validatie

Validatie

- Eerste keer geen validatie
  - Speciale methode van ActionSupport
    - input → resultaat = input
  - Configuratie in struts.xml (example.xml)

```
<action name="Login_*" method="{1}" class="example.Login">
    <result name="input">/Login.jsp</result>
    <result type="redirect-action">Menu</result>
</action>
```

  - Actie: Login_input
    - Validatie + execute worden NIET uitgevoerd
- Anders
  - Validatie + execute
  - Bij fout → input

Beschikbare validatie

- required validator
  - Niet null?
- requiredstring validator
  - Niet null? Niet allemaal spaties?
- int validator
  - Int? In interval?
- date validator
  - Int? In interval?
- expression validator
  - Logische OGNL-uitdrukking
- fieldexpression validator
  - Logische OGNL-uitdrukking, gekoppeld aan een veld
- email validator
  - Email?
- url validator
  - url?
- conversion validator
  - Conversie gelukt?
- stringlength validator
  - Lengte in interval?
- regex validator
  - Voldoet aan reguliere uitdrukking?

Voorbeelden validatie

```
<validators>
  <!-- Field Validators for email field -->
  <field name="email">
    <field-validator type="required">
      <message>...</message>
    </field-validator>
    <field-validator type="email">
      <message>...</message>
    </field-validator>
  </field> ...
```

```
<field name="bar">
  <field-validator type="required">
    <message>...</message>
  </field-validator>
  <field-validator type="int">
    <param name="min">6</param>
    <param name="max">10</param>
    <message>bar must be between
    ${min} and ${max}, current value is
    ${bar}.</message>
  </field-validator>
</field>  ...
```

Voorbeelden validatie

```
<field name="bar2">
  <field-validator type="regex">
    <param name="expression">[0-9],[0-9]</param>
    <message>...</message>
  </field-validator>
</field>
<field name="date">
  <field-validator type="date">
    <param name="min">12/22/2002</param> <param
    name="max">12/25/2002</param> <message>...</message>
  </field-validator>
</field>
```

ActionContext

- Context waarin een actie wordt uitgevoerd

```
import com.opensymphony.xwork2.ActionContext;
ActionContext context = ActionContext.getContext();
```

- Toegang tot
  - Sessie
  - Parameters
  - …

Referenties

- Starting with Struts 2, Ian Roughley (http://www.infoq.com/minibooks/starting-struts2)
- Struts tutorial (http://struts.apache.org/2.0.9/docs/bootstrap.html)
- Struts guide (http://struts.apache.org/2.0.9/docs/guides.html)

Referenties

- Struts API (http://struts.apache.org/2.0.12/struts2-core/apidocs/index.html)
- Struts tags (http://struts.apache.org/2.0.12/docs/tag-reference.html)

Hoofdstuk 9

Webservices

Webservices

Veerle Ongenae

Overzicht

- Gegevens uitwisselen
- SOAP
- Webservices
  - Technologieën
  - Gebruik

Gegevens uitwisselen

- Data uitwisselen tussen programma's op verschillende computers
- Tot nu toe
  - EDI
  - CORBA, RMI, DCOM

EDI

- Electronic Data Interchange
- WAN voor een geografische regio, waar partners kunnen inpluggen
- Afspraak: boodschappen en protocol voor datatransfer
- Networkdetails overgelaten aan de implementatie
- Verschillende netwerken
  - Duur om in te stappen
  - Moeilijk en duur om andere netwerken te bereiken

CORBA, RMI, DCOM

- Afkortingen
  - Common Object Request Broker Architecture
  - Remote Method Invocation
  - Distributed Component Object Model
- Gedistribueerde object-infrastructuur over het Internet
- Transportprotocol bovenop TCP/IP (verschillend voor CORBA, RMI en DCOM) voor verschillende platformen
- Afspraken over communicatie tussen objecten

CORBA, RMI, DCOM

- Voordeel: de verschillende servers hoeven niet tot het zelfde netwerk te behoren
- Kunnen enkel met dezelfde infrastructuur praten, tenzij extra lagen op een reeds ingewikkelde structuur

SOAP

- SOAP zorgt voor verandering:
  - Mogelijkheid tot het uitwisselen van data overal op het web
  - Geen koppeling tussen data en transport
  - Niet binair

Overzicht

- Gegevens uitwisselen
- SOAP
- Webservices
  - Technologieën
  - Gebruik

SOAP

- Simple Object Acces Protocol
- Protocol gebaseerd op XML
- Uitwisseling van informatie in een gedecentraliseerde, gedistribueerde omgeving
- Transport via het web (HTTP, …)
- SOAP-bericht
  - Verstuurd door zender
  - Naar ontvanger

SOAP en HTTP

- De eerste lijn en de headerlijnen blijven bestaan
- Het corpus van een HTTP-aanvraag en HTTP-antwoord bevat XML ipv HTML, …
- Analoog via FTP of SMTP

SOAP

- XML-envelop
  - Met XML-inhoud
- Een aantal regels voor servers die een SOAP-boodschap ontvangen
- Maakt gebruik van bestaande transportprotocols (bv. HTTP)
- Oorspronkelijk Microsoft, ondertussen standaard van W3C

Wat is er nodig om data uit te wisselen via SOAP?

- Afgesproken DTD of XML Schema voor de uit te wisselen gegevens
- SOAP-server
  - Afhandelen aanvragen
- Client
  - Software om XML-documenten te genereren

Overzicht SOAP

- XML-tags die een SOAP-bericht beschrijven en die een framework geven voor de inhoud van het bericht
- Regels voor het uitwisselen van zelf gedefinieerd gegevenstypes (o.a. aanvaarden, verwerpen, uitzonderingen)
- Conventies voor het uitvoeren van methodes op een andere server en het voorstellen van het resultaat

SOAP Bericht: voorbeeld aanvraag

```
POST /ZwiftBooks HTTP/1.1
Host: www.zwiftbooks.com
Content-Type:text/xml
Content-Length: 134
SOAP Action: "Some-URI"

<Envelope xmlns="http://www.w3.org/2001/09/soap-envelope"
   encodingStyle="http://www.w3.org/2001/09/soap-encoding">
    <Body>
        <zwiftbooks:GetGuaranteedDeliveryTime
          xmlns:zwiftbooks="www.zwiftbooks.com">
            <zwiftbooks:isbn>0-13-18188-222-2</zwiftbooks:isbn>
            <zwiftbooks:zipcode>75240</zwiftbooks:zipcode>
        </zwiftbooks:GetGuaranteedDeliveryTime>
    </Body>
</Envelope>
```


SOAP Bericht: voorbeeld antwoord

```
HTTP/1.1 200 OK
Content-Type: text/xml
Content-Length: 122

<Envelope xmlns="http://www.w3.org/2001/09/soap-envelope">
    <Body>
        <zwiftbooks:GetBestDeliveryTimeResponse
          xmlns:zwiftbooks="www.zwiftbooks.com">
            <zwiftbooks:Time>2 hours</zwiftbooks:Time>
        </zwiftbooks:GetBestDeliveryTimeResponse>
    </Body>
</Envelope>
```


Structuur SOAP Bericht

- SOAP-envelop
- SOAP-hoofding
- SOAP-corpus

Structuur SOAP Bericht: envelop

- SOAP-envelop:
  - Envelope-element
    - Attributen
      - encodingStyle: hoe geserialiseerd wordt
  - Rootelement van het XML-document dat een SOAP-bericht voorstelt
  - Verplicht

Structuur SOAP Bericht: hoofding

- SOAP-hoofding:
  - Header-element
  - Extra richtlijnen en informatie voor SOAP-server
    - Transacties
    - Beveiliging
    - Filters
    - …
  - Optioneel

Voorbeeld: SOAP-hoofding

```
<Header>
    <n:alertcontrol
      xmlns:n="http://example.org/alertcontrol">
        <n:priority>1</n:priority>
        <n:expires>2001-06-22T14:00:00-05:00</n:expires>
    </n:alertcontrol>
</Header>
```


Structuur SOAP Bericht: corpus

- SOAP-corpus:
  - Body-element
  - Bevat de uit te wisselen XML
    - Onafhankelijk van SOAP-structuur
    - Domein-specifieke structuur
  - Inhoud
    - Data/Resultaat
    - Aanroep van een methode
  - Verplicht

SOAP-berichtenpad

- Bestaat uit SOAP-knopen:
  - SOAP-zender
  - Nul of meerdere tussenliggende SOAP-servers
  - SOAP-ontvanger
- Tussenliggende SOAP-servers
  - Stuurt SOAP-bericht door
  - Behandelt de voor hem bedoelde SOAP-hoofdingen

SOAP-rollen

- Elke SOAP-knoop heeft één of meerdere rollen
- Mogelijke rollen
  - Gespecifieerd door SOAP:
    - none (niet bedoeld voor een specifieke knoop)
    - next (ontvanger en alle tussenliggende)
    - ultimateReceiver (ontvanger)
  - Gespecifieerd door de applicatie
    - Proxy
    - Beveiliging
    - Caching
    - …

SOAP-rollen

- Een SOAP-hoofding kan aangeven voor welke SOAP-knoop de hoofding bedoeld is door gebruik te maken van rollen
  - Attribuut role
  - Waarde attribuut
    - Rol gespecifieerd door SOAP
    - URI SOAP-knoop

Voorbeeld: SOAP-hoofding

```
<Header
  role="http://schemas.xmlsoap.org/soap/actor/next">
    <n:alertcontrol xmlns:n="http://example.org/alertcontrol"
      mustUnderstand="true">
        <n:priority>1</n:priority>
        <n:expires>2001-06-22T14:00:00-05:00</n:expires>
    </n:alertcontrol>
</Header>
```

Regels voor SOAP-server

- Indentificatie van de stukken van het SOAP-bericht die voor de huidige server bedoeld zijn
- Nagaan of deze server alle delen van de headers die bedoeld zijn voor deze server en die het attribuut mustUnderstand hebben ondersteunt. Indien niet fout genereren

Regels voor SOAP-server

- Uitvoeren van de stukken van de headers bedoeld voor deze server
- Indien niet de SOAP-ontvanger: alle headers bedoeld voor deze server verwijderen en bericht doorsturen

SOAP-fouten

- Wanneer
  - Niet begrijpen bericht
  - Fout bij behandelen van bericht
- fault-element
  - Kind van body-element
  - Kinderen
    - faultcode: foutcode
    - faultstring: leesbare verklaring
    - detail: informatie over het probleem

Overzicht

- Gegevens uitwisselen
- SOAP
- Webservices
  - Technologieën
  - Gebruik

Webservices

- Kader
  - Applicaties bouwen op basis van losse componenten
- Wat?
  - Diensten aanbieden voor applicaties
  - Vergelijking
    - Presentatielaag maakt gebruik van diensten van de applicatielogica
- Technologie
  - Verschillende protocollen

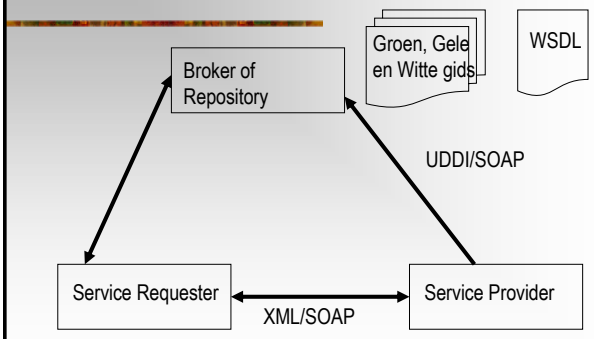
Verschillende facetten van webservices

- Beschrijving
  - Aangeboden diensten
- Publiceren
  - Bij een ‘repository’
  - Vergelijkbaar met telefoonboek: witte, gele en groene gids
- Uitvoeren
  - Een applicatie maakt gebruik van de aangeboden diensten
- Antwoorden
  - De dienst stuurt een resultaat terug naar de applicatie

Webservices: onderdelen

- Service provider
  - Biedt een interface
  - Voor software
  - Bepaalde set taken
- Service requester
  - Doet een aanvraag bij service provider
  - Krijgt antwoord van service provider
  - Maakt deel uit van een applicatie
- Broker
  - Publiceert beschikbare diensten

Overzicht webservices



Webservices: technologieën

- UDDI
  - Universal Description, Discovery and Integration
- WSDL
  - Web Services Definition Language
- SOAP

UDDI

- UDDI
  - XML-Protocol
  - Webservices beschrijven en registreren bij een broker
  - Communicatie tussen
    - Service provider en broker
    - Broker en service requester
  - Inhoud van een SOAP-envelop

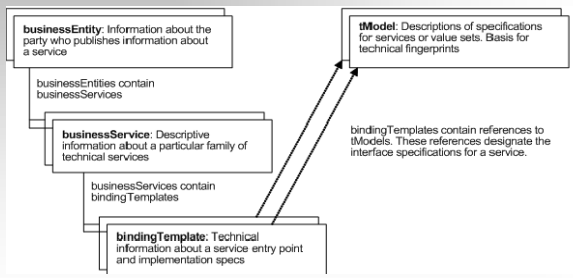
Informatie in een register

- Witte gids
  - Contactinformatie: naam, adres, webstek, …
  - Registratie van de provider
- Gele gids
  - Categorieën van diensten
  - Hoe vind een client een webservice
- Groene gids
  - Technische informatie
  - Hoe kan een client gebruik maken van een webservice

Elementen van UDDI

- businessEntity
  - Informatie over bepaalde provider
- businessService
  - Beschrijving van een bepaalde webservice bestaande uit één of meerdere diensten
- bindingTemplate
  - Technische informatie, informatie voor implementatie
- tModel
  - Technische informatie per dienst

Elementen van UDDI



WSDL

- WSDL
  - XML-standaard W3C
  - Beschrijven webservice
    - Interface
    - Implementatiedetails
  - Document kan meegestuurd worden met UDDI
  - Bepaalt hoe de clientapplicatie de aangeboden dienst moet aanspreken

WSDL

- Twee versies
  - WSDL 1.1 (http://www.w3.org/TR/2001/NOTE-wsdl-20010315)
  - WSDL 2.0 (http://www.w3.org/TR/2007/REC-wsdl20-primer-20070626/)

Voorbeeld WSDL 1.1

```
<definitions name="BoekAgent"
   targetNamespace="http://www.zwiftbooks.com/"
   xmlns="http://schemas.xmlsoap.org/wsdl/"
   xmlns:tns="http://www.zwiftbooks.com/"
   xmlns:xsd="http://www.w3.org/2001/XMLSchema"
   xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/" >

   <message name="GetDeliveryInfoRequest">
      <part name="zipcode" type="xsd:integer"/>
      <part name="isbn" type="xsd:string"/>
   </message>

   <message name="GetDeliveryInfoResponse">
      <part name="result" type="xsd:duration"/>
   </message>
```


Voorbeeld WSDL 1.1

```
<portType name="BoekAgent">
   <operation name="GetDeliveryInfo" parameterOrder="zipcode
     isbn">
      <input message="tns:GetDeliveryInfoRequest"
        name="GetDeliveryInfoRequest "/>
      <output message="tns:GetDeliveryInfoResponse "
        name="GetDeliveryInfoResponse"/>
   </operation>
   <!-- andere diensten -->
</portType>
```


Voorbeeld WSDL 1.1

```
<binding name="BoekAgentBinding" type="tns:BoekAgent">
   <soap:binding style="rpc"
     transport="http://schemas.xmlsoap.org/soap/http" />
   <operation name="GetDeliveryInfo">
      <soap:operation
        soapAction="http://zbooks.org/action/ZBooks.Get-
        DeliveryInfo" />
      <input>
         <soap:body use="encoded"
           namespace="http://zbooks.org/message/"
           encodingStyle="http://schemas.xmlsoap.org/soap/en
           coding/" />
      </input>
```

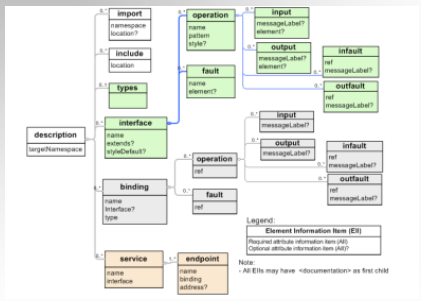
Voorbeeld WSDL 1.1

```
            <output>
                <soap:body use="encoded"
                  namespace="http://zbooks.org/message/"
                  encodingStyle="http://schemas.xmlsoap.org
                  /soap/encoding/" />
            </output>
        </operation>
    </binding>
</definitions>
```


WSDL 1.1 -structuur

- Messages
  - Berichten
- Port Types
  - Verzameling van alle diensten van een webservice
- Bindings
  - Beschrijving concreet formaat van de berichten en het gebruikte protocol
- Services
  - Beschrijving van een webservice bestaande uit verschillende “port types”

WSDL 2.0 -structuur



WSDL 2.0 -structuur

- types: beschrijving berichten in XML Schema
- interface: abstracte funcionaliteit
- binding: hoe toegang tot diensten
- service: waar toegang tot diensten

Voorbeeld WSDL 2.0

```
<?xml version="1.0" encoding="utf-8" ?>
<description xmlns="http://www.w3.org/ns/wsdl"
    targetNamespace= "http://greath.example.com/2004/wsdl/resSvc"
    xmlns:tns= "http://greath.example.com/2004/wsdl/resSvc" . . . >
    . . .
</description>
```


Voorbeeld WSDL 2.0

```
<?xml version="1.0" encoding="utf-8" ?>
<description  . . . >
    ...
<types>
    <xs:schema ...>
     <xs:element name="checkAvailability"
    type="tCheckAvailability"/>
    <xs:complexType name="tCheckAvailability">
    <xs:sequence>
          <xs:element name="checkInDate" type="xs:date"/>
          <xs:element name="checkOutDate" type="xs:date"/>
          <xs:element name="roomType" type="xs:string"/>
    </xs:sequence> </xs:complexType>
          <xs:element
          name="checkAvailabilityResponse"
          type="xs:double"/>
          <xs:element name="invalidDataError"
          type="xs:string"/>
    </xs:schema>
    </types>
          . . .
</description>
```

Voorbeeld WSDL 2.0

```
<?xml version="1.0" encoding="utf-8" ?>
<description …>
    . . . <types> ... </types>
<interface name = "reservationInterface" >
<fault name = "invalidDataFault" element = "ghns:invalidDataError"/>
<operation name="opCheckAvailability" pattern="http://www.w3.org/ns/wsdl/in-out"
    style="http://www.w3.org/ns/wsdl/style/iri" wsdlx:safe = "true">
    <input messageLabel="In" element="ghns:checkAvailability" />
    <output messageLabel="Out" element="ghns:checkAvailabilityResponse" />
    <outfault ref="tns:invalidDataFault" messageLabel="Out"/>
</operation> </interface> . . . </description>
```


Voorbeeld WSDL 2.0

```
<?xml version="1.0" encoding="utf-8" ?>
<description … > . . . <types> . . . </types>
<interface name = "reservationInterface" > ... </interface>
<binding name="reservationSOAPBinding" interface="tns:reservationInterface"
    type="http://www.w3.org/ns/wsdl/soap"
    wsoap:protocol="http://www.w3.org/2003/05/soap/bindings/HTTP/">
    <operation ref="tns:opCheckAvailability"
    wsoap:mep="http://www.w3.org/2003/05/soap/mep/soap-response"/>
    <fault ref="tns:invalidDataFault" wsoap:code="soap:Sender"/>
</binding>
. . .
</description>
```


Voorbeeld WSDL 2.0

```
<?xml version="1.0" encoding="utf-8" ?>
<description …>
. . . <types> . . . </types>
<interface name = "reservationInterface" > . . . </interface>
<binding name="reservationSOAPBinding" interface="tns:reservationInterface" . . . >
. . . </binding>
<service name="reservationService" interface="tns:reservationInterface">
    <endpoint name="reservationEndpoint" binding="tns:reservationSOAPBinding"
    address ="http://greath.example.com/2004/reservation"/>
</service>
</description>
```


Webservices: technologieën

- SOAP
  - Communicatie tussen broker, service provider en service requester
  - SOAP-corpus bevat UDDI

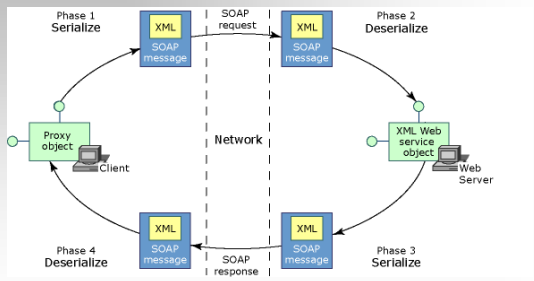
Overzicht

- Gegevens uitwisselen
- SOAP
- Webservices
  - Technologieën
  - Gebruik

Wat is een webservice?

- Uitvoeren van een methode op een andere server
- Een dienst
  - Eén of meerdere methodes
  - Een stuk programma met een bepaalde functionaliteit
  - Toegankelijk via XML (SOAP) en HTTP

Overzicht



Versturen SOAP-aanvraag

- Proxy-object: een lokaal object voor de clientapplicatie die webservice “voorstelt”
- Client-applicatie roept methodes van het proxy-object op
- Infrastructuur serialiseert → SOAP-bericht
- Bericht wordt verstuurd via het netwerk
  - Communicatie via HTTP en het SOAP-protocol

Verwerken SOAP-bericht

- Serverinfrastructuur
  - Deserialisatie van het SOAP-bericht
  - Al dan niet aanmaken object webservice
  - Methode webservice uitvoeren
  - Antwoord serialiseren → SOAP-bericht
  - SOAP-antwoord wordt verstuurd via het netwerk

SOAP-antwoord vertalen

- Clientinfrastructuur
  - Ontvangt SOAP-antwoord
  - Deserialiseert
  - Geeft resultaat door aan proxy-object
- Proxy-object genereert resultaat voor clientapplicatie

Voorbeeld

- Boek-object opvragen aan de hand van ISBN/ID
- In .NET
  - http://localhost/WebCatalogus/BoekenLijst.asmx
- In J2EE
  - http://localhost:8084/WebCatalogus/CatalogusService

SOAP

- Structureert HTTP-corpus
- Envelop-tag
  - Header-tag(s)
  - Body-tag
    - Bevat XML
    - Aan te roepen methode + argumenten
    - Resultaat van een methode

SOAP-aanvraag

```
POST /iit/webservices/Catalogus.asmx HTTP/1.1
Host: localhost
Content-Type: text/xml; charset=utf-8
Content-Length: ...
SOAPAction: "http://be.hogent.iii/webservices/GetBoek"

<?xml version="1.0" encoding="utf-8"?>
<soap:Envelope xmlns:xsi="..." xmlns:xsd="..."
  xmlns:soap="...">
    <soap:Body>
      <GetBoek xmlns="...">
        <ISBN>...</ISBN>
      </GetBoek>
    </soap:Body>
</soap:Envelope>
```


SOAP-antwoord

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

<?xml version="1.0" encoding="utf-8"?>
<soap:Envelope xmlns:xsi="..." xmlns:xsd="..."
  xmlns:soap="..."> <soap:Body>
    <GetBoekResponse xmlns="...">
      <GetBoekResult>
        <Isbn>...</Isbn>
        <Titel>...</Titel>
        <Prijs>...</Prijs>
      </GetBoekResult>
    </GetBoekResponse>
</soap:Body> </soap:Envelope>
```


Informatie voor proxy-object

- De structuur van de SOAP-berichten
  - Methodes
  - Argumenten
  - Teruggeefwaarden
- Locatie webservices
- Transportprotocol
- WSDL: Webservice Description Language

Structuur WSDL

- Types: Type-declaraties van argumenten, methodes, teruggeefwaardes, …
- Message: Beschrijving bericht (aanvraag/antwoord)
- PortType: Beschrijving mogelijke opdracht (naam, input, output)
- Binding: Verbinden portType met een locatie en een protocol (SOAP, HTTP GET, …)
- Service: Uit welke portTypes bestaat de webservice

Voorbeeld WSDL

- .NET
  - http://localhost/WebCatalogus/BoekenLijst.asmx?WSDL
- J2EE
  - http://localhost:8080/WebCatalogus/catalogusservice?WSDL

Werkwijze

- Maken webservice
  - Klassen, interface, … met eigenlijke functionaliteit maken
  - Webservice maken
  - WSDL genereren
- Clientapplicatie
  - Op basis van WSDL proxy-klassen genereren
  - Clientapplicatie maakt gebruik van proxy-klassen/objecten

Voorbeeld in .NET

Werkwijze in .NET

- Maken webservice
  - Klassen, interface, … met eigenlijke functionaliteit maken (Catalogus.dll)
  - Webservice maken (.asmx-bestand + codebehind)
  - WSDL genereren (automatisch)
- Clientapplicatie
  - Op basis van WSDL proxy-klassen genereren (Wsdl.exe)
  - Clientapplicatie maakt gebruik van proxy-klassen/objecten

Eigenlijke functionaliteit

- Project Catalogus
  - Klasse Boek
    - Resultaat methode
    - Moet geserialiseerd worden
      - Default-constructor
      - set- en get-declaraties voor eigenschappen → anders NIET in XML
  - Klasse BoekenMagazijn
    - Indexer
      - Boek op basis van isbnnummer

Werkwijze in .NET

- Maken webservice
  - Klassen, interface, … met eigenlijke functionaliteit maken (Catalogus.dll)
  - Webservice maken (.asmx-bestand + codebehind)
  - WSDL genereren (automatisch)
- Clientapplicatie
  - Op basis van WSDL proxy-klassen genereren (Wsdl.exe)
  - Clientapplicatie maakt gebruik van proxy-klassen/objecten

Webservice

- Klasse afgeleid van System.Web.Services.WebService
- Klasse en methode
  - Extra attribuut
    - WebService
    - WebMethod
    - Tussen [en] voor klassedefinitie of methodedeclaratie
- WebCatalogus.BoekenLijst.cs

Webservice

- asmx-bestand
- Bestaat uit één richtlijn nl.

```
<%@ WebService Language="c#"
   Codebehind="BoekenLijst.asmx.cs"
   Class="WebCatalogus.BoekenLijst" %>
```

Werkwijze in .NET

- Maken webservice
  - Klassen, interface, … met eigenlijke functionaliteit maken (Catalogus.dll)
  - Webservice maken (.asmx-bestand + codebehind)
  - WSDL genereren (automatisch)
- Clientapplicatie
  - Op basis van WSDL proxy-klassen genereren (Wsdl.exe)
  - Clientapplicatie maakt gebruik van proxy-klassen/objecten

Proxy-klassen genereren

Wsdl /namespace:Be.Hogent.III.ClientWebservice http://localhost/WebCatalogus/BoekenLijst.asmx

- Wsdl.exe → BoekenLijst.cs
- BoekenLijst.cs
  - BoekenLijst -klasse
    - GetBoek-methode
  - Boek-klasse

Proxy-klassen genereren

- Alternatief: menu “Add Service References”
  - url webservice opgeven
- ClientWebCatalogus\Service References\ServiceWebCatalogus\Reference.cs
- Klassen bekijken in objectbrowser

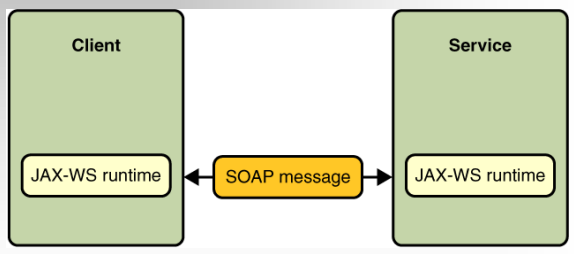
Werkwijze in .NET

- Maken webservice
  - Klassen, interface, … met eigenlijke functionaliteit maken (Catalogus.dll)
  - Webservice maken (.asmx-bestand + codebehind)
  - WSDL genereren (automatisch)
- Clientapplicatie
  - Op basis van WSDL proxy-klassen genereren (Wsdl.exe)
  - Clientapplicatie maakt gebruik van proxy-klassen/objecten

Clientapplicatie

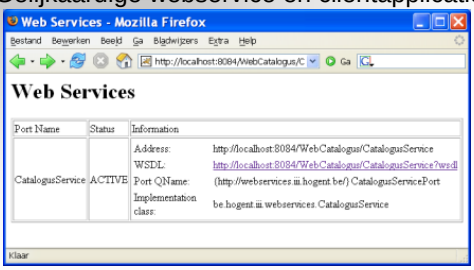
- Boeken met ID/ISBN van 1 tot 5 uitschrijven
- TestGetBoek.cs
- In objectbrowser beschikbare klassen bekijken

Webservices in J2EE



Webservices in J2EE

- Gelijkaardige webservice en clientapplicatie



Werkwijze in J2EE

- Maken webservice
  - Klassen, interface, … met eigenlijke functionaliteit maken (bo en jdbc)
  - Webservice maken
  - WSDL genereren (wsgen.exe)
- Clientapplicatie
  - Op basis van WSDL proxy-klassen genereren (wsimport.exe)
  - Clientapplicatie maakt gebruik van proxy-klassen/objecten

Eigenlijke functionaliteit

- be.hogent.iii.catalogus.Boek
  - Default-constructor
  - get/set-methodes voor id, titel en prijs
- be.hogent.iii.catalogus.BoekenLijst
  - Interface
    - Methode geefBoek
- be.hogent.iii.catalogus.impl.BoekenLijstImpl
  - Implementatie interface BoekenLijst
- In jar catalogus.jar

Werkwijze in J2EE

- Maken webservice
  - Klassen, interface, … met eigenlijke functionaliteit maken (bo en jdbc)
  - Webservice maken
  - WSDL genereren (wsgen.exe)
- Clientapplicatie
  - Op basis van WSDL proxy-klassen genereren (wsimport.exe)
  - Clientapplicatie maakt gebruik van proxy-klassen/objecten

Webservice

- Klasse met annotatie @Webservice()
- Methode met annotatie @WebMethod
  - Parameters met annotatie @WebParam(…)

```
@WebService()
public class CatalogusService {
   @WebMethod
   public Boek geefBoek(@WebParam(name = "isbn") String isbn) {
      try {
         BoekenLijst boeken = new BoekenLijstImpl();
         return boeken.geefBoek(isbn);
      } catch (Exception e) {
         return null;
      } } }
```


Werkwijze in J2EE

- Maken webservice
  - Klassen, interface, … met eigenlijke functionaliteit maken (bo en jdbc)
  - Webservice maken
  - WSDL genereren (wsgen.exe)
- Clientapplicatie
  - Op basis van WSDL proxy-klassen genereren (wsimport.exe)
  - Clientapplicatie maakt gebruik van proxy-klassen/objecten

WSDL genereren

- Automatisch bij build project in Netbeans
  - CatalogusServiceService.wsdl
  - CatalogusServiceService_schema1.xsd
    - beschrijft types
  - Wsdl af te halen via http://localhost:8084/WebCatalogus/CatalogusService
  - wsgen: tool om dit handmatig te doen

Webservice deployen

- War-bestand
  - Aantal extra XML-bestanden o.a.
    - sun-jaxws.xml
    - wsdl
    - web.xml bevat gegenereerd info
  - Doel
    - wrappen in een servlet
- Publiceren webservice

Werkwijze in J2EE

- Maken webservice
  - Klassen, interface, … met eigenlijke functionaliteit maken (bo en jdbc)
  - Webservice maken
  - WSDL genereren (wsgen.exe)
- Clientapplicatie
  - Op basis van WSDL proxy-klassen genereren (wsimport.exe)
  - Clientapplicatie maakt gebruik van proxy-klassen/objecten

Genereren proxy-klassen

- Automatisch in Netbeans in een webproject
  - New → Web Service Client
    - WSDL doorgeven
- Klassen terug te vinden in
  - build\generated\
- In Projectsvenster
  - Map Web Service References
    - CatalogusServiceService

Werkwijze in J2EE

- Maken webservice
  - Klassen, interface, … met eigenlijke functionaliteit maken (bo en jdbc)
  - Webservice maken
  - WSDL genereren (wsgen.exe)
- Clientapplicatie
  - Op basis van WSDL proxy-klassen genereren (wsimport.exe)
  - Clientapplicatie maakt gebruik van proxy-klassen/objecten

Clientapplicatie

- Project GebruikWebservice
  - gebruikwebservice.Main.java
- Oproepen methode webservice
  - Rechts klikken in editeervenster
    - Web Service Client Resource
      - Call Web Service Operation
    - Genereert de nodige code om de proxyobjecten te gebruiken

Informatie over webservices

- W3C SOAP Tutorial (http://www.w3.org/TR/2007/REC-soap12-part0-20070427/)
- W3C SOAP 1.2 (http://www.w3.org/TR/soap12-part1/)
- UDDI (http://uddi.org/pubs/uddi_v3.htm)

Informatie over webservices

- XML Web Services Created Using ASP.NET and XML Web Service Clients

(http://msdn2.microsoft.com/en-us/library/7bkzywba.aspx)

- The JavaEE 5 Tutorial (http://java.sun.com/javaee/5/docs/tutorial/doc/)
- Java™ 2 Platform Enterprise Edition, v 5.0 API Specification (http://java.sun.com/javaee/5/docs/api/)
- Webservices in Netbeans (http://www.netbeans.org/kb/trails/web.html)

Deel II

Netwerkprogrammatie

Hoofdstuk 10

Voorbereiding : C

10.1 Speciale karakters in C

In C zijn verschillende *speciale karakters* gedefinieerd die handig van pas komen bij het programmeren :

Omschrijving	*Karakter*
null	\0
biep	\a
nieuwe lijn	\n
horizontal tab	\t
vertical tab	\v
backspace	\b
carriage return	\r
formfeed	\f
single quote	\’
double quote	\”
backslash	\\
question mark	\?

Tabel 10.1: Karakters in C

10.2 Geformateerde uitvoer met *printf*

In C-programma’s wordt doorgaans gebruik gemaakt van *printf* om data naar het scherm te schrijven omdat het toelaat op eenvoudige manier deze data te formatteren. Syntax:

```
#include <stdio.h>

int printf(const char *format, argument1, argument2, ...);
```

Teken	*Actie*
d, i	conversie van *int* naar *signed* decimaal getal
u	conversie van *int* naar *unsigned* decimaal getal
o	conversie van *unsigned* naar *unsigned* octaal getal
x, X	conversie van *int* naar *unsigned* hexadecimaal getal
c	conversie van *int* ï¿½ï¿½n karakter
s	conversie van string
f	conversie naar *signed* decimaal floating-point getal
e, E	conversie naar *signed* decimaal floating-point getal in wetenschappelijke notatie
g, G	conversie d.m.v. *e* (of *E*) of *f* conversie; degene die het minst plaats inneemt
p	conversie naar pointer
n	het aantal tekens dat *printf* tot nu toe geschreven heeft

Tabel 10.2: Conversie operaties bij *printf*

```
Return: aantal karakters geschreven
```

printf zal de waarden van de argumenten formatteren aan de hand van de *format*-string. Deze string kan twee soorten informatie bevatten : gewone karakters, die naar de uitvoer worden gekopieerd, en *conversie specificaties* die aangeven hoe de argumenten moeten worden geformatteerd.

Het eenvoudigste voorbeeld van *printf* is met een formatstring waarin geen conversietekens staan en zonder argumenten. In dit geval wordt de formatstring gewoon naar de standaard uitvoer gekopieerd :

```
printf("Deze zin is vals.\n")
```

Voor elk argument dat wordt meegegeven aan *printf* moet een conversiespecificatie worden opgegeven in de formatstring. Dit doe je door een percentteken (%) gevolgd door een reeks van één of meer conversie tekens. Volgende conversie specificaties zijn geldig :

c d E f g G i n o p s u x X %

We bespreken de voornaamste; de rest vind je in de *man* pages, zie ook tabel 10.2.

Een argument van het type integer wordt gespecifieerd met de *d* conversie operator. Voorbeeld:

```
int a,b,som;
a=10; b=15;
printf("De som van %d en %d is %d\n", a, b, a+b);
```

Geeft volgende uitvoer :

```
De som van 10 en 15 is 25
```

In een format kan ook een minimale veldbreedte staan. In de eenvoudigste vorm is dit een integer constante.

```
printf("De som van %d en %d is %4d\n", a, b, a+b);
```

Geeft als uitvoer :

```
De som van 10 en 15 is  25
```

Als de geconverteerde waarde van het argument meer karakters omvat dan de opgegeven veldbreedte, wordt het veld voldoende verbreed.

Floating-point waarden worden met de *%f* specificatie weergegeven. De geconverteerde waarde bevat een decimaal punt en (default) 6 decimalen na de komma.

```
float x,y;
x = 1.234; y = 56.78;

printf("%f afgetrokken van %f is %f\n", y, x, x-y);
```

geeft dan:

```
56.780000 afgetrokken van 1.234000 is -55.546000
```

Hierbij kan men minimale veldbreedte en precisie opgeven als twee integer constanten gescheiden door een punt. Voorbeeld:

```
printf("%5.2f afgetrokken van %4.2f is %7.2f",y,x,x-y);
```

geeft als uitvoer:

```
56.78 afgetrokken van 1.23 is -55.55}
```

10.3 Geformateerde invoer met *scanf*

Deze functie verloopt analoog aan *printf*: de formatstring geeft aan welke types moeten worden ingelezen en de argumenten zijn pointers naar de variabelen waaraan de ingelezen waarden moeten worden toegekend. De uitdrukking

```
scanf("%d %d %f", &a, &b, &x);
```

verwacht dus twee integers en een float als invoer. Indien de invoer niet overeenkomt met de formatstring, dan stopt de functie. De returnwaarde is het aantal toegekende waarden.

10.4 Toekenningen en testen

Volgend soort programmacode kom je dikwijls tegen:

```
int n;
n = lees_iets();
while (n > 0){
  schrijf_iets();
  n = lees_iets();
}
```

In C kan je dit afkorten tot :

```
int n;
while ( n = lees_iets()) > 0)
  schrijf(n);
```

De reden hierachter is dat de waarde van een toekenningsexpressie gelijk is aan de waarde van het rechterlid. De toekenning *a=b+c* heeft dus, in haar geheel, de waarde *b+c*, zodat we kunnen schrijven *x = a=b+c* of dingen als *if(* $a = b + c < 0$*)*.

10.5 Foutverwerking en wrapper functies

Vele systeemfuncties in C werken met de globale integer variabele *errno*. Wanneer een functie een fout detecteert, wordt de waarde van *errno* aangepast. De functie zelf geeft een bepaalde waarde terug die duidt op een fout; meestal -1. Vervolgens kan men met de functie *perror* (print error) een geschikte boodschap naar het standaard error-kanaal schrijven. Een typisch voorbeeld zou zijn :

```
#include <stdio.h>
#include <errno.h>

main(){
  char buf[5];

  if ( recv(3, buf, 5, 0) < 0 ){ // 3: slechte descriptor
    perror("read");
    exit(1);
  }
}
```

De functie *perror* leest de waarde van *errno* en zoekt de bijhorende foutmelding op (terug te vinden in *errno.h*).

Omdat de opeenvolging *if functie* < 0 *– perror("melding") – exit(1)* zeer dikwijls voorkomt in programma's, is het een goed idee zogenaamde *wrapper* functies te schrijven. Wrapper-functies krijgen dezelfde naam als de functie zelf maar de eerste letter wordt een hoofdletter.[1] Bijvoorbeeld :

```
int Recv(int kanaal, char *buffer, int aantal, int vlaggen){
  int result;
  if (result = recv(kanaal, buffer, aantal, vlaggen) < 0){
```

[1] Andere notaties zijn natuurlijk ook mogelijk, zolang het maar consistent en duidelijk is.

Filedescriptor	**K**anaal
0	Standaard invoerkanaal
1	Standaard uitvoerkanaal
2	Standaard foutkanaal

Tabel 10.3: Standaard file descriptors

```
    perror("recv");
    exit(1);
  }
  return(result);
}
```

Met deze functie kunnen we nu het vorige programma afkorten tot:

```
#include <stdio.h>
#include <errno.h>

void main(){
  char buf[5];
  Recv(3, buf, 5, 0);
}
```

In deze cursus zullen we voor veelgebruikte functies telkens een wrapperfunctie gebruiken, te herkennen aan de hoofdletter, zodat de code overzichtelijk blijft.

10.6 Invoer en uitvoer op laag niveau

Wanneer je vanuit een programma berichten over een netwerk verstuurt of ontvangt, gebruik je daarvoor in principe dezelfde functies en procedures om gegevens naar een bestand te schrijven of van een bestand te lezen. De interface met het besturingssysteem is zodanig opgebouwd dat je voor dergelijke eenvoudige invoer- en uitvoerbewerkingen niet hoeft te weten of je uitvoer dan wel naar het scherm, naar een bestand of naar een programma op een andere computer wordt verstuurd. In de C-omgeving, zeker onder Unix, kan je stellen dat zowat alles een bestand is; bestanden op de harde schijf, directories, het scherm en toetsenbord, een netwerkverbinding, een FIFO, ...

10.6.1 Filedescriptors

Invoer- en uitvoerkanalen worden door UNIX voorgesteld met behulp van zogenaamde *filedescriptors*. Dit zijn kleine gehele getallen die gedurende de loop van het programma een bepaald kanaal (toetsenbord, scherm, bestand, printer, netwerkverbinding, ...) identificeren. Het zijn eigenlijk indexen in een interne tabel van gegevensstructuren die door het besturingssysteem worden bijgehouden en die niet rechtstreeks toegankelijk zijn voor de programmeur.

In tabel 10.3 vind je de drie filedescriptors die standaard worden toegewezen bij de start van een programma.

Nieuwe filedescriptors worden toegekend aan elk nieuw invoer- of uitvoerkanaal dat je in je programma creëert — elk bestand dat je opent en elke netwerkverbinding die je legt.

10.6.2 Lezen van een kanaal

Lezen en schrijven van en naar een kanaal gebeurt steeds in groepen van opeenvolgende bytes van het type **char** — zeg maar lettertekens. Gegevens worden bij het inlezen niet geïnterpreteerd : de enige manier om bijvoorbeeld reële getallen uit een tekstbestand in te lezen, is door ze achteraf zelf van letterreeks naar reëel getal om te zetten (bijvoorbeeld met behulp van *strtod* uit §10.7).

De functie die je gebruikt om een aantal bytes van een bestand in te lezen (of van het toetsenbord, of van het netwerk), heet *read* en heeft de volgende hoofding :

```
int read (int fd, char *buffer, int aant)
```

Deze functie doet een poging om *aant* bytes in te lezen van het kanaal met descriptor *fd* en stopt die in de tabel waar de pointer *buffer* naar verwijst[2]. De functie geeft het effectief aantal ingelezen bytes terug. Dit is meestal gelijk aan *aant*, maar kan ook minder zijn. Wanneer *read* 0 teruggeeft, betekent dit het einde van het bestand (of het einde van de netwerkverbinding als het om een netwerkkanaal gaat). Gebeurt er een leesfout, dan is de waarde van *read* negatief.

Het gebeurt vaak dat het aantal bytes dat effectief wordt ingelezen niet hetzelfde is als het aantal dat is opgegeven in de derde parameter. Zo zal een *read* van het toetsenbord (kanaal 0, het standaard invoerkanaal) meestal niet meer dan één volledige lijn tegelijk inlezen. Ook invoer van het netwerk is vaak niet volledig : door vertragingen in het netwerk of in de onderliggende netwerksoftware kan het best zijn dat een bericht slechts gedeeltelijk door de eerste *read* wordt ingelezen.

Daarom kan je best de derde parameter van *read* interpreteren als een soort maximum — als een indicatie van hoeveel plaats er in de tabel *buffer* is. Merk op dat je zelf in je programma voldoende geheugen moet voorzien voor de *buffer* die het ingelezen resultaat zal moeten bevatten.

Heb je het veel nodig om een vast aantal bytes in te lezen, dan kan je hiervoor een functie schrijven die *read* verschillende keren achter elkaar oproept :

```
int readn (int fd, char *buffer, int aantal){
// Lees aantal bytes in buffer
  int gelezen, totaal;
  totaal = 0;
  gelezen = read (fd, buffer, aantal);
  while (gelezen > 0 && totaal < aantal) {
    totaal += gelezen;
    gelezen = read (fd, buffer+totaal, aantal-totaal);
  }
  if (gelezen < 0)
    return gelezen;
  else
    return totaal+gelezen;
}
```

[2] De parameter *buffer* hoeft niet noodzakelijk een pointer naar een tabel van chars te zijn, maar kan een pointer zijn naar een willekeurig datatype. *Aant* wordt echter altijd in *bytes* gerekend.

Het is nog steeds mogelijk dat er minder dan *aantal* bytes worden ingelezen, maar dan enkel omdat er zich een leesfout voordeed, omdat het einde van het bestand werd bereikt of omdat de netwerkverbinding vroegtijdig werd afgesloten. De functiewaarde van *readn* voldoet aan dezelfde conventies als bij *read*.

Bij netwerkprogramma's is het dikwijls nodig om lettertekens in te lezen tot en met het *einde van een lijn* (aangeduid door een linefeed, *'\n'*). Hiervoor kan je een functie *readline* schrijven die op haar beurt gebruik maakt van *read* :

```
int readline (int fd, char *buffer, int max){
// Lees t.e.m. het einde van de lijn of buffer vol
 int gelezen, totaal, result;
  gelezen = read (fd, buffer, 1);
  totaal = 1;
  while ( gelezen > 0 &&
          buffer[totaal-1] != '\n' &&
          totaal < max) {
    gelezen = read (fd, &buffer[totaal], 1);
    totaal++;
  }

  if (gelezen < 0)
    result = gelezen;
  else if (gelezen==0)
    result = totaal - 1;
  else
    result = totaal;

return result;
}
```

Merk op dat een *read* van één byte per keer niet echt efficiënt is, en dat je beter steeds zoveel mogelijk bytes in één keer probeert in te lezen.

Bij het lezen van sockets kan men de functie *read()* probleemloos gebruiken. We opteren in deze cursus echter voor de iets uitgebreidere functie *recv()*. Alles wat voor *read()* geldt, geldt ook voor *recv()*. Bovendien kan je aan deze laatste een extra parameter meegeven, *flags*, maar dat valt buiten het bestek van deze cursus.

10.6.3 Schrijven naar een kanaal

Schrijven naar een kanaal gebeurt op een gelijkaardige manier als lezen, maar dan met de functie *write* :

```
#include <unistd.h>

int write (int fd, char *buffer, int aant)
```

Deze functie probeert de eerste *aant* bytes van de opgegeven *buffer* weg te schrijven naar het kanaal met filedescriptor *fd*. De waarde van de functie is het aantal bytes dat effectief werd weggeschreven of een negatief getal bij een fout.

In de meeste gevallen zal de waarde van deze functie gelijk zijn aan *aant*, maar dit is opnieuw niet noodzakelijk. Zijn deze aantallen niet gelijk, dan betekent dit niet dat er ergens een fout is opgetreden,

maar enkel maar dat je in je programma de rest van de gegevens met een nieuwe *write* moet versturen. We kunnen opnieuw een functie schrijven die ons hierbij helpt :

```
int writen (int fd, char *buffer, int aantal){
// Schrijf aantal bytes uit buffer,
// eventueel in meerdere keren
  int geschreven, totaal;
  geschreven = write (fd, buffer, aantal);
  totaal = 0;
  while (geschreven >= 0 && geschreven < aantal) {
    totaal += geschreven;
    geschreven=write(fd, buffer+totaal, aantal-totaal);
  }
  if (geschreven < 0)
    return geschreven;
  else
    return totaal + geschreven;
}
```

Bij het schrijven naar sockets kan men de functie *write()* probleemloos gebruiken. We opteren in deze cursus echter voor de iets uitgebreidere functie *send()*. Aan deze laatste kan je een extra parameter meegeven, *flags*, maar dat valt buiten het bestek van deze cursus.

10.6.4 Bestanden openen en sluiten

We geven nog in het kort aan hoe bestanden in deze context worden geopend en gesloten. Hiervoor gebruik je de volgende functies :

```
#include <fcntl.h>

int fd = open (bestandsnaam, O_RDONLY);
int fd = open (bestandsnaam, O_WRONLY | O_CREAT | O_TRUNC, 0644);

close (fd);
```

Met de eerste *open* openen we een bestand om er later van te lezen. De eerste parameter van *open* is een string met de naam van het bestand, het tweede argument geeft aan dat je van het bestand enkel wil lezen.

De tweede *open* opent een bestand om er naar te schrijven. De symbolische constanten geven aan dat het bestand moet worden aangemaakt indien het nog niet bestaat en dat het anders eerst moet worden uitgeveegd. Het octaal getal 0644 geeft de ‘mode’ van het bestand aan, in dit geval ‘iedereen lezen, eigenaar lezen en schrijven’. Meer informatie vind je in de manpages.

De functie *close* sluit het bestand. Doorgaans worden alle bestanden automatisch gesloten wanneer een UNIX-programma eindigt (ook bij het gebruik van *exit*) zodat je deze functie in de praktijk niet veel zal ontmoeten. Het is echter geen slechte gewoonte om al je bestanden toch steeds expliciet te sluiten wanneer je ermee klaar bent.

De procedure *open* geeft een negatief getal terug wanneer om één of andere reden het opgegeven bestand niet kon worden geopend (of aangemaakt). In andere gevallen wordt er een filedescriptor teruggeven die dan in verdere *read*s en *write*s op dat bestand als parameter moet worden gebruikt.

In het voorbeeld hieronder maken we een volledig copie van een opgegeven bestand. Bestandsnamen van bron en bestemming bevinden zich op de opdrachtlijn (zie §10.9). Het bestand wordt verwerkt in stukken van 4096 bytes.

Merk op hoe bij *read* de constante *BLOKGROOTTE* als derde argument wordt gebruikt, terwijl dit bij *write* de variabele *n* is. We lezen een maximum van 4096 bytes in, maar schrijven slechts het effectief ingelezen aantal uit naar het tweede bestand.

```
#include <unistd.h>
#include <fcntl.h>
#include <iostream.h>
#include <stdio.h>

const int BLOKGROOTTE = 4096;

int main (int argc, char **argv){
  int n;
  int ifd = open (argv[1], O_RDONLY);
  char buffer[BLOKGROOTTE];

  if (ifd < 0) {
    perror (argv[1]); return 2;
  }
  int ufd = open (argv[2],
                  O_WRONLY | O_CREAT | O_TRUNC, 0644);

  if (ufd < 0) {
    perror (argv[2]); return 2;
  }

  n = read (ifd, buffer, BLOKGROOTTE);
  while (n > 0) {
    if (write (ufd, buffer, n) < 0) {
      perror (argv[2]); return 2;
    }
    n = read (ifd, buffer, BLOKGROOTTE);
  }
  if (n < 0) {
    perror (argv[1]);
    return 2;
  }
  close (ifd);
  close (ufd);
  return 0;
}
```

(Let op de eerste twee *#include*-lijnen. De eerste heb je nodig voor *read*, *write* of *close*, de tweede voor *open*.)

10.7 Omzetten van strings naar getallen

Dikwijls is het nodig strings om te zetten naar getallen en omgekeerd. Voor de ene richting kan je de bibliotheekfuncties *strtod* (*string to* **double**) en *strtol* (*string to* **long**) gebruiken.

Strtod heeft twee argumenten. Het eerste is de string die moet worden omgezet en het tweede is een pointer naar een stringvariabele waarvan we de betekenis straks zullen uitleggen. De waarde van de

functie is het gezochte reëel getal.

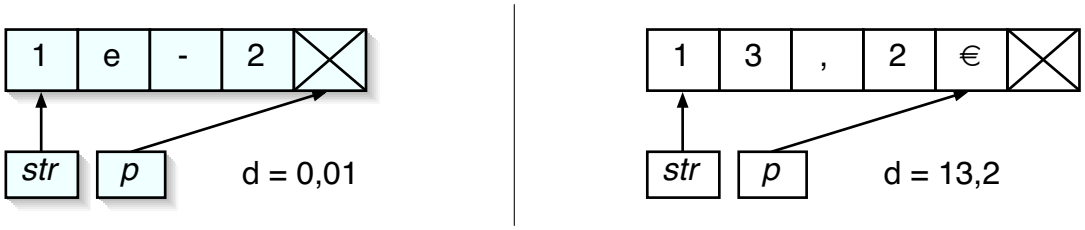
Strtod probeert een zo groot mogelijk stuk van de opgegeven string om te zetten naar een reëel getal en geeft dit als waarde terug. Als je voor het tweede argument een nullpointer invult, is dit het enige wat *strtod* doet. De waarde van '*strtod* (*"1e-2"*, 0)' is dus 0.01 en de waarde van '*strtod* (*"13.2BEF"*, 0)' is 13.2.

Geef je als tweede argument het adres van een stringvariabele op (in plaats van een nullpointer) dan vult *strtod* die na afloop in met een pointer naar het eerste letterteken van de string dat net niet meer tot het reëel getal behoort. We gebruiken dit argument bijvoorbeeld op de volgende manier :

```
#include <stdlib.h>
...
char *str = ...;
char *p;

double d = strtod (str, &p);
...
```

(Let op de *#include*-opdracht !)

Onderstaande figuur geeft twee voorbeelden van het effect van deze opdrachten.



Je kan dus de variabele *p* gebruiken om na afloop te weten te komen of wel degelijk de volledige string werd omgezet.

De opdracht *strtol* werkt op dezelfde manier, maar produceert een geheel getal. Bovendien heeft *strtol* een derde parameter die het grondtal van de omzetting aangeeft. Je vult hier dus 10 in voor de gewone decimale notatie, of bijvoorbeeld 16 voor hexadecimale getallen.

```
...
int aantal = strtol (argv[2], 0, 10);
...
```

10.8 Omzetten van getallen naar strings

Om een getal om te zetten naar een string kan je gebruik maken van de bibliotheekroutine *gcvt*[3]. Deze routine heeft de volgende hoofding :

[3]De routines *ecvt* en *fcvt* bieden gelijkaardige mogelijkheden. Voor meer informatie verwijzen we opnieuw naar de manpages.

```
#include <stdlib.h>

char *gcvt(double getal, int aantcijf, char *buffer);
```

De eerste parameter *getal* is het reëel getal dat je wil omzetten, *aantcijf* bepaalt het aantal *significante* cijfers en *buffer* is een tabel van lettertekens waarin de resultaatstring moet terechtkomen. De functiewaarde van *gcvt* is een copie van de parameter *buffer*.

gcvt probeert bij de omzetting altijd zo weinig mogelijk plaats nodig te hebben. Soms gebruikt hij hiervoor exponentiële notatie en soms wordt het decimale punt weggelaten. Het effectieve aantal lettertekens in de resultaatstring is echter moelijk op voorhand te bepalen en kan een stuk groter zijn dan het aantal significante cijfers. Zorg er dus voor dat je in je programma op voorhand voldoende plaats in de tabel *buffer* hebt voorzien.

Hoewel deze routine er eigenlijk niet voor bedoeld is, kan je ze ook gebruiken om *gehele* getallen naar strings om te zetten :

```
#include <stdlib.h>
  ...
  char buffer[12];
  int getal;
  ...
  gcvt (getal, 10, buffer);
```

De variabele *getal* in dit voorbeeld wordt dan eerst door de compiler stilzwijgend naar een reëel getal omgezet. Een geheel getal van 32 bits kan slechts 10 decimalen bevatten. We voorzien 12 plaatsen in *buffer* om ook de eindnull en een eventueel minteken toe te laten.

10.9 Argumenten aan *main*

Stel dat we een programma *som* willen schrijven dat twee argumenten meekrijgt bij aanroep en de som van die twee naar het scherm schrijft:

```
[jvm@gonzo home]> som 4 15
                  19
[jvm@gonzo home]>
```

In C kan je de argumenten die werden meegegeven bij de executie achterhalen via twee parameters van de *main* functie. Het eerste, *argc* geeft het aantal argumenten dat werd meegegeven. De naam van het programma zelf wordt ook meegeteld. In het bovenstaande voorbeeld is *argc* dus gelijk aan 3. Het tweede, *argv* is een array van pointers naar *char*, pointers naar het eerste karakter van null-terminated strings. *argv[0]* is de programmanaam. In het voorbeeld zijn dan *argv[0]*, *argv[1]* en *argv[2]* resp. “som”, “4” en “15”.

Het programma zou er als volgt kunnen uitzien :

```
#include <stdio.h>
#include <stdlib.h>
int main(int argc, char *argv[]){
  if (argc!=3)
    printf("Fout aantal argumenten.\n");
  else
    printf("%d\n",
           strtol(argv[1],0,10) + strtol(argv[2],0,10));
}
```

Hoofdstuk 11

Basis Sockets

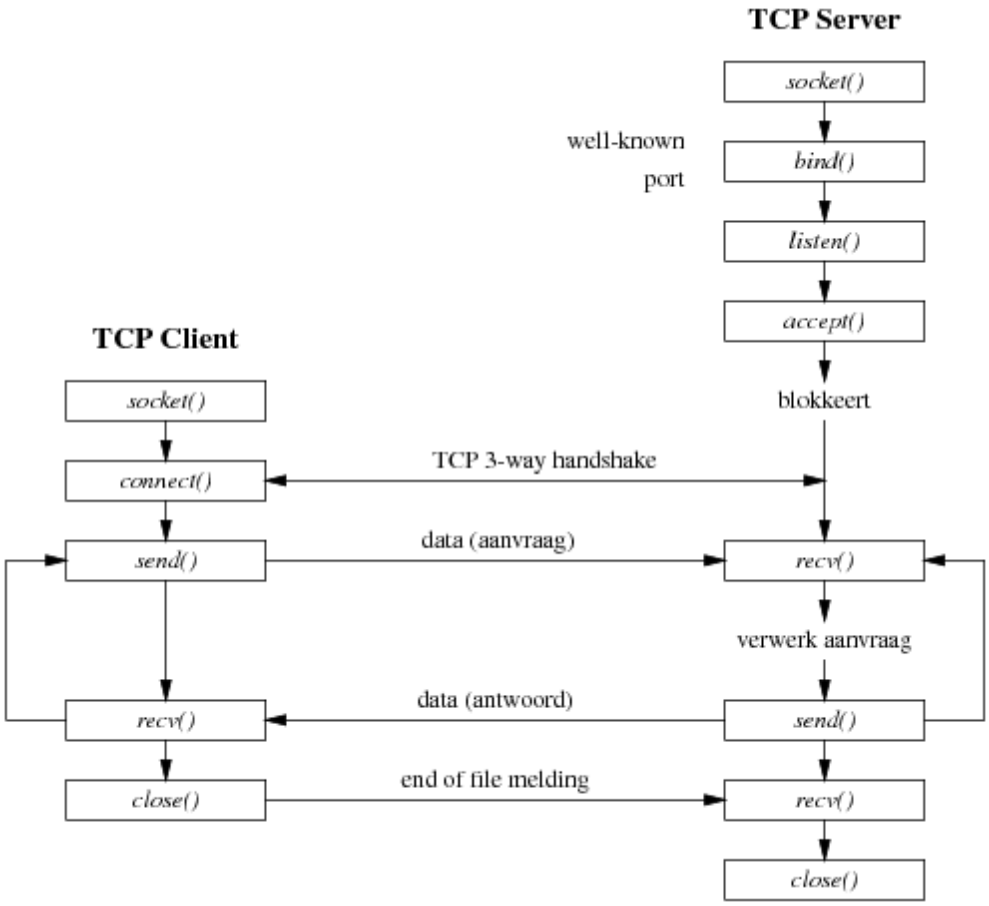
11.1 Wat is een socket ?

Een socket is een abstractie waarmee applicaties gegevens kunnen sturen en ontvangen, net als een open bestand kan worden gebruikt om data te lezen en te schrijven naar een schijf. Via sockets kunnen applicaties "inpluggen" op een netwerk en met elkaar communiceren. Data die door een applicatie naar een socket wordt geschreven, kan door een andere applicatie, op een andere of zelfde machine binnen het netwerk, worden gelezen, en omgekeerd.

Voor de veelheid aan netwerkprotocollen, bestaan er verschillende soorten sockets. In deze cursus behandelen we twee soorten: *stream sockets* en *datagram sockets*. Stream sockets zijn betrouwbare, bi-directioneel verbonden communicatie streams. Als je twee pakketten verstuurt in volgorde "1, 2", dan komen die pakketten ook in die volgorde toe aan de andere zijde van de connectie. Bovendien zullen ze foutvrij aankomen. Deze hoge transmissie-kwaliteit wordt bereikt door het protocol dat dit type socket gebruikt : TCP of *Transmission Control Protocol*. TCP zorgt ervoor dat de data sequentieel en foutvrij wordt doorgestuurd. Bij eventuele fouten zal TCP er zelf voor zorgen dat de data opnieuw wordt verstuurd. Toepassingen als telnet, WWW browsers, email gebruiken allen stream sockets.

Datagram sockets zijn connectieloos, onbetrouwbaar en snel. Ze gebruiken niet TCP maar UDP : *User Datagram Protocol*. Eventuele fouten worden niet opgevangen door het protocol; daar moet de programmeur zelf voor zorgen. Het voordeel van UDP is dat je geen connectie hoeft op te zetten naar de andere computer; veel ellende van TCP programma's ontstaat door het niet of slecht onderhouden van connecties. Internet toepassingen als DHCP, traceroute, RIP, NFS gebruiken datagram sockets.

In dit hoofdstuk bekijken we alle functies die nodig zijn om sockets aan te maken en ermee te werken. Er wordt een voorbeeld gegeven van een eenvoudige TCP client en server. De programma's worden in de loop van de volgende hoofdstukken verder aangevuld en verbeterd. In figuur 11.1 wordt een grafisch overzicht gegeven van de verschillende functies en hun opeenvolging in tijd.



Figuur 11.1: Opeenvolging van de socket functies bij TCP server en client

11.2 De *socket()* functie

Sockets worden gecreëerd met de *socket* functie :

```
#include <sys/socket.h>

int socket(int family, int type, int protocol);

Return : descriptor (>=0) indien OK, -1 bij fout
```

De *family* specifieert het protocol en moet een van de constanten uit tabel 11.1 zijn. In de cursus gebruiken we enkel sockets die Internet protocollen ondersteunen en zullen steeds PF_INET gebruiken. Socket *type* is een constante uit tabel 11.2. De derde parameter *protocol* is IPPROTO_TCP voor TCP sockets en IPPROTO_UDP voor UDP sockets. Je kan ook de waarde 0 opgeven en dan kiest het systeem voor het standaard protocol voor het gevraagde type.

family	Omschrijving
PF_INET	IPv4 protocollen
PF_INET6	IPv6 protocollen
PF_LOCAL	Unix domain protocol
PF_ROUTE	Routing socket
PF_KEY	Key socket

Tabel 11.1: Protocol familie constanten voor de *socket* functie

type	Omschrijving
SOCK_STREAM	stream socket
SOCK_DGRAM	datagram socket
SOCK_RAW	raw socket

Tabel 11.2: Socket *type* voor de *socket* functie

Afhankelijk van de combinatie *family* en *type* verkrijgt men de protocollen uit tabel 11.3. Ongeldige combinaties zijn aangeduid met een vraagteken. Geldige combinaties waar geen speciale benaming voor bestaat zijn de vakjes gemarkeerd met “Ja”.

	PF_INET	PF_INET6	PF_LOCAL	PF_ROUTE	PF_KEY
SOCK_STREAM	TCP	TCP	Ja	?	?
SOCK_DGRAM	UDP	TCP	Ja	?	?
SOCK_RAW	IPv4	IPv6	?	Ja	Ja

Tabel 11.3: Socket *type* voor de *socket* functie

Bij succes geeft de functie *socket* een geheel getal weer ($>= 0$): de *socket descriptor*. Merk op dat we enkel de protocol familie (IPv4, IPv6, . . .) en het socket type (stream, datagram, raw) opgeven; er is nog geen sprake van IP adressen.

11.3 Adressen specifiëren

Applicaties die data versturen en ontvangen via sockets, moeten in staat zijn Internet adressen en poortnummers te specifiëren. In de sockets API gebeurt dit via een generische structure *sockaddr* :

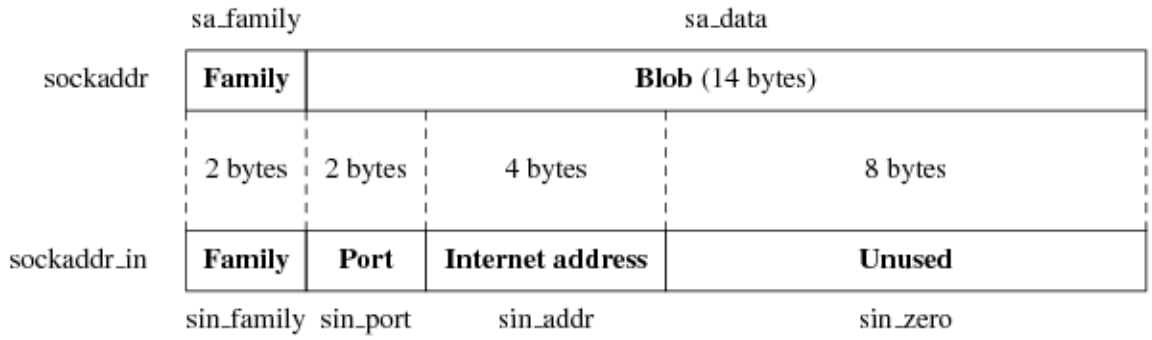
```
struct sockaddr {
  unsigned short sa_family;   /* adres familie, AF_xxx */
  char           sa_data[14]; /* 14 bytes adres info   */
};
```

Het eerste veld geeft aan over welk soort adres het gaat. In deze cursus maken we enkel gebruik van Internet adressen en zal de familie steeds *AF_INET* (= Address Family InterNet) zijn. Het tweede veld is een reeks bits (een *blob*) waarvan de structuur wordt bepaald door de familie. In het geval van IP sockets bestaat het adres steeds uit een 32-bit Internet adres en een 16-bit poortnummer.

Om het invullen van IP adressen en poortnummers mogelijk (of makkelijker) te maken, is volgende structure gedefinieerd: *struct sockaddr_in* (waarbij "in" staat voor Internet) :

```
struct in_addr{
  unsigned long s_addr;           /* Internet adres (32 bits) */
};

struct sockaddr_in {
  unsigned short int sin_family;  /* adres familie: AF_INET   */
  unsigned short int sin_port;    /* poort nummer (16 bits)   */
  struct in_addr     sin_addr;    /* Internet adres (32 bits) */
  unsigned char      sin_zero[8]; /* opvulsel                 */
};
```

Het is belangrijk dat je inziet dat een *sockaddr_in* structuur gewoon een andere kijk is op de data in een *sockaddr* structuur, aangepast aan sockets die met het Internet protocol werken. In figuur 11.2 is dit grafisch weergegeven. Wanneer we de velden van een *sockaddr_in* invullen met een poort en adres, kunnen deze *casten* naar een *sockaddr* structure en doorgeven aan een socket-functie.



Figuur 11.2: Adres structuren

11.4 De *connect()* functie

Wanneer een TCP client een socket heeft aangemaakt, kan hij een connectie maken met een TCP server via de *connect()* functie :

```
#include <sys/socket.h>

int connect(int sockfd,
            const struct sockaddr *servaddr,
            int addrlen);

Return : 0 als OK, -1 bij fout
```

Hierbij is *sockfd* de socket descriptor die werd teruggeven door *socket*. De structure *servaddr* bevat het IP-adres en poortnummer waarnaar connectie moet gemaakt worden (zie 11.3). Bij TCP sockets zal *connect* de 3-way-handshake initialiseren. De functie keert enkel terug wanneer de connectie

is gemaakt of een fout is opgetreden. Bij een fout krijgt de globale variabele *errno* een specifieke foutcode. Volgende fouten zijn mogelijk:

- Als de client geen antwoord ontvangt, wordt *ETIMEDOUT* teruggegeven
- Als de server *RST* (ReSeT) antwoordt, wil dit zeggen dat op de gespecifieerde poort geen proces luistert. In dit geval krijgt *errno* de waarde *ECONNREFUSED*.
- "destination unreachable" fouten : een router geeft aan dat de bestemming niet bereikbaar is. *errno* is dan *EHOSTUNREACH* of *ENETUNREACH* terug.

11.5 Sturen en ontvangen van data

Wanneer je een socket hebt verbonden met een server met *connect* dan kan je beginnen data te sturen en ontvangen met de functies *send()* en *recv()*.

```
int send(int socket, const void *msg, unsigned int msgLength, int flags);
int recv(int socket, void *rcvBuffer, unsigned int bufferLength, int flags);
```

Beide functies hebben gelijkaardige parameters. *socket* is de descriptor van de verbonden socket waarlangs data wordt gestuurd en ontvangen. Bij de functie *send()* wordt een boodschap meegegeven en de lengte van die boodschap. De parameter is *void* * zodat je alle mogelijke types data kan meesturen. Voor de functie *recv* moet je ook een pointer naar een buffer meegeven waarin de ontvangen bytes zullen worden opgeslagen. De parameter *bufferLength* geeft de grootte van die buffer aan en *recv()* zal niet meer bytes dan deze waarde ontvangen. Beide functies blokkeren het proces tot het moment dat minstens één byte kan worden verstuurd of ontvangen. De laatste parameter *flags* bepaalt het gedrag van de functies. Wanneer je het standaard gedrag wenst geef je de waarde 0. Beide functies geven het aantal bytes terug dat werd verstuurd of ontvangen. Bij een fout wordt -1 teruggegeven. Wanneer *recv()* de waarde 0 teruggeeft, wil dit zeggen dat het proces aan de andere kant van de socket de verbinding heeft verbroken.

11.6 Beëindigen van een verbinding

Een TCP verbinding kan men beëindigen door de functie *close()* aan te roepen.

```
#include <unistd.h>

int close(int socket);
```

De returnwaarde van *close()* is 0 bij succes en -1 bij fout.

Indien *socket* een verbonden TCP socket is, dan heeft de aanroep van *close()* tot gevolg dat de verbinding wordt gesloten. De andere kant kan dit detecteren via de returnwaarde van *recv()* die dan gelijk is aan 0.

11.7 Voorbeeld van een TCP client

Het volgende programma is een zogenaamde *echo client*: het programma maakt connectie met een echo server en schrijft het ontvangen antwoord op het scherm. Zoals de meeste programma's in deze cursus wordt hier gebruik gemaakt van het header bestand *netwerk.h*. Dit bestand bevat alle nodige include-opdrachten voor de meeste netwerk functies. Je kan dit bestand vinden op amerigo.

```
1  #include  "netwerk.h"
2
3  #define RCVBUFSIZE 32 /* grootte receive buffer */
4  #define ECHO_PORT  7
5
6  int main(int argc, char **argv){
7    int sockfd;
8    char buffer[RCVBUFSIZE];
9    struct sockaddr_in servaddr;
10
11   if (argc != 3){
12     printf("Gebruik: TCPEchoClient <IPaddress> <string>\n");
13     exit(1);
14   }
15
16   /* maak TCP socket */
17   if( (sockfd = socket(PF_INET, SOCK_STREAM, IPPROTO_TCP)) < 0)
18     ExitWithMessage("socket() failed");
19
20   /* vul server adres structure in */
21   memset(&servaddr, 0, sizeof(servaddr));        /* maak struct leeg */
22   servaddr.sin_family = AF_INET;                 /* Internet adres */
23   servaddr.sin_addr.s_addr = inet_addr(argv[1]); /* server adres */
24   servaddr.sin_port = htons(ECHO_PORT);          /* echo poort */
25
26   /* maak verbinding met server */
27   if (connect(sockfd, (struct sockaddr *) &servaddr, sizeof(servaddr)) < 0)
28     ExitWithMessage("connect() failed");
29
30   /* stuur string naar de server */
31   int echoStringLen = strlen(argv[2]);
32   if( send(sockfd, argv[2], echoStringLen, 0) != echoStringLen )
33     ExitWithMessage("send() failed");
34
35   /* lees antwoord van de server */
36   int totalBytesRcvd = 0;
37   printf("Ontvangen:");
38   while ( totalBytesRcvd < echoStringLen) {
39     int bytesRcvd;
40     if( (bytesRcvd = recv(sockfd, buffer, RCVBUFSIZE-1, 0) ) <= 0){
41       ExitWithMessage("recv failed");
42     }
43     totalBytesRcvd += bytesRcvd;
44     buffer[bytesRcvd] = '\0'; /* string eindigt met 0 */
45     printf(buffer);
46   }
47   printf("\n"); /* final endline */
48   close(sockfd); /* sluit de verbinding */
49   return 0;
50 }
```

Listing 11.1: TCPEchoClient.c

Enkele verduidelijkingen bij dit voorbeeld.

1. **Include :** lijn 1
   Het bestand **netwerk.h** bevat alle nodige includes. Te downloaden op http://amerigo.hogent.be/np.

2. **TCP socket aanmaken:** lijnen 16-18

3. **Server adres invullen:** lijnen 21-24

   - Met de functie *memset()* krijgt elke byte van de structure de waarde 0. Dit is nodig om er zeker van te zijn dat er geen ongewenst data in de structure zit.
   - In de structure moeten we de familie (*AF_INET*), het poortnummer en het IP adres invullen. De functie *inet_addr* converteert de string-representatie van het IP adres naar de correcte 32-bit representatie. Het poortnummer moet worden omgezet naar *network byte order* met de functie *htons()*, waarover meer in 13.2.

4. **verbinding maken:** lijnen 34-35
   De functie *connect* zorgt ervoor dat de socket wordt verbonden met de socket die gespecifieerd wordt in de *sockaddr_in* structure. Omdat de sockets API generiek is, moeten we de pointer naar deze structure *casten* naar het algemene type (*sockaddr*) en bovendien moeten we de grootte van de structure meegeven.

5. **stuur string:** lijnen 27-28
   De string die we naar de server sturen zit in de array *argv* en we geven gewoon dit element mee aan *send()*. Merk op dat we de *terminator* van de string, het nul-karakter, niet meesturen.

6. **Ontvang het antwoord van de server:** lijnen 36-46
   TCP is een *byte-stream* protocol. Een gevolg hiervan is dat een *send()* aan de ene kant van de verbinding een aantal bytes kan versturen die door de ontvanger aan de andere kant van de verbindig in meerdere aanroepen van *recv* moet worden gelezen. Daarom blijven we *send()* aanroepen tot we evenveel bytes hebben ontvangen als we er hebben verstuurd. In dit programma zal waarschijnlijk slechts een enkele aanroep voldoende zijn, maar dit is *niet gegarandeerd.*

De applicatie gebruikt de functie *ExitWithMessage()* om de gebruiker een foutboodschap te geven en vervolgens de applicatie af te sluiten. Er kan een boodscap worden meegegeven en de functie print deze boodschap en een systeem-specifieke boodschap af dmv *perror*. Deze laatste functie bekijkt de waarde van de globale systeemvariabele *errno* die aangeeft wat de laatst opgetreden fout in het systeem is. Er wordt een gepaste melding bij gegeven.

```
#include <stdio.h>
#include <stdlib.h>

void ExitWithMessage(char *message){
  perror(message);
  exit(1);
}
```

Listing 11.2: ExitWithMessage.c

Proces specifieert		Resultaat
IP adres	poort	
wildcard	0	kernel kiest adres en poort
wildcard	nonzero	kernel kiest adres, proces kiest poort
lokaal IP adres	0	proces kiest IP adres, kernel kiest poort
lokaal IP adres	nonzero	proces kiest beide

Tabel 11.4: Kiezen van poort en adres bij *bind* functie

11.8 De *bind()* functie

De functie *bind* kent een lokaal adres toe aan een socket. Een adres is bij Internet protocollen een combinatie van een 32-bit IPv4 of 128-bit IPv6 adres en een TCP of UDP poortnummer.

```
#include <sys/socket.h>

int bind(int sockfd,
         const struct sockaddr *myaddr,
         int addrlen);

Return : 0 bij OK, -1 bij fout
```

Het eerste argument is de socket descriptor, het tweede een pointer naar een protocol-specifiek adres en het derde de lengte van deze address structure. Bij TCP kunnen we een IP adres, een poortnummer, beide of geen specifiëren. TCP clients roepen *bind* doorgaans niet aan. Hierdoor zal de kernel zelf een poortnummer kiezen. Servers daarentegen roepen de functie doorgaans wel aan en specifiëren hierbij een poortnummer; men moet immers weten naar welke poort clients connecties kunnen maken. Wanneer geen IP adres wordt opgegeven, kiest de kernel zelf een lokaal IP adres dat wordt gebruikt voor de connectie. Alle vier combinaties worden samengevat in tabel 11.4

In IPv4 zijn de wildcards voor poortnummers 0 en voor IP adressen *INADDR_ANY*.

11.9 De *listen()* functie

Enkel TCP servers roepen deze functie aan :

```
#include <sys/socket.h>

int listen(int sockfd, int backlog);

Return: 0 als OK, -1 bij fout
```

Wanneer men een socket aanmaakt met de *socket()* functie, is dit automatisch een *actieve* socket: een client kan een actieve socket gebruiken om een connectie te maken met een server. De *listen()* functie verandert de status in *passief* waardoor de kernel de socket kan gebruiken om inkomende connectieaanvragen te verwerken. Het eerste argument is de socket die in passieve modus dient gezet

te worden, het tweede geeft aan hoeveel clients in de wachtrij kunnen. De exacte betekenis van het backlog argument verschilt nogal sterk van systeem tot systeem. Het geeft aan hoeveel aanvragen er in de wachtrij mogen komen. Stel dat een server slechts één client tegelijk kan bedienen en er melden zich twee clients aan, dan krijgt de eerste een connectie en gaat de tweede in de wachtrij. Op het moment dat de server klaar is met de eerste, wordt de tweede uit de wachtrij gehaald en wordt een connectie opgezet. Wanneer de wachtrij vol is en een client doet een connectieaanvraag, dan krijgt deze een "Connection refused" foutboodschap. Zie ook de man pages.

11.10 De *accept()* functie

Deze functie wacht totdat de volgende client zich aanmeldt (of totdat een client die zich reeds eerder had aangemeld door het systeem uit de wachtrij wordt gehaald) en geeft een socketdescriptor terug voor een verbinding met de client. De functie *accept()* aan de serverkant reageert met andere woorden op een *connect()* aan de clientkant. De waarde van *accept()* is negatief wanneer er zich een fout voordoet.

```
#include <sys/types.h>
#include <sys/socket.h>

int accept(int s, struct sockaddr *addr, int *addrlen);
```

De eerste parameter van *accept()* is de descriptor van de *luisterende* socket. De descriptor die je als waarde terugkrijgt, correspondeert met de *verbonden* socket. Het is deze nieuwe descriptor die je dan verder gebruikt om berichten over het netwerk te verzenden of te ontvangen (met *send()* en *recv()*) en het is deze descriptor die met *close* wordt afgesloten om de verbinding met de client te verbreken.

accept() kan je ook informatie bezorgen over het IP-adres en het poortnummer dat door de client wordt gebruikt. Deze informatie wordt teruggeven in de socketadresstructuur waar de tweede parameter (*adres*) naar verwijst. Tegelijkertijd wordt ook de lengte van het socketadres van de client ingevuld in het geheel getal waar de laatste parameter (*addrlen*) naar refereert.

Merk op dat deze laatste parameter een *pointer* naar een geheel getal voorstelt en niet het geheel getal zelf, zoals bij *connect()* en *bind()*. Dit is vaak een bron van verwarring. Je moet de variabele waar *addrlen* naar verwijst reeds op voorhand met de (maximale) lengte van het socketadres initializeren.

Het volgende fragment wacht op een connectie van een client, drukt het IP-adres en het poortnummer van deze client af, verstuurt een kort bericht en verbreekt daarna de verbinding. Let op het gebruik van de variabele *csalen*.

```
sockaddr_in csa;
unsigned int csalen = sizeof (sockaddr_in);
int sd = accept (lsd, (SA *)(&csa), &csalen);
printf("%s:%d\n",inet_ntoa(csa.sin_addr),ntohs (csa.sin_port));
char * sendString = "Hallo";
send(sd, sendString, strlen(sendString), 0);
close (sd);
```

Wanneer je niet in het adres van de client bent geïnteresseerd, vul dan de nullpointer in voor beide laatste argumenten van *accept()*.

11.11 Voorbeeld van een TCP server

We bekijken nu een uitgewerkt voorbeeld van een TCP echo server. De server ontvangt connectie-aanvragen van clients, leest data van de socket en stuurt dezelfde data terug.

```
1  #include "netwerk.h"
2
3  #define MAXLINE 256
4  #define POORT 7
5
6  int main(int argc, char **argv){
7    int listenfd, connfd;
8    unsigned int len;
9    int n;
10   struct sockaddr_in servaddr, cliaddr;
11   char buff[MAXLINE];
12
13   /* Maak een TCP socket */
14   if( (listenfd = socket(PF_INET, SOCK_STREAM, IPPROTO_TCP)) < 0)
15     ExitWithMessage("socket() failed");
16
17   /* Vul server adres struct in */
18   memset(&servaddr, 0, sizeof(servaddr));       /* maak struct leeg */
19   servaddr.sin_family = AF_INET;                /* Internet familie */
20   servaddr.sin_addr.s_addr = htonl(INADDR_ANY); /* elke interface */
21   servaddr.sin_port = htons(POORT);             /* lokale poort*/
22
23   /* Bind socket aan lokaal adres */
24   if (bind(listenfd, (struct sockaddr *)&servaddr, sizeof(servaddr)) < 0)
25     ExitWithMessage("bind()  failed");
26
27   /* Maak de socket passief zodat hij luistert naar connectie-aanvragen */
28   if( listen(listenfd, 10) < 0)
29     ExitWithMessage("listen() failed");
30
31   /* oneindige lus */
32   while(1) {
33     len = sizeof(cliaddr);
34     if( (connfd = accept(listenfd, (SA *) &cliaddr, &len)) < 0)
35         ExitWithMessage("accept()  failed");
36
37     /* er is een verbinding; handel client af */
38     printf("connectie met %s\n", inet_ntoa(cliaddr.sin_addr));
39     while( (n = recv(connfd,buff,MAXLINE-1, 0)) > 0){
40       buff[n]=0;
41       if( (n=send(connfd, buff,strlen(buff), 0))<0)
42         ExitWithMessage("send() failed");
43     }
44
45     /* sluit de verbindig */
46     close(connfd);
47   }
48   /* onbereikbaar */
49 }
```

Listing 11.3: TCPEchoServer.c

Een beetje uitleg bij de code :

1. **Initialisatie :** lijn 1-4
2. **TCP socket aanmaken:** lijnen 14-15
3. **Lokale adres invullen:** lijnen 18-21
   We vullen een *sockaddr_in* struct in met de gegevens van de server. We geven hiermee aan op welke interface en poort de applicatie moet luisteren naar binnenkomende connectieaanvragen. Voor het adres gebruiken we de *wildcard INADDR_ANY* waarmee we aangeven dat we op elke interface van het systeem naar aanvragen willen luisteren. Dit bespaart ons de moeite om het werkelijke ip-adres van de server in te vullen. Zowel het adres als de poort worden in network byte order geplaatst met de functies *htonl()* en *htons()* (zie 13.2).
4. **Bind socket aan lokale poort en adres:** lijnen 24-25
   bind() koppelt het opgegeven adres en poort aan de socket. Dit kan mislukken om verschillende redenen. De meest voorkomende reden is dat een andere socket reeds gekoppeld is aan het adres. Een andere mogelijkheid is dat *super user* privileges vereist zijn om een socket te koppelen aan een poort lager dan 1024 (*well known ports*).
5. **Maak de socket passief:** lijnen 28-29
   listen() zorgt ervoor dat het mogelijk is connectieaanvragen te ontvangen op de socket.
6. **Verwerk clients:** lijnen 32-47
   De server gaat in een oneindige lus luisteren naar connectieaanvragen en vervolgens de client bedienen. Eerst wordt een verbinding aangemaakt met *accept()*. Deze functie blokkeert het proces tot een aanvraag binnenkomt. Wanneer dit gebeurt, geeft de functie een descriptor. Dit is een actieve socket en de eigenlijke verbinding met de client. De tweede parameter van *accept* is een pointer naar een *sockaddr_in* struct die, bij succes, het adres en poortnummer van de client bevat. De socket die wordt teruggegeven door *accept()* is reeds verbonden met de client en is klaar om gegevens te ontvangen en versturen. De server leest een boodschap van de client en stuurt diezelfde boodschap terug. Ook hier gebruiken we een while-lus om de data te lezen omdat TCP niet garandeert dat alle data van de client met één *recv()* wordt binnengehaald.

Op het moment dat de client *close()* aanroept, wordt de verbinding verbroken. In de server heeft dit tot gevolg dat de functie *recv()* terugkeert en de waarde 0 teruggeeft. De server springt uit de binnenste while-lus, sluit zijn kant van de verbinding en gaat terug naar de buitenste while-lus waar *accept()* wordt aangeroepen om te wachten op een volgende client.

Hoofdstuk 12

IP adresconversies en DNS

12.1 *gethostbyname*

Hoewel de relatie tussen machinenamen en IP-adressen wordt bijgehouden door DNS, hoef je hiervoor gelukkig het DNS-protocol niet te kennen of rechtstreeks connecties te leggen met de verschillende DNS-servers op het Internet. Dit alles wordt voor jou gedaan door de zogenaamde *resolver* — in de hoedanigheid van de functie *gethostbyname*. Deze functie heeft de volgende hoofding :

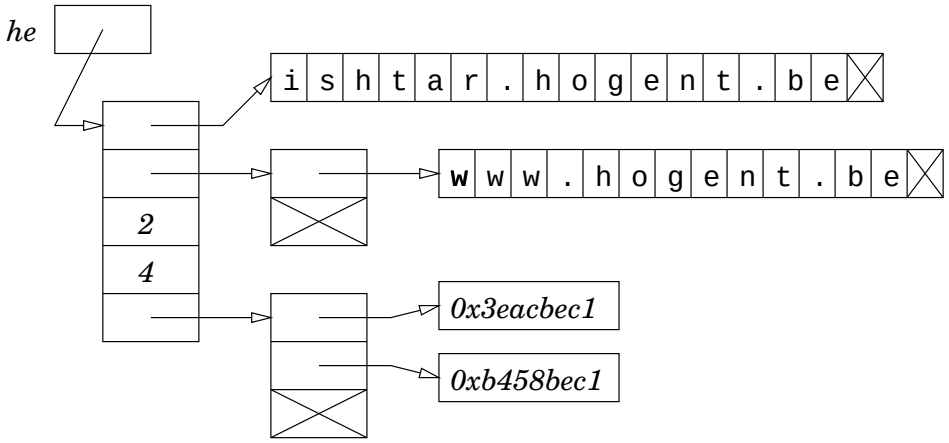
```
#include <netdb.h>

hostent *gethostbyname(const char *name)
```

Het argument van deze functie is de naam van een machine, bijvoorbeeld "amerigo.hogent.be" of een IP adres in presentatievoorstelling, bijvoorbeeld "192.145.65.32". Het resultaat is een pointer naar een structuur van het type *hostent* die door de resolver wordt ingevuld. Deze structuur wordt, in het bestand *netdb.h*, als volgt gedefinieerd :

```
struct hostent {
  char    *h_name;
  char    **h_aliases;
  int     h_addrtype;
  int     h_length;
  char    **h_addr_list;
};
```

Het eerste veld *h_name* is een string die de zogenaamde *canonische* naam bevat van de machine — de *officiële* naam. Heeft een toestel nog andere namen, dan worden die bewaard in het veld *h_aliases* dat een tabel van strings bevat, afgesloten met een nullpointer. De velden *h_addrtype* en *h_len* zijn hier niet belangrijk en bevatten altijd de waarden 2 en 4.

Tot slot bevat *h_addr_list* een tabel van pointers naar IP-adressen van het opgegeven toestel. Deze tabel wordt opnieuw afgesloten met een nullpointer. Hoewel het type hier '**char** $**$' is, kan je dit beter beschouwen als een veld van het type '*in_addr* $**$'. Je zal in je programma wel de nodige omzettingen moeten doen. Merk op dat de IP-adressen reeds in netwerkbytevolgorde staan.



Figuur 12.1: *hostent* na aanroep *gethostbyname*

Indien het argument een Ip adres is (dus geen hostnaam) dan zal de resulterende *hostent* struct enkel dit IP adres bevatten in *h_addr_list*.

In figuur 12.1 zie je hoe deze structuur er uitziet na de oproep '*he* = *gethostbyname* (*"www"*)' wanneer die wordt uitgevoerd op een computer van het domein *hogent.be*. *Www* heet dus eigenlijk *ishtar* en heeft twee verschillende IP-adressen.

Vaak heb je genoeg aan slechts een gedeelte van de informatie in deze structuur, namelijk het eerste adres in de tabel van IP-adressen. Dit bekom je bijvoorbeeld op de volgende manier (we halen de computernaam uit de opdrachtlijn) :

```
#include <netdb.h>
...
hostent *he = gethostbyname (argv[1]);
if (he == NULL) {
    herror (argv[0]); return 1;
}
in_addr **p = (in_addr **)he->h_addr_list;
sa.sin_addr = **p;
```

Merk op dat we nu niet *perror* gebruiken om foutberichten te tonen, maar *herror*. Deze functie doet hetzelfde als *perror* maar moet gebruikt worden voor foutboodschappen die met de resolver te maken hebben. De voorlaatste lijn zet een *pointer naar een pointer naar een* ***char*** om naar een *pointer naar een pointer naar een IP-adres*.

Een tweede codevoorbeeld laat zien hoe je een functie kan schrijven die een hostnaam omzet in de binaire representatie, in netwerk-bytevolgorde :

```
#include <stdio.h>      /* fprintf() */
#include <netdb.h>      /* gethostbyname() */
#include <stdlib.h>     /* exit() */

unsigned long ResolveName(char name[]){
    struct hostent *host;  /* host information */
```

```
    if ((host = gethostbyname(name)) == NULL){
        herror("gethostbyname()");
        return 0;
    }

    /* Return binary, network byte ordered address */
    return *((unsigned long *) host->h_addr_list[0]);
}

int main(int argc, char **argv){
  printf("binair adres = %d\n", ResolveName(argv[1]));
  return 0;
}
```

Listing 12.1: ResolveName.c

Een uitbreiding op *gethostbyname* is de functie *gethostbyname2* die ook voor Ipv6 werkt.

12.2 *gethostbyaddr*

Deze functie doet precies het omgekeerde van *gethostbyname*; ze converteert een binair IP adres naar zijn naam.

```
  #include <sys/socket.h>        /* voor AF_INET */
  #include <netdb.h>
  extern int h_errno;

  struct hostent *gethostbyaddr( const char *addr, int len, int family);

Return : pointer naar hostent, NULL pointer bij fout
```

Het eerste argument bevat het IP adres in binaire representatie. *len* is de grootte van dit adres (4 bij IPv4 en 16 bij IPv6). Het laatste argument geeft aan of we met IPv4 (*AF_INET*) of IPv6 (*AF_INET6*) werken. Het resultaat wordt bewaard in een *hostent* structure.

Volgend programma levert de officiële naam van een host :

```
#include <stdio.h>      /* printf() */
#include <netdb.h>      /* gethostbyname() */
#include <stdlib.h>     /* exit() */
#include <string.h>     /* strlen() */

unsigned long ResolveName(char name[]){
    struct hostent *host;  /* host information */

    if ((host = gethostbyname(name)) == NULL){
        herror("gethostbyname()");
        return 0;
    }

    /* Return binary, network byte ordered address */
    return *((unsigned long *) host->h_addr_list[0]);
}

char * ResolveAddress(unsigned long bin){
    struct hostent *host;  /* host information */
```

```
    if ((host=gethostbyaddr((char *)&bin, sizeof(bin), AF_INET)) == NULL){
        herror("gethostbyaddress()");
        return NULL;
    }

    /* Return hostname */
    char * ret = malloc(strlen(host->h_name) + 1);
    strcpy(ret, host->h_name);
    return ret;

}

int main(int argc, char **argv){
  printf("we onderzoeken %s\n", argv[1]);
  unsigned long binair = ResolveName(argv[1]);
  printf("binair = %ld\n", binair);
  char *naam = ResolveAddress(binair);
  printf("naam = %s\n", naam);
  free(naam);
  return 0;
}
```

Listing 12.2: ResolveAddress.c

12.3 Opzoeken van Service Informatie

Je kan ook informatie opzoeken over diensten (*services*) die worden aangeboden. Met de functie *getservbyname* kan je te weten komen op welke poort een bepaalde service wordt aangeboden. Indien je, omgekeerd, wil weten welke service op een bepaalde poort wordt aangeboden, dan kan je de functie *getservbyport* gebruiken. Je geeft telkens op voor welk protocol je informatie wenst; “tcp” of “udp”.

```
  #include <netdb.h>

  struct servent *getservbyname(const char *name, const char *proto);

  struct servent *getservbyport(int port, const char *proto);

  Return : pointer naar servent, NULL pointer bij fout
```

Het resultaat van beide functies zit in een *servent* struct :

```
  struct servent {
    char    *s_name;        /* official service name */
    char    **s_aliases;    /* alias list */
    int     s_port;         /* port number */
    char    *s_proto;       /* protocol to use */
}
```

De informatie die de functies teruggeven wordt gehaald uit een lokale databank of */etc/services* en biedt geen garantie dat ze correct of universeel geldig is.

In volgend programma wordt het gebruik van *getservbyname* gedemonstreerd :

```
#include <stdio.h>      /* for printf() and fprintf() */
#include <netinet/in.h> /* for htons() */
#include <netdb.h>      /* for getservbyname() */
#include <stdlib.h>     /* for atoi() */

unsigned short ResolveService(char service[], char protocol[]){
    struct servent *serv;        /* service information */
    unsigned short port;         /* Port to return */

    if ((port = atoi(service)) == 0){  /* Is poort numeriek? */
        /* Niet numeriek.  Zoek op */
        if ((serv = getservbyname(service, protocol)) == NULL){
            fprintf(stderr, "getservbyname() failed\n");
            exit(1);
        }
        else
            port = serv->s_port;   /* poort gevonden (network byte order) */
    }
    else
        port = htons(port);  /* zet om naar network byte order */

    return port;
}

int main(int argc, char **argv){
  if(argc!=3){
    printf("gebruik: %s <naam> <protocol>\n", argv[0]);
    exit(1);
  }
  printf("we onderzoeken service %s voor protocol %s\n", argv[1], argv[2]);
  unsigned short poort = ResolveService(argv[1], argv[2]);
  printf("deze service wordt aangeboden op poort %d\n", ntohs(poort));
  return 0;
}
```

Listing 12.3: ResolveService.c

Hoofdstuk 13

Boodschappen opstellen

Wanneer applicaties data willen versturen en ontvangen naar en van andere applicaties, dan moet er een communicatieprotocol worden afgesproken. De ene moet immers precies weten in welke vorm de andere data zal versturen en omgekeerd. In de voorgaande hoofdstukken was dit eenvoudig; een echo-server ontvangt een willekeurig aantal karakters en stuurt deze terug. Normaalgezien zal een protocol er iets complexer uitzien. In dit hoofdstuk bespreken we enkele aspecten hiervan.

13.1 Data coderen

Stel dat een financiële applicatie een boodschap moet kunnen sturen naar een andere applicatie, via sockets, waarin twee getallen staan : een bedrag dat aangeeft hoeveel geld er is gestort die dag (*deposits*) en een tweede bedrag dat aangeeft hoeveel geld er werd afgehaald die dag (*withdrawals*).

Een mogelijke manier om getallen te versturen is door er een string van te maken en deze string aan *send()* door te geven. Men zou kunnen afspreken dat het einde van een getal wordt aangegeven door een spatie.

```
sprintf(message, "%d %d ", deposits, withdrawals);
send(s, message, strlen(message), 0);
```

Stel dat de applicatie de getallen 17.998 en 47.034 wil versturen. De boodschap ziet er dan als volgt uit :

49	55	57	57	56	32	52	55	48	51	52	32
'1'	'7'	'9'	'9'	'8'	'_'	'4'	'7'	'0'	'3'	'4'	'_'

Deze methode heeft een voordeel maar ook enkele gevaren :

- Je kan willekeurige grote getallen representeren op deze manier. Het is namelijk zo dat computers beperkt zijn in de binaire voorstelling van integers. Indien op een systeem gekozen wordt voor bijvoorbeeld 4-byte integers, dan is de hoogste positieve waarde 2.147.483.647 [1] Indien de applicatie met grotere getallen moet kunnen werken dan zit er niet veel anders op dan een eigen representatie te construeren, bijvoorbeeld met strings.

- De buffer *message* moet groot genoeg zijn om de hele boodschap te bevatten. De maximale lengte in dit geval zou 23 zijn : de cijfers in de twee getallen, de eventuele mintekens, de spaties en het afsluitende 0-karakter (*null terminator*).

- De functie *send()* stuurt de null-terminator niet mee in de boodschap. Deze moet de ontvanger er zelf aanplakken. Een foute manier is de volgende :

```
#define BUFSIZE 100
...
char message[BUFFSIZE];
...
sprintf(message, "%d %d ", deposits, withdrawals);
send(s, message, BUFSIZE, 0);
```

  Dit is fout want je stuurt hiermee meer bytes dan nodig. De ontvangende partij kan hiermee in de problemen komen omdat ze niet weet wat het einde van de ene boodschap is en het begin van een tweede. De rommel die wordt meegestuurd met de eerste boodschap kan verkeerdelijk voor het begin van een tweede boodschap worden gezien. Merk op dat het in TCP helemaal niet vanzelfsprekend is om dit probleem foutvrij aan te pakken! Een correcte manier is het gebruik van *strlen(message)*.

- Het encoderen van getallen als strings is weinig efficiënt. In het slechtste geval wordt een int van 4 bytes gecodeerd als een string van 11 bytes. Bovendien is het ook vervelend om strings die getallen voorstellen te manipuleren (tel maar eens twee strings op). Het converteren neemt op zich ook tijd en geheugen in beslag.

We kunnen de boodschap ook op een binaire manier versturen. De functie *send()* stuurt immers alle bytes die ze binnenkrijgt netjes door naar de socket, of dit nu karakters uit een string voorstellen of bytes van een integer zijn. Een mogelijke implementatie is deze :

```
struct{
  int dep;
  int wd;
} msgStruct;

...

msgStruct.dep = deposits;
msgStruct.wd = withdrawals;
send(s, &msgStruct, sizeof(msgStruct), 0);
```

[1] Aangenomen dat het systeem een *two-complement* voorstelling hanteert.

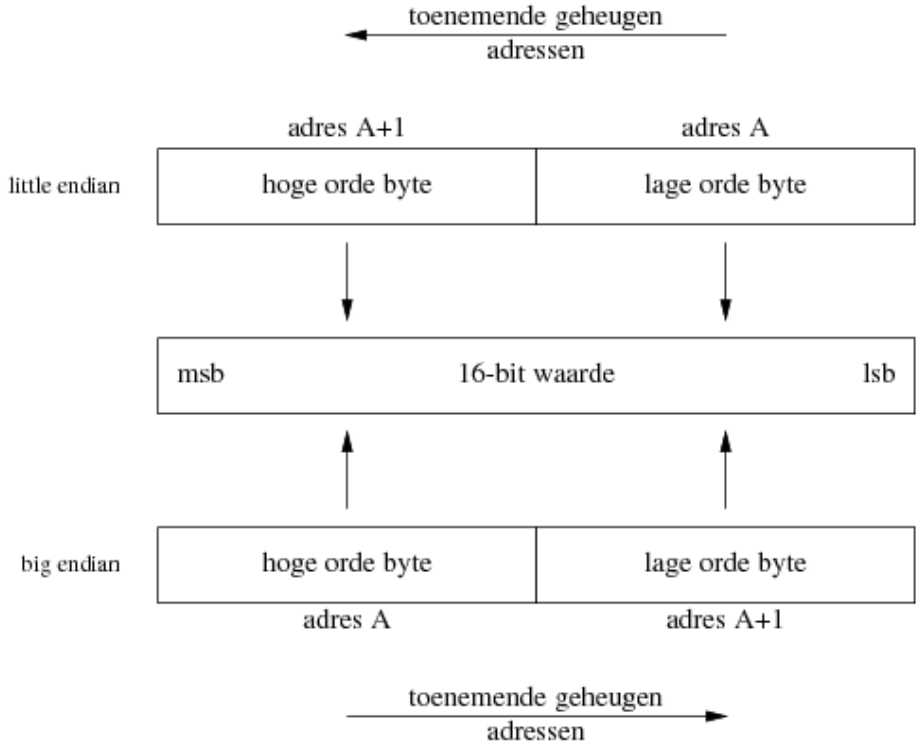
We zouden het in TCP ook eenvoudiger kunnen doen[2] :

```
send(s, &deposits, sizeof(deposits), 0);
send(s, &withdrawals, sizeof(withdrawals), 0);
```

In beide gevallen worden 8 bytes verstuurd, 4 voor de eerste integer en 4 voor de tweede. Er is echter een mogelijk probleem : de machine waarop de integers verstuurd worden en de machine waarop ze ontvangen worden hebben niet noodzakelijk dezelfde integer-representatie. In de volgende sectie wordt dit verder onderzocht.

13.2 Bytevolgorde

Een 16-bit integer bestaat uit 2 bytes. De twee bytes kunnen op twee verschillende manieren in een computergeheugen worden opgeslagen : met de lage orde byte (*least significant byte*) op het eerste adres (*little endian* byte-volgorde) of met de hoge orde byte (*most significant byte*) op de eerste plaats (*big endian* byte-volgorde). De termen “little endian” en “big endian” duiden aan welk eind van de multibyte-waarde, het kleine of het grote eind, in het begin van het geheugen staat.



Figuur 13.1: Little en big endian byte orde bij een 16bits integer

[2]In UDP gaat dit niet omdat dit twee afzonderlijk verzonden datagrammen zou opleveren.

De bytevolgorde die wordt gevolgd door de computer waarop het programma zal draaien, noemt men de *host*volgorde. De bytevolgorde op het netwerk noemt men de *netwerk*volgorde en is steeds big endian. Er bestaan een aantal functies om gehele getallen die in de ene volgorde zijn opgeslagen, om te zetten naar de andere volgorede. Voor *shorts*, getallen van 16 bits (bijvb. poortnummers), zijn dit *htons* en *ntohs* die resp. omzetten van hostvolgorde naar netwerkvolgorde en van netwerkvolgorde naar hostvolgorde. Voor *longs*, getallen van 32 bits (bijvb. IP-adressen), zijn dit resp. *htonl* en *ntohl*.

```
#include <netinet/in.h>

unsigned long int  htonl(unsigned long int hostlong);
unsigned short int htons(unsigned short int hostshort);
unsigned long int  ntohl(unsigned long int netlong);
unsigned short int ntohs(unsigned short int netshort);
```

Op machines die big endian zijn, doen deze functies niets (ze zijn gedefinieerd als lege macro's), maar dat betekent nog niet dat je ze zomaar mag weglaten. Het is namelijk een goed idee om al je programma's (en netwerkprogramma's in het bijzonder) zo veel mogelijk machineonafhankelijk te maken. Pentiumprocessoren bijvoorbeeld zijn little endian, Macs gebruiken dan weer big endian.

Met volgend programmaatje kan je gemakkelijk nagaan of het systeem big dan wel little endian is :

```
#include <stdio.h>

int main(){
  union {
    short  s;
    char   c[2];
  } un;

  un.s = 0x0102;  /* 2-byte waarde */
  if (un.c[0] == 1 && un.c[1] == 2)
    printf("big endian\n");
  else printf("little endian\n");

  return 0;
}
```

Listing 13.1: WhichEndian.c

Het programma gaat er vanuit dat een *short* twee bytes geheugenruimte inneemt. Merk op dat de byte-volgorde-functies geen effect hebben op *char*s, aangezien deze slechts één byte innemen.

Je zal merken dat het soms nodig is getallen of velden om te zetten naar Netwerk Byte Orde en soms juist niet. In een *struct sockaddr_in* bijvoorbeeld, moeten de velden *sin_addr* en *sin_port* in netwerk byte orde omdat deze velden worden meegestuurd in de IP-paketten terwijl het veld *sin_family* enkel door de lokale kernel wordt gebruikt, en dus in Host Byte Orde moet staan.

In ons voorbeeld van de bankapplicatie is het duidelijk dat de twee integers in netwerkvolgorde moeten worden verstuurd :

```
msgStruct.dep = htonl(deposits);
msgStruct.wd = htonl(withdrawals);
send(s, &msgStruct, sizeof(msgStruct), 0);
```

De ontvangende partij moet uiteraard de integers terug omzetten naar hostvolgorde :

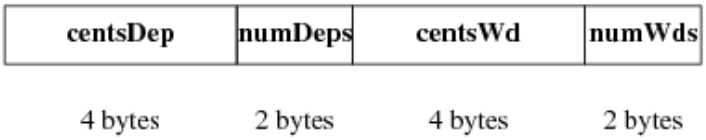
```
recv(s, &msgStruct, sizeof(msgStruct), 0);
deposits = ntohl(msgStruct.dep);
withdrawals = ntohl(msgStruct.wd);
```

13.3 Uitlijning en opvulling

Wanneer boodschappen bestaan uit verschillende velden van verschillende grootte, speelt de *alignment* van deze velden een rol. Stel dat we in het bank-voorbeeld niet enkel de bedragen maar ook het aantal stortingen en afhalingen willen doorsturen. Indien deze aantal maximaal 65.535 bedragen, kunnen we ze opslaan in twee bytes; in C zou een *unsigned short* hier geschikt voor zijn en kunnen we de struct als volgt uitbreiden :

```
struct{
  int centsDep;               // 4 bytes
  unsigned short numDeps;     // 2 bytes
  int centsWd;                // 4 bytes
  unsigned short numWds;      // 2 bytes
} msgStruct;                  // 12 bytes ???

...
send(s, &msgStruct, sizeof(msgStruct), 0);
```

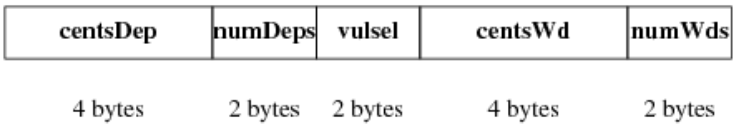
Grafisch kan je de datastructuur als volgt voorstellen :



Op sommige systemen worden hiermee de verwachte 12 bytes verstuurd, maar op andere 14 of zelfs 16. De oorzaak hiervan is de zogenaamde *uitlijning* of *alignment* van de struct. De compiler zorgt ervoor dat de velden van de datastructuur worden uitgelijnd. Volgende principes worden hierbij gehanteerd :

1. datastructuren worden maximaal uitgelijnd; d.w.z. dat hun adres deelbaar is door de grootte van de grootste *native* integer.
2. de velden van een datastructuur worden uitgelijnd volgens hun grootte; het adres van een 4-byte integer is deelbaar door 4; het adres van een 2-byte integer is deelbaar door 2 enzovoort.

In het bovenstaande voorbeeld, is het adres van *msgStruct*, noem het *A*, deelbaar door 4 (grootte van een long integer), alsook het veld *centsWd*. Maar dit veld ligt op een 2-byte grens : $A + 4 + 2$. De compiler zal dus zelf een 2-byte veld invoegen, vlak voor *centsWd* :

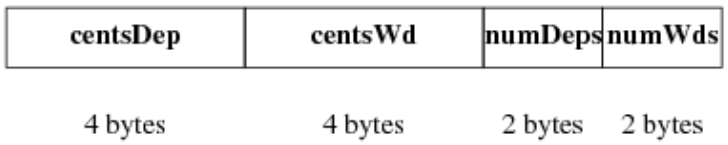


Dit is uiteraard een zeer gevoelige verandering wanneer het erop aan komt precieze afspraken te maken over het sturen en ontvangen van boodschappen. Wanneer de ontvanger immers volgende code zou gebruiken om een struct te ontvangen :

```
recv(s, &msgStruct, sizeof(msgStruct), 0);
```

dan zouden de laatste twee bytes, het veld *numWds*, niet ontvangen worden.

Een mogelijke oplossing is het zelf invoegen van deze opvulling of het herschikken van de velden in de struct. Wanneer we bijvoorbeeld volgende schikking zouden nemen :



dan zijn alle velden reeds correct uitgelijnd. De structure zou er dan als volgt uitzien :

```
struct{
  int centsDep;               // 4 bytes
  int centsWd;                // 4 bytes
  unsigned short numDeps;     // 2 bytes
  unsigned short numWds;      // 2 bytes
} msgStruct;                  // 12 bytes
```

Nu is de uitkomst van *sizeof(msgBuf)* gelijk aan 12, zoals gewenst.

13.4 Het verwerken van boodschappen

Wanneer we een protocol opstellen moeten we op de een of andere manier duidelijk maken hoe de boodschappen eruit gaan zien. De ontvanger moet onder andere weten wanneer een boodschap volledig ontvangen is.

In TCP is dit niet vanzelfsprekend. Twee boodschappen, verstuurd met twee aanroepen van *send()*, kunnen bij de ontvanger in één keer ontvangen worden door één aanroep van *recv()*, en omgekeerd.

Indien de velden in de boodschap een vaste lengte hebben dan is het niet zo moeilijk. De ontvanger leest dan van de socket tot hij het vereiste aantal bytes heeft binnengekregen.

```
struct msgStruct msg;
void *buffer = (void *) &msg;
int rBytes, rv;
...
for(rBytes=0; rBytes< sizeof(msg), rBytes += rv){
  rv = recv(s, buffer+rBytes, sizeof(msg)-rBytes, 0);
  if(rv<=0)
    ExitWithError("fout bij lezen data");
}
/* byte order ! */
msg.centsDep = ntohl(msg.centsDep);
...
```

Wanneer we met text-string representatie werken, zoals in het begin van dit hoofdstuk, dan gebruikt men doorgaans een *begrenzer* of *delimiter* (in ons voorbeeld het spatie-karakter) en bestaat een boodschap uit een vast aantal velden (in het voorbeeld dus 2; *dep* en *wd*). Deze aanpak is helemaal niet triviaal in TCP wegens het samennemen en splitsen van boodschappen over verschillende pakketten.

Een functie die een aantal velden inleest, begrensd door een bepaalde *delimiter* zou er als volgt kunnen uitzien :

```
int ReceiveMesg(int s, char *buf, int numFields, int maxLen, char delimiter){
  int received=0;
  int delimCount=0;
  int rv;
  while((received<maxLen)&&(delimCount<numFields)){
    rv = recv(s, buf+received, 1, 0);
    if(rv<0)
      ExitWithError("recv() failed");
    else if(rv==0)
      ExitWithError("onverwacht einde van verbinding");
    if(*(buf+received)==delimiter)
      delimCount += 1;
    received += 1;
  }
}
```

Merk dat de functie slechts één byte per keer van de socket haalt. Dit is niet zo efficiënt maar het bespaart ons wel een boekhouding bij te moeten houden van de ontvangen data.

Hoofdstuk 14

Socket Opties

14.1 Inleiding

Om meer geavanceerde socketapplicaties te schrijven, is het soms nodig bepaalde opties in te stellen die het gedrag van sockets verfijnen. Dit gebeurt voornamelijk met de functies *getsockopt* en *setsockopt*. De functie *fcntl* wordt gebruikt om opties in te stellen m.b.t. nonblocking I/O, signal-driven I/O en de eigenaar van een socket in te stellen.

14.2 De *getsockopt* en *setsockopt* functies

```
#include <sys/socket.h>

int getsockopt(int sockfd, int level, int optname,
               void *optval, int *optlen);

int setsockopt(int sockfd, int level, int optname,
               const void *optval, int optlen);

Beide return: 0 als OK, -1 bij error
```

sockfd is een open socket. De *level* geeft aan welk deel van het systeem de optie moet behandelen: de socket (*SOL_SOCKET*)of een protocol-specifiek onderdeel (*IPPROTO_IP* voor IPv4, *IPPROTO_IPV6* voor IPv6 en *IPPROTO_TCP* TCP).

optval is een pointer naar de waarde die werd verkregen via *getsockopt* of die moet worden ingesteld via *setsockopt*. De lengte van *optval* wordt aangegeven in *optlen*.

Er zijn talloze opties mogelijk voor sockets en we bespreken er hier bij wijze van voorbeeld slechts 3: *SO_REUSEADDR*, *SO_RCVBUF* en *SO_SNDBUF*. Zie *man 7 socket* voor een volledig overzicht van socket opties.

14.2.1 *SO_REUSEADDR*

Wanneer je *bind* aanroept, kan je de foutmelding "Socket already in use" krijgen. Dit gebeurt wanneer je een socket tracht te binden aan een adres en poort waaraan voorheen een andere socket was gebonden. De oude socket houdt de kernel verkeerdelijk voor dat de poort nog steeds in gebruik is. Ofwel wacht je dan enkele minuten en start je de server opnieuw op, ofwel geef je aan de socket die je gaat gebruiken voor listener socket, de optie *SO_REUSEADDR* (Re-Use Address) mee :

```
...
int yes=1;
setsockopt(listen_sd, SOL_SOCKET,
           SO_REUSEADDR, &yes, sizeof(int));

listen(listen_sd, BACKLOG);
...
```

Het is een goed idee om telkens voor het aanroepen van *listen* deze optie in te stellen.

14.2.2 *SO_RCVBUF* en *SO_SNDBUF*

De prestaties van een netwerktoepassing kunnen grotendeels worden bepaald door de groottes van de lees- en schrijfbuffer die TCP gebruikt. Interactieve toepassingen als telnet werken best met een kleine buffer:

- De client stuurt doorgaans kleine hoeveelheden data naar de server (meestal commando's als *ls* en dergelijke). Het gebruik van een grote buffer zou hier een verspilling van geheugen betekenen.
- Bij een interactief programma is het meestal de bedoeling dat de software redelijk snel reageert op commando's van de gebruiker. Wanneer men met grote buffers werkt, kan het lang duren eer een bepaald commando (bijvoorbeeld control-C om de uitvoer van een vorig commando te onderbreken) effectief verstuurd wordt.

Toepassingen die daarentegen grote hoeveelheden data moeten versturen (bijvb. een ftp server) zijn beter af met een grote buffer; vele kleine pakketten versturen duurt bij TCP, wegens de uitwisseling van ACK pakketten voor betrouwbaarheid, immers veel langer dan één groot. Bij toepassingen die veel data versturen gaat men ervan uit dat de ideale buffergrootte gelijk is aan of groter dan de *bandbreedtevertraging*. De bandbreedte is het aantal bits dat per tijdseenheid op een bepaald fysiek medium kan verzonden worden. Bandbreedte wordt doorgaans gemeten in bits per seconde. Ethernet heeft bijvb. een bandbreedte van tien megabits per seconde. De vertraging wordt veroorzaakt door het feit dat een pakket een bepaalde tijd nodig heeft om van de ene host naar de andere te reizen. Bij TCP wordt immers bij het verzenden van een pakket steeds een ACK teruggestuurd door de ontvanger van het pakket. Men spreekt in dit geval van de RTT : de Round Trip Time; de tijd die nodig is voor een pakket om van host A naar host B en terug te reizen.

Bandbreedtevertraging is dan als volgt te berekenen :

$$\text{BWD} = \text{bandbreedte} \times \text{RTT}$$

Als we RTT in seconden meten, krijgen we:

$$[\text{BWD}] = \frac{bits}{seconden} \times seconden = bits$$

In theorie is dit allemaal mooi maar als programmeur weet je meestal niet wat de bandbreedtevertraging zal zijn in de omgeving waar het programma moet draaien; er zijn vele factoren die invloed hebben en enkel het effectief meten geeft betrouwbare resultaten. Men heeft dan voorgesteld om de buffergroottes dynamisch te maken; eerst wordt met behulp van een ping-achtig programma de BWD berekend en vervolgens de buffers aangemaakt. Het probleem hierbij is dan nog dat in de loop van de uitvoer van een programma de BWD kan veranderen en dat de buffers dus constant moeten aangepast worden. Hoedanook is dit een zeer complexe zaak en het resultaat is verre van gegarandeerd.

In 1995 ontdekten Comer en Lin een eenvoudige regel die in bijna alle gevallen goede resultaten biedt :

$$\text{verzendbuffer} \geq 3 \times \text{MTU}$$

waarbij MTU de *maximum transmission unit* is, de grootste fragmentgrootte die door het netwerk kan worden verwerkt. Voor Ethernet is dit 1500 bytes. Andere datalinks, als PPP, hebben een instelbaar MTU. Je kan de MTU van een netwerkinterface te weten komen met de opdracht *netstat -i*. De grootte van de ontvangstbuffer speelt niet zo'n rol en kan je op standaard laten.

In de praktijk :

```
...
int sd;
int bufgrt = 3 * 1500 // voor ethernet
...
sd = socket(AF_INET, SOCK_STREAM, 0);
setsockopt(sd, SOL_SOCKET, SO_SNDBUF,
           (char *)&bufgrt, sizeof(bufgrt));
connect(sd, ...);
...
```

14.3 De functie *fcntl*

Deze *file control* functie laat toe enkele opties in te stellen voor descriptoren, dus ook sockets. Meer bepaald nonblocking (zie hoofdstuk 22) en asynchrone functionaliteit wordt verkregen door *fcntl* aan te roepen met *O_NONBLOCK* en *O_ASYNC*.

```
#include <fcntl.h>

int fcntl(int fd, int cmd, int flags );

Return : -1 bij error
```

De huidige status van de *flags* van een descriptor verkrijg je met de *F_GETFL* opdracht. Je verandert de status met het *F_SETFL* commando. Een typisch code-voorbeeld:

```
int flags;

flags=fcntl(fd,F_GETFL,0);
flags = flags | O_NONBLOCK;
fcntl(fd, F_SETFL, flags);
```

Merk op dat je eerst de vlagstatus opvraagt en dan de nodige opties toevoegt door logische XOR (met de | operator). Opties verwijderen doe je met een logische AND (&) en logische NOT (~) :

```
flags = flags &  ~ O_NONBLOCK
```

Hoofdstuk 15

Processen

15.1 Eigenschappen

Een proces is een instantie van een programma dat wordt uitgevoerd. Het bevat de volgende elementen:

- de huidige status van het proces
- de actieve directory (*working directory*) van het proces
- de bestanden waartoe het proces toegang heeft
- de toegangsrechten van het proces (wordt bepaald door wie de eigenaar is van het proces)
- de bronnen die aan het proces zijn toegewezen door de kernel; geheugen, CPU rechten, etc.

De kernel houdt een overzicht bij van alle processen die actief zijn en zorgt voor een proces-planning (*schedule*); elke proces krijgt op haar beurt een beetje (*time slice*) CPU tijd. Hierdoor lijkt het alsof verschillende processen simultaan door een CPU worden uitgevoerd.

Elk proces heeft een aantal eigenschappen die door de kernel intern worden bijgehouden:

- een *pid* : de proces identifier; een uniek getal voor elk proces. Deze waarde wordt teruggegeven door de functie *getpid.*
- een *ppid* : de pid van het ouder (*parent*) proces. Deze waarde wordt teruggegeven door de functie *getppid*. Een proces is een kind van het proces waardoor het werd opgestart, het ouder-proces.
- de user en group ID van de eigenaar van het proces, dit is de gebruiker die het proces heeft opgestart.
- de CPU-tijd die het proces reeds heeft gebruikt

Vanuit een shell kan je de processen die actief zijn opvragen met het programma *ps*. Een voorbeeld zou zijn:

```
bsd> ps -o pid,ppid,command
  PID  PPID COMMAND
 3837  3737 /bin/bash
 4353  3837 ps -o pid,ppid,command
```

Zie man pages voor alle opties van *ps*.

In het volgende programmavoorbeeld worden enkele eigenschappen van een proces opgevraagd en naar het scherm geschreven:

```
#include <unistd.h>
#include <stdio.h>

int main(){
  printf("PID       = %d\n", getpid() );
  printf("PPID      = %d\n", getppid() );
  printf("user ID   = %d\n", getuid() );
  printf("groupd ID = %d\n", getgid() );

  return 0;
}
```

Listing 15.1: proces eigenschappen

15.2 Het maken van processen met *fork*

De functie *fork* start een nieuw proces dat een kopie is van het ouderproces. Dit nieuwe proces noemen we het *kindproces*.

```
#include <sys/types.h>
#include <unistd.h>

pid_t fork(void);

Return : Bij succes de PID van het kind in het ouderproces,
         en 0 in het kindproces.
Bij fout = -1.
```

Bij succes geeft *fork* de PID van het kindproces terug aan het ouderproces. In het kindproces is de returnwaarde 0. Het verschil in returnwaarde is de manier waarop we het onderscheid kunnen maken tussen ouder- en kindproces. Het volgende voorbeeld illustreert dit.

```
#include <unistd.h>
#include <sys/types.h>
#include <stdio.h>

int main(){
  pid_t PID;

  PID = fork();
  if (PID==-1){
    perror("fork");
    return -1;
```

```
  } else if (PID == 0){
    printf("in kind\n\tPID = %d\n\tPPID = %d\n", getpid(), getppid());
    return 0;
  } else {
    printf("in ouder\n\tPID = %d\n\tPPID = %d\n", getpid(), getppid());
    return 0;
  }

  // hier geraakt men niet
  return 0;
}
```

Listing 15.2: Een kindproces opstarten

Een typische uitvoer voor bovenstaand programma zou zijn :

```
in kind
        PID = 4244
        PPID = 4243
in ouder
        PID = 4243
        PPID = 3837
```

Maar het had evengoed dit kunnen zijn :

```
in ouder
in kind
        PID = 4244
        PPID = 4243
        PID = 4243
        PPID = 3837
```

waarbij de volgorde van de uitvoer veranderd is. Je kan immers onmogelijk voorspellen welke code uit welk proces het eerst wordt uitgevoerd; de kernel heeft hiervoor een planning waardoor elk proces beurtelings wat code mag uitvoeren. Er is geen enkele manier om te voorspellen hoe dit gebeurt. Zorg er dus voor dat je code nooit betrouwt op het reeds uitgevoerd zijn van code in een ander proces (tenzij je speciale maatregelen neemt; zie verder).

15.3 Het wachten op een proces

Wanneer een ouderproces een kindproces heeft opgestart met *fork*, kan het op het einde van dit kindproces wachten met de functie *wait*.

Indien een kindproces eindigt en het ouderproces vangt dit niet op met een aanroep aan *wait*, dan is dit afgesloten kindproces een *zombie proces*. Het geheugen en de toebedeelde bronnen zijn wel vrijgegeven, maar er bestaat nog steeds een verwijzing naar het proces in de kernel proces tabel, samen met de exit status van het proces. Dit wordt door de kernel bijgehouden tot het moment dat het ouderproces het ervan haalt met de *wait* functie.

Het is niet zo erg als er een of twee zombieprocessen in de kernel procestabel staan maar als het er honderden of meer worden dan geraakt de procestabel van de kernel volzet en moet het systeem herstart worden. Zombieprocessen herken je in de uitvoer van *ps* aan het label *defunct*.

Het omgekeerde is ook mogelijk : het ouderproces sluit af voor het kindproces afsluit. In dit geval wordt het kind een wees (*orphan*) en het *init*proces neemt de taak van ouder over waardoor het geen zombie wordt. Het init proces is het eerste proces dat door de kernel wordt opgestart en heeft PID 1.

Het is met andere woorden belangrijk voor een ouderproces om de kindprocessen op een correcte manier af te handelen. De functies *wait* en *waitpid* zijn hiervoor het best geschikt.

```
#include <sys/types.h>
#include <sys/wait.h>

pid_t wait(int *status);
pid_t waitpid(pid_t pid, int *status, int options);

Return: de PID van het afgesloten proces of -1 bij fout.
```

Met het argument *status* haal je de exit status van het proces binnen. In de tweede functie *waitpid* kan je het PID opgeven van het proces waarop je wil wachten met het argument *pid*. Je kan hiervoor ook -1 opgeven, en dan krijg je hetzelfde gedrag als *wait*, namelijk wachten op een willekeurig kindproces. Met het argument *options* kan je opgeven dat *waitpid* niet moet blokkeren als er niet direct een kind te vinden is dat afsluit of afgesloten is (optie *WNOHANG*) en dat de functie moet terugkeren voor kinderen die geen status hebben gerapporteerd bij het afsluiten (optie *WUNTRACED*). Je kan beide ook combineren met een logische of (|) operator.

Voorbeeld : een proces maakt een kindproces. Vervolgens wacht het ouderproces op het afsluiten van het kind en sluit dan zelf af.

```
#include <sys/wait.h>
#include <unistd.h>
#include <sys/types.h>
#include <stdio.h>

int main(){
  pid_t PID;
  int status;

  PID = fork();
  if (PID==-1){
    perror("fork");
    return -1;
  } else if (PID == 0){
    printf("in kind\n\tPID = %d\n\tPPID = %d\n", getpid(), getppid());
    return 0;
  } else {
    waitpid(PID, &status, 0);
    printf("in ouder\n\tPID = %d\n\tPPID = %d\n", getpid(), getppid());
    printf("\tkind eindigde met exit status %d\n", status);

    return 0;
  }
  return 0;
}
```

Listing 15.3: Wachten op kindproces

De exit status die de functie *wait* teruggeeft is een integer waarin enkele gegevens zijn verzameld. Met volgende macro's kan je de exit status interpreteren:

- *WEXITSTATUS* : deze macro geeft de waarde van de exit code van het kind; de waarde die werd meegegeven aan *exit* of *return*.
- *WIFEXITED* : geeft aan of het proces normaal is beëindigd; d.w.z. met een aanroep van *exit* of *true* als het proces normaal werd beëindigd en *false* als het door een niet afgehandeld signaal werd afgesloten (zie hoofdstuk 16).
- *WTERMSIG* : geeft het signaal nummer waardoor het proces werd afgesloten, als het werd afgesloten door een niet afgehandeld signaal.

We kunnen het vorige programma nu uitbreiden als volgt :

```
#include <sys/wait.h>
#include <unistd.h>
#include <sys/types.h>
#include <stdio.h>

int main(){
  pid_t PID;
  int status;

  PID = fork();
  if (PID==-1){
    perror("fork");
    return -1;
  } else if (PID == 0){
    printf("in kind\n\tPID = %d\n\tPPID = %d\n", getpid(), getppid());
    return 0;
  } else {
    waitpid(PID, &status, 0);
    printf("in ouder\n\tPID = %d\n\tPPID = %d\n", getpid(), getppid());
    if (WIFEXITED(status))
      printf("code van kindproces %d\n", WEXITSTATUS(status));
    else
      printf("kind afgesloten door signaal %d\n", WTERMSIG(status));

    return 0;
  }
  return 0;
}
```

Listing 15.4: exit status van kindproces

15.4 Descriptor referentie tellers

Bij het maken van kindprocessen wordt het volledig ouderproces gedupliceerd. Bovendien houdt de kernel voor elke descriptor een teller bij in de bestandstabel : de *descriptor reference count*. Deze teller geeft aan hoeveel hoeveel processen gebruik maken van een descriptor. Elk nieuw kindproces dat wordt aangemaakt erft alle descriptoren van het ouderproces en dus wordt voor elk nieuw proces de descriptor teller van alle open descriptoren met 1 verhoogd in de tabel.

Wanneer nu een proces de functie *close* aanroept met de bedoeling een descriptor (bestand of TCP connectie) te sluiten, dan zal de kernel in de tabel de teller met 1 verminderen. De descriptor wordt pas echt gesloten wanneer de teller de waarde 0 heeft bereikt.

Een probleem dat optreedt wanneer je niet nauwkeurig bijhoudt hoeveel processen een descriptor open houden, is dat een proces dat een TCP connectie wil verbreken door *close* aan te roepen daar niet in zal slagen als andere processen dezelfde descriptor open hebben. Zolang de referentie teller niet gelijk is aan 0 zal de TCP connectie behouden blijven.

Een andere oplossing is het gebruik van de functie *shutdown* (zie 21.3).

Zie voor een voorbeeld programma listing 15.5 in de volgende sectie.

15.5 Processen en netwerkprogrammatie

Processen kunnen en worden op vele manieren gebruikt bij netwerkprogrammatie. De bekendste technieken zijn:

- een server die voor elke binnenkomende connectie een nieuw kindproces start waarin de client wordt behandeld. Op deze manier kan een server op eenvoudige wijze talloze clients tegelijk bedienen.
- om blocking te voorkomen van invoer of uitvoer, worden deze acties in verschillende processen behandeld. Een echo client kan bijvoorbeeld invoer lezen van het toetsenbord en data sturen naar de server in een kindproces en data lezen van de server in het ouderproces. Op deze manier blijft communicatie met de server en de gebruiker mogelijk niettegenstaande de client geblokkeerd zit in een *read* van het toetsenbord of van de socket.
- wanneer een server langdurige operaties moet ondernemen bij het afhandelen van een client, bijvoorbeeld een ftp server die grote bestanden doorstuurt, dan kan m'n controle over de connectie bewaren via een tweede connectie die wordt opgezet in een kindproces, bij zowel server als client. Op die manier kan een client een data-overdracht gemakkelijk annuleren door op de tweede connectie een bericht te sturen.

We bekijken een voorbeeld van een echo-server die meerdere clients aankan via kindprocessen :

```
#include "../wrapper.h"

#define POORT   6789
#define BUFSIZE  256 /* max aantal bytes te lezen */

int verwerk_client(int client_sd){
  char buffer[BUFSIZE];
  int n;

  printf("nieuw proces : %d\n", getpid());

  n = Recv(client_sd, buffer, BUFSIZE, 0);
  while(n>0){
    Send(client_sd, buffer, n, 0);
    n = Recv(client_sd, buffer, BUFSIZE, 0);
  }

  printf("client verbreekt connectie\n");
```

```
  Close(client\_sd); /* descriptor ref. teller wordt 0*/
  exit(0); /* stop dit proces */
}

int main(){
  struct sockaddr_in client_adres;
  int listen_sd, client_sd; // socket descriptoren
  int adres_lengte;
  pid_t stop_PID;
  int status; // voor waitpid
  listen_sd = Tcp_listener(POORT);

  while(1){
    client_sd = Accept(listen_sd,
                       (struct sockaddr *)&client_adres, &adres_lengte);
    if (Fork()==0){
      /* kind */
      printf("nieuw proces : %d\n", getpid());
      Close(listen_sd); /* toch niet nodig */
      verwerk_client(client_sd);
    } else {
      /* ouder */
      Close(client_sd); /* descriptor ref. teller wordt 1 */
      /* kijk of er kindprocessen zijn gestopt */
      stop_PID = waitpid(0, &status, WNOHANG | WUNTRACED);
      if(stop_PID>0)
    printf("proces %d is gestopt.\n", stop\_PID);
    }
  }

  return 0;
}
```

Listing 15.5: echo server met processen

Merk op dat het kindproces begint met het sluiten van de *listen_sd* socket; die heeft het toch niet nodig. De socket wordt niet echt gesloten maar de descriptor referentie teller wordt met 1 verminderd; de socket is dus nog steeds open in het ouderproces.

Het ouderproces sluit de *client_sd* socket; ook hier hetzelfde verhaal. Indien de ouder dit niet zou doen, zou het kindproces niet in staat zijn de TCP verbinding te sluiten door het aanroepen van *close*.

Het opvangen van gestopte kindprocessen verloopt niet zo optimaal; omdat de server blokkeert in de aanroep van *accept*, wordt slechts wanneer een nieuwe connectie is aangegaan eenmaal gecheckt of er een kindproces is gestopt. Een betere oplossing is door het instellen van een timer op accept met de select functie (zie hoofdstuk 21) of door het opvangen van signalen (zie hoofdstuk 16).

Hoofdstuk 16

Signalen

Een manier waarop processen met elkaar kunnen communiceren is door het sturen van signalen. Zo'n signaal bevat geen data. Signalen worden *asynchroon* doorgestuurd, een proces kan dus op elk moment onderbroken worden door de ontvangst van een signaal. Dit wil zeggen: middenin het uitvoeren van een lijn code!

Er zijn verscheiden soorten signalen. Ze hebben elk hun eigen naam die steeds begint met *SIG* en je vindt ze allemaal terug in *bit/signum.h*:

Naam	**Nr**	**Omschrijving**
SIGHUP	1	Hangup (POSIX).
SIGINT	2	Interrupt (ANSI).
SIGQUIT	3	Quit (POSIX).
SIGILL	4	Illegal instruction (ANSI).
SIGTRAP	5	Trace trap (POSIX).
SIGABRT	6	Abort (ANSI).
SIGIOT	6	IOT trap (4.2 BSD).
SIGBUS	7	BUS error (4.2 BSD).
SIGFPE	8	Floating-point exception (ANSI).
SIGKILL	9	Kill, unblockable (POSIX).
SIGUSR1	10	User-defined signal 1 (POSIX).
SIGSEGV	11	Segmentation violation (ANSI).
SIGUSR2	12	User-defined signal 2 (POSIX).
SIGPIPE	13	Broken pipe (POSIX).
SIGALRM	14	Alarm clock (POSIX).
SIGTERM	15	Termination (ANSI).
SIGSTKFLT	16	Stack fault.
SIGCLD	SIGCHLD	Same as SIGCHLD (System V).
SIGCHLD	17	Child status has changed (POSIX).
SIGCONT	18	Continue (POSIX).
SIGSTOP	19	Stop, unblockable (POSIX).
SIGTSTP	20	Keyboard stop (POSIX).
SIGTTIN	21	Background read from tty (POSIX).

SIGTTOU	22	Background write to tty (POSIX).
SIGURG	23	Urgent condition on socket (4.2 BSD).
SIGXCPU	24	CPU limit exceeded (4.2 BSD).
SIGXFSZ	25	File size limit exceeded (4.2 BSD).
SIGVTALRM	26	Virtual alarm clock (4.2 BSD).
SIGPROF	27	Profiling alarm clock (4.2 BSD).
SIGWINCH	28	Window size change (4.3 BSD, Sun).
SIGPOLL	SIGIO	Pollable event occurred (System V).
SIGIO	29	I/O now possible (4.2 BSD).
SIGPWR	30	Power failure restart (System V).
SIGSYS	31	Bad system call.
SIGUNUSED	31	

Tabel 16.1: Mogelijke signalen

De meeste van deze signalen worden door de kernel naar processen gestuurd. Wanneer bijvoorbeeld een proces een ongeldig geheugenadres zou willen adresseren, ontvangt het een *SIGSEGV* signaal (segmentation violation). Het standaard gedrag van een proces is dan stoppen.

Wanneer een proces een signaal ontvangt, kan het er drie dingen mee doen:

- het signaal negeren
- het signaal opvangen (*catch a signal*) waardoor een speciale functie, de signal handler, wordt uitgevoerd. Wanneer deze functie terugkeert, kan het proces verdergaan met de uitvoering van het programma.
- toestaan dat de standaard actie die met het signaal is verbonden wordt uitgevoerd. Doorgaans is dit het beëindigen van het proces.

Wat een proces precies met een signaal doet, noemt men de *signaal dispositie*.

16.1 Signalen sturen

Het sturen van een signaal naar een proces gebeurt met de de functie *kill*. Je hoeft enkel de PID van het proces waarnaar je het signaal wil sturen en de naam of nummer van het signaal op te geven :

```
#include <sys/types.h>
#include <signal.h>

int kill(pid_t pid, int sig);

Return: bij succes 0, bij fout -1 en errno krijgt waarde
```

Merk op dat je met deze functie alle signalen kan sturen, niet enkel *SIGKILL*.

Het volgende voorbeeld is een programma dat het signaal *SIGTERM* naar zichzelf stuurt. De standaardactie die hiermee verbonden is, is dat het proces wordt afgesloten.

```
#include <sys/types.h>
#include <signal.h>
#include <unistd.h>
#include <stdio.h>

int main(){
  printf("proces gestart\n");

  kill(getpid(), SIGTERM);

  while(1)
    ; /* oneindige loop */

 return 0;
}
```

Listing 16.1: Stuur SIGTERM naar zelf

De uitvoer ziet er als volgt uit:

```
bsd> ./a.out
proces gestart
Terminated
```

16.2 Opvangen van signalen

Het ontvangen en afhandelen van signalen is iets complexer. Een proces kan elk signaal op eigen manier afhandelen, behalve *SIGSTOP* en *SIGKILL*; twee signalen die het proces onvermijdelijk doen stoppen.

Een signaal opvangen doe je met een signal handler. Deze breng je in gebruik met de functie *sigaction*:

```
#include <signal.h>

int sigaction( int signum, const struct sigaction *act,
               struct sigaction *oldact);

0 bij succes, -1 bij fout
```

Het eerste argument is het signaal waarvoor je de dispositie wil veranderen. Het tweede en derde argument zijn twee *sigaction* structures. De eerste bevat een bepaling van hoe de nieuwe dispositie moet zijn, de tweede krijgt bij return de waarden ingevuld van de oude dispositie. Het belangrijkste veld van deze structure is *sa_handler*. Hiermee geef je aan wat er precies moet gebeuren wanneer het proces het signaal ontvangt:

- *SIG_DFL* : de default dispositie van het signaal
- *SIG_IGN* : negeer het signaal
- een pointer naar een signal handler functie. Deze functie neemt één parameter mee; het nummer van het signaal.

In het volgende voorbeeld wordt de dispositie van zowel *SIGTERM* als *SIGUSR1* en *SIGUSR2* veranderd. Die laatste zijn overigens twee signalen die speciaal ter beschikking van de programmeur staan. Telkens een proces het signaal *SIGUSR1* stuurt, wordt een teller met één verhoogd. Wanneer het *SIGTERM* signaal wordt ontvangen, wordt de teller naar het scherm geschreven en het proces sluit zichzelf af.

```
#include <sys/types.h>
#include <signal.h>
#include <unistd.h>
#include <stdio.h>
#include <string.h> // bzero
#include <stdlib.h> // exit
int count = 0;

void usr1_handler(int signum){
  count++;
}

void term_handler(int signum){
  printf("SIGUSR1 werd %d keer opgevangen\n", count);
  exit(0); // stop proces
}

int main(){
  struct sigaction sa;

  bzero(&sa, sizeof(sa));

  sa.sa_handler = &usr1_handler;
  sigaction(SIGUSR1, &sa, NULL);

  sa.sa_handler = &term_handler;
  sigaction(SIGTERM, &sa, NULL);

  while(1)
    pause(); // wacht op signaal


 return 0;
}
```

Listing 16.2: Signalen opvangen in handler

In de *main* functie wordt herhaaldelijk de functie *pause* aangeroepen. Deze functie plaatst het proces in slaapmodus, tot hetwordt gewekt door een ontvangen signaal. Wanneer het *SIGTERM* signaal binnenkomt, wordt wordt de functie *term_handler* uitgevoerd en stopt het proces.

Je kan dit programma testen door er zelf signalen naar te sturen vanuit een shell met het commando *kill*. Als argumenten geef je het signaal mee dat je wil sturen en het PID van het proces waarnaar het signaal moet worden verstuurd. Bijvoorbeeld:

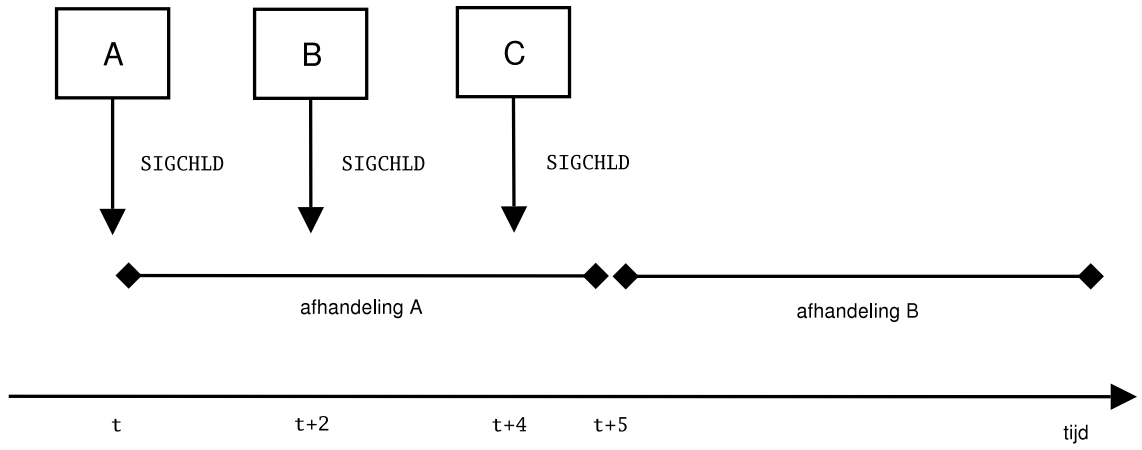
```
bsd> ./handler &}
bsd> ps -e | grep handler
 5639 pts/3    00:00:00 handler
bsd> kill -s SIGUSR1 5639
bsd> kill -s SIGUSR1 5639
bsd> kill -s SIGTERM 5639
SIGUSR1 werd 2 keer opgevangen
bsd>
```

Wanneer een signaal handler bezig is met het verwerken van een signaal, worden de signalen van dat zelfde type geblokkeerd.

16.3 Kindprocessen volgen met signalen

Signalen geven een ouderproces veel controle over hun kindprocessen doordat een kindproces een *SIGCHLD* signaal stuurt naar het ouderproces wanneer het stopt. Op deze manier kan het ouderproces ten gepaste tijde, nl. na het ontvangen van het signaal, *waitpid* aanroepen. Dit wordt verduidelijkt in het volgende voorbeeld:

```
#include <sys/types.h>
#include <signal.h>
#include <unistd.h>
#include <stdio.h>
#include <string.h> // bzero
#include <wait.h>
#include <stdlib.h> // random
#include <time.h>

#define AANTAL_KINDEREN 5
int stop_count = 0;

void chld_handler(int signum){
  int status;
  pid_t pid;
  while( (pid = waitpid(-1, &status, WNOHANG))>0){
    stop_count++;
    printf("proces %d gestopt\n", pid);
  }
}

void doe_iets(){
  int rust;
  srand(getpid()); /* init random generator */
  rust= 1 + random()%3;
  printf("kindproces %d slaapt %d seconden\n", getpid(), rust);
  sleep(rust);
  exit(0);
}

int main(){
  int i,pid;
  struct sigaction sa;

  bzero(&sa, sizeof(sa));

  sa.sa_handler = &chld_handler;
  sigaction(SIGCHLD, &sa, NULL);

  for(i=0;i<AANTAL_KINDEREN; i++){
    pid=fork();
    if(pid==0){
      doe_iets();
    } else if (pid<0){
      perror("fork");
      exit(0);
    }
  }
  printf("kinderen gestart\n");
```



Figuur 16.1: problemen met *SIGCHLD*

```
  while(stop_count < AANTAL_KINDEREN)
    pause();

  printf("alle kinderen gestopt!\n");
  return 0;
}
```

Listing 16.3: ouder roept waitpid aan na signaal

Dit programma geeft bijvoorbeeld volgende uitvoer :

```
bsd> ./waitpid
kindproces 8232 slaapt 3 seconden
kindproces 8233 slaapt 2 seconden
kindproces 8234 slaapt 1 seconden
kindproces 8235 slaapt 2 seconden
kindproces 8236 slaapt 2 seconden
kinderen gestart
proces 8234 gestopt
proces 8233 gestopt
proces 8235 gestopt
proces 8236 gestopt
proces 8232 gestopt
alle kinderen gestopt!
bsd>
```

Het is belangrijk in de signal handler voor het *SIGCHLD* signaal de functie *waitpid* aan te roepen en niet *wait*. Met de eerste functie kunnen we namelijk opgeven dat de functie onmiddellijk moet terugkeren wanneer er geen zombie-processen zijn door de optie *WNOHANG*. Op deze manier blokkeert de functie niet als er geen zombies zijn en kunnen we ze zonder gevaar voor blokkering een while lus aanroepen, zoals in het voorbeeld. Indien we dit niet zouden doen, dan zouden we waarschijnlijk enkele zombie-processen missen. Het is namelijk zo dat slechts één signaal in een wachtrij kan worden geplaatst. Stel het volgende scenario voor:

Er zijn drie kindprocessen A, B en C. In het ouderproces doet de signal handler voor *SIGCHLD* er 5 tijdseenheden over om een kindproces af te sluiten. Op tijdstip t sluit proces A af en wordt een *SIGCHLD*-signaal gestuurd naar het ouderproces. Op tijdstip t+2 sluit B af en op tijdstip t+4 sluit C

af. De afhandeling van proces A duurt tot t+5. Ondertussen ontvangt het ouderproces de twee signalen van B en C. Slechts één signaal kan in de wachtrij staan; er gaat dus één verloren. Op tijdstip t+5 wordt de signal handler weer aangeroepen wegens het ene signaal dat nog in de wachtrij zit en het volgende zombie-proces, B, wordt afgehandeld. Het zombieproces C zal nooit meer worden afgehandeld omdat er geen signalen meer toekomen bij het ouderproces.

In de oplossing die hierboven gegeven werd verloopt het afhandelen wel goed; op tijdstip t wordt begonnen met het eerste proces af te handelen. Op tijdstip t+5 wordt, nog steeds in dezelfde eerste aanroep van de handler, het tweede proces afgehandeld en op tijdstip t+10 het derde. De signal handler wordt dus maar een keer aangeroepen en er gaan twee signalen ”verloren”, maar alle processen worden correct afgesloten.

Hoofdstuk 17

Pipes

17.1 Basisconcepten

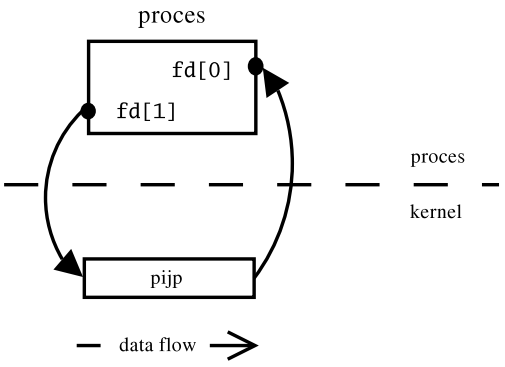
Een *pijp* is een manier om de *standaard output* van een proces te verbinden met de *standaard input* van een ander. Pijpen bieden één-richtingsverkeer (half-duplex) tussen processen.

Wanneer je op de commandolijn van een unix-machine het commando

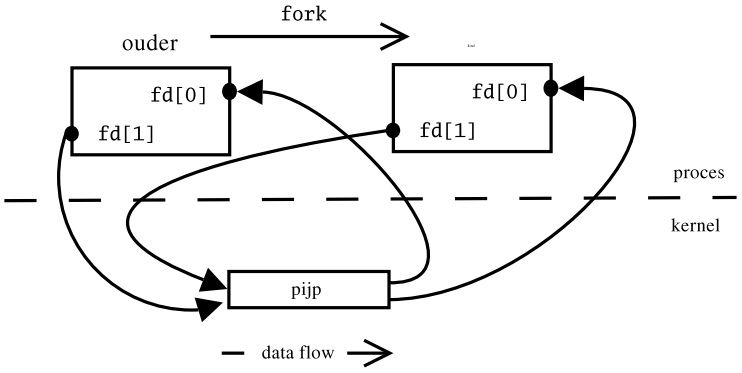
```
bsd> ls | sort | lp
```

geeft, wordt er een pijplijn opgezet waardoor de uitvoer van ls de invoer van sort wordt en de uitvoer van sort de invoer van lp. De pijp wordt gemaakt door het shell-proces (Bourne, C-shell, ...).

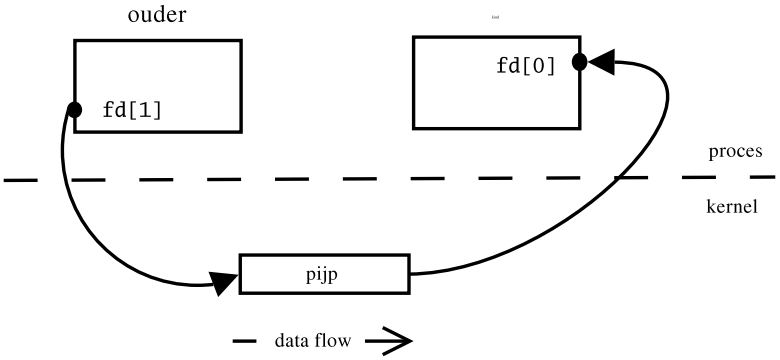
Wanneer je in een proces een pijp maakt, dan maakt de kernel twee descriptoren aan die door de pijp zullen worden gebruikt. De ene descriptor is de ingang van de pijp (input, write) en de andere de uitgang (output, read). De situatie is dan:



Figuur 17.1: een pijp in een proces



Figuur 17.2: een pijp in een proces met kindproces meteen na *fork*.



Figuur 17.3: een pijp tussen twee processen

Wanneer het proces data stuurt naar de pijp (*fd[1]*) dan kan het diezelfde data terug lezen via de uitgang, *fd[0]*. De data die het proces verstuurt reist dan door de pijp naar de kernel en de kernel stuurt alles weer terug door de pijp, in de andere richting. Dit is nog niet echt nuttig te noemen. Stel nu dat het proces een kindproces creëert dmv *fork* (zie hoofdstuk 15). Aangezien de bestandsdescriptoren van het ouderproces beschikbaar zijn in het kindproces, kunnen de twee processen nu via de pijp communiceren.

Beide processen hebben nu toegang tot de pijplijn via de gemeenschappelijke bestandsdescriptoren. Nu moet worden uitgemaakt in welke richting de data moet stromen. Stuurt het kind naar de ouder of omgekeerd? Beide processen moeten dit op voorhand hebben afgesproken. Ze sluiten de kant van de pijp die ze niet gaan gebruiken. Stel dat de ouder data naar het kind moet zenden, dan verkrijgen we de situatie in figuur 17.1.

De pijp is nu klaar voor gebruik. Via *read* en *write* kan data worden uitgewisseld. De pijp kan dus behandeld worden als een bestand maar hou er rekening mee dat functies als *lseek* ongeldig zijn op pijpen.

17.2 Pipes in C

In C maak je een pijp met de *pipe* functie:

```
#include <unistd.h>

int pipe(int fd[2]);

Return: 0 bij succes, -1 bij error, zie errno
```

Het enige argument is een array van twee integers. Bij succes bevat de array de twee descriptoren van de pijp. Na het aanmaken van de pijp wordt doorgaans geforkt. De eerste integer in de array wordt geopend voor lezen en de tweede voor schrijven.

Als de ouder data wil ontvangen van het kind, moet hij *fd[1]* sluiten en het kind *fd[0]*. Als de ouder data wil sturen sluit hij *fd[1]*. Als je er geen rekening mee houdt dat het ongebruikte einde moet gesloten worden, zal de uitvoer van de pijp geen *EOF* kunnen teruggeven. Eens de pijplijn is opgezet kunnen de descriptoren worden behandeld als standaard bestanden. Het volgende programma maakt een pijp, forkt, het kind stuurt naar de ouder en de ouder leest van het kind, via de pijp.

```
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
#include <string.h>
#include <stdlib.h>

int main(){
  int fd[2], nbytes;
  pid_t childpid;
  char string[]="Hello, world.\n";
  char readbuffer[80];
  pipe(fd);
  childpid=fork();
  if (childpid==0){
    // kind
    close(fd[0]);
    write(fd[1],string,strlen(string));
    exit(0);
  }else{
    // ouder
    close(fd[1]);
    nbytes=read(fd[0],readbuffer,sizeof(readbuffer));
    readbuffer[nbytes]=0; // 0 zelf zetten op einde string !
    printf("Ontvangen string:%s\n", readbuffer);
  }
  return 0;
}
```

Listing 17.1: Pipe en vervolgens fork

Wanneer het ouderproces *read* aanroept, blokkeert het proces tot op z'n minst één byte kan worden gelezen van de descriptor of een *EOF* toekomt. Het doet er bijgevolg niet toe of nu eerst het kind *write* dan wel de ouder *read* uitvoert.

17.3 Voorbeeld : som van de elementen van een tabel

In volgend voorbeeld wordt de som berekend van alle elementen van een twee-dimensionale tabel. De som van elke rij wordt in een apart proces berekend en via de pijp worden de resultaten teruggegeven aan het ouderproces.

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

#define N 3

int som_vector(int v[]);
void error_stop(char *s);

int main(){
  int a[N][N]={{1,1,1},{2,2,2},{3,3,3}},
      i,rij_som,som=0, pd[2];

  if (pipe(pd)==-1)
      error_stop("pipe");

    for (i=0;i<N;++i){
      if (fork()==0){
        close(pd[0]);
        rij_som = som_vector(a[i]);
        if (write(pd[1], &rij_som, sizeof(int))==-1)
          error_stop("write");
        return 0;
      }
    }
    close(pd[1]);
    for (i=0;i<N;++i){
      if(read(pd[0],&rij_som,sizeof(int))==-1)
        error_stop("read");
      som+=rij_som;
    }
    printf("De som = %d", som);
    return 0;
}

int som_vector(int v[]){
  int i,vector_som=0;
  for (i=0;i<N;++i)
    vector_som+=v[i];
  return vector_som;
}

void error_stop(char *s){
  printf("Fout bij %s\n", s);
  exit(1);
}
```

Listing 17.2: Bereken som van array met kindprocessen

17.4 Bemerkingen bij halfduplex pipes

- je kan een full-duplex pijp creëeren door twee half-duplex pijpen te maken en de descriptoren aan te passen.

- let erop dat je *pipe* aanroept voor *fork*, anders zijn de descriptoren niet gemeenschappelijk.
- half-duplex pijpen werken enkel wanneer de verbonden processen een gemeenschappelijke voorouder heeft die de pijp creëerde.

Hoofdstuk 18

FIFO's

18.1 Basisconcepten

FIFO is het acronym van *First in First Out*. Een FIFO wordt ook wel een *named pipe* of *benoemde pijp* genoemd. FIFO's werken ongeveer als gewone pijpen, maar er zijn enkele belangrijke verschillen.

- FIFO's zijn speciale devices in het bestandssysteem
- Processen die geen gemeenschappelijke voorouder hebben kunnen data delen door een FIFO.
- Wanneer alle IO-operaties zijn uitgevoerd blijft een FIFO in het bestandssysteem voor later gebruik

18.2 Een FIFO maken

Een FIFO maak je aan met de systeemfunctie *mkfifo*.

```
#include <sys/types.h>
#include <sys/stat.h>

int mkfifo(const char *pathname, mode_t mode);

Return: 0 bij succes, -1 bij error, zie errno
```

Het eerste argument is de naam die je de FIFO wil geven, het tweede de mode waarin je de FIFO wil aanmaken. Dit is een combinatie van waarden uit volgende tabel (terug te vinden in *stat.h*).

Een eenvoudige FIFO maak je door volgende parameters mee te geven

Constante	Omschrijving
S_IRUSR	user read
S_IWUSR	user write
S_IRGRP	group read
S_IWGRP	group write
S_IROTH	other read
S_IWOTH	other write

Tabel 18.1: mode constanten voor bij het aanmaken van nieuwe IPC objecten

```
mkfifo("/tmp/MYFIFO", S_IRUSR | S_IWUSR);
```

waarbij een bestand */tmp/MYFIFO* wordt aangemaakt als een FIFO bestand. De permissies zijn user read/write.

18.3 FIFO Operaties

Gewone Unix half-duplex pijpen bestaan niet in het bestandssysteem maar enkel in de kernel. FIFO zijn echter echte bestanden en ze moeten dan ook geopend en gesloten worden. Het doet er niet zo veel toe welke manier je kiest om met de FIFO-als-bestand te werken. We doen het hier met streams. In volgende listing 18.1 vind je de FIFO server.

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/stat.h>
#include <unistd.h>
#define FIFO_BEST "MYFIFO"
#define MODE (S_IRUSR | S_IWUSR)

int main(){
  FILE *fp;
  char readbuf[80];

  umask(0);
  mkfifo(FIFO_BEST, MODE);

  while ( 1 ){
    fp=fopen(FIFO_BEST,"r");
    fgets(readbuf,80,fp);
    printf("Ontvangen string:%s\n", readbuf);
    fclose(fp);
  }
  return(0);
}
```

Listing 18.1: Lezen van een FIFO

Laat de server lopen in de achtergrond (gezien de FIFO blokkeert, zie verder):

```
bsd> fifoserver &
```

Een client voor deze server zou er bijvoorbeeld als volgt uit kunnen zien:

```
#include <stdio.h>
#include <stdlib.h>
#define FIFO_BEST "MYFIFO"

int main(int argc, char **argv){
  FILE *fp;
  if (argc!=2){
    printf("Gebruik : fifoclient [string]\n");
    exit(1);
  }

  if ((fp=fopen(FIFO_BEST,"w"))==NULL){
    perror("fopen");
    exit(1);
  }

  fputs(argv[1],fp);
  fclose(fp);

  return(0);
}
```

Listing 18.2: Schrijven naar FIFO

18.4 Blocking FIFO's

Als je een FIFO opent voor lezen, blokkeert het proces tot een ander proces dezelfde FIFO opent voor schrijven. Vice versa geldt hetzelfde. Als je dat niet wil, kan je de FIFO openen met de *O_NONBLOCK* vlag in de aanroep van *open*.

Hoofdstuk 19

Remote procedure calls

19.1 Wat is RPC ?

Via *Remote procedure calls* (RPCs) kunnen processen functies aanroepen die deel uitmaken van een ander proces. Dit tweede proces kan op dezelfde of een andere host lopen. In het eerste geval gebruikt men *doors*, in het laatste geval gebruikt men *Sun RPC*.

We beperken ons tot Sun RPC. Uiteraard kan men hier ook RPC uitvoeren tussen twee processen die op dezelfde host lopen.

Het voordeel van RPC is dat je geen expliciete netwerkprogrammatie nodig hebt; men spreekt van *impliciete netwerkprogrammatie*. De client roept een RPC aan en krijgt een returnwaarde van de server zonder het gebruik van sockets. De eigenlijke netwerkcommunicatie wordt verzorgd door het RPC subsysteem.

19.2 Een voorbeeld

In dit voorbeeld wordt een systeem gebouwd waarin een client een kwadraatsfunctie aanroept bij een server. De client geeft één *integer* argument mee en krijgt als return waarde ook één *int* terug; het kwadraat van de invoer.

Een eenvoudig RPC systeem waarbij een client een functie aanroept bij een server (op een andere of dezelfde host) verloopt in drie stappen. Eerst moet men een *.x* bestand maken waarin de functies die de server aanbiedt worden gedefinieerd. Dan schrijven we een client programma dat de RPC aanroept en tenslotte maken we het server programma dat de RPC aanbiedt.

19.2.1 Het *.x* bestand

De syntax van zo'n *.x* bestand is standaard C, met toevoeging van twee nieuwe structures : *program* en *version*.

```
struct kwadraat_in {   /* input (argument) */
  long  arg1;
};

struct kwadraat_out {  /* output (result) */
  long  res1;
};

program KWADRAAT_PROG {
  version KWADRAAT_VERS {
    kwadraat_out KWADRAATPROC(kwadraat_in) = 1; /* procedure nr = 1 */
  } = 1;                /* versie nummer */
} = 0x31230000;         /* program nummer */
```

Listing 19.1: kwadraat.x

Eerst worden er twee structures gedefinieerd, een voor invoer en een voor uitvoer. We houden het eenvoudig in dit voorbeeld en definiëren slechts één veld voor zowel in- als uitvoer.

Vervolgens wordt een RPC programma *KWADRAAT_PROG* en versie *KWADRAAT_VERS* gedefinieerd. In die versie is er een enkele procedure *KWADRAATPROC*. Het enige argument is een structure *kwadraat_in* en de return is van het type *kwadraat_out*. Zowel programma, versie als procedure moeten een nummer krijgen. We kiezen voor de eerste twee de waarde 1, en het programma moet een 32-bit hexadecimale waarde krijgen. Voor user programma's moet deze waarde in het bereik *0x20000000* en *0x3fffffff* liggen. Dit bestand compileer je met de speciale Sun RPC compiler *rpcgen*.

19.2.2 De client

De volgende stap is het schrijven van het client programma dat de RPC aanroept.

```
#include "../wrapper.h"
#include "kwadraat.h"   /* gemaakt door rpcgen */

int main(int argc, char **argv){
  CLIENT    *cl;
  kwadraat_in  in;
  kwadraat_out  *outp;

  if (argc != 3)
    err_quit("usage: client host waarde");

  cl = Clnt_create(argv[1], KWADRAAT_PROG, KWADRAAT_VERS, "tcp");

  in.arg1 = atol(argv[2]);
  if ( (outp = kwadraatproc_1(\in, cl)) == NULL)
    err_quit(clnt_sperror(cl, argv[1]));

  printf("result: %ld\n", outp->res1);
  exit(0);
}
```

Listing 19.2: RPC client

Het begint met de *include* van het bestand *rpcgen*.

We hebben een *client handle* nodig, gedeclareerd als *CLIENT* *. Een handle wordt aangemaakt met de functie *clnt_create* :

```
#include <rpc/rpc.h>

CLIENT *clnt_create(const char *host, unsigned long prognum,
                    unsigned long versnum, const char *protocol);

Return: pointer naar client handle als OK, NULL bij fout
```

Hierbij geven we op met welk programma en welke versie daarvan we willen communiceren. We geven ook aan met welk protocol moet worden gewerkt. Je kan hier kiezen tussen “tcp” en “udp”.

Vervolgens roept de client de remote procedure aan met de functie *kwadraatproc_1*. Deze functie hebben we gedefinieerd in *rpcgen* heeft de hoofdletters omgezet naar kleine letters en het versienummer eraan toegevoegd. Het eerste argument is een *kwadraat_in* pointer en het tweede argument de client handle *cl*. De return waarde is een pointer naar *kwadraat_uit*. Merk op dat we hier zelf geen geheugen moeten alloceren voor de structure; dit gebeurt door het RPC systeem.

Het compileren van de client moet als volgt gebeuren :

```
bsd> rpcgen -C kwadraat.x
bsd> gcc -o client client.c kwadraat_clnt.c kwadraat_xdr.c
```

Indien je wrapper functies gebruikt, moet je uiteraard ook je wrapper bestand meecompileren, bijvoorbeeld

```
bsd> gcc -o client client.c kwadraat_clnt.c kwadraat_xdr.c ../wrapper.c
```

19.2.3 De server

Langs de serverzijde hoeven we enkel de RPC procedure te schrijven. *rpcgen* zorgt zelf voor een *main* functie.

```
#include "../wrapper.h"
#include "kwadraat.h"

kwadraat_out *kwadraatproc_1_svc( kwadraat_in *inp,
                                  struct svc_req *rqstp){

  static kwadraat_out out;

  out.res1 = inp->arg1 * inp->arg1;
```

```
  return(&out);
}
```

Listing 19.3: RPC server

Merk op dat de naam van de functie nu *kwadraat_proc_1_svc* is; dit is de naam uit *kwadraat.x* in kleine letters, gevolgd door het versienummer en de letters *_svc*.

De functie heeft twee argumenten; een pointer naar een *kwadraat_in* structure, gebruikt voor invoer, een een tweede structure met informatie over de aanroep, ingevuld door het RPC systeem. In dit voorbeeld negeren we het tweede argument.

De functie berekent het kwadraat en geeft die waarde terug via een pointer naar *kwadraat_out* structure. Aangezien er een *pointer* naar het resultaat moet worden teruggeven, moet je ervoor zorgen dat het geheugenadres waar de pointer naar wijst behouden blijft na het terugkeren van de functie. Die doe je door *out* als *static* te declareren. Een andere oplossing zou zijn dat je in de structure *kwadraat_in* een veld voorziet voor het resultaat.

Compilatie van de server gebeurt als volgt :

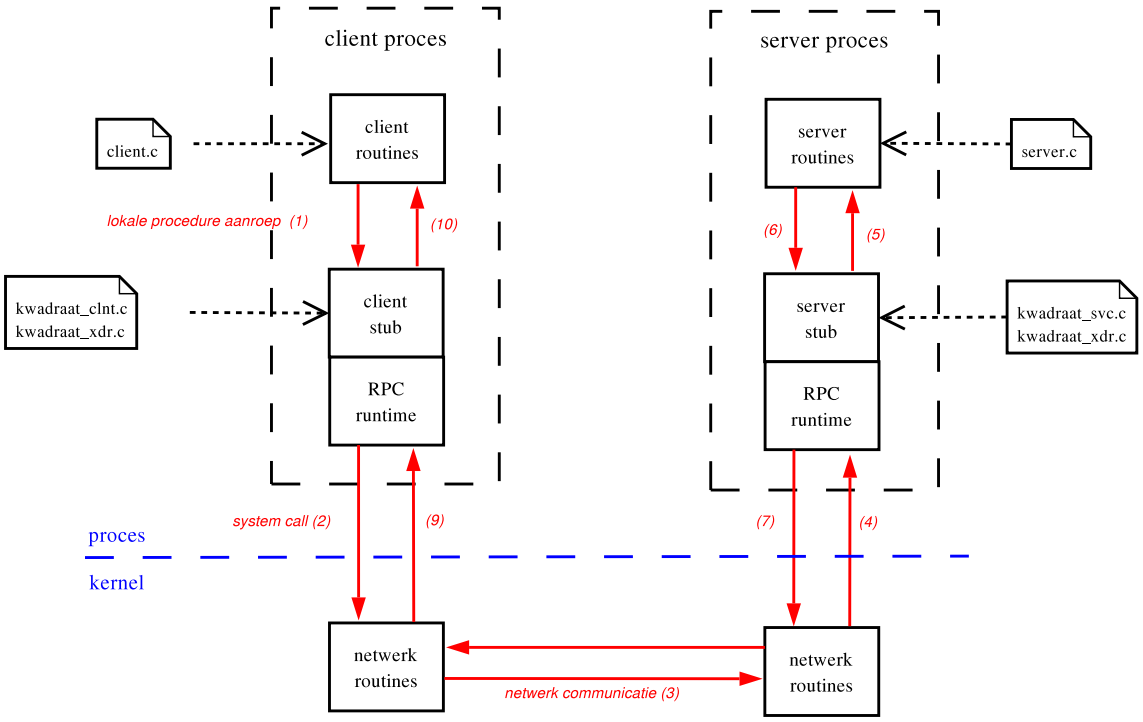
```
bsd> gcc -o server server.c kwadraat_svc.c kwadraat_xdr.c
```

19.3 Werking van de RPC runtime

In figuur 19.3 wordt schematisch weergegeven welke stappen er worden doorlopen bij het aanroepen van een remote procedure.

Volgende stappen worden uitgevoerd:

1. de client roept een lokale procedure aan : *kwadraatproc_1*. Deze procedure noemt men de *client stub*. Deze procedure werd aangemaakt door *rpcgen* en bevindt zich in het bestand *kwadraat_clnt.c*. De stub zorgt ervoor dat de argumenten aan de remote procedure correct verpakt worden in een formaat dat de RPC runtime nodig heeft. De stub roept vervolgens de nodige procedures aan in de RPC runtime.
2. De netwerk boodschappen worden door de stub naar de andere host gestuurd. Dit gebeurt via een systeem aanroep als *sendto*.
3. De netwerk boodschappen worden verstuurd via TCP of UDP.
4. Op de host van de server komen de boodschappen toe in de server stub. Deze transformeert de data weer in leesbaar formaat voor het server proces.
5. De server hub roept de procedure van de server (*kwadraat_1_svc* uit het voorbeeld) aan en geeft de argumenten mee die het ontving van de client.
6. De server stub ontvangt de return waarde van de server procedure.



Figuur 19.1: stappen van een remote procedure call

7. De netwerk boodschappen worden door de server stub terug naar de client host gestuurd.
8. De boodschappen worden teruggestuurd naar de client host via het netwerk
9. de client stub ontvangt de boodschappen van de kernel en transformeert ze in leesbaar formaat
10. de stub keert terug naar de client functie

Hoofdstuk 20

Threads

20.1 Wat zijn threads

Threads (eng.: draden) maken het mogelijk dat een proces meer dan één taak tegelijk kan uitvoeren. Een thread is in vele opzichten gelijk aan een proces; men noemt ze wel eens *lichtgewicht processen*.

Threads lopen schijnbaar tegelijkertijd; in feite zorgt de kernel ervoor dat elke thread beurtelings wat CPU tijd krijgt om enkele taken uit te voeren.

Threads bestaan binnen een proces. Wanneer je een programma opstart, maakt de kernel een initiële thread waarin het programma wordt uitgevoerd. Deze thread kan dan bijkomende threads aanmaken. Al deze threads voeren hetzelfde programma uit, binnen hetzelfde proces, maar verschillende threads kunnen op een gegeven moment verschillende delen van het programma uitvoeren.

Threads in eenzelfde proces delen alle globale variabelen. Op deze manier kan gemakkelijk informatie tussen twee threads worden uitgewisseld. Wanneer een thread een bepaalde variabele een waarde geeft, zien alle andere threads onmiddellijk de verandering.

Bestandsdescriptoren worden ook gedeeld. Dat wil zeggen dat wanneer een thread een bestandsdescriptor sluit, de andere threads er ook niet meer naar kunnen schrijven of ervan lezen.

In netwerkprogrammatie worden threads veelvuldig gebruikt. Een situatie die zich zeer goed leent tot het gebruik van threads is een server die verschillende clients tegelijk moet kunnen bedienen (bijvb. een webserver). Elke client wordt dan in een aparte thread afgehandeld. Een andere context waarin threads goed van pas komen is bij invoer/uitvoer operaties. Een programma dat invoer moet lezen van zowel toetsenbord als een socket (bijvb. een chat client), kan voor elke invoerkanaal een aparte thread opstarten die het lezen van dat kanaal verzorgt. Op die manier vermijd men blokkering; omdat de twee lees-threads simultaan lopen, moet het programma bijvoorbeeld niet eindeloos wachten op invoer van het toetsenbord zonder invoer van de socket te kunnen lezen.

We beperken ons in deze cursus tot POSIX threads, ook wel *pthreads* genoemd. Deze worden het meest gebruikt en zijn het best ondersteund.

20.2 Threads maken met *pthread_create*

Elke thread binnen een proces wordt uniek geïdentificeerd door een *thread ID*. Deze is van het type *pthread_t*.

Bij het opstarten van een thread wordt een *thread function* uitgevoerd. Dit is een normale functie die de code bevat die moet worden uitgevoerd door de thread. Wanneer de functie terugkeert, eindigt de thread.

De *pthread_create* functie maakt een nieuwe thread aan.

```
#include <pthread.h>

int pthread_create( pthread_t*      thread,
                    pthread_attr_t* attr,
                    void*           (*func)(void *),
                    void*           arg
                  );

Return: 0 als OK, positief Exxx bij error
```

Volgende parameters worden meegeven :

1. Een pointer naar een *pthread_t* variabele waarin de ID van de nieuwe thread wordt bewaard.
2. Een pointer naar een *thread attribute* object. Hierin kan je aangeven hoe de thread zich moet interageren met de rest van het programma. Wanneer je *NULL* meegeeft, worden de default waarden gebruikt.
3. Een pointer naar de thread functie die moet worden uitgevoerd in de nieuwe thread. Dit is een functiepointer van het type *void * (*)(void*)*.
4. Een thread argument van het type *void**. Hiermee kan je data doorgeven aan de nieuwe thread.

De functie *pthread_create* keert onmiddellijk terug en de originele thread gaat verder met het uitvoeren van instructies. Tegelijkertijd wordt in de nieuwe thread de threadfunctie uitgevoerd. De kernel voert beide threads asynchroon uit en je kan de volgorde waarin welke instructies uit beide threads worden uitgevoerd niet voorspellen.

In het volgende voorbeeld programma wordt een thread gemaakt die voortdurend het karakter ”1”schrijft naar het standaard foutkanaal. Na het aanmaken van de thread, wordt in de hoofdthread voortdurend het karakter ”0”naar het foutkanaal geschreven.

```
#include <pthread.h>
#include <stdio.h>

void *print_1(void *unused){
  while(1)
    fputc('1', stderr);
  return NULL;
}
```

```
int main(){
  pthread_t thread_id;
  pthread_create(&thread_id, NULL, &print_1, NULL);
  while(1)
    fputc('0', stderr);
  return 0;
}
```

Listing 20.1: Threads schrijven 1 en O

Compileer en link dit programma als volgt :

```
% gcc -o thread-create thread-create.c -lpthread
```

Het laatste argument aan *gcc* geeft aan dat de bibliotheek *pthread* moet worden meegelinkt.

Voer het programma uit en bekijk het resultaat. Merk op dat de 1 en 0 karakters onregelmatig afwisselend naar het scherm worden geschreven.

Normaal eindigt een thread op twee mogelijke manieren; ofwel wanneer de threadfunctie terugkeert ofwel wanneer de thread de functie *pthread_exit* aanroept. Hier kan de thread een return-waarde meegeven, zodat een andere thread data kan terugkrijgen. Meer hierover vind je in de sectie 20.5.

Wanneer een thread binnen een proces de functie *exit* aanroept, stoppen *alle* threads binnen dat proces. Wanneer de hoofd thread stopt (doorgaans wanneer de *main* functie terugkeert) stoppen ook alle threads binnen het proces.

20.3 Argumenten doorgeven aan threads

Een eenvoudige manier om data door te geven aan een thread is via globale variabelen. Alle threads binnen een proces hebben toegang tot dezelfde globale variabelen. Een nadeel van deze methode is dat voor elke thread eventueel een aparte globale variabele moet worden gedeclareerd.

Een elegantere oplossing gaat via de laatste parameter van de *pthread_create* functie. Hiermee kan men een argument doorgeven aan de threadfunctie. Je kan slechts één argument doorgeven en dit is van het type *void**. Wanneer er meer dan één waarde moet worden doorgegeven, moet je een structure of array meegeven.

Doorgaans wordt voor elke threadfunctie een structure gedefiniëerd die de parameters bevat die de functie nodig heeft. De functie moet dan beginnen met een *void* * argument naar het juiste type. Gebruik hiervoor een pointer.

Het programma in listing 20.2 is een uitbreiding van het vorige; er worden nu twee threads opgestart en aan elke wordt andere data meegegeven via de *struct print_char_parms*.

```
#include <pthread.h>
#include <stdio.h>
```

```
struct print_char_parms{
  char karakter; // het karakter dat geprint wordt
  int aantal;    // aantal keer dat het karakter geprint wordt
};

void *print_char(void *parameters){
  struct print_char_parms *p = (struct print_char_parms*)parameters;
  int i;
  for (i=0; i<p->aantal; i++)
    fputc(p->karakter, stderr);
  return NULL;
}

int main(){
  pthread_t thread1_id, thread2_id;
  struct print_char_parms thread1_args, thread2_args;

  thread1_args.karakter='x';
  thread1_args.aantal = 30000;
  pthread_create(&thread1_id, NULL, &print_char, &thread1_args);

  thread2_args.karakter='y';
  thread2_args.aantal = 20000;
  pthread_create(&thread2_id, NULL, &print_char, &thread2_args);

  return 0;
}
```

Listing 20.2: Twee schrijvende threads

Wanneer je dit programma uitvoert merk je dat het niet naar behoren werkt; er wordt noch 30000 maal x noch 20000 maal y afgedrukt. De verklaring is dat de hoofd thread stopt meteen na de creatie van de twee threads. Hierdoor worden de twee threads ook gestopt.

Een oplossing voor dit probleem is dat de hoofd thread moet wachten tot de twee andere threads klaar zijn.

20.4 Wachten op het einde : *pthread_join*

Je kan wachten op het einde van een thread door aanroep van de functie *pthread_join*:

```
#include <pthread.h>

int pthread_join(pthread_t thread, void **thread_return);

Return: 0 als OK, positief Exxx bij error
```

Je specifieert de ID van de thread waarop je wil wachten in het eerste argument. De return waarde van de thread wordt bewaard in *thread_return*. Indien je geen return waarde wil ontvangen, kan je voor het tweede argument *NULL* meegeven.

De *main* functie uit listing 20.2 kunnen we als volgt herschrijven:

```
int main(){
```

```
  pthread_t thread1_id, thread2_id;
  struct print_char_parms thread1_args, thread2_args;

  thread1_args.karakter='x';
  thread1_args.aantal = 30000;
  pthread_create(&thread1_id, NULL, &print_char, &thread1_args);

  thread2_args.karakter='y';
  thread2_args.aantal = 20000;
  pthread_create(&thread2_id, NULL, &print_char, &thread2_args);

  // wacht op beide threads
  pthread_join(thread1_id, NULL);
  pthread_join(thread2_id, NULL);

  return 0;
}
```

Listing 20.3: wachten met *pthread_join*

Let er dus steeds op dat alle data die je via een pointer doorgeeft aan een thread niet wordt vernietigd (door deallocatie of *free*) voor de thread klaar is.

20.5 Thread return waarden

Er zijn een aantal manieren om waarden terug te krijgen van een thread. De ene is al wat 'properder' dan de andere.

20.5.1 Teruggeven van kleine waarden (int, char, ...)

Als je een return waarde wil ontvangen van een thread, doe je dat door een pointer argument mee te geven aan *pthread_join*. De return waarde van de thread functie zal dan op die plaats worden bewaard. De return waarde is van het type *void* *. De thread functie moet dus de return waarde casten naar een *void* * en de hoofd thread kan deze waarde dan weer casten naar het geschikte type.

Het volgende programma start een thread die het kwadraat berekent van het meegestuurde argument en geeft die waarde terug.

```
#include <pthread.h>
#include <stdio.h>

void *kwadraat(void *parameter){
  int x = *((int *)parameter);
  int resultaat;
  resultaat = x * x;
  return (void *)resultaat;
}

int main(){
  pthread_t thread_id;
  int getal = 7;
  int *resultaat;
```

```
  pthread_create(&thread_id, NULL, &kwadraat, &getal);

  pthread_join(thread_id, (void **)&resultaat);

  printf("%d\n", resultaat);
  return 0;
}
```

Listing 20.4: return-waarde van een thread in *pthread_join*

Deze manier werkt enkel wanneer de waarde die de threadfunctie binnen vier bytes past, omdat de waarde naar een *pointer variabele* wordt geschreven. Zodra je een *double* (8 bytes) of meer moet teruggeven, werkt deze methode niet meer. Een pointer naar een *double* is zelf immers slechts 4 bytes groot.

20.5.2 Uitvoer naar het argument

Een andere manier om uitvoer van een thread te krijgen is dat je in de thread functie via de pointer naar het argument de terug te geven waarde bewaart op diezelfde plaats, zoals in het volgende voorbeeld.

```
#include <pthread.h>
#include <stdio.h>

void *tf(void *parameter){
  double *p = ((double *)parameter);
  *p=*p * *p;
  return (NULL);
}

int main(){
  pthread_t thread_id;
  double getal = 7.0;

  pthread_create(&thread_id, NULL, &tf, &getal);
  pthread_join(thread_id, NULL);
  printf("resultaat=%e\n", getal);

  return 0;
}
```

Listing 20.5: Thread met uitvoer naar argument

Het nadeel van die methode is dat je de oorspronkelijke waarde van de variabele kwijtspeelt.

20.5.3 Een pointer naar een element van een tabel als uitvoer

Indien je een threadfunctie hebt die als invoer een array krijgt en als uitvoer een element uit die array, bijvoorbeeld het element met de grootste waarde, dan is volgende manier van werken aangewezen :

```
#include <pthread.h>
#include <stdio.h>
```

```
void *zoek_max(void *parameter){
  int *p = (int *)parameter;
  int *max_index = p;

  while (*p!=0){
    if(*p > *max_index)
      max_index = p;
    p++;
  }
  return (void *)max_index;
}

int main(){
  pthread_t thread_id;
  int getallen[] = {1,5,8,4,12,6,0};
  int *maxp;

  pthread_create(&thread_id, NULL, &zoek_max, getallen);
  pthread_join(thread_id, (void **)&maxp);
  printf("resultaat=%d\n", *maxp);
  return 0;
}
```

Listing 20.6: Thread met array element als uitvoer

20.6 Threads beëindigen

Een thread eindigt in volgende gevallen :

- de threadfunctie terugkeert
- de *main* functie van het proces terugkeert
- een van de threads binnen het proces *exit* aanroept. In dit geval stoppen alle threads van het proces.
- binnen de thread de functie *pthread_exit* wordt aanroepen
- een andere thread de thread beëindigt met de functie *pthread_cancel*

Een thread kan zichzelf beëindigen met de functie *pthread_exit*.

```
#include <pthread.h>

void pthread_exit( void *status );

Return: keert niet terug naar aanroeper
```

Het argument *status* is de return-waarde van de thread en kan nadien worden opgevraagd in de *pthread_join* functie.

Om een thread te annuleren, gebruik je de functie *pthread_cancel*.

```
#include <pthread.h>

int pthread_cancel( pthread_t thread );

Return: 0 als OK, positief Exxx code bij error
```

De enige parameter is de thread ID van de thread die je wil laten stoppen. Een geannuleerde thread kan en moet later ge*join*ed worden om de gebruikte bronnen vrij te geven. De waarde die een geannuleerde thread teruggeeft aan een eventuele aanroep van *pthread_join* is een speciale waarde : *PTHREAD_CANCELED*.

Een thread kan zelf bepalen wanneer hij kan worden geannuleerd. Dit is nodig om ervoor te zorgen dat gebruikte bronnen (bijvb. gealloceerd geheugen) weer kunnen worden vrijgegeven. Deze controle gebeurt aan de hand van drie toestanden waarin een thread zichzelf kan plaatsen.

1. *asynchronously cancelable* : de thread kan op elke punt van zijn uitvoering worden geannuleerd.
2. *synchronously cancelable* : de thread kan slechts op bepaalde punten (*cancel points*) van zijn uitvoering worden geannuleerd. Annulatieaanvragen worden in een wachtrij gestopt en behandeld wanneer de thread in zijn uitvoering aan zo'n punt belandt. De precieze punten waarop een thread kan geannuleerd worden zijn afhankelijk van de kernel. Typische punten zijn aanroepen van functies als *read* en *write*.
3. een thread kan ook *uncancelable* zijn; elke aanvraag tot annulatie van de thread wordt genegeerd.

Wanneer een thread wordt aangemaakt, bevindt hij zich standaard in de *synchronously cancelable* toestand.

Om de toestand van de thread te veranderen, gebruik je de functie *pthread_setcanceltype*.

```
#include <pthread.h>

int pthread_setcanceltype( int type, int *oldtype);

Return: 0 als OK, positief Exxx code bij error
```

Het eerste argument geeft de toestand aan waarin de thread moet terechtkomen. Volgende twee toestanden zijn mogelijk:

1. PTHREAD_CANCEL_ASYNCHRONOUS : asynchroon annuleerbaar
2. PTHREAD_CANCEL_DEFERRED : synchroon annuleerbaar, *default*

Het vorige type waarin de thread zich bevond wordt opgeslagen op de plaats waar de tweede parameter naar wijst. Geef hier *NULL* als je deze oude waarde niet moet weten.

Je kan zelf *cancel points* toevoegen door de functie *pthread_testcancel*. Deze functie doet niets behalve checken of er een annulatieaanvraag is binnengekomen. Indien zo wordt de thread beëindigd. Plaats deze functieaanroep op regelmatige, veilige plaatsen in de thread.

Om een thread *uncancelable* te maken gebruik je de functie *pthread_setcancelstate*

```
#include <pthread.h>

int pthread_setcancelstate( int state, int *oldstate);

Return: 0 als OK, positief Exxx code bij error
```

waarbij *state* aangeeft in welke toestand je de thread wil brengen : *PTHREAD_CANCEL_ENABLE* of *PTHREAD_CANCEL_DISABLE*.

Stel dat je een thread hebt die een kritische handeling moet uitvoeren, bijvoorbeeld het overschrijven van een geldbedrag van een rekening naar een andere. Tijdens de uitvoering van de overschrijving mag de thread in geen geval geannuleerd worden. Dit kan je dan als volgt implementeren :

```
#include <pthread.h>

int transactie(float rekeningen[ ], int van_rekening,
               int naar_rekening, int bedrag){
  int old_state;
  if (rekeningen[van_rekening] < bedrag){
    return 1; // fout ! niet genoeg geld op rekening
  }
  // begin kritische sectie
  pthread_setcancelstate(PTHREAD_CANCEL_DISABLE, &old_state);
  rekeningen[van_rekening] -= bedrag;
  rekeningen[naar_rekening] += bedrag;
  // einde kritische sectie
  pthread_setcancelstate(old_state, NULL);

  return 0;
}
```

Listing 20.7: Kritische transactie beveiligen in een thread

20.6.1 De return waarde van een gecancelde thread

Wanneer een thread wordt geannuleerd dan is de return waarde (de waarde die wordt opgevangen in de tweede parameter van *pthread_join*) de constante *PTHREAD_CANCELED*. Indien je threads kunnen worden geannuleerd, dan moet je hier op controleren. Volgend programma start een thread en annuleert hem direct erna. Merk op dat ook hier de variabele *getal* dient als uitvoerparameter van de thread functie.

```
#include <pthread.h>
#include <stdio.h>

void *tf(void *parameter){
  double *p = ((double *)parameter);
  *p=*p * *p;
```

```
  read(0,NULL,100); // blokkeer zodat thread kan worden geannuleerd
  return (void *)p;
}

int main(){
  pthread_t thread_id;
  double getal = 7.0;
  int * dp;

  pthread_create(&thread_id, NULL, &tf, &getal);
  pthread_cancel(thread_id);
  pthread_join(thread_id, (void **)&dp);

  if(dp==PTHREAD_CANCELED)
    printf("thread canceled\n");
  else
    printf("resultaat=%d\n", *dp);

  return 0;
}
```

Listing 20.8: Return van een geannuleerde thread

20.7 Andere thread functies

Hier volgen nog enkele functies die sporadisch van pas kunnen komen.

```
#include <pthread.h>

int pthread_self(void);

Return: thread ID van aanroepende thread.
```

Met deze functie kan je het ID van de eigen thread bekomen. Soms handig bij het gebruik van thread functies.

```
#include <pthread.h>

int pthread_detach(pthread_t thread_id);

Return: 0 als OK, Exxx waarde bij fout
```

Een thread is ofwel *joinable* ofwel *detached*. Wanneer een *joinable* thread eindigt, worden zijn ID en exit status bewaard tot een andere thread *pthread_join* aanroept. Een detached thread gedraagt zich anders: wanneer die eindigt worden alle bronnen vrijgemaakt en er kan niet op gewacht worden. Wanneer een thread moet weten wanneer een andere thread eindigt, moet je deze laatste joinable laten.

De functie wordt dikwijls als volgt aangeroepen :

```
pthread_detach(pthread_self());
```

20.8 Voorbeelden van threads in netwerkprogrammatie

Hierna volgen enkele scenario's waarin threads goed van pas komen.

20.8.1 echo server met meerdere clients

Het volgende programma is een echo server die een willekeurig aantal clients aankan. Voor elke nieuwe connectie die binnenkomt wordt een thread opgestart. De socket descriptor wordt als argument meegegeven.

```
void * verwerk_client(void *arg){
  char buffer[MAXLINE];
  int n, fd;
  fd = *((int *)arg);
  Pthread_detach(pthread_self()); // wacht niet op mij
  n = recv(fd, buffer, MAXLINE, 0);
  while( n > 0){
    send(fd, buffer, n, 0);
    n = recv(fd, buffer, MAXLINE, 0);
  }
  close(fd);
  return (NULL);
}

int main(){
  int listenfd, connfd;
  socklen_t len;
  struct sockaddr_in cliaddr;
  pthread_t id;
  listenfd = Tcp_listener(ECHO_PORT);
  while(1){
    len = sizeof(cliaddr);
    connfd = Accept(listenfd, (SA *) &cliaddr, &len);
    Pthread_create(&id, NULL, &verwerk_client, (void *) &connfd);
  }
  return 0;
}
```

Listing 20.9: Echo server met threads

20.8.2 echo client met aparte thread voor lezen en schrijven

Volgend programma is een echo client waarbij het lezen van het toetsenbord en schrijven naar de socket in een thread gebeurt, terwijl het lezen van de socket en schrijven naar het scherm in een tweede plaatsvindt.

```
char SERVERSTRING = "127.0.0.1";

pthread_t thread_socket, thread_toetsenbord;
int lees_toetsenbord_actief = 0;

void * lees_socket(void *arg){
  char buffer[MAXLINE];
```

```
  int n, fd;
  fd = *((int *)arg);
  n = recv(fd, buffer, MAXLINE, 0);
  while( n > 0){
    printf("%s", buffer);
    n = recv(fd, buffer, MAXLINE, 0);
  }
  // server sluit de connectie -> sluit socket
  close(fd);

  // sluit de andere thread, die geblokkeerd zit in toetsenbord-read
  if (lees_toetsenbord_actief==1)
    pthread_cancel(thread_toetsenbord);
  else
    printf("lees_toetsenbord sluit zelf af\n");

  printf("lees_socket sluit af\n");
  return (NULL);
}

void * lees_toetsenbord(void *arg){
  char buffer[MAXLINE];
  int n, fd;
  lees_toetsenbord_actief = 1;
  fd = *((int *)arg);
  n = read(0, buffer, MAXLINE);
  while( n > 0){
    send(fd, buffer, n, 0);
    n = read(0, buffer, MAXLINE);
  }
  lees_toetsenbord_actief = 0;
  shutdown(fd, SHUT_WR);

  printf("lees_toetsenbord sluit af\n");

  return (NULL);
}

int main(){
  int n, connfd, *p;
  connfd = Tcp_client(ECHO_PORT, SERVERSTRING);

  Pthread_create(&thread_socket, NULL,
                 &lees_socket, (void *)&connfd);

  Pthread_create(&thread_toetsenbord, NULL,
                 &lees_toetsenbord, (void *)&connfd);

  Pthread_join(thread_socket, NULL);
  n = Pthread_join(thread_toetsenbord, (void **)&p);

  if(p==PTHREAD_CANCELED)
    printf("server sloot connectie : thread canceled\n");
  return 0;
}
```

Listing 20.10: Echo client met threads

Deze echoclient bevat een interessante manier van afsluiten van threads; wanneer het toetsenbord EOF doorgeeft, sluit de *lees_toetsenbord* thread de socket descriptor voor schrijven met de *shutdown* functie. De server ontvangt een *FIN* en sluit op zijn beurt de socket. Hierdoor ontvangt de *lees_socket* thread een EOF en sluit zichzelf. Het voordeel is dat de *lees_socket* thread eventueel hangende data kan ontvangen van de server en ook de *lees_toetsenbord* thread is afgesloten.

Hoofdstuk 21

Multiplexing I/O

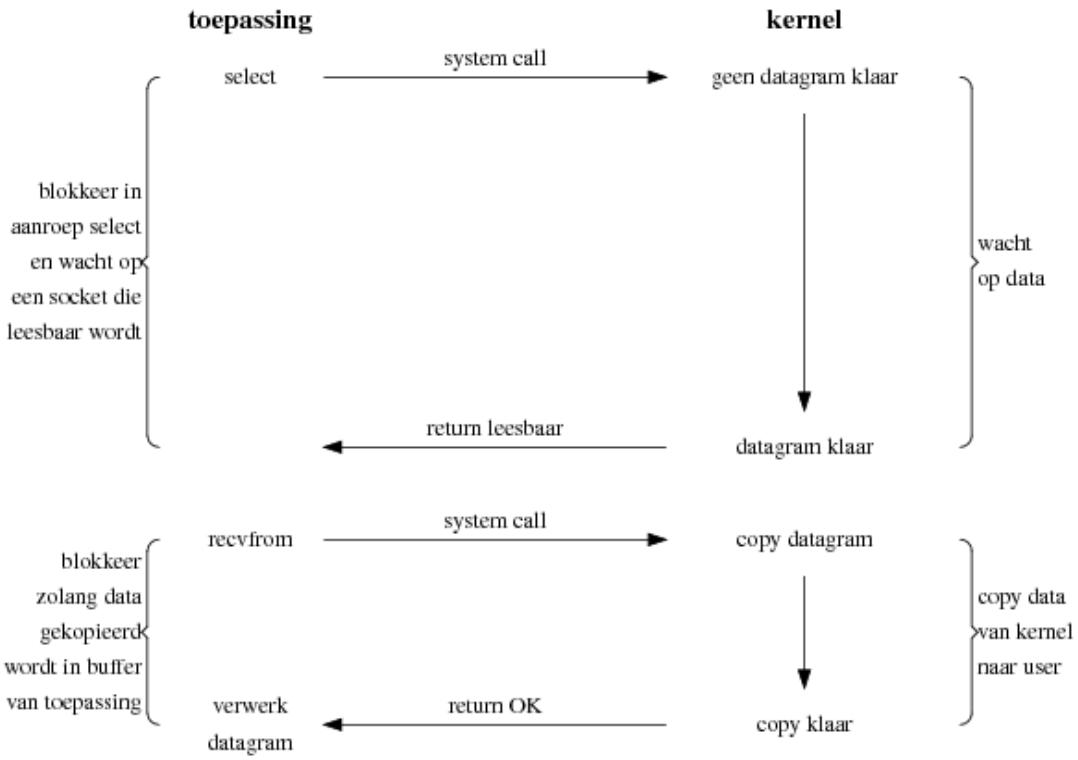
21.1 Inleiding

In een TCP applicatie waarbij de client invoer ontvangt van zowel het toetsenbord als de server, kan volgend probleem ontstaan. Terwijl de client wacht op invoer van de gebruiker (bijvb. door het aanroepen van de functie *readline*), is de client *blocked* of geblokkeerd. Stel nu dat het server proces wordt beëindigd (bijvb. door een netwerk administrator of een systeemfout). De server TCP stuurt dan een FIN naar de client, maar de client zal deze *end connection request* pas lezen wanneer de functie *readline* terugkeert (wanneer de gebruiker op *enter* drukt) en dit kan lange tijd duren. Ondertussen kan de TCP connectie niet worden afgesloten. Dit probleem is opgelost wanneer we een systeem zouden hebben waarbij de kernel tegen het client proces zegt wanneer een of meer I/O operaties mogelijk zijn; het lezen van of het schrijven naar een descriptor. Op die manier wordt vermeden dat het proces in een *blocked status* terecht komt of dat de periode waarin de client geblokkeerd is wordt geminimaliseerd. Dit systeem heet *I/O multiplexing* en wordt geïmplementeerd door de functie *select*.

I/O multiplexing wordt in volgende situaties gebruikt :

- wanneer een proces verschillende descriptoren gebruikt (bijvb. interactieve invoer en een netwerk-socket)
- wanneer een client verschillende sockets tegelijk gebruikt (bijvb. Web clients die meerdere verbindingen maken met één of meerdere servers)
- wanneer een server zowel de listen als de connection socket afhandelt
- wanneer een server verschillende diensten vervult en/of protocollen hanteert, bijvb. de *inetd super server* (zie hoofdstuk 25).

In figuur 21.1 zie je een samenvatting van wat er gebeurt in het multiplexing model. De applicatie blokkeert in een aanroep van *select* en niet in een I/O-operatie.



Figuur 21.1: I/O multiplexing model

21.2 De functie *select*

Deze functie zegt tegen de kernel het proces in slaapmodus te plaatsen en het pas op te wekken wanneer een of andere I/O operatie (lezen of schrijven) mogelijk is. Deze functie is niet enkel zinvol bij netwerkprogrammatie maar bij alle applicaties die met verschillende descriptoren werken.

De syntax van *select* :

```
#include <sys/select.h>
#include <sys/time.h>

int select ( int maxfdp1,
             fd_set *readset,
             fd_set *writeset,
             fd_set *exceptset,
             struct timeval *timeout);

Return : aantal descriptoren die klaar zijn, 0 als timeout, -1 bij fout.
```

De functie keert terug wanneer een van de volgende gebeurtenissen optreedt :

- er kan worden gelezen van een of meer descriptoren

- er kan worden geschreven naar een of meer descriptoren
- er treedt een uitzondering op bij een of meerdere descriptoren
- een bepaalde tijd is verstreken zonder dat er iets gebeurde

Parameters :

- maxfdp1 : het aantal descriptoren dat moet worden getest. De waarde is de waarde van de hoogste descriptor + 1. De reden hiervoor is dat in C de eerste descriptor gelijk is aan 0. Wanneer de hoogste descriptor de waarde 3 heeft, moeten er dus 4 descriptoren, beginnend bij 0, worden getest.
- readset : de verzameling descriptoren die moeten worden getest op lezen (zie lager)
- writeset : de verzameling descriptoren die moeten worden getest op schrijven (zie lager)
- exceptset : de verzameling descriptoren die moeten worden getest op uitzonderingen Er zijn twee mogelijke uitzonderingen waarop kan worden getest :
  - out-of-band data
  - status-informatie van een pseudo-terminal
- timeout : hierin specifieer je hoelang *select* moet wachten op een gebeurtenis. Hiervoor maak je gebruik van de timeval structure :

```
struct timeval{
  long tv_sec;  /* seconden*/
  long tv_usec; /* microseconden */
};
```

Voor het specifiëren van de verzamelingen descriptoren, kan je volgende macro's uit *select.h* gebruiken :

```
void FD_ZERO(fd_set *fdset); /* ledig de set*/
void FD_SET(int fd, fd_set *fdset) /* voeg toe */
void FD_CLR(int fd, fd_set *fdset) /* verwijder  */
bool FD_ISSET(int fd, fd_set *fdset) /* fd in set?*//
```

Wanneer de functie terugkeert, zijn de verzamelingen readset, writeset en exceptset gevuld met die descriptoren waar resp. gelezen, geschreven worden of een fout optrad. Hieronder zie je een voorbeeld van een echo client die gebruik maakt van I/O multiplexing:

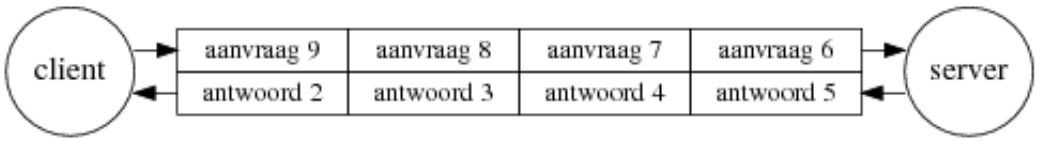
```
void verwerk_klant(int invoer_fd, int sockfd){
  bool einde = false;
  int maxfdp1;
  fd_set rset, rset_all;
  char sendline[MAXLINE], recvline[MAXLINE];

  FD_ZERO(&rset_all);
  FD_SET(invoer_fd, &rset_all);
  FD_SET(sockfd, &rset_all);
```

```
  while (!einde) {
    rset = rset_all;
    maxfdp1 = max(invoer_fd, sockfd) + 1;
    Select(maxfdp1, &rset, NULL, NULL, NULL);

    if (FD_ISSET(sockfd, &rset)) { /* socket leesbaar */
      if (Readline(sockfd, recvline, MAXLINE) == 0)
        einde=true; /*keer terug*/
      else
        printf("%s",recvline);
    }

    if (FD_ISSET(invoer_fd, &rset)){ /* invoer leesbaar */
      if (Readline(invoer_fd, sendline, MAXLINE) == 0)
        FD_CLR(invoer_fd, &rset_all); /* klaar */
      else
        Writestr(sockfd, sendline);
    }
  }
}
```

Listing 21.1: echo client met *select*

21.3 De functie *shutdown*

Aangezien TCP en UDP connecties full-duplex zijn (er kan tegelijk geschreven en gelezen worden), kan volgende situatie ontstaan : nadat de client een FIN heeft gestuurd naar de server (door het aanroepen van de functie *close*) met de bedoeling de TCP connectie te sluiten, is de socket van de client onbruikbaar geworden; er kan niet meer van worden gelezen, noch geschreven. Maar het is mogelijk dat er nog pakketten van de server moeten worden gelezen. In figuur 21.2 zie je een voorbeeld van een echo client die 9 pakketten (lijnen tekst) heeft gestuurd en vervolgens de connectie wil verbreken, niettegenstaande het nog maar 1 pakket van de server heeft teruggekregen; de rest zweeft nog ergens rond op het netwerk.



Figuur 21.2: De pijplijn tussen client en server

Wanneer de client, na het sturen van de negen paketten, de socket zou sluiten, dan kan er ook niet meer van gelezen worden. Acht antwoord-paketten van de server zouden dan verloren gaan. De oplossing ligt in het gebruik van de functie *shutdown* :

```
#include <sys/socket.h>

int shutdown(int sockfd, int howto);
```

waarbij *howto* volgende waarden kan hebben :

- SHUT_RD : sluit de connectie voor lezen, niet voor schrijven
- SHUT_WR : sluit de connectie voor schrijven, nietvoor lezen
- SHUT_RDWR : sluit de connectie volledig

Een uiteindelijke echo client zie je hieronder.

```
#include    "netwerk.h"

void str_cli(int in_fd, int sockfd){
  bool einde=false;
  int maxfdp1;
  fd_set rset, rset_all;
  char sendline[MAXLINE], recvline[MAXLINE];

  FD_ZERO(&rset_all);
  FD_SET(in_fd, &rset_all);
  FD_SET(sockfd, &rset_all);
  maxfdp1 = max(in_fd, sockfd) + 1;

  while (!einde){
    rset = rset_all;

    Select(maxfdp1, &rset, NULL, NULL, NULL);

    if (FD_ISSET(sockfd, &rset)) { /* socket leesbaar */
      if (Readline(sockfd, recvline, MAXLINE) == 0) {
        einde = true; /* normaal einde */
      else
        printf(``%s'', recvline);
    }

    if (FD_ISSET(in_fd, &rset)) {  /* input is readable */
      if (Readline(0, sendline, MAXLINE)==0) {
        shutdown(sockfd, SHUT_WR);
        FD_CLR(in_fd, &rset);
      }
      else
        Writestr(sockfd, sendline);
    }
  }
}
```

Listing 21.2: echo client met *select* en *shutdown*

Hoofdstuk 22

Nonblocking IO

Standaard zijn sockets blocking; als je een actie wil uitvoeren die niet meteen kan worden gestart, blokkeert het proces (slaapmode) en wacht tot de actie wel kan worden ondernomen. Er zijn vier groepen socketaanroepen die kunnen blokkeren:

- invoer operaties: *recv*, *recvfrom* een aanverwanten. Wanneer je deze functies aanroept en er is geen data aanwezig in de socket ontvangstbuffer, wordt het proces in slaap gezet tot er data arriveert.

  Bij non-blocking sockets wordt onmiddellijk teruggekeerd uit de aanroep met een *EWOULDBLOCK* fout.

- uitvoer operaties: *send*, *sendto* enz. Wanneer een applicaties schrijft naar een socket, kopieert de kernel de data naar de socket zendbuffer. Wanneer deze buffer vol is, blokkeert het proces tot er ruimte vrijkomt in de buffer.

  Met een nonblocking socket zal de functie onmiddellijk terugkeren wanneer de buffer vol is. Ook hier met de error *EWOULDBLOCK*. Indien er wel nog plaats was in de buffer maar niet genoeg om alles te kopiëren, is het resultaat van de functie het aantal bytes dat de kernel kon kopiëren.

- wachten op inkomende verbindingen: *accept*. Het proces slaapt tot er een verbinding wordt aangevraagd.

  Nonblocking sockets keren onmiddellijk terug met de *EWOULDBLOCK* fout.

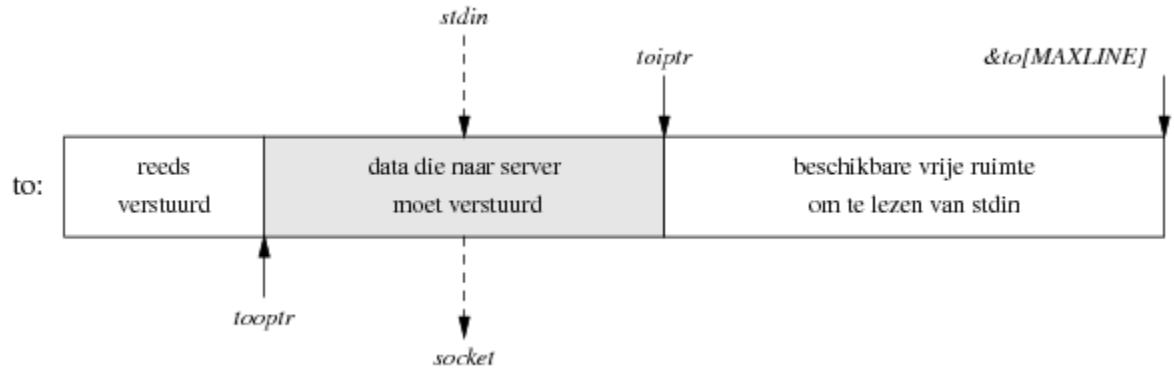
- maken van verbindingen: *connect*. Wegens de three-way handshake is er op z'n minst een periode round-trip-time (RTT) dat de functie blokkeert omdat de functie niet terugkeer voor ze een ACK ontvangt van de server.

  Wanneer je een connect uitvoert op een nonblocking socket en de verbinding kan niet direct worden gemaakt, wordt het eerste pakket (SYN) gestuurd en keert de functie terug met de melding *EINPROGRESS*.

22.1 Nonblocking client

Het is niet zo eenvoudig om efficiënte nonblocking applicaties te schrijven. Het volstaat immers niet de optie "NONBLOCK" in te stellen voor de socket die de netwerkverbinding verzorgt. In het volgende programma gebruiken we een veel geziene techniek die gebruik maakt van twee buffers (op applicatie-niveau); een lees- en een schrijfbuffer *to* en *fr*. Verder gebruiken we *select* om uit te zoeken welke operaties kunnen worden uitgevoerd. Het programma is de echo-client die reeds in vorige hoofdstukken voorkwam.

De twee buffers en hun functionaliteit is als volgt :



Figuur 22.1: Buffer met data van de standaard input die naar de socket gaat

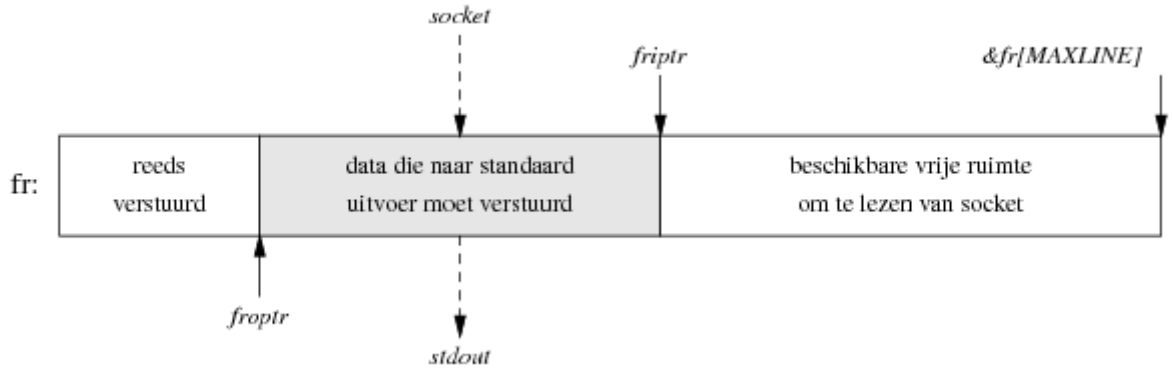
De pointer *toiptr* wijst naar de volgende byte waar data kan bewaard worden die gelezen wordt van de standaard invoer. De *tooptr* wijst naar de volgende byte die naar de socket moet worden geschreven. Het aantal bytes dat kan gelezen worden van de invoer is *&to[MAXLINE]* min *toiptr*[1]. Van zodra *tooptr* gelijk wordt aan *toiptr*, worden beide pointers weer naar het begin van de buffer geplaatst.

In figuur 22.2 zie je een analoge opzet voor de *fr* buffer.

We bekijken de code van een typische nonblocking applicatie, in dit geval een echo client :

```
1  #include "netwerk.h"
2  #include "wrapper.h"
3  #define MAXLINE 10 // speciaal kleine waarde om te kunnen testen
4  void doNonBlock(int sockfd){
5    int maxfdp1, val, stdineof;
6    int n, nwritten;
7    fd_set rset, wset;
8    char to[MAXLINE], fr[MAXLINE];
9    char *toiptr, *tooptr, *friptr, *froptr;
10
11   /* maak socket, toetsenbord en scherm nonblocking */
12   val = Fcntl(sockfd, F_GETFL, 0);
```

[1] In C geeft het verschil van twee pointers die naar elementen van eenzelfde tabel wijzen immers de afstand tussen die twee pointers



Figuur 22.2: Buffer met data van de socket die naar stdout gaat

```
13  Fcntl(sockfd, F_SETFL, val | O_NONBLOCK);
14
15  val = Fcntl(STDIN_FILENO, F_GETFL, 0);
16  Fcntl(STDIN_FILENO, F_SETFL, val | O_NONBLOCK);
17
18  val = Fcntl(STDOUT_FILENO, F_GETFL, 0);
19  Fcntl(STDOUT_FILENO, F_SETFL, val | O_NONBLOCK);
20
21  toiptr = tooptr = to; /* init buffer pointers */
22  friptr = froptr = fr;
23  stdineof = 0;
24
25  maxfdp1 = max(max(STDIN_FILENO,STDOUT_FILENO),sockfd)+1;
26
27  while ( 1 ) {
28    /* plaats gepaste descriptoren in sets */
29    FD_ZERO(&rset);
30    FD_ZERO(&wset);
31    if (stdineof == 0 && toiptr < &to[MAXLINE])
32      FD_SET(STDIN_FILENO, &rset); /* lees stdin */
33    if (friptr < &fr[MAXLINE])
34      FD_SET(sockfd, &rset);       /* lees socket */
35    if (tooptr != toiptr)
36      FD_SET(sockfd, &wset);       /* schrijf socket */
37    if (froptr != friptr)
38      FD_SET(STDOUT_FILENO, &wset);/* schrijf stdout */
39
40    select(maxfdp1, &rset, &wset, NULL, NULL);
41
42    /* lees van toetsenbord ? */
43    if (FD_ISSET(STDIN_FILENO, &rset)) {
44      n = read( STDIN_FILENO, toiptr, &to[MAXLINE] - toiptr);
45      if ( n < 0 && errno != EWOULDBLOCK){
46          perror("leesfout op stdin");
47      } else if (n == 0) {
48        stdineof = 1;   /* klaar met stdin */
49        if (tooptr == to) /* alles verstuurd */
50          Shutdown(sockfd, SHUT_WR);/* stuur FIN */
51      } else if (n > 0){
52          toiptr += n;   /* net gelezen */
53          FD_SET(sockfd, &wset); /* probeer socket schrijven */
54      }
55    }
56
```

```
57      /* lees van socket ? */
58      if (FD_ISSET(sockfd, &rset)) {
59        n = recv(sockfd, friptr, &fr[MAXLINE] - friptr, 0);
60        if ( n < 0 && errno != EWOULDBLOCK) {
61            perror("leesfout op socket");
62        } else if (n == 0) {
63            if (stdineof)
64              return;  /* normale beeindiging */
65            else
66              ExitWithMessage("str_cli: server onverwacht gestopt");
67        } else if (n>0){
68          friptr += n;  /* net gelezen */
69          FD_SET(STDOUT_FILENO, &wset); /* probeer scherm te schrijven */
70        }
71      }
72
73      /* schrijf naar scherm ? */
74      if ( FD_ISSET(STDOUT_FILENO, &wset) && ( (n = friptr - froptr) > 0)) {
75        nwritten = write(STDOUT_FILENO, froptr, n);
76        if (nwritten < 0 && errno != EWOULDBLOCK){
77          perror("schrijffout stdout");
78        } else if (nwritten>0){
79          froptr += nwritten;  /* net geschreven */
80          if (froptr == friptr)
81            froptr = friptr = fr; /* naar begin buffer */
82        }
83      }
84
85      /* schrijf naar socket ? */
86      if ( FD_ISSET(sockfd, &wset) && ( (n = toiptr - tooptr) > 0)) {
87        nwritten = send(sockfd, tooptr, n, 0);
88        if ( nwritten < 0 && errno != EWOULDBLOCK) {
89          perror("socket schrijffout");
90        } else if (nwritten>0){
91          tooptr += nwritten; /* net geschreven */
92          if (tooptr == toiptr) {
93            toiptr = tooptr = to; /* naar begin buffer */
94            if (stdineof)
95              Shutdown(sockfd, SHUT_WR); /* stuur FIN */
96          }
97        }
98      }
99    }
100  }
101
102
103 int main(int argc, char **argv){
104   int sockfd;
105   struct sockaddr_in servaddr;
106
107   if (argc != 2){
108     printf("Gebruik: NonBlocking <IPaddress>\n");
109     exit(1);
110   }
111
112   /* maak TCP socket */
113   if( (sockfd = socket(PF_INET, SOCK_STREAM, IPPROTO_TCP)) < 0)
114     ExitWithMessage("socket() failed");
115
116   /* vul server adres structure in */
117   memset(&servaddr, 0, sizeof(servaddr));        /* maak struct leeg */
118   servaddr.sin_family = AF_INET;                 /* Internet adres */
119   servaddr.sin_addr.s_addr = inet_addr(argv[1]); /* server adres */
120   servaddr.sin_port = htons(7);                  /* echo poort */
121
122   /* maak verbinding met server */
```

```
123   if (connect(sockfd, (struct sockaddr *) &servaddr, sizeof(servaddr)) < 0)
124     ExitWithMessage("connect() failed");
125
126   doNonBlock(sockfd);
127
128   return 0;
129 }
```

Listing 22.1: Nonblocking.c

We bespreken kort de code :

- **lijnen 1 - 40 :** In het eerste deel van de functie *doNonBlock* worden alle IO kanalen (socket, toetsenbord en scherm) de optie NONBLOCK gegeven met de functie *fcntl* (zie 14.3). De constanten STDIN_FILENO en STDOUT_FILENO zijn vooraf gedefinieerd (doorgaans 0 en 1). Vervolgens worden de pointers geïnitialiseerd. De read en write sets worden klaargezet en *select* wordt aangeroepen met de descriptor sets *rset* en *wset*.
- **lijnen 43 - 71 :** In het volgende deel van de functie wordt getest op mogelijkheid tot lezen van de standaard invoer en de socket. Als we kunnen lezen van de standaard invoer, wordt *toiptr* opgeschoven en zetten we de bit die overeenkomt met de socket aan in de schrijfset *wset*, zodat later in de lus bij de test voor schrijven naar de socket de data onmiddellijk wordt geschreven.

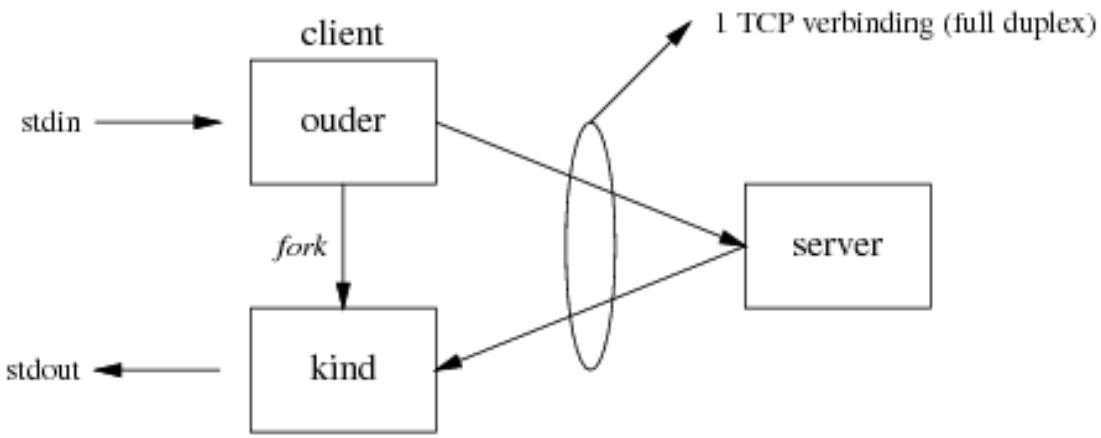
  Bij het lezen van de socket gebeurt er niets als *recv()* terugkeert met de fout *EWOULDBLOCK* (zoals het hoort, natuurlijk). Als we een EOF lezen is dit OK als ook de standaard invoer reeds EOF gaf (en dus waarde 1 gekregen heeft). In het andere geval komt er een foutmelding. Als data gelezen werd van de socket, wordt *friptr* aangepast en de bit voor standaard uitvoer wordt aangezet in de schrijf descriptor set, zodat in het volgende deel zal geschreven worden naar de standaard uitvoer.
- **lijnen 74 - 100 :** Het laatste deel van de functie geeft de *if*constructie voor schrijven naar standaard uitvoer en socket.
- **lijnen 103 - 129 :** Dit is de code voor een standaard TCP client.

22.2 Eenvoudige nonblocking met kindproces

Een eenvoudige manier om nonblocking sockets te simuleren is de applicatie in twee deel-processen te splitsen (met *fork()* bijvb.). We bekijken nog eens het voorbeeld van de echo server/client applicatie; de client stuurt alles wat hij van het toetsenbord leest naar de server en de server stuurt alles wat hij krijgt van de client terug.

Het ene proces neemt dan de invoer van het toetsenbord en schrijven naar de server voor zijn rekening en het andere proces verzorgt het lezen van de server en het schrijven naar het scherm. In figuur 22.3 zie je de situatie van de verbinding na een fork van de client.

Merk op dat er slechts één TCP connectie is; de ouder en kind delen gewoon dezelfde socket. De ouder schrijft naar de socket en het kind leest er van. De code is eenvoudiger dan de vorige uitwerking, maar helaas ook minder snel in uitvoering.



Figuur 22.3: Nonblocking client; verbinding met twee processen.

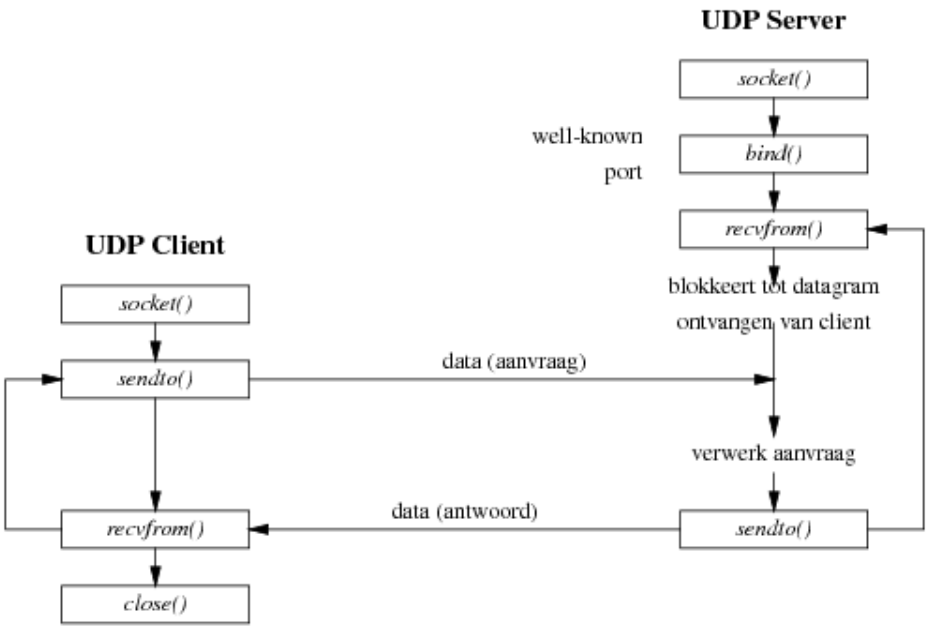
```
void echo_client(int invoer, int sockfd){
  int pid;
  char sendline[MAXLINE], recvline[MAXLINE];

  if ( (pid = Fork()) == 0) {
    /* kindproces: server -> stdout */
    while (recv(sockfd, recvline, MAXLINE, 0) > 0){
      printf(''%s'',recvline);
    }
    exit(0); /* stop dit proces */
  } else{
    /* ouderproces : invoer->server */
    while (read(invoer, sendline, MAXLINE) > 0){
      send(sockfd, sendline, strlen(sendline), 0);
    }
    Shutdown(sockfd, SHUT_WR); /* EOF op stdin */
  }
  return;
}
```

Hoofdstuk 23

UDP Sockets

23.1 Inleiding

UDP applicaties verschillen nogal van hun TCP tegenhangers wegens de verschillen in transportlagen; UDP is connectieloos, onbetrouwbaar en datagram-georiënteerd terwijl TCP in een geconnecteerde, betrouwbare byte stream voorziet. Figuur 23.1 laat de typische opbouw zien van een UDP client-server omgeving.



Figuur 23.1: Opeenvolging van de socket functies bij UDP server en client

23.2 *recvfrom* en *sendto* functies

Deze functies vervangen de *recv* en *send* bij TCP sockets.

```
#include <sys/socket.h>

int  recvfrom(int  s,  void  *buf,  int len, int flags,
       struct sockaddr *from, int *fromlen);
int  sendto(int s, const void *msg, int len, int flags,
       const struct sockaddr *to, int tolen);

Beide return: aantal bytes gelezen/geschreven als OK,
              -1 bij error
```

De eerste drie parameters zijn hetzelfde als bij *read* en *write*; descriptor, buffer en aantal bytes te lezen of te schrijven. Met de vierde parameter *flags* kan een aantal speciale opties worden ingesteld (zie *man*) maar wij houden het hier op 0. Het *to* argument in de *sendto* functie bevat het adres (IP adres en poortnummer) waar de data naartoe moet. De grootte van deze socket address structure zit in *addrlen*. De *recvfrom* functie vult de structure *from* met de adresgegevens van degene die de data stuurde.

Een datagram versturen met lengte 0 is OK; in dat geval zal *recvfrom* de waarde 0 teruggeven maar dit betekent niet dat de andere kant de connectie heeft verbroken, zoals bij TCP sockets. Bij UDP zijn er geen connecties en kunnen er bijgevolg ook geen gesloten worden.

23.3 UDP Echo server

In afdeling 11.11 gaven we een voorbeeld van een eenvoudige echo server; een client stuurt er tekst naartoe en de server stuurt diezelfde tekst onmiddellijk terug. We herschrijven deze server nu voor UDP:

```
#define SERV_PORT 4567
#define MAXLINE 256

int main(int argc, char **argv){
  int sockfd, len, n;
  struct sockaddr_in servaddr, cliaddr;
  char mesg[MAXLINE];

  sockfd = Socket(AF_INET, SOCK_DGRAM, 0);

  bzero(&servaddr, sizeof(servaddr));
  servaddr.sin_family = AF_INET;
  servaddr.sin_addr.s_addr = htonl(INADDR_ANY);
  servaddr.sin_port = htons(SERV_PORT);

  Bind(sockfd, (SA *) &servaddr, sizeof(servaddr));

  while ( 1 ) {
    len = sizeof(cliaddr);
    n = Recvfrom(sockfd, mesg, MAXLINE, 0, &cliaddr, &len);
    Sendto(sockfd, mesg, n, 0, &cliaddr, len);
  }
```

```
}
```

Listing 23.1: echo server met UDP

De server is *iteratief*, niet *concurrent*. Er is slechts één proces dat alle clients afhandelt. UDP servers zijn doorgaans iteratief en TCP servers meestal concurrent. De datagrammen gestuurd door de clients komen terecht in de leesbuffer van de UDP socket in FIFO orde.

23.4 UDP echo client

Een eenvoudige UDP echo client zou men als volgt kunnen programmeren :

```
#include "netwerk.h"
#define SERV_PORT 4567

int main(int argc, char **argv){
  int sockfd;
  struct sockaddr_in servaddr;

  if (argc != 2){
    printf("gebruik: udpcli <IPaddress>");
    exit(1);
  }
  bzero(&servaddr, sizeof(servaddr));
  servaddr.sin_family = AF_INET;
  servaddr.sin_port = htons(SERV_PORT);
  inet_pton(AF_INET, argv[1], &servaddr.sin_addr);

  sockfd = Socket(AF_INET, SOCK_DGRAM, 0);

  dg_cli(stdin, sockfd, (SA *) &servaddr, sizeof(servaddr));

  exit(0);
}
```

Een IPv4 socket adres structure wordt ingevuld met het IP adres en poortnummer van de server. Deze structure wordt dan doorgegeven aan *dg_client*.

```
void dg_cli(int fd, int sockfd, const SA *pservaddr, socklen_t len){
  int n;
  char buffer[MAXLINE+1];

  while (Read(fd, buffer, MAXLINE) > 0) {
    Sendto(sockfd, buffer, strlen(buffer), 0, pservaddr, len);

    n = Recvfrom(sockfd, buffer, MAXLINE, 0, NULL, NULL);

    buffer[n] = 0; /* null terminate */
    printf(``%s'',buffer);
  }
}
```

De functie leest een lijn tekst van de descriptor *fd*, stuurt de tekst naar de server met *sendto*, leest het antwoord van de server met *recvfrom* en stuurt dit antwoord naar het scherm. Merk op dat de laatste

argumenten van *recvfrom* NULL pointers zijn, waarmee we aangeven dat we niet geïnteresseerd zijn in het adres of poortnummer van de server. Het systeem zal aan de client pas een poortnummer toekennen bij de eerste aanroep van *sendto* (bij een TCP client gebeurt dit bij de aanroep van *connect*). Een UDP client zou eventueel *bind* kunnen aanroepen om een welbepaalde client-poort toe te kennen aan een socket.

23.5 Betrouwbaarheid

Bovenstaande client is onbetrouwbaar; wanneer een client datagram verloren gaat zal de client geblokkeerd blijven in de aanroep van *recvfrom*, wachtend op een antwoord van de server dat nooit zal komen omdat de server nooit een aanvraag kreeg van de client. Evenzo zal de client "eeuwig" blokkeren wanneer een server datagram verloren gaat. Dit probleem is op te lossen door een time-out op te geven voor *recvfrom*. Dit is gemakkelijkst te implementeren dmv de *select* functie. Hieronder een aangepaste timeout versie van de *recvfrom* functie:

```
int recvtimeout(int s, char *buf, int len, int timeout){
  fd_set fds;
  int n;
  struct timeval tv;

  // initialiseer de descriptoren
  FD_ZERO(&fds);
  FD_SET(s, &fds);

  // init de  struct timeval voor de timeout
  tv.tv_sec = timeout;
  tv.tv_usec = 0;

  // wacht tot data of timeout
  n = select(s+1, &fds, NULL, NULL, &tv);
  if (n == 0) return -2; // timeout!
  if (n == -1) return -1; // error

  // er is data, dus standaard recv()
  return recv(s, buf, len, 0);
}
```

Voorbeeld aanroep van recvtimeout():

```
  n = recvtimeout(s, buf, sizeof(buf), 10);

  if (n == -1)
    perror("recvtimeout");
  else if (n == -2)
    printf("timeout!\n");
  else
    printf("%s", buf);
```

Een ander probleem met de client is dat een willekeurig proces een datagram kan sturen naar het IP adres en poortnummer van de client (gegeven dat het proces op een of andere wijze het poortnummer van de client kent). Een oplossing voor dit probleem is controleren of het datagram dat we

binnenkregen met *recvfrom* wel degelijk van de server kwam waar we een datagram naartoe stuurden :

```
void dg_cli( int fd, int sockfd,
             const SA *pservaddr, int servlen){

  int n;
  char buffer[MAXLINE+1];
  int len;
  struct sockaddr *preply_addr;

  preply_addr = malloc(servlen);

  while (Read(fd, buffer, MAXLINE) > 0) {

    Sendto( sockfd, buffer, strlen(buffer), 0,
            pservaddr, servlen);

    len = servlen;
    n = Recvfrom(sockfd, buffer, MAXLINE, 0, preply_addr, &len);
    if (len != servlen || memcmp(pservaddr, preply_addr, len) != 0) {
      printf("antwoord van %s (genegeerd)\n",
             Sock_ntop(preply_addr, len));
    }
    else{
      buffer[n] = 0;    /* null terminate */
      printf(``%s'', buffer);
    }
  }
}
```

Hoofdstuk 24

Multicasting

24.1 Inleiding

Programma's die via multicast pakketten versturen zijn zeer vergelijkbaar met programma's die via UDP data versturen naar een enkele ontvanger. Het grootste verschil zit hem in het IP adres dat wordt gebruikt om de data heen te zenden. Hiervoor gebruikt men zogenaamde *multicast adressen*. Programma's die multicast data ontvangen zijn heel verschillend van programma's die UDP pakketten ontvangen, zoals we zullen zien.

Multicast adressen zijn IP adressen van klasse D en liggen dus in het bereik 224.0.0.0 tot en met 239.255.255.255. Sommige van deze adressen zijn gereserveerd, zoals 224.0.0.1; het *all hosts* adres. Meer info over specifieke multicast adressen vind je op http://www.ia-na.org/assignments/multicast-addresses.

24.2 Multicast onder Linux

Om multicast te gebruiken op een linux machine, moet de kernel dit ondersteunen. Een gemakkelijke manier om dit te testen is door te kijken in het bestand ".config" in de kernel source directory. Daar moet de optie *CONFIG_IP_MULTICAST* de waarde "y" (yes) hebben. Indien dit niet het geval is kan je het aanpassen en de kernel opnieuw compileren. Het compileren van een nieuwe linux kernel is niet moeilijk en op het Net zijn er veel goede gidsen.

Vervolgens voeg je een route toe :

```
# route add -net 224.0.0.0 netmask 240.0.0.0  dev lo
```

Dit commando voer je uit als **root**, uiteraard.

24.3 Multicast data sturen

Dit verloopt analoog aan het sturen van UDP pakketten naar een bepaald IP adres. In het geval van multicasting behoort dit IP adres tot klasse D. Zie voor het sturen van UDP pakketten hoofdstuk 23.

Volgend voorbeeldprogramma is een multicast server die een string stuurt naar het *all hosts* multicast adres *224.0.0.1*. Het is een aangepaste versie van de zender uit hoofdstuk 23.

```
#include "wrapper.h"

#define POORT         6789
#define MULTIADRES    "224.0.0.1"
#define TEST_STRING   "testboodschap van multicast server"


int main(){
  struct sockaddr_in server_adres;
  int socket_sd; /* socket descriptor */

  bzero(&server_adres, sizeof(server_adres));
  server_adres.sin_family = AF_INET;
  inet_pton(AF_INET, MULTIADRES, &server_adres.sin_addr);
  server_adres.sin_port = htons(POORT);

  /* maak een UDP socket */
  socket_sd =  socket(AF_INET, SOCK_DGRAM, 0);


  /* stuur multicast */

  while(1) {
    sendto(socket_sd, TEST_STRING,
           sizeof(TEST_STRING), 0,
           (struct sockaddr *)&server_adres, sizeof(server_adres));
    sleep(2);
  }
  close(socket_sd);
  return 0;
}
```

Listing 24.1: multicast server

24.4 Multicast data ontvangen

Een programma dat multicast pakketten wil ontvangen doet dit via een UDP socket. Met de functie *setsockopt* (zie hoofdstuk 14) wordt aan de kernel duidelijk gemaakt dat de socket multicast pakketten, bestemd voor een specifiek multicast adres, moet ontvangen. Men noemt dit wel eens *de socket inschrijven in een multicast groep*.

Het commando dat wordt meegegeven aan *setsockopt* is een *ip_mreq structure*. Hier moet je twee velden invullen: het multicastadres waar de socket data van moet ontvangen en de interface waarop dit moet gebeuren.

setsockopt wordt aangeroepen met de optienaam *IP_ADD_MEMBERSHIP*.

Voorbeeld:

```
struct ip_mreq commando;

commando.imr_multiaddr.s_addr = inet_addr("224.0.0.1");
commando.imr_interface.s_addr = htonl(INADDR_ANY);

setsockopt( socket, IPPROTO_IP, IP_ADD_MEMBERSHIP,
            &commando, sizeof(commando));
```

Wanneer je niet langer data van de multicastgroep wenst te ontvangen, roep je *setsockopt* aan maar nu met de optie *IP_DROP_MEMBERSHIP*. Dit gaat als volgt :

```
struct ip_mreq commando;

commando.imr_multiaddr.s_addr = inet_addr("224.0.0.1");
commando.imr_interface.s_addr = htonl(INADDR_ANY);

setsockopt( socket, IPPROTO_IP, IP_DROP_MEMBERSHIP,
            &command, sizeof(command));
```

Een volledig voorbeeld van een programma dat multicast data ontvangt van het adres *224.0.0.1* ziet er dan als volgt uit :

```
#define POORT         6789
#define MULTIADRES    "224.0.0.1"

int main(void){
  struct ip_mreq command;
  int yes = 1;
  int sin_len;
  char message[256];
  int socket;
  struct sockaddr_in sin;

  bzero(&sin, sizeof(sin));
  sin.sin_family = AF_INET;
  sin.sin_addr.s_addr = htonl(INADDR_ANY);
  sin.sin_port = htons(POORT);
  socket = Socket(PF_INET, SOCK_DGRAM, 0);
  Setsockopt(socket, SOL_SOCKET, SO_REUSEADDR, &yes, sizeof(yes));
  Bind(socket, (struct sockaddr *)&sin, sizeof(sin));

  command.imr_multiaddr.s_addr = inet_addr(MULTIADRES);
  command.imr_interface.s_addr = htonl(INADDR_ANY);

  Setsockopt( socket, IPPROTO_IP, IP_ADD_MEMBERSHIP,
              &command, sizeof(command));
  int iteratie;
  for(iteratie=0; iteratie<10; iteratie++) {
    sin_len = sizeof(sin);
    Recvfrom( socket, message, 256, 0,
              (struct sockaddr *)&sin, &sin_len);
    printf("Boodschap #%-2d van server: %sn", iteratie, message);
    sleep(2);
  }

  Setsockopt( socket, IPPROTO_IP, IP_DROP_MEMBERSHIP,
              &command, sizeof(command));

  Close(socket);
```

```
  return 0;
}
```

Listing 24.2: Multicast client

Merk op dat *SO_REUSEADDR* geen overbodige luxe is. Verschillende processen kunnen immers dezelfde data willen binnenhalen.

Hoofdstuk 25

De inetd superserver en SFTP

Een geheel andere manier om een serverprogramma te implementeren is door gebruik te maken van **inetd** (ook wel *Internet daemon* of *superserver* genoemd. Deze techniek is vooral nuttig bij eenvoudige serverprogramma's en wordt in de praktijk dikwijls toegepast.

25.1 Algemeen principe

Als we hem op de juiste manier configureren, kunnen we de ingewikkelde taak van het wachten op een verbinding met een client (en het eventueel behandelen van verschillende clients tegelijkertijd) aan de **inetd**-server overlaten. Het programma dat we zelf schrijven, hoeft zich dan enkel bezig te houden met het berichtenverkeer van en naar de client.

Concreet betekent dit dat **inetd** in jouw plaats op een verbinding zal wachten op een bepaalde poort en dat hij jouw programma pas zal opstarten wanneer de verbinding effectief gemaakt is. **Inetd** zorgt er dan voor dat tijdens de loop van het programma filedescriptors 0, 1 en 2 nu niet meer met toetsenbord en scherm verbonden zijn, maar in de plaats de netwerkverbinding met de client representeren. Lezen van filedescriptor 0 komt dus overeen met het binnenhalen van berichten van de client, en schrijven naar filedescriptor 1 met berichten versturen[1].

Als voorbeeld zullen we met deze techniek een eenvoudige *file transfer* service implementeren waarmee bestanden van en naar een vaste *basisdirectory* op de server kunnen worden gedownload en geupload. We hebben dit protocol *SFTP* gedoopt (*Simple File Transfer Protocol*). Het serverprogramma heet **sftpd** en heeft één opdrachtlijnparameter, de basisdirectory van het protocol. Opdat dit programma door **inetd** zou worden behandeld, moeten we eerst twee configuratiebestanden aanpassen.

In het bestand `/etc/services` moeten we de naam van ons protocol met een poortnummer associeren (hier 3124),

[1] Je kan deze filedescriptors door elkaar gebruiken. Je kan dus lezen van descriptor 2 en schrijven naar 0. Er zijn echter goede redenen om je bij de conventies te houden.

```
...
sftp            3124/tcp
...
```

en in `/etc/inetd.conf` moeten we het serverprogramma opgeven. (We gebruiken `/usr/local/sftp` als basisdirectory.)

```
...
sftp stream tcp nowait root
                /usr/local/bin/sftpd sftpd /usr/local/sftp
...
```

Vergeet ook niet aan **inetd** te laten weten dat zijn configuratiebestand werd gewijzigd. Meestal gebeurt dit met behulp van een **kill**-opdracht :

```
# ps ax | fgrep inetd
  344 ?        S      0:00 inetd
# kill -HUP 344
#
```

25.2 Het SFTP-protocol

Vooraleer we de server implementeren, geven we eerst een korte beschrijving van het protocol.

We maken gebruik van drie soorten berichten die tussen server en client worden uitgewisseld. Je kan zien over welk soort bericht het gaat aan de eerste byte (eerste letterteken) van het bericht. Berichten worden meestal afgesloten door een *linefeed* of een combinatie van *carriage return* en *linefeed* (om de DOS-wereld ter wille te zijn).

Dit zijn de mogelijkheden :

1. Een *opdracht*bericht wordt van client naar server gestuurd als begin van een bestandsoverdracht of om de verbinding af te sluiten. Dit zijn de mogelijke opdrachten :

Gbestandsnaam	Stuur een bestand van server naar client (*get*).
Pbestandsnaam	Stuur een bestand van client naar server (*put*).
Q	Beëindig de verbinding.

   Merk op dat er verschillende opdrachten kunnen verstuurd worden tijdens één enkele verbindingssessie.

2. Een *status*bericht dient als antwoord op een ander bericht. Het eerste letterteken is + om aan te geven dat alles goed gaat, en - wanneer er een fout optreedt. De rest van deze berichtlijn (voor het protocol van minder belang) bevat een korte tekst met meer informatie. Bijvoorbeeld :

```
+bestand ontvangen
-kan het bestand niet lezen
-ongeldig bericht
```

3. Een bestand wordt niet in één keer van de ene computer naar de andere overgedragen, maar wordt opgesplitst in verschillende *datapakketten*. Elk datapakket dat verstuurd wordt, wordt voorafgegaan door een lijn van de vorm '*Daantal*'. Dit geeft het aantal bytes aan (in decimale notatie) van het datapakket dat erop volgt. De *D*-lijn wordt afgesloten met een linefeed, het pakket zelf echter niet.

Datapakketten mogen hoogstens 8192 bytes bevatten. Opdrachtlijnen en statuslijnen mogen slechts 256 bytes beslaan (inclusief de linefeed op het einde.)

Het SFTP-protocol begint met de client die een opdrachtbericht naar de server stuurt. Is dit een ongeldige opdracht, dan stuurt de server een negatief statusbericht terug en begint het protocol opnieuw. Een *G*- of een *P*-opdracht worden verwerkt zoals hieronder beschreven, waarna de server op een nieuwe opdracht wacht. Dit herhaalt zich totdat de client het protocol afsluit met een *Q*-opdracht.

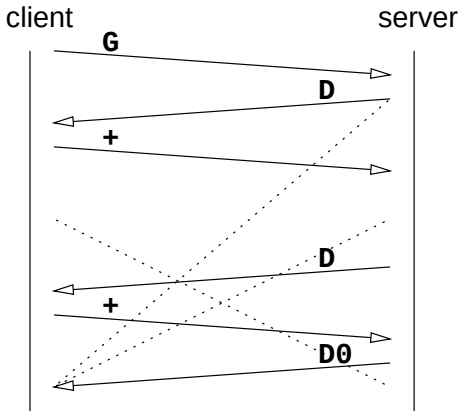
Bij een *G*-opdracht controleert de server eerst of de meegestuurde bestandsnaam een geldige naam is (om veiligheidsredenen wordt het teken '/' niet in een bestandsnaam toegelaten[2]), of het bestand in de basisdirectory bestaat en of dit bestand kan gelezen worden. Is dit niet het geval dan wordt een negatief statusbericht gestuurd en beginnen we weer van voor af aan.

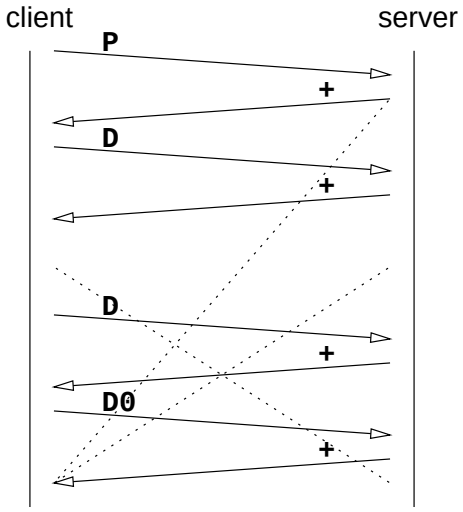
Anders begint de server meteen met de gegevensoverdracht : het bestand wordt in afzonderlijke *D*-pakketten stuk voor stuk naar de client gestuurd. De client stuurt na elk pakket een statusbericht terug. De server sluit de pakkettenstroom af met een pakket van lengte 0 (waarop door de client *niet* meer mag worden gereageerd). De gegevensoverdracht wordt onderweg stopgezet wanneer de client antwoordt met een negatief statusbericht. Ook de server kan een negatieve status sturen in plaats van een *D*-pakket om de gegevensoverdracht vroegtijdig te beëindigen.

Een *P*-opdracht verloopt op een gelijkaardige manier. Eerst antwoordt de server met een statusbericht en wanneer dit positief is begint de client met het verzenden van *D*-pakketten. Op elk van die pakketten antwoordt de server met een statusbericht, ook op het laatste pakket (van lengte nul).

De figuur hieronder geeft schematisch weer hoe de gegevensoverdracht in beide gevallen verloopt. (Stippellijnen corresponderen met negatieve statusberichten.)

[2]Wanneer de basisdirectory bijvoorbeeld */tmp* is, zou je anders een bestandsnaam kunnen opgeven zoals *../etc/passwd*





25.3 Het serverprogramma

Door de manier waarop **inetd** werkt, kunnen we tijdens de opbouw van het programma abstractie maken van de netwerkverbinding en doen alsof berichten van de client van het toetsenbord komen, en alsof berichten naar de client naar het scherm mogen gestuurd worden. We kunnen het programma ook op die manier testen, vooraleer we het aan de zorgen van **inetd** overdragen.

Omdat het protocol voornamelijke lijngericht is, zullen we uitvoerig gebruik maken van de functie *readline* uit vorig hoofdstuk, maar ook *readbuf*, *writebuf* en *writebuf* komen aan bod. We gebruiken ook nog een vierde functie *writestr* waarmee een C++-string kan worden uitgeschreven :

```
int writestr (int fd, char *str)
// Zoals writebuf, maar met een C++-string
{
  writebuf (fd, str, strlen(str));
}
```

We bekijken eerst het hoofdprogramma :

```
int main (int argc, char **argv)
{
  if (argc != 2) return 1;
  basisdirectory (argv[1]);

  char aanvraag[256];
  int n = readline (0, aanvraag, 256);
  while (n > 0 && aanvraag[0] != 'Q') {
    if (aanvraag[n-1] != '\n')
      writestr (1, "-aanvraag te lang\n");
    else if (aanvraag[0] == 'G')
      verwerkGet (aanvraag+1);
    else if (aanvraag[0] == 'P')
      verwerkPut (aanvraag+1);
    else
      writestr (1, "-onbekend opdrachtbericht\n");
    n = readline (0, aanvraag, 256);
  }
```

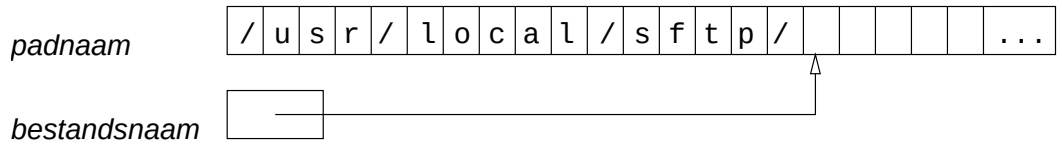
```
  return n <= 0;
}
```

Het hoofdprogramma is weinig meer dan een lus die telkens de opdrachtberichten van de client analyseert en door een gepaste hulpprocedure laat verwerken.

Deze hulpprocedures heten *verwerkGet* en *verwerkPut*. De bedoeling is dat ze alle protocolberichten verwerken die bij een *volledige* bestandsoverdracht horen, dus eventueel inclusief het afsluitende statusbericht van server naar client (dit komt overeen met de schema's in §25.2). Als argument nemen ze de bestandsnaam uit het opdrachtbericht.

De hoofdlus zelf probeert reeds een aantal fouten op te vangen en aan de client te melden[3]. Wanneer de *read* een *end-of-file* signaleert, betekent dit dat de client de verbinding vroegtijdig heeft afgesloten. In dat geval heeft het weinig zin om een statusbericht naar de client terug te sturen en sluiten we gewoon ons serverprogramma af. Hetzelfde doen we wanneer zich een netwerkleesfout voordoet.

Vooraleer deze hoofdlus start, roept het programma de procedure *basisdirectory* op die een aantal globale variabelen initialiseert (zie figuur). Het argument van *basisdirectory* komt van de opdrachtlijn.



Om het de gebruiker iets gemakkelijker te maken, plaatst *basisdirectory* een schuine streep vóór en na de opgegeven directorynaam, als deze er nog niet staan.

```
char padnaam [512];
char *bestandsnaam;

void basisdirectory (char *s)
{
  bestandsnaam = padnaam;
  if (*s != '/')
    *bestandsnaam++ = '/';
  while (*s != 0) *bestandsnaam++ = *s++;
  if (bestandsnaam[-1] != '/')
    *bestandsnaam++ = '/';
}
```

Later in het programma (in *verwerkGet* en *verwerkPut*) wordt de naam van het bestand dat moet worden overgezet, gecopieerd naar de plaats waarnaar *bestandsnaam* verwijst. Als volledige naam van het bestand wordt dan *padnaam* gebruikt.

Dit gebeurt met behulp van de functie *maakpadnaam*. Merk op dat deze functie het einde van een lijn omzet naar een nullteken en bovendien signaleert wanneer de bestandsnaam het ongeldige letterteken '/' bevat.

[3] Een opdrachtbericht dat langer is dan 256 bytes, wordt niet perfect opgevangen. We laten het aan de lezer over om het programma hier een beetje op te poetsen.

```
bool maakpadnaam (char *s)
{
  char *d = bestandsnaam;
  while (*s != '/' && *s != '\n')
    *d++ = *s++;
  if (d[-1] == '\r') d--;
  *d = 0;
  return *s != '/';
}
```

De procedure *verwerkGet* roept eerst *maakpadnaam* op en probeert daarna het opgegeven bestand te openen om ervan te lezen. Fouten die hierbij optreden worden aan de client gesignaleerd.

```
void verwerkGet (char *s)
{
  if (!maakpadnaam(s)) {
    writestr (1, "-ongeldige bestandsnaam\n");
    return;
  }
  int fd = open (padnaam, O_RDONLY);
  if (fd < 0) {
    writestr (1, "-kan het bestand niet openen om te lezen\n");
    return;
  }
  ...
```

Daarna wordt het bestand in stukken van maximaal 8192 bytes naar de client doorgestuurd, volgens het protocol dat we eerder hebben besproken (§25.2).

Voor elk stuk wordt een afzonderlijk datapakket opgebouwd. De *D*-lijn waarmee dit pakket begint, construeren we met behulp van *gcvt*.

Het getal moet immers in decimale vorm worden doorgegeven naar de client. Daarna sturen we de gegevens zelf door met behulp van *writebuf*[4].

De lus waarin deze blokken worden verstuurd, eindigt wanneer het bestand volledig is getransfereerd, wanneer er een leesfout optreedt, of wanneer de client antwoordt met een negatief statusbericht. In het eerste geval versturen we nog een *D0*-lijn en in het tweede geval versturen we zelf een negatieve statuslijn.

```
void verwerkGet (char *s)
{
  ...
  char buffer[8192];
  bool positievestatus = true;
  int n = readbuf (fd, buffer, 8192);
  while (positievestatus && n > 0) {
    char dataheader [7] ;
    dataheader[0] = 'D';
    gcvt (n, 4, dataheader+1);
    strcat (dataheader+1, "\n");
    writestr (1, dataheader);
    writebuf (1, buffer, n);
    char antwoord [256];
    if (readline (0, antwoord, 256) <= 0)
      exit (1);
```

[4]We veronderstellen hier voor de eenvoud dat er geen netwerkschrijffouten optreden.

```
    else if (antwoord[0] != '+')
      positievestatus = false;
    else
      n = readbuf (fd, buffer, 8192);
  }
  close (fd);
  if (positievestatus) {
      if (n < 0)
          writestr (1, "-fout bij het lezen van het bestand\n");
      else
          writestr (1, "D0\n");
  }
}
```

De procedure *verwerkPut* vertoont hiermee heel wat gelijkenis. Nadat de opgegeven bestandsnaam werd gecontroleerd en het bestand werd geopend, sturen we eerst een positief statusbericht[5]. Daarna verwerken we het inkomend bestand in een lus, die opnieuw om verschillende redenen kan stoppen : omdat we het inkomende bestand volledig hebben verwerkt, of omdat er onderweg een fout is opgetreden.

```
void verwerkPut (char *s)
{
  ...
  writestr (1, "+OK, het bestand mag doorgezonden worden\n");

  bool positievestatus = true;
  int n = leesDheader ();
  while (n > 0 && positievestatus) {
    char buffer[8192];
    if (readbuf (0, buffer, n) != n)
      exit (1);
    if (writebuf (fd, buffer, n) < 0) {
      writestr (1, "-fout bij het schrijven van het bestand\n");
      positievestatus = false;
    }
    else
      writestr (1, "+OK, volgend stuk graag\n");
    n = leesDheader ();
  }
  close (fd);
  if (positievestatus) {
    if (n < 0)
      writestr (1, "-Protocolfout\n");
    else
      writestr (1, "+OK, bestand ontvangen\n");
  }
}
```

Voor elk stuk van het bestand stuurt de client een afzonderlijk datapakket dat uit twee stukken bestaat : een *D*-lijn en de gegevens zelf. De *D*-lijn verwerken we in de functie *leesDheader*, de rest wordt door *verwerkPut* ontvangen met een eenvoudige *readbuf*.

De functiewaarde van *leesHeader* is de lengte van het pakket dat volgt, of -1 als er een fout optrad. Net zoals de andere leeroutines in dit programma, wordt het programma onmiddellijk beëindigd met *exit* bij een leesfout op het netwerk.

```
int leesDheader (void)
{
```

[5]Deze eerste fase van *verwerkPut* hebben we niet afgedrukt.

```
  char buffer[256];
  int n = readline (0, buffer, 256);
  if (n <= 0 )
    exit (1);
  if (buffer[0] != 'D' || buffer[n-1]!= '\n') return -1;

  char *p;
  int aantal = strtol (buffer+1, &p, 10);
  if (*p == '\r') p++;
  if (*p == '\n' && aantal >= 0 && aantal < 8192)
    return aantal;
  else
    return -1;
}
```

25.4 Foutmeldingen met syslog

Zoals reeds eerder gezegd, kunnen we foutmeldingen niet naar het standaard foutkanaal sturen omdat dit door **inetd** aan de netwerkverbinding wordt gekoppeld. Als alternatief kan je foutmeldingen eventueel opslaan in een eigen logbestand. UNIX voorziet echter een ander eenvoudig mechanisme dat gebruik maakt van de **syslogd** daemon.

Dit daemon wordt bij het opstarten van de computer automatisch geactiveerd en heeft als enige taak foutberichten van allerhande systeemprogramma's op een aantal vooraf geconfigureerde manieren te rapporteren[6]. Al naargelang de ernst van de fout kan het foutbericht rechtstreeks op het consolescherm worden afgedrukt, achteraan één of ander logbestand worden toegevoegd of zelfs naar een bepaalde persoon worden doorgemailed.

Je kan in je programma foutberichten doorgeven aan **syslogd** met behulp van de functie *syslog*. Een typische oproep van deze functie heeft de volgende vorm :

```
#include <syslog.h>
...
   syslog (LOG_WARNING | LOG_DAEMON,
           "Netwerkverbinding onderbroken: %m");
```

De tweede parameter van *syslog* bevat het foutbericht. Deze string mag bovendien een aantal speciale codes bevatten (die met een %-teken beginnen). De code *%m* die we hierboven gebruiken, wordt bijvoorbeeld vervangen door het foutbericht dat bij de laatste systeemfunctie hoort, hetzelfde bericht dat anders door *perror* zou worden afgedrukt.

Het eerste argument van *syslog* is een combinatie van twee soorten systeemconstanten : de eerste geeft de ernst aan van de foutmelding, de tweede de contekst waarin de fout zich voordoet. De volgende tabel geeft een overzicht van de verschillende mogelijkheden voor de eerste constante, van zeer ernstig tot minder ernstig.

[6]Wegens plaatsgebrek moeten we ons in deze tekst beperken tot een aantal hoofdlijnen. We vertellen je bijvoorbeeld niet hoe dit daemon moet worden geconfigureerd en we laten ook heel wat details weg over de functie *syslog* zelf. Voor meer informatie verwijzen we — zoals steeds — naar de (elektronische) handleiding.

LOG_EMERG	Computer is onbruikbaar geworden
LOG_ALERT	Gelieve onmiddellijk actie te ondernemen
LOG_CRIT	Kritieke fout
LOG_ERR	'Gewone' fout
LOG_WARNING	Waarschuwing
LOG_NOTICE	Opmerking (belangrijke gebeurtenis, maar geen fout)
LOG_INFO	Ter informatie
LOG_DEBUG	Enkel gebruikt om programma's te debuggen

(Wat **syslogd** met elk van deze berichten doet, hangt af van hoe hij geconfigureerd werd.)

De tweede constante geeft aan in wat voor programma deze *syslog*functie voorkomt. Dit wordt door **syslogd** voornamelijk gebruikt om onderscheid te kunnen maken tussen verschillende logbestanden.

LOG_PRIVATE	Beveiligings- en authorizatieberichten
LOG_DAEMON	Systeemdaemons
LOG_MAIL	Berichten van de postverwerking
...	(Voor een volledige lijst verwijzen we naar de manpages)

Hoofdstuk 26

Remoting in C#

Remoting (C#)

Veerle Ongenae

Inhoud

- Structuur Remoting
- Remotable object
- Server
- Client
- Opmerkingen
- Voorbeeld
- Beveiliging

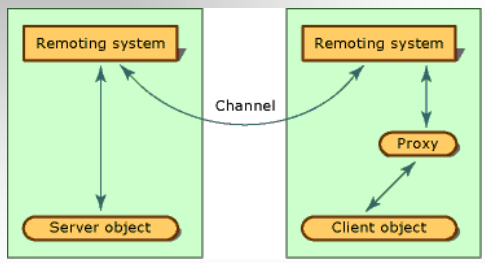
Remoting

- Gedistribueerde applicaties
- Objecten gebruiken
  - In andere processen op een zelfde computer
  - Op een andere computer (via netwerk)
- Eigenlijke netwerkcommunicatie is afgeschermd

Structuur Remoting applicatie

- Een “remotable” object
- Host application domain → Server
  - Luisteren naar aanvragen voor het object
- Client application domain → Client
  - Aanvragen opstellen voor object

Structuur Remoting applicatie



Inhoud

- Structuur Remoting
- Remotable object
- Server
- Client
- Opmerkingen
- Voorbeeld
- Beveiliging

“Remotable object”

- Klasse afgeleid van MarshalByRefObject

```
using System;
public class RemotableType : MarshalByRefObject
{
   public string SayHello()
   {
      Console.WriteLine("RemotableType.SayHello() was called!");
      return "Hello, world";
   }
}
```

- Maak dll
  - Gebruikt in client en server

Inhoud

- Structuur Remoting
- Remotable object
- Server
- Client
- Opmerkingen
- Voorbeeld
- Beveiliging

Hosting application → Server

- Nodig opdat van buiten af het object aangemaakt en opgeroepen kan worden
- Een kanaal kiezen en registreren
  - Netwerkprotocol afhandelen
  - Serialisatie afhandelen
- Type object registreren bij .NET remoting system
  - Gebruikt kanaal om naar aanvragen voor het type te luisteren
- In configuratie-bestand

Configuratie

- *app-name*.exe.config

```
<configuration>
  <system.runtime.remoting>
    <application>
      <service>
        <wellknown mode="Singleton" type="RemotableType,
RemotableAssembly" objectUri="RemotableType.rem" />
      </service>
      <channels>
        <channel ref="http" port="8989"/>
      </channels>
    </application>
  </system.runtime.remoting>
</configuration>
```

Gekend buiten de applicatie

assembly

type

Kanalen

- Drie standaard mogelijkheden
  - HttpChannel
    - SOAP over HTTP
  - TcpChannel
    - Tcp sockets
    - Binair
  - IpcChannel
    - Named pipes
    - Verschillende processen op zelfde computer

Host activation

- Server objecten worden aangemaakt bij het oproepen van een methode
  - Niet bij het oproepen van de constructor bij de client
    - Proxy object wordt dan aangemaakt
    - → default constructor
  - Mode
    - Singleton
      - Één object op server voor alle aanvragen
    - SingleCall
      - Voor elke aanvraag een nieuw object

Hosting application: code

- Gebruik bib System.Runtime.Remoting
- Configuratie in laden

```
public class Listener
{
   public static void Main()
   {
      RemotingConfiguration.Configure("Listener.exe.config", false);
      Console.WriteLine("Listening for requests. Press enter to exit...");
      Console.ReadLine();
   }
}
```

Enable security?

Inhoud

- Structuur Remoting
- Remotable object
- Server
- Client
- Opmerkingen
- Voorbeeld
- Beveiliging

Client applicatie

- Registreren als client voor het remote object
- Methodes oproepen

Client configuratie

- *app-name*.exe.config

```
<configuration>
  <system.runtime.remoting>
    <application>
      <client>
        <wellknown
          type="RemotableType, RemotableAssembly"
          url="http://localhost:8989/RemotableType.rem"
        />
      </client>
    </application>
  </system.runtime.remoting>
</configuration>
```


Client code

- Configuratie inlezen
- Remote object aanmaken en gebruiken

```
public static void Main(){
    RemotingConfiguration.Configure("Client.exe.config");
    RemotableType remoteObject = new RemotableType();
    Console.WriteLine(remoteObject.SayHello());
  }
```


Inhoud

- Structuur Remoting
- Remotable object
- Server
- Client
- Opmerkingen
- Voorbeeld
- Beveiliging

TcpChannel ipv HttpChannel

- Aanpassen configuratie
  - Server

```
<channel ref="tcp" port="8989"/>
```

  - Client

```
<wellknown
  type="RemotableType, RemotableAssembly"
  url="tcp://localhost:8989/RemotableType.rem"
/>
```


Opmerkingen

- Dll met remote object zowel op client als op server
- Alternatief
  - Dll met interface
    - Client
    - Server
  - Server
    - Implementatie interface
  - Client
    - Geen constructor
    - Methode System.Activator.GetObject(Type, String)
      - Type: type
      - String url

Opmerkingen

- Dll met remote object zowel op client als op server
- Alternatief
  - Stub genereren
    - Commandolijn tool soapsuds.exe

Inhoud

- Structuur Remoting
- Remotable object
- Server
- Client
- Opmerkingen
- Voorbeeld
- Beveiliging

Voorbeeld

- VbRemoteObject.RemoteTijd
  - VbRemoteObject.dll
    - In client en server project
- VbRemoteServer.cs
  - VbRemoteServer.exe.config
    - Manueel kopiëren
- VbRemoteClient.cs
  - VbRemoteClient.exe.config
    - Manueel kopiëren

Inhoud

- Structuur Remoting
- Remotable object
- Server
- Client
- Opmerkingen
- Voorbeeld
- Beveiliging

Beveiliging

- Iedereen kan methode van het remote object oproepen
  - Authenticatie nodig
- Of vertrouwelijke informatie onderscheppen
  - Encryptie communicatie nodig
- Standaard geen beveiliging

Encryptie communicatie

- HttpChannel
  - Enkel indien via IIS
- TcpChannel
  - Configuratie server én client
- IpcChannel
  - Geen encryptie

Encryptie communicatie TCP

Client

```
<configuration>
 <system.runtime.remoting>
  <application>
   <client>
    <wellknown  … />
   </client>
   <channels>
    <channel ref="tcp"
  secure="true"/>
   </channels>
  </application>
 </system.runtime.remoting>
</configuration>
```

Server

```
<configuration>
 <system.runtime.remoting>
  <application>
   <service>
    <wellknown  …  />
   </service>
   <channels>
    <channel ref="tcp"
  port="8989" secure="true" />
   </channels>
  </application>
 </system.runtime.remoting>
</configuration>
```


Beveiliging

- Iedereen kan methode van het remote object oproepen
  - Authenticatie nodig
- Of vertrouwelijke informatie onderscheppen
  - Encryptie communicatie nodig
- Standaard geen beveiliging

Authenticatie en impersonation

- Authenticatie
  - Identificatie client
- Impersonation
  - Remote object voert code uit als een andere gebruiker
    - Bepaald door client

Authenticatie HttpChannel

- Server
  - IIS
- Client

Gebruiker clientapplicatie

```
<configuration>  <system.runtime.remoting>   <application>
   <channels>
    <channel ref="http" useDefaultCredentials="true"/>
   </channels>
   <client>       <wellknown
      url="http://MyServer/IISSec/MyRemoteObj.rem"
      type="Shared.MyRemoteObj, Shared"/>
   </client>
  </application>  </system.runtime.remoting></configuration>
```

Impersonation TcpChannel

Server

```
<configuration>
  <system.runtime.remoting>
    <application>
      <channels>
        <channel ref="tcp"
  port="..." secure="true"
  impersonate="true" />
      </channels>
      …
    </application>
  </system.runtime.remoting>
</configuration>
```

Client

```
<configuration>
  <system.runtime.remoting>
    <application>
      <channels>
        <channel ref="tcp"
  secure="true"
  tokenImpersonationLevel="imp
  ersonation"/>
      </channels>
      …
    </application>
  </system.runtime.remoting>
</configuration>
```


Authenticatie IpcChannel

- Authenticatie is standaard
  - Gebruiker client = gebruiker server
  - Andere gebruikers toelaten
    - configuratie server

```
<channels>
  <channel ref="ipc" portName="MyIpcChannel"
  authorizedGroup="Users"/>
</channels>
```


Impersonation IpcChannel

Server

```
<channel ref="ipc"
  portName="MyIpcChannel"
  secure="true"
  impersonate="true"
  authorizedGroup="Users"/>
      </channels>
```

Client

```
<channels>
      <channel ref="ipc"
  secure="true"
  tokenImpersonationLevel="imp
  ersonation" />
      </channels>
```


Informatie

- MSDN: .NET Framework Remoting Overview (http://msdn.microsoft.com/en-us/library/bb552670.aspx)

Hoofdstuk 27

RMI

RMI

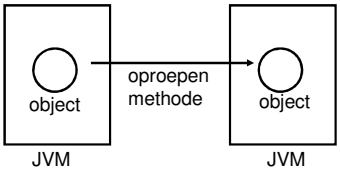
Veerle Ongenae

Inhoud

- Wat is RMI?
- Structuur RMI-applicatie
- Ontwikkelen RMI-applicatie

RMI

- Remote Method Invocation
- Java



- Communicatie tussen Java-programma's op afstand

RMI-applicatie

- Vaak client-server applicatie
  - Server
    - Maakt objecten aan
    - Biedt referentie naar die objecten aan
    - Wacht op clients
      - Oproepen methodes objecten
  - Client
    - Verkrijgt referentie naar objecten van de server
    - Roept methode op
- RMI
  - Bepaalt communicatie tussen client en server

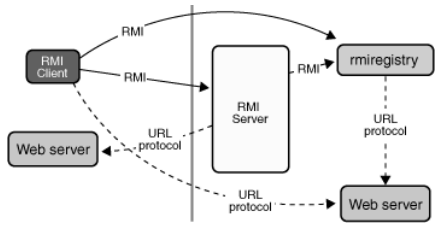
Inhoud

- Wat is RMI?
- Structuur RMI-applicatie
- Ontwikkelen RMI-applicatie

RMI-applicatie

- Gedistribueerde object applicatie
- Taken
  - Object op afstand lokaliseren
  - Communicatie met object op afstand
  - class-definities laden van objecten die doorgestuurd worden
    - Argumenten
    - Resultaat

Structuur RMI-applicatie



Bron: java.sun.com

Web server: laden class-definities

Remote objects

- Remote objects
  - Objecten met methodes
    - Kunnen uitgevoerd worden vanuit een andere JVM
- Implementeren een remote interface
  - Interface afgeleid van java.rmi.Remote
  - Elke methode van de interface
    - throws java.rmi.RemoteException

Stub

- RMI bezorgt de oproepende JVM een stub voor het remote object
  - GEEN KOPIE
- Stub
  - Lokale voorstelling van remote object (proxy)
  - Remote reference
- Client
  - Oproepen methode stub
    - Roept methode remote object op

Inhoud

- Wat is RMI?
- Structuur RMI-applicatie
- Ontwikkelen RMI-applicatie

Aanmaken RMI-applicatie

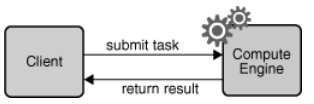
- Componenten ontwerpen en implementeren
  - Remote interfaces definiëren
  - Remote objecten implementeren
  - Clients ontwikkelen
- Compileren
- Klassen beschikbaar maken via het netwerk
  - Remote interfaces, af te halen klassen, …
  - Bv. via webserver
- Applicatie starten

Ontwikkelen RMI-applicatie

- Voorbeeld
- Server
  - Remote interface(s)
  - Remote object(s): klassen schrijven
- Client
- Compilatie
- Applicatie starten

Voorbeeld

- Compute engine
  - Server
    - Remote object
      - Voert taken voor een client uit
      - Geeft resultaat van de taak terug aan de client
      - Taken worden uitgevoerd op de server



Bron: java.sun.com

Voorbeeld

- Taken
  - Hoeven niet gekend te zijn bij het starten van de compute engine
  - Op elk moment kunnen nieuwe taken gedefinieerd worden → uitgevoerd op server
  - Implementeren interface
  - Code wordt dynamisch geladen in server

Ontwikkelen RMI-applicatie

- Voorbeeld
- Server
  - Remote interface(s)
  - Remote objects: klassen schrijven
- Client
- Compilatie
- Applicatie starten

RMI-server schrijven

- Functionaliteit
  - Taken aanvaarden
  - Taken uitvoeren
  - Resultaat doorspelen
- Realisatie
  - Interface
    - Zicht van de client
  - Klasse
    - Implementatie interface

Ontwikkelen RMI-applicatie

- Voorbeeld
- Server
  - Remote interface(s)
  - Remote objects: klassen schrijven
- Client
- Compilatie
- Applicatie starten

Remote interface

- Compute.java
  - Maakt gebruik van de interface Task
- Bepaalt communicatieprotocol tussen client en server
- Interface afgeleid van java.rmi.Remote
- Elke methode van de interface
  - throws java.rmi.RemoteException

Task

- Opdracht
  - Uitvoeren door server
- Task.java

```
package compute;
public interface Task<T> {
   T execute();
}
```

- T: type resultaat uitvoeren taak

Serialisatie

- RMI
  - Transport objecten via serialisatie
- Dus
  - Implementatie Task-interface
  - Resultaat execute methode (T)
  - Serialiseerbaar
    - Implementeren interface java.io.Serializable

Transport objecten

- Mogelijke argumenten/resultaten van remote methods
  - Primitieve types
  - Remote objecten
  - Serialiseerbare objecten
- Doorgeven objecten aan remote method
  - Remote objecten: passed by reference
  - Lokale objecten: passed by copy

Ontwikkelen RMI-applicatie

- Voorbeeld
- Server
  - Remote interface(s)
  - Remote objects: klassen schrijven
- Client
- Compilatie
- Applicatie starten

Implementatie Remote interface

- ComputeEngine.java
  - Implementatie interface Compute
  - Opstarten server
    - Aanmaken en installeren security manager
    - Aanmaken en exporteren remote object
    - Registreren remote object bij RMI registry of …

Security Manager

- Beschermen systeem
- Bepaalt toegang van de gedownloade code
  - Toegang tot bestandssysteem
  - Welke opdrachten uitgevoerd mogen worden
- Geen security manager
  - Klassen kunnen niet gedownload worden

Remote object beschikbaar maken

- Exporteren → binnenkomende aanvragen ontvangen
  - Methode van java.rmi.server.UnicastRemoteObject

```
public static Remote exportObject(Remote obj, int port) throws
    RemoteException
```

- Tweede argument: poort
  - 0: anoniem, RMI of systeem kiest
- Resultaat: stub (vb. Compute)
  - Enkel remote methods

Referenties naar remote objecten ophalen

- Client
  - Remote object gebruiken
  - Referentie nodig
- Verkrijgen referentie remote objects
  - Resultaat methode
  - Member object
- Speciaal remote object: RMI registry
  - Vinden referenties van remote objects
  - Meestal enkel eerste object
- Server registreert remote object bij RMI registry

Registratie remote objects

- java.rmi.registry.LocateRegistry
  - Registratie object vinden (standaardpoort)

```
Registry registry = LocateRegistry.getRegistry();
```

  - Registratie object creëren
- java.rmi.registry.Registry
  - Interface
  - Bewaren en vinden referenties naar remote objects

```
void bind(String name,Remote obj) throws RemoteException,
    AlreadyBoundException, AccessException
void rebind(String name, Remote obj) throws
    RemoteException, AccessException
```

  - Referentie naar een remote object koppelen aan een naam in het register

Opruimen remote object

- Programma eindigt
  - Remote object blijft bestaan zolang er een referentie naar is in een andere JVM
- ComputerEngine-object verdwijnt
  - Verwijdert uit register
  - Én
  - Geen remote references meer bij de client

Ontwikkelen RMI-applicatie

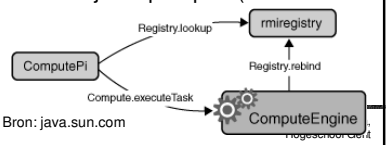
- Voorbeeld
- Server
  - Remote interface(s)
  - Remote objects: klassen schrijven
- Client
- Compilatie
- Applicatie starten

Client

- Voorbeeld
  - ComputePi.java
    - Roept de methode executeTask van een remote object op
  - Pi.java
    - Implementeert de interface Task
    - Berekent waarde van π
    - Constructor
      - Aantal cijfers na komma

Client: verschillende stappen

- Aanmaken en installeren security manager
  - Bescherming tegen gedownloade stub-klassen
- Referentie naar registerobject op server ophalen
- Referentie naar remote object opzoeken in registerobject (vb. Compute)
- Methode van remote object oproepen (vb. executeTask)



Bron: java.sun.com

Referentie naar registerobject op server ophalen

- java.rmi.registry.LocateRegistry
  - Registratie object vinden (standaardpoort)

public static Registry getRegistry(String host)
throws RemoteExceptionegistry

Referentie naar remote object opzoeken in registerobject

- java.rmi.registry.Registry

Remote lookup(String name) throws RemoteException, NotBoundException, AccessException

- Referentie naar een remote object bepalen op basis van de naam in het register

Argument remote method

- Serialiseerbaar object

private static final long serialVersionUID = 227L;

  - Compabiliteit verschillende versies
  - Eerste versie van de klasse
    - Willekeurige long
  - Volgende versies zelfde nummer

```
Pi task = new Pi(Integer.parseInt(args[1]));
BigDecimal pi = comp.executeTask(task);
```

  - Code klasse Pi wordt door server opgeladen

Ontwikkelen RMI-applicatie

- Voorbeeld
- Server
  - Remote interface(s)
  - Remote objects: klassen schrijven
- Client
- Compilatie
- Applicatie starten

Compilatie

- 3 pakketten
  - compute → compute.jar
    - Interfaces Compute en Task
    - Gebruikt door server en client
  - engine
    - Klasse ComputeEngine (impl. Compute)
  - client
    - Klasse ComputePi en Pi (impl.Task)
- Klassen die moeten gedownload worden
  - In publiek toegankelijk netwerkmap (.class-bestanden of jar)

Code downloaden

- Code downloaden in RMI
  - maakt gebruik van bestaande URL-protocols in Java (bv. HTTP)
- Eenvoudige webserver nodig op client en server
- Te downloaden klassen
  - Publieke netwerkmap

Ontwikkelen RMI-applicatie

- Voorbeeld
- Server
  - Remote interface(s)
  - Remote objects: klassen schrijven
- Client
- Compilatie
- Applicatie starten

Applicatie starten

- Beveiliging
- RMI-register starten
- Server-programma starten

Beveiliging

- server.policy
- client.policy
- Lokale class-bestanden
  - Alle permissies
- Gedownloade bestanden
  - Geen rechten

RMI-register starten

- Programma
  - rmiregistry
    - CLASSPATH mag niet ingesteld zijn
  - Windows
    - start rmiregistry
  - Linux
    - rmiregistry &
- Standaardpoort
  - 1099

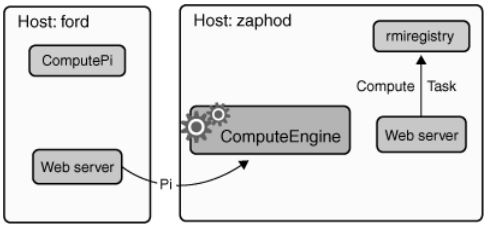
Server-programma starten

- Classpath
  - Lokale klassen
- Systeemeigenschappen instellen
  - Map waarin te downloaden klassen staan
    - java.rmi.server.codebase
  - Eventueel hostname te gebruiken door clients
    - java.rmi.server.hostname
  - policy-bestand
    - java.security.policy
  - Hoe instellen?
    - Optie –D van het java-commando

Client-programma starten

- Classpath
  - Lokale klassen
- Systeemeigenschappen instellen
  - Map waarin te downloaden klassen staan
    - java.rmi.server.codebase
  - policy-bestand
    - java.security.policy
  - Hoe instellen?
    - Optie –D van het java-commando
- Commando-lijn opties
  - Naam van de server
  - Aantal cijfers na komma

Overzicht: te downloaden klassen



Bron: java.sun.com

Informatie

- Java tutorial (http://java.sun.com/docs/books/tutorial/rmi/)

Bijlage A

Lijst veelgebruikte netwerkfuncties

```
int socket(int domain, int type, int protocol)
    maakt socket descriptor aan
    <sys/socket.h>
    return :  -1 bij fout
    vb. int sd = socket(AF_INET, SOCK_STREAM, 0);

int connect(int sd, sockaddr **adres, unsigned int addrlen)
    maakt connectie met een server
    <sys/socket.h>
    return : -1 bij fout, 0 bij succes
    vb. if (connect(sd, (sockaddr *)(&sa), sizeof(sockaddr_in)==0)

int bind(int lsd, sockaddr *adres, unsigned int adrlen)
    bindt een socket lsd aan de poort en ip-adres in adres
    <sys/socket.h>
    return : -1 bij fout, 0 bij succes
    vb. if (bind(lsd, (sockaddr *)(&sa), sizeof(sockaddr_in)==0)

int listen(int lsd, int aantal)
    luister naar connectie aanvragen op een socket
    <sys/socket.h>
    return : -1 bij fout, 0 bij succes
    vb. if (listen(lsd, 10)==0){ ... }

int accept (int connsd, sockaddr *adres, unsigned int adrlen)
    accepteer een connectie met een socket
    <sys/socket.h>
    return : -1 bij fout, 0 bij succes
    vb. if (listen(lsd, 10)==0){ ... }

unsigned long int htonl(unsigned long int host)
    converteert long van host naar network bytevolgorde
    <netinet/in.h>
    vb. sa.sin_addr.s_addr = htonl(0xc1bead01);

unsigned short int htons(unsigned short int)
    converteert short van host naar network bytevolgorde
    <netinet/in.h>
    vb. sa.sin_port = htons(7);

int inet_aton(const char *str, in_addr *p)
    zet dotted decimal notation om naar intern formaat
    <arpa/inet.h>
    return : 0 bij fout
```

```
    vb. inet_aton("127.0.0.1", &sa.sin_addr);

int inet_ntoa(in_addr p)
    zet intern formaat om naar dotted decimal notation
    <arpa/inet.h>
    return : 0 bij fout
    vb. inet_aton(sa.sin_addr); // vb.  "127.0.0.1"

hostent *gethostbyname(const char *naam)
    zet naam om naar intern ip adres, via DNS
    <netdb.h>
    return : 0 bij fout
    vb. :
hostent *he = gethostbyname("www.gonzo.be");
in_addr **p = (in_addr **)he->h_addr_list;
sa.sin_addr = **p;
```

Bibliografie

[APA] Apache http server project. http://httpd.apache.org/.

[Cha01] Steve Champeon. Javascript: How did we get here? http://www.oreillynet.com/pub/a/javascript/2001/04/06/js_history.html, June 2001.

[CSS] Cascading style sheets. http://www.w3.org/Style/CSS/.

[DOM] Document object model (dom). http://www.w3.org/DOM/.

[DYNa] Dhtml tutorial. http://www.w3schools.com/dhtml/default.asp.

[DYNb] Dynamic html. http://nl.wikipedia.org/wiki/Dynamic_HTML.

[Fla06] David Flanagan. *JavaScript: The Definitive Guide, Fifth Edition*. O'Reilly, August 2006.

[Gar05] Jesse James Garrett. Ajax: A new approach to web applications. February 2005. http://www.adaptivepath.com/ideas/essays/archives/000385.php.

[GOOa] Google maps. http://maps.google.com/.

[GOOb] Google suggest. http://www.google.com/webhp?complete=1&hl=en.

[HTM] Html 4.01 specification. http://www.w3.org/TR/html401/.

[IIS] Internet information services. http://www.iis.net/.

[ISO] Ecmascript language specification, iso/iec 16262. http://www.iso.org/iso/iso_catalogue/catalogue_tc/catalogue_detail.htm?csnumber=33835.

[MYS] Mysql. http://www.mysql.com/.

[PHPa] Php: A simple tutorial. http://www.php.net/manual/en/tutorial.useful.php.

[PHPb] Php: Hypertext preprocessor. http://www.php.net/.

[RAI] Ruby on rails. http://www.rubyonrails.com/.

[RUB] Ruby. http://www.ruby-lang.org/en/.

[Seb08] Robert W. Sebesta. *Programming the World Wide Web, 4th edition*. Pearson Education, 2008.

[WEB] Webdevelopment. http://nl.wikipedia.org/wiki/Webdevelopment.

[XHT] Extensible hypertext markup language (xhtml). http://www.w3.org/MarkUp/.

[XMLa] Extensible markup language (xml). http://www.w3.org/XML/.

[XMLb] The xmlhttprequest object. http://www.w3.org/TR/XMLHttpRequest/.

[XSL] The extensible stylesheet language family (xsl). http://www.w3.org/Style/XSL/.