FACULTEIT INGENIEURSWETENSCHAPPEN
EN ARCHITECTUUR

Gedistribueerde toepassingen

Bert De Saffel

Master in de Industriële Wetenschappen: Informatica Academiejaar 2018–2019

Gecompileerd op 1 januari 2019

UNIVERSITEIT
GENT

Inhoudsopgave

Deel I

Theorie

Hoofdstuk 1

Extensible Stylesheet Language Family

1.1 Inleiding

XSL staat voor *Extensible Stylesheet Language Family* en is een standaard om XML-documenten te presenteren en te transformeren. De drie componenten van XSL zijn:

- **XSLT** (Extensible Stylesheet Language Transformations) : dit is een XML-taal dat XML-documenten kan omvormen naar opnieuw XML, maar kan ook HTML, LaTeX, JSON, enz... zijn.
- **XPath** : dit is een taal dat waarmee bepaalde stukken van een XML-document kunnen gemanipuleerd worden aan de hand van het opbouwen van paden.
- XSL-FO (XSL Formatting Objects)

1.2 XSLT

1.2.1 Structuur

Een XSLT-bestand moet altijd voldoen aan volgende structuur:

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet version="1.0"
        xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
        <xsl:output method="html"/>
        <xsl:template match="patroon">
        ...
        </xsl:template>
        ...
</xsl:stylesheet>
```

1. `xsl:stylesheet` is het basiselement
2. `xsl:output` is het element dat het type transformatie vastlegt in het attribuut *method*

3. `xsl:template` bevat wat er moet gebeuren voor elk element dat aan *patroon* voldoet.

1.2.2 Instructie-elementen

- **xsl:value-of** : Bepaalt de waarde van een XPath uitdrukking.

  `<xsl:value-of select='uitdrukking'/>`

- **apply-templates** : Zal de templates van de onmiddelijke kindelementen van het huidig element toepassen. Indien er geen templates zijn, zal de inhoud van het element naar de uitvoer uitgeschreven worden. Het optionele *select* attribuut zal enkel de templates uitvoeren voor elk element die aan de uitdrukking voldoet.

  `<xsl:apply-templates [select='uitdrukking']/>`

- **xsl:for-each** : Voert een actie uit voor alle knopen die door het *select*-attribuut bepaalt wordt. Binnen de for-each kunnen meerdere instructie-elementen voorkomen.

  `<xsl:for-each select'uitdukking'>...</xsl:for-each>`

- **xsl:sort** :Dit is een kindelement van *xsl:apply-templates* of een *xsl:for-each* element. Dit element zal de knopen sorteren die aan de waarde van het select attribuut voldoen. Optionele attributen zijn

  - data-type : legt het datatype vast (*text* of *number*). Dit heeft een invloed op de sorteervolgorde (bij *text* is 10 kleiner dan 2).
  - order : bepaalt de sorteervolgorde (*ascending* of *descending*).

  `<xsl:sort select='uitdrukking' [data-type='text|number'] [order='ascending|descending]/>`

- **xsl:if** : Dit is een typische selectiestructuur. Het bevat als enig attribuut *test* met als waarde een predicaat.

  `<xsl:if test='logische uitdrukking'> ... </xsl:if>`

- **xsl:choose** : Een alternatieve selectiestructuur equivalent zoals een switch in programeertalen zoals Java en C#. Dit element wordt gevolgd door één of meerdere **xsl:when** elementen. Een xsl:when element is equivalent met een xsl:if element en heeft ook als enig attribuut *test*.

```
<xsl:choose>
        <xsl:when test="logische uitdrukking 1">
                ...
        </xsl:when>
        <xsl:when test="logische uitdrukking 2">
                ...
        </xsl:when>
        ...
</xsl:choose>
```

- **xsl:text** : Wanneer tekst spaties bevat is het aan te raden om xsl:text te gebruiken zodat deze ook opgenomen worden in het resultaat. Spaties die niet in een xsl:text element staan worden genegeerd.

  `<xsl:text>stukje tekst</xsl:text>`

- **xsl:variable** : Legt een constante variabele vast. Het verplichte attribuut is *name*, wat de naam van de variabele vastlegt. Het invullen van deze waarden kan op twee manieren :

  1. via het select attribuut `<xsl:variabele name='varnaam' select='uitdrukking'/>`

2. via de inhoud van het element: `<xsl:variabele name ="varnaam"> ... </xsl:variable>` In dit geval kan de waarde ook opgebouwd worden uit meerdere instructies.

- **xsl:param** : Indien dit element gedeclareerd wordt in het begin van een xls:template element, dan is het mogelijk om parameters mee te geven aan dit template. De declaratie kan op dezelfde manier gebeuren als bij xsl:variabele.
- **xsl:with-param** : Dit element is een kindobject van xsl:apply-templates. Dit element vult voor een bepaalde *name* de waarde die in *select* staat.

```
<xsl:apply-templates ...>
        <xsl:with-param name='varnaam' select='uitdrukking'>
</xsl:apply-templates>
```

Hoofdstuk 2

XML Path Language (XPath)

XPath is een taal om knopen te selecteren van een XML-document. Het resultaat van een XPath-uitdrukking kan een getal, string of logische waarde zijn.

2.1 Paden

Belangrijke XPath uitdrukkingen zijn paden. Een pad identificeert nul, één of meerdere knopen in een XML-document. Een pad bestaat uit *location steps* die verbonden zijn met een /. Bij elk pad zijn volgende wildcards mogelijk:

- *: selecteert elk elementknoop in de huidige context.
- node(): selecteert alle knopen, dus niet enkel elementen
- @*: selecteert alle attributen in de huidige context.

2.1.1 Basispad (Root Location Path)

Het eenvoudigste pad is `/` en selecteert het root-element van het XML-document. Verder is dit een absoluut pad, dus deze uitdrukking zal **altijd** het root-element teruggeven.

2.1.2 Kinelement-stap (Child Element Location Steps)

Dit pad bestaat uit de naam van een element en selecteert alle elementen met deze naam in de huidige context. De context is afhankelijk van de ouder.

```
<xsl:template match ="author">
        <xls:apply-templates select="name" />
</xls:template>
```

In dit voorbeeld wordt de XPath uitdrukking van het selectattribuut `/author/name/`.

2.1.3 Attribuut-stap (Attribute Location Steps)

Een attribuut selecteren begint met een @ en wordt gevolgd door de naam van het attribuut.

```
<xsl:template match ="query">
        <xls:value-of select="@isbn"/>
</xls:template>
```

2.1.4 Voorwaarden

```
<xsl:apply-templates select="//book[title='XML']//name[.='Ongenae]"/>

<xsl:apply-templates select="//person[@born<=1976]"/>
```

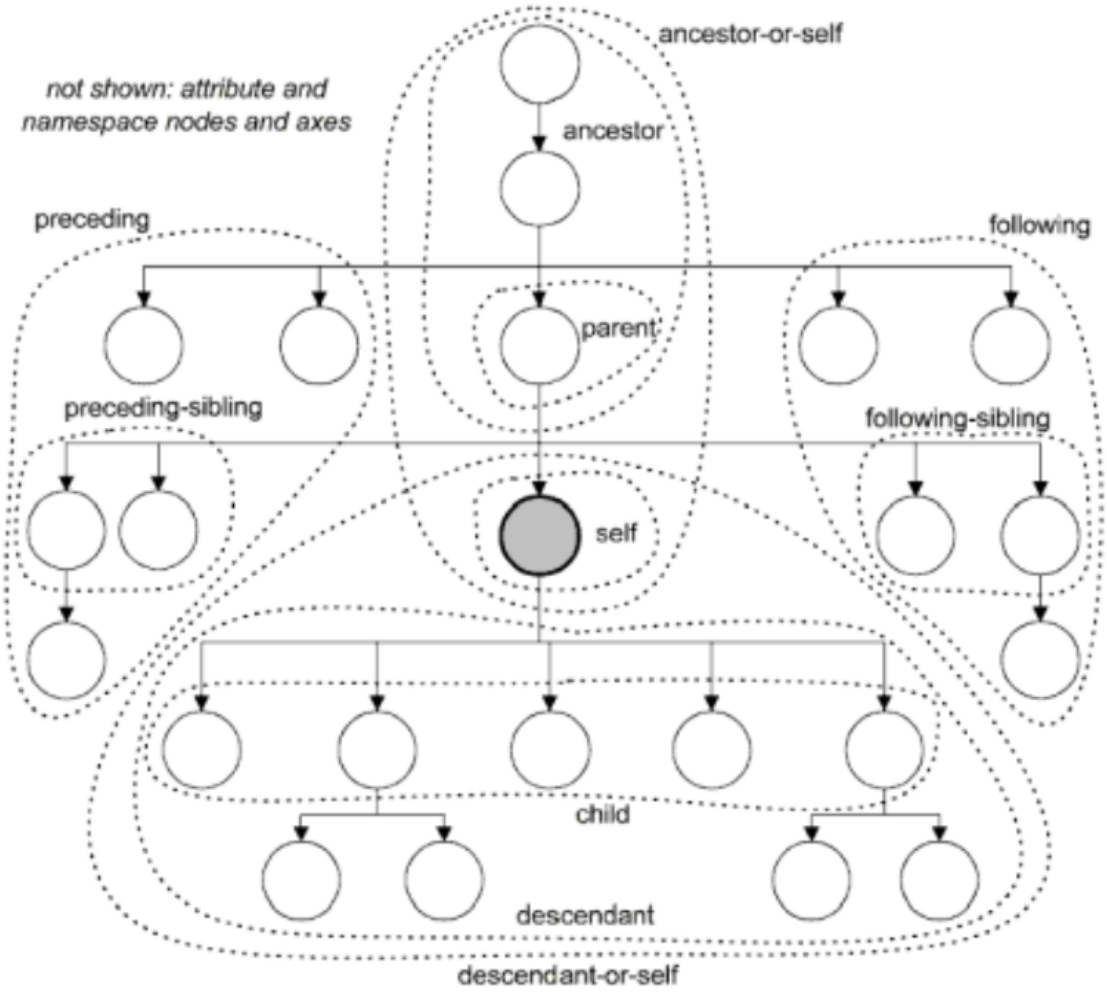
2.1.5 Lange padnamen

In elke XPath uitdrukking wordt elke stap voorafgegaan door de child:: as.

`books/book/isbn => child::books/child::book/child::isbn`

andere assen:

- parent (of ..)
- self (of .)
- descendant-or-self (of //)
- ancestor
- ancestor-or-self
- namespace
- descendant
- following-sibling
- preceding-sibling
- following
- preceding



2.2 Andere XPath-uitdrukkingen

XPath kan ook getallen, logische waarden en strings manipuleren.

2.2.1 Datatypes

- **Getallen** : Stel dat *born* het geboortejaar van een persoon is, dan zal de volgende uitdrukking de eeuw uitschrijven van dit jaar. `<xls:value of select="(@born - (@born mod 100)) div 100 + 1"`
- **Strings** : Strings staan tussen enkele of dubbele aanhalingstekens. Strings vergelijken kan met = en != operatoren.
- **Logische waarden** : De sleutelwoorden true en false bestaan niet in XPath, wel worden ze respectievelijk voorgesteld door de functies true() en false(). Op logische waarden kunnen de operatoren *and* en *or* toegepast worden. Met not(...) wordt de negatie toegepast.

2.3 Functies

XPath kent een aantal functies.

- **Knopenfuncties** zoals *position()*, *last()*, *count(...)* en *id(...)* bepalen respectievelijk
  - het volgnummer van de huidige knoop in de context,
  - het aantal knopen in de huidige context (volgnummer van de laatste knoop),
  - telt het aantal knopen van het argument, dat een verzameling knopen is en
  - bepaalt een verzameling knopen. Deze verzameling bevat alle elementen van een XML-document met één van de gespecifieerde ID's. Het argument is een string bestaande uit ID's gescheiden door spaties.
- **Stringfuncties** zoals *concat(...)*, *contains(...)*, *starts-with(...)*, *string(...)* (conversie naar string) en *string-length(...)*.
- **Logische functies** zoals *true()*, *false()*, *not(...)*, *boolean(...)* (conversie naar bool waarde).
- **Numerieke functies** zoals *ceiling(...)*, *floor(...)*, *number(...)*, *round(...)*.

2.4 Groeperen met de methode van Meunchian

Groeperen op een bepaald attribuut kan op twee manieren.

- Zonder `xsl:key` te gebruiken: Je kan in een variabele alle elementen steken die geen broeren hebben, die hiërarchisch voor hun liggen, met dezelfde waarde voor dat attribuut:

```
<xsl:variable
  name='varnaam'
  select='element[not(attr = preceding-sibling::element/attr)]'
/>
```

  De variabele bevat nu het eerste element voor elke verschillende waarde voor het attribuut. Om nu de verschillende elementen op te vragen, die dezelfde waarde hebben voor het gegroepeerde attribuut, kan je volgende as gebruiken:

```
element[attr = current()/attr]
```

  Deze methode is uiteraard niet performant, aangezien elke knoop eerst al zijn voorgaande broeren moet overlopen en daarna moet elke knoop overlopen worden om de elementen met de huidige waarde voor het attribuut op te zoeken.

- Door `xsl:key` te gebruiken: Deze xsl instructie geeft een sleutel aan een element op basis van een één of meerdere attributen. Elementen die dezelfde waarde voor een attribuut hebben, krijgen dezelfde sleutel toegekend. Het `use` attribuut specifieert het attribuut waarop de sleutel gebaseerd moet worden.

```
<xsl:key match='element' use='attribuut' name='sleutelnaam'/>
```

  Nu kan eenvoudig alle elementen opgehaald worden die een bepaalde waarde voor het attribuut hebben:

```
<xsl:for-each select="key('sleutelnaam','attribuutwaarde')">
```

Men kan ook alle elementen gegroepeerd ophalen, zonder expliciet de attribuutwaarde mee te geven, door nog een buitenlus te definieëren die elke waarde van het attribuut achterhaald door enkel de eerste tegenkomst van die waarde te selecteren:

```
<xsl:for-each select='element[count(.  | key('..', attr)[1]) = 1]">
```

waarbij `attr` de naam van het attribuut is en `..` (wegens plaatsgebrek) de naam van de sleutel.

Hoofdstuk 3

Webservices

Een gedistribueerd systeem zal over verschillende servers APIs beschikbaar hebben. Tabel 3.1 geeft de voornaamste verschillen tussen een monolitische architectuur en een gedistribueerde architectuur.

	Monolitisch	Gedistribueerd
Communicatie	Tussen processen, gedeeld geheugen	Over een netwerk
Globale toestand	Mogelijk	Niet mogelijk
Globale tijd	Mogelijk via lokaal besturingssysteem	Niet mogelijk
Fouten	Eenvoudig te ontdekken	Gedeeltelijk falen moeilijk te ontdekken
Locatie	Alle componenten op één machine	Variabel
Beveiliging	Taak van het besturingssysteem	Inherent kwetsbaar door networkcommunicatie.

Tabel 3.1: Monolitisch versus gedistribueerd.

Er zijn twee belangrijke componenten bij gedistribueerde systemen :

- **EDI (Electronic Data Interchange)**
- **RPC (Remote Procedure Call)**.

3.1 Componenten

3.1.1 EDI

Dit soort toepassingen digitaliseren het papierwerk dat vaak plaatsvindt in eender welke organisatie. Dit werkt in drie stappen:

1. Verzamel de informatie die verzonden moet worden. Deze data komt vaak uit interne bronnen, die een willekeurige structuur kunnnen hebben?
2. Zet deze informatie om naar het EDI formaat door gebruik te maken van vertaler die de interne informatie kan omzetten naar het EDI formaat.
3. Verstuur deze informatie naar de bestemmeling. Een zender en ontvanger zijn direct aangesloten met elkaar via software.

3.1.2 RPC

RPC is het aanroepen van een procedure (functie, methode, ...) op een ander toestel dan waarop een programma fysiek uitgevoerd wordt. Dit proces bevat volgende stappen:

1. De client maakt een RPC aan en verstuurt deze naar een server die deze call can behandelen.
2. De server verwerkt de procedure en stuurt het resultaat terug naar de client.
3. De client kan gebruik maken van dit resultaat op lokaal niveau.

3.2 Gedistribueerde objectsystemen

Gedistribueerde systemen maken gebruik van EDI en RPC om een systeem te implementeren. Er worden enkele besproken zoals **RMI (Remote Method Invocation)**, **DCOM (Distributed Component Object Model)** en **CORBA (Common Object Request Broker Architecture)**

3.2.1 RMI

Deze technologie is ontwikkelt door Oracle en bestaat typisch uit een **Server** en **Client** applicatie. Een server zal methoden beschikbaar stellen en zal wachten tot dat een client deze methode uitvoert (invocation). RMI is het mechanisme dat de verbinding en informatieuitwisseling tussen een client en een server behandelt. Een gedistribueerd systeem gebasseerd op RMI kan de volgende 3 zaken uitvoeren.

1. *Lokalisatie van remote objecten.* Remote objecten moeten opgeslagen worden in een zogenaamde **RMI registery**.
2. *Communicatie met remote objecten.* Het aanroepen van methoden van een remote object wordt volledig door RMI behandelt. Dit heeft als gevolg dat een programmeur niet eens moet weten dat hij met een remote object bezig is.
3. *Klassedefinities laden van remote objecten.* Aangezien dat RMI objecten kan terugsturen/versturen, voorziet RMI de mogelijkheid om klassedefinities van deze objecten op te halen.

3.2.2 DCOM

DCOM is een uitbreiding op COM om communicatie tussen verschillende toestellen, hetzij op een LAN, hetzij op een WAN of zelfs op het internet aan te spreken. DCOM behandelt het low-level gedeelte van de netwerkcommunicatie, zodat een programmeur hier zich niet meer mee bezig hoeft te houden.

3.2.3 CORBA

CORBA is zoals de anderen, ook een manier om communicatie te hebben tussen verschillende systemen. CORBA baseert zich op 4 punten:

1. Applicatieobjecten die specifiek zijn voor de applicatie. Een CORBA object is een encapsulatie van zulke objecten.

2. Een **ORB (Object Request Broker)** dat de requests van verschillende services behandelt.
3. Een verzameling van services die waarschijnlijk gebruikt zullen worden door veel applicaties.
4. Een verzameling van componenten die bovenop deze services gebouwd zijn.

3.3 MOM

MOM (Message Oriented Middleware) is een software systeem dat instaat voor het aanmaken, versturen, ontvangen en lezen van berichten op zowel synchrone of aysnchrone manier. Een client applicatie moet dus enkel deze middleware aanspreken om berichten op te halen of te versturen.

3.4 SOA

SOA (Service Oriented Architecture) is een architectuurstijl dat gebruik maakt van services die onafhankelijk zijn van andere services. Een service kan individueel aangepast worden zonder dat andere services hiervan last hebben. Deze architectuurstijl kent een aantal uitdagingen:

- Gelijktijdige toegang tot bronnen
- Wat indien een deel van de opdracht mislukt?
- Wat als één partner incompatibel wordt?
- Geen gedeeld geheugen voor requester en provider
- Traagheid en onbetrouwbaarheid gebruike transport

Er blijkt ook een gelijkheid te zijn met SOA en microservices. Het verschil tussen microservices en SOA is _ToDo: saai

3.5 Webservices in Java

De java klasse die een SOAP aanvraag moet verwerken wordt geannoteerd met

```
@WebService(serviceName = "...")
```

Deze klasse bevat een aantal methoden, die geannoteerd worden met

```
@WebMethod(operationName = "...")
```

De waarde van `operationName` in de SOAP aanvraag moet overeenkomen met één van de methoden in de webservice. Elke methode kan eventuele parameters hebben, geannoteerd met

```
@WebParam(name = "...")
```

Een eenvoudige webservice dat een boek geeft op basis van een ISBN, ziet er als volgt uit:

```
@WebService(serviceName = "Catalogus")
public class Catalogus {
  @WebMethod(operationName = "geefBoek")
   public Boek geefBoek(@WebParam(name = "isbn") String isbn){
```

```
    try {
      BoekenLijst boeken = new BoekenLijstImpl();
      return boeken.geefBoek(isbn);
    } catch(Exception e) {
      return null;
    }
  }
}
```

De bijhorende SOAP aanvraag wordt:

```
<?xml version="1.0" encouding="UTF-8"?>
<S:Envelope
  xmlns:S="http://schemas.xmlsoap.org/soap/envelope/">
  <S:Header/>
  <S:Body>
    <ns2:geefBoek xmlns:ns2="http://web.boeken.iii.be/">
      <isbn>isbn2</isbn>
    </ns2:geefBoek>
  </S:Body>
</S:Envelope>
```

Het antwoord dat we terugkrijgen is dan (bijvoorbeeld):

```
<?xml version="1.0" encouding="UTF-8"?>
<S:Envelope
  xmlns:S="http://schemas.xmlsoap.org/soap/envelope/">
  <S:Header/>
  <S:Body>
    <ns2:geefBoekResponse xmlns:ns2="http://web.boeken.iii.be/">
      <return>
        <isbn>isbn2</isbn>
        <prijs>36.0</prijs>
        <titel>Java Servlet Programming</titel>
      </return>
        </ns2:geefBoekResponse>
  </S:Body>
</S:Envelope>
```

3.6 Webservices in C#

Webservics in C# kunnen geïmplementeerd worden met **Windows Communication Foundation (WCF)**. WCF is een .NET framework om servicegeoriënteerde applicaties te ontwikkelen. De communicatie verloopt tussen endpoints via berichten. Een service bestaat uit meerdere endpoints. De architectuur wordt eerst weergegeven op figuur 3.1 en bestaat uit vier lagen.

- **Contracts:** Deze laag specificeert onder andere welke parameters er verwacht worden, de structuur van het bericht, de beschikbare interfaces/methodes en hoe er gecommuniceerd moet worden.
- **Service runtime:** Deze laag specificeert het gedrag van de applicatie zoals het aantal berichten dat verwerkt kan worden, wat er moet gebeuren als er iets fout loopt, welke metadata er

beschikbaar moet zijn voor de buitenwereld en hoeveel instanties er van deze service mogen zijn.

- **Messaging:** Deze laag is opgebouwd uit kanalenn, ook wel de *channel stack* genoemd. Een kanaal verwerkt berichten en komen in twee vormen voor:
  - Transport kanaal: Verantwoordelijk voor het lezen en schrijven van berichten van en naar het netwerk.
  - Protocol kanaal: Implementeert message processing protocols door bijkomende headers te lezen en schrijven.
- **Activation and hosting:** Een service kan enerzijds als een executable (self-hosted service) uitgevoerd worden en anderzijds gehost worden in een externe omgeving als webserver of Windows service.

3.6.1 Ontwerp servicecontract

Het ontwerp van het servicecontract is de eerste stap. Een methode krijgt het `[OperationContract]` attribuut (equivalent met annotaties in Java). Een methode die dit attribuut heeft, zal nooit referenties teruggeven, maar kopieën van de objecten. Het is ook mogelijk om eigen datatypes te definieren met het `[DataContract]` en `[DataMember]` attribuut. De werkelijke interface krijgt het `[ServiceContract]` attribuut. Een voorbeeld:

```
[ServiceContract(Namespace = "...")]
public interface ICalculator {
        [OperationContract]
        double Add(double x, double y);
        [OperationContract]
        double Subtract(double x, double y);
        [OperationContract]
        double Multiply(double x, double y);
        [OperationContract]
        double Divide(double x, double y);
}
```

De normale communicatie methode, waarbij de cliënt een vraag stelt waarop de server antwoord, is mogelijk. Ook is er een one-way contract, waarbij de cliënt geen antwoord verwacht van de server, en een duplex contract, waarbij de server zal ook informatie kan ophalen van clients. Een one-way contract is eenvoudig te definieren door de eigenschap `IsOneWay` van het attribuut `[OperationContract]` op `True` te zetten.

```
[OperationContract(IsOneWay = true)]
void Hello(string greeting);
```

Een duplex contract moet gedefinieerd worden in het `[ServiceContract]` attribuut in de server. In de client moet er ook een callback voorzien worden.

```
        // SERVER
[ServiceContract
        (
        Namespace = "...",
        SessionMode = SessionMode.Required,
        CallbackContract = typeof(ICalculatorDuplexCallback)
        )
```

```
]
public interface ICalculatorDuplex {
        [OperationContract(IsOneWay = true)]
        void Clear();
        [OperationContract(IsOneWay = true)]
        void AddTo(double n);
        [OperationContract(IsOneWay = true)]
        void SubtractFrom(double n);
        [OperationContract(IsOneWay = true)]
        void MultiplyBy(double n);
        [OperationContract(IsOneWay = true)]
        void DivideBy(double n);
}
```

```
        // CLIENT
public interface ICalculatorDuplexCallback {
        [OperationContract(IsOneWay = true)]
        void Result(double result);
        [OperationContract(IsOneWay = true)]
        void Equation(string eqn);
}
```

3.6.2 Implementatie contract

Als voorbeeld van de implementatie nemen we de standaardcommunicatiemethode. Na de vorige stap hebben we een interface ontworpen die er als volgt uit ziet:

```
[ServiceContract]
public interface IMath {
        [OperationContract]
        double Add(double x, double y);
        [OperationContract]
        double Multiply(double x, double y);
}
```

De implementatie van deze interface is vrij eenvoudig:

```
public class MathService : IMath {
        public double Add(double x, double y) {
                return x + y;
        }
        public double Multiply(double x, double y){
                return x * y;
        }
}
```

3.6.3 Configuratie

De configuratie specificeert per endpoint welke service (interface en implementatie) gebruikt moet worden, waar (URL) deze zich bevindt en hoe (binding) er moet gecommuniceerd worden. Een endpoint specificeren kan op twee manieren:

△ In code:

```
Uri base = new Uri("http://localhost:8080/Service");
ServiceHost host = new ServiceHost(typeof(MathService), base);
try {
        selfHost.AddServiceEndPoint(
                typeof(IMath),
                new WSHttpBinding(),
                "MathService");
        // gedrag instellen, service runnen
} catch (CommunicationException ce){
        Console.WriteLine(ce.Message);
        host.Abort();
}
```

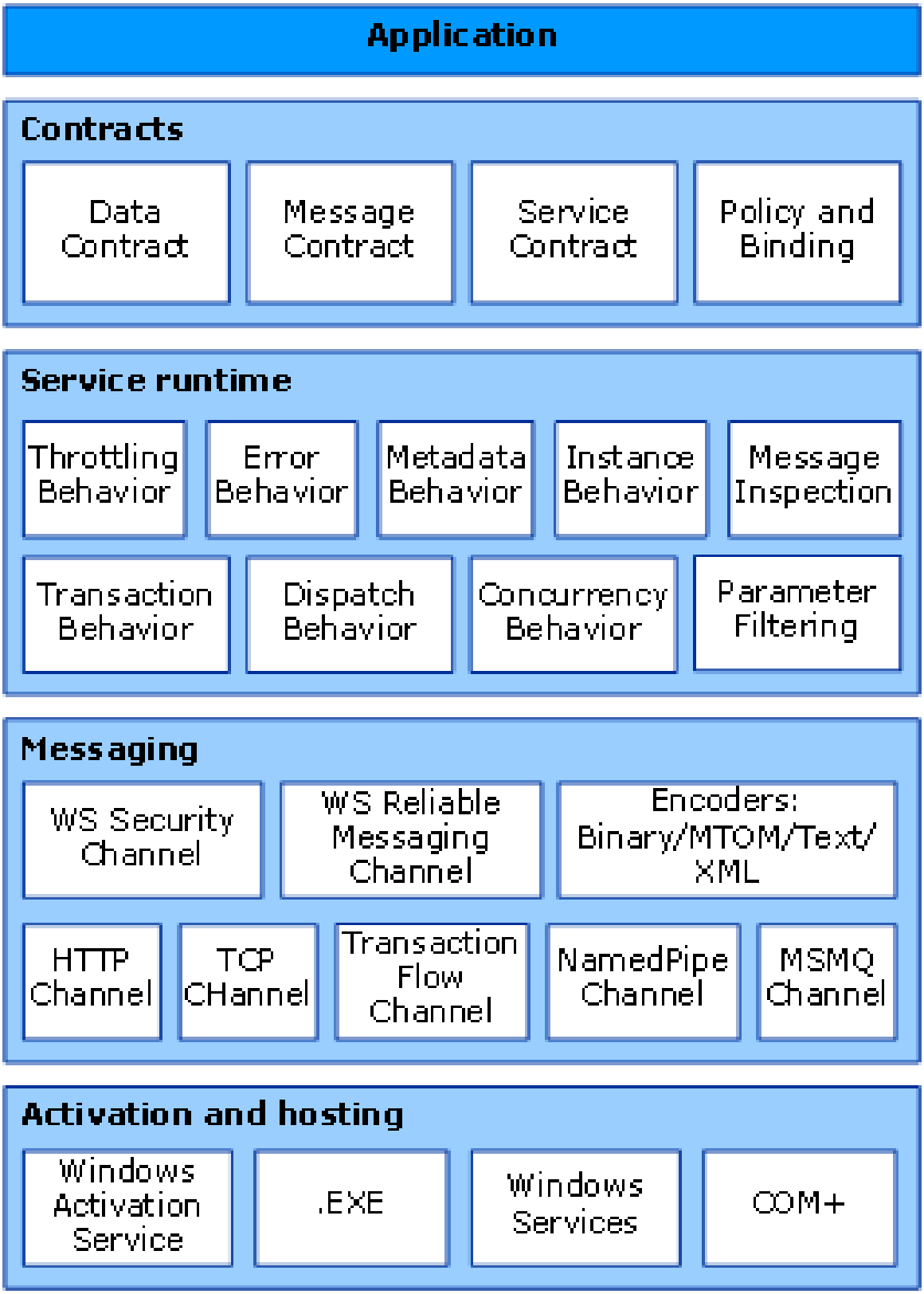
△ In configuration:

```
<system.ServiceModel>
  <services>
    <service name ="x.y.z.MathService">
      <endpoint address="http://localhost:8080/Service"
                binding="basicHttpBinding"
                bindingConfiguration="customConfiguration"
                contract = "x.y.z.IMath"/>
    </service>
  </services>
  <bindings>
    <basicHttpBinding>
      <binding name="customConfiguration"
               closeTimeout="00:01:00"/>
    </basicHttpBinding>
  </bindings>
  <behaviors>
    <serviceBehaviours>
      <behaviour>
        <serviceMetadata httpGetEnabled="True"/>
      </behaviour>
    </serviceBehaviours>
  </behaviors>
</system.ServiceModel>
```

Er zijn een aantal bindings voorzien zoals `BasicHttpBinding`, `WSHttpBinding`, `NetTcpBinding` en `NetNamedPipeBinding`.

3.6.4 Hosting

Je kan ofwel de serivce publiceren _ToDo: holy shit dit is retarded



Figuur 3.1: Architectuur WCF.

Hoofdstuk 4

API Gateways

Het publiceren van meerdere services blijft een uitdagen. Services kunnen onderling verschillende protocollen gebruiken en meerdere clients moeten de service kunnen aanspreken. Een webapplicatie kan bijvoorbeeld meer data krijgen dan een mobiele applicatie, omdat er meer plaats is om deze informatie te tonen. Een API ontwerpen die voor alle type clients zou werken is niet zinvol, en praktisch onrealiseerbaar. Men zou elke client de service rechtstreeks kunnen laten aanspreken. Dit komt erop neer dat de client de rol van API composer krijgt. Dit heeft drie nadelen:

- ! **Slechte gebruikerservaring:** De client moet complexe code bevatten om de resultaten van de APIs te combineren. Meerdere calls naar verschillende services vertraagd de aanvraag enorm.
- ! **Backend is niet afgeschermd:** Een wijziging in de backend vereist een wijziging in de frontend. Het uitrollen van een nieuwe versie van de service en de daarbijhorende client applicatie, wordt belemmerd. Niet iedereen gebruikt namelijk de nieuwste versie van elk product. Externe partners hebben ook nood aan een stabiele API.
- ! **Services kunnen IPC-mechanismen gebruiken:** Het aanroepen van diverse protocollen wordt niet altijd ondersteund in de client.

De oplossing is de API gateway, dat gebaseerd is op het facade patroon zodat de interne structuur afgeschermd wordt. De API gateway is het toegangspunt van de applicatie en is verantwoordelijk voor:

- Δ Request routing.
- Δ API-compositie, aggregatie/samenvoegen resultaten.
- Δ Vertaling naar protocollen geschikt voor de client.
- Δ Randfunctionaliteiten: authenticatie, autorisatie, caching, logging, ...

De client stuurt slechts één enkele request naar de API gateway, waarop de API gateway één enkel resultaat terugstuurt. De API gateway zal intern één of meerdere services aanspreken. De API gateway voorziet voor elke client een API.

Hoofdstuk 5

Netwerkprogrammatie in Java

Belangrijke netwerkklassen in java, in de package `java.net`:

- **TCP**
  - URL
  - URLConnection
  - Socket
  - ServerSocket
- **UDP**
  - DatagramPacket
  - DatagramSocket
  - MulticastSocket

5.1 Sockets

De cliënt maakt een `Socket` object aan. De constructor heeft de locatie van een bepaalde server, en een poortnummer, die de server heeft opengezet (cfr. 5.1).

```
Socket socket = new Socket("localhost", 8901);
```
(5.1)

Communicatie met de socket gebeurt met de `PrintWriter` klasse, die naar de server zal sturen, en de `BufferedReader` klasse, die berichten van de server zal ontvangen. Code 5.2 toont hoe deze objecten geïnitialiseerd kunnen worden.

```
  PrintWriter out = new PrintWriter(socket.getOutputStream(), true);
BufferedReader in = new BufferedReader(new InputStreamReader(socket.getInputStream()));
```
(5.2)

Gebruik best de try-met-resources om instanties van deze objecten te maken, zodat ze automatisch verdwijnen indien de code uit het `try` block komt. Na de initialisatie kan er met de server gecommuniceerd worden door weg te schrijven via de `PrintWriter`. Wachten op een bericht van de server kan door de `readLine()` methode van de `BufferedReader`klasse op te roepen. Dit is een synchrone methode dus de applicatie zal blokkeren.

De server zal gebruik maken van een `ServerSocket` klasse. Deze werkt op identiek dezelfde manier als de `Socket` klasse die de cliënt gebruik, behalve dat een instantie van de `ServerSocket` klasse meerdere sockets kan accepteren. De constructor van deze socket heeft als enige argument de poort die opengesteld moet worden (cfr 5.3).

```
ServerSocket socket = new ServerSocket(8901);
```
(5.3)

Tot slot is er nog een protocol nodig, die de communicatie tussen de server en communicatie beschrijft. Een eenvoudig protocol vereist bijvoorbeeld dat een bericht start met een bepaald sleutelwoord.

```
// CLIENT
String hostName = "localhost";
int portNumber = 4444;
try(
    Socket socket = new Socket(hostName, portNumber);
    PrintWriter out = new PrintWriter(socket.getOutputStream(), true);
    BufferedReader in = new BufferedReader(
        new InputStreamReader(socket.getInputStream())
    );
){
    String input = ... // input ophalen van gebruiker
    while(!input.equals("exit")){
        out.println(input); // naar server
        System.out.println("echo: ) + in.readLine()); // van server
        input = ... // input ophalen van gebruiker
    }
} catch (IOException ex) {
    System.err.println(ex.getMessage());
}

// SERVER
try {
    ServerSocket socket = new ServerSocket(4444);
    while(true) {
        try(
            PrintWriter out = new PrintWriter(
                socket.getOutputStream(), true
            );
            BufferedReader in = new BuffereadReader(
                new InputStream(socket.getInputStream())
            );
        ){
            String input, output;
            while((input = in.readLine()) != null){
                output = protocol.processInput(input);
                out.println(output);
                if(output.equals("Bye")){
                    break;
                }
            }
        }
    }
```

```
}
```

5.2 Multithreading

5.2.1 Threads

Een socket wordt best geïmplementeerd met behulp van threads, zodat verschillende cliënts op hetzelfde moment behandeld kunnen worden. Er zijn een aantal opties om multithreading te implementeren.

- **De interface `Runnable` implementeren.** Dit vereist dat de methode `run()` geïmplementeerd wordt. Het object moet dan verpakt worden in een `Thread`, zodat het kan uitgevoerd worden (cfr. 5.4) met de `start()` methode van de klasse `Thread`.

```
public class HelloRunnable implements Runnable {
        @Override
        public void run() {
                System.out.println("Hello from a thread!");
        }
}
```

```
(new Thread(new HelloRunnable())).start();
```
(5.4)

- **Overerven van de klasse `Thread`.** Om nu eigen functionaliteit aan te bieden, kan de methode `run()` overschreven worden. De thread kan nu direct gestart worden met de `start()` methode (cfr. 5.5).

```
public class HelloThread extends Thread {
        @Override
        public void run() {
                System.out.println("Hello from a thread!");
        }
}
```

```
(new HelloThread()).start();
```
(5.5)

- **Inline definitie.** Deze laatste methode zal een anonieme `Runnable` instantie aanmaken, die dan op exact dezelfde manier kan gestart worden zoals (cfr. 5.4).

```
public static void main(String args[]){
        Runnable task = () -> {
                System.out.println("Hello from a thread!");
        };
        new Thread(task).start(); // cfr. 5.4
}
```

5.2.2 Executors

Een Executor is verantwoordelijk voor het beheren en aanmaken van threads. De statische factoryklasse `Executors` laat toe om verschillende typen executors te initialiseren. De eerste twee voorbeelden geven een `ExecutorService` object terug, het laatste een `ScheduledExecutorService` object:

- `Executors.newSingleThreadExecutor();` Maakt gebruik van slechts één enkele thread. Indien de thread afsterft wordt er een nieuwe aangemaakt die zijn taken overneemt.
- `Executors.newFixedThreadPool(10);` Maakt een thread pool dat een vast aantal threads *her*gebruikt. De parameter is het aantal threads dat in de thread pool zullen zitten.
- `Executors.newScheduledThreadPool(10);` Maakt een thread pool waarmee threads na een bepaalde delay worden uitgevoerd.

De `ExecutorService` kan op meerdere manieren threads starten:

- `void execute(Runnable r);`

  Zal een `Runnable` object uitvoeren en verwacht geen returnwaarde.

- `Future<?> submit(Runnable r);`

  Analoog met de `execute()`-methode, enkel is het returntype nu een `Future` object die de taak voorstelt. Een `Future` object is een object speciaal ontworpen voor asynchrone operaties. Het object bevat een aantal methodes zoals `isDone()`, om na te gaan of dat de taak voltooid is en `get()`, om de waarde op te halen.

- `<T> Future<?> submit(Callable c);`

  Deze methode verwacht kan ook een instantie van type `Callable` krijgen. De functionaliteit is compleet analoog met `Runnable`, enkel kan een instantie van `Callable` een returnwaarde geven. Deze versie geeft een `Future` object met de returnwaarde van de overschreven methode `call()` van het `Callable` object.

- `<T> List<Future<T>> invokeAll(Collection<? extends Callable<T>> tasks);`

  Een lijst van `Callable` objecten uitvoeren. Het returntype is een lijst van `Future` objecten, die voor elke thread het returnresultaat bijhouden.

- `<T> T invokeAny(Collection<? extends Callable<T>> tasks);`

  Zal ook een lijst van `Callable` objecten uitvoeren, maar enkel het resultaat van de thread die het eerst klaar is wordt teruggegeven.

Hoofdstuk 6

Java NIO

Java NIO (New IO) biedt een alternatieve manier om met IO te werken.

- **Channels en Buffers.** Data wordt altijd gelezen van een bepaald kanaal naar een buffer, of geschreven van een buffer naar een kanaal.
- **Non-blocking IO.** Een thread kan aan een kanaal vragen om data van een buffer in te lezen. Terwijl het kanaal dit doet, kan de thread andere zaken uitvoeren en wachten tot het kanaal klaar is.
- **Selectors.** Een selector is een object that meerdere kanalen kan monitoren. Het reageert op events (connectie opened, data arrived, ...). Een enkele thread kan meedere kanalen monitoren met behulp van selectors. Een selector registreren gebeurt door een kanaal mee te geven, en de `select()` methode op te roepen van de `Selector` klasse.

De kern van Java NIO zijn de klassen `Channel`, `Buffer` en `Selector`. Andere klassen zoals `Pipe` en `FileLock` dienen gebruikt te worden in samenwerking met de drie kernklassen. Elke klasse kent meerdere implementaties. Zo kent de `Channel` klasse bijvoorbeeld:

`FileChannel`, `DatagramChannel`, `SocketChannel` en `ServerSocketChannel`

De klasse `Buffer` kent meerdere implementies afhankelijk van het datatype:

`ByteBuffer`, `CharBuffer`, `DoubleBuffer`, `FloatBuffer`, `IntBuffer`, `LongBuffer`, `ShortBuffer`

De verschillen tussen de `Channel` en `Stream` klasse zijn:

- Lezen en schrijven is mogelijk met één kanaal. Een stream is ofwel leesmode of schrijfmode.
- De lees en schrijfoperatie kan asynchroon gebeuren met kanalen.
- Een kanaal kan lezen van, of schrijven naar een buffer

Het gebruik van een eenvoudig `Channel` object:

```
RandomAccessFile file = new RandomAccessFile("data/file.txt", "rw");
FileChannel inChannel = file.getChannel();
ByteBuffer buffer = ByteBuffer.allocate(48);
int bytesRead = inChannel.read(buf);
while(bytesRead != -1){
```

```
    System.out.println("Read " + bytesRead);
    buf.flip();
    while(buf.hasRemaining()){
        System.out.print((char) buf.get());
    }
    buf.clear();
    bytesRead = inChannel.read(buf):
}
file.close();
```

6.1 Buffers

Om een buffer te verkrijgen, moet er altijd geheugen gereserveerd worden met de `allocate(int n)` methode. De waarde n is het aantal bytes. De `read(Buffer buffer)` methode van een `Channel` object zal naar de buffer schrijven. Het returntype van deze methode is het aantal bytes dat geschreven wordt, of -1 indien er niets geshreven wordt. De `flip()` methode van een `Buffer` object zal de buffermodus van schrijven naar lezen omzetten, of omgekeerd. Een buffer in leesmode laat toe om de methode `get()` van de buffer te gebruiken, of de `write()`-methode van een `Channel` object. De naamgeving van de methoden is hier verwarrend:

- `FileChannel.read(Buffer buffer)`: Zal data VAN het kanaal NAAR de buffer schrijven. Dit vereist dat de buffer in schrijfmode staat.
- `FileChannel.write(Buffer buffer)`: Zal data NAAR het kanaal VAN de buffer schrijven. Dit vereist dat de buffer in leesmode staat.

Na het uitlezen van de buffer moet deze opnieuw ingesteld worden met de `clear()`-methode, zodat de buffer klaar staat om opnieuw naar geschreven te worden.

Het is ook mogelijk om handmatig data in de buffer te plaatsen, zonder behulp van een `Channel`. De `put()`-methode (en al de varianten zoals `putInt`, ...) laat toe om rechtstreeks bytes te schrijven naar de buffer.

Terminologie:

- **Capaciteit:** Een buffer is in werkelijkheid een blok van geheugen en heeft daarom een vaste grootte, hier capaciteit genoemd. Eens de buffer vol is, kan er niet meer naar geschreven worden en moet dan ofwel gelezen worden, ofwel gereset worden.
- **Positie:** De positie hangt af van de buffermode:
  - Schrijfmode: Data schrijven naar een buffer gebeurd op een bepaalde positie. De positie wordt bij elke toevoeging (byte, int, long, ...) verplaatst naar de volgende schrijfbare geheugenlocatie van de buffer. De positie kan maximaal 1 minder dan de capaciteit worden.
  - Leesmode: Na het schrijven wordt de `flip()`-methode opgeroepen, die de positie terug op 0 zal zetten. Bij elke leesoperatie zal de positie zich verplaatsen naar de volgende leesbare geheugenlocatie.
- **Limiet:** Ook hier wordt er een onderscheidt gemaakt afhankelijk van de buffermode:
  - Schrijfmode: De limiet is gelijk aan de capaciteit.

– Leesmode: Vooraleer de methode `flip()` opgeroepen wordt, heeft de capaciteit een bepaalde waarde. Na de `flip()`-methode zal de limiet gelijk zijn aan deze capaciteit. Het komt erop neer dat er maar zoveel bytes mogen gelezen worden als er geschreven zijn.

Bij berichten die een header en een body hebben met vaste grootte, kan het handig zijn om scattering reads te gebruiken. Een kanaal kan schrijven naar een array van buffers. Deze variant zal de buffers in sequentie overlopen. Wanneer een buffer vol is, zal het de volgende buffer gebruiken.

```
ByteBuffer header = ByteBuffer.allocate(128);
ByteBuffer body   = ByteBuffer.allocate(1024);

ByteBuffer[] bufferArray = { header, body };
channel.read(bufferArray);
```

Op analoge manier kan ook de data van meerdere buffers rechstreeks in één kanaal geschreven worden. Deze manier werkt wel voor dynamische grootten, aangezien er maar zoveel bytes geschreven zal worden als de positie aangeeft van elke buffer.

```
ByteBuffer header = ByteBuffer.allocate(128);
ByteBuffer body   = ByteBuffer.allocate(1024);

// data naar buffer schrijven

ByteBuffer[] bufferArray = { header, body };
channel.write(bufferArray);
```

6.2 Channels

Het is mogelijk om rechtstreeks `FileChannel` objecten te verplaatsen met de `transferTo()` en `transferFrom()` methoden.

```
RandomAccessFile fromFile = new RandomAccessFile("fromFile.txt", "rw");
FileChannel fromChannel = fromFile.getChannel();

RandomAccessFile toFile = new RandomAccessFile("toFile.txt", "rw");
FileChannel toChannel = toFile.getChannel();


toChannel.transferFrom(fromChannel, 0, fromChannel.size());
// of
fromChannel.transferTo(0, fromChannel.size(), toChannel);
```

Op die manier kan men voorkomen dat men constant bestanden moet kopiëren.

6.3 Selectors

Een selector kan meerdere kanalen monitoren, en nagaan of ze klaar zijn om naar geschreven of van gelezen te worden. Een selector kan aangemaakt worden met de `open()` methode van de statische klasse `Selector` (cfr. 6.1).

```
Selector selector = Selector.open();
```
(6.1)

Vooraleer een kanaal kan geregistreerd worden moet deze in non-blocking mode staan (cfr. 6.2).

```
channel.configureBlocking(false);
```
(6.2)

Uiteindelijk kan het kanaal regegistreerd worden aan de selector (cfr. 6.3). Bij de registratie moeten de operaties gespecificeerd worden die van belang zijn:

- Connect
- Accept
- Read
- Write

```
SelectionKey key = channel.register(selector, SelectionKey.OP_READ);
```
(6.3)

Een kanaal dat één van de vier events afvuurt is 'ready' voor dat event. Een kanaal meerdere events registreren, door de constanten met de OR operatie mee te geven. Een selector kan opvragen welke kanalen er 'ready' zijn voor de gespecificeerde events Het is eerst interessant om het aantal kanalen op te vragen, met één van de `select()`-methoden (cfr. 6.4).

```
int readyChannels = selector.select();
int readyChannels = selector.select(1000);
int readyChannels = selector.selectNow();
```
(6.4)

De eerste variant is een blocking methode die oneindig lang zal wachten, tot er minstens één kanaal klaar is. De tweede variant zal slechts 1 seconde (1000 milliseconden) blocken. De laatste variant blokkeert niet en zal direct het aantal kanalen teruggeven. De `select()`-methoden geven niet altijd het aantal kanalen dat effectief 'ready' is, maar enkel de kanalen die 'ready' geworden zijn tussen twee `select()` calls.

De beschikbare kanalen kunnen wel overlopen worden via de set gegeven door de `selectedKeys()`-methode. Na het overlopen van de kanalen moet de set leeggemaakt worden aangezien de selector dat niet zelf doet (cfr. 6.5)

```
selector.selectedKeys().clear();
```
(6.5)

Voorbeeld eenvoudig gebruik van een selector:

```
while(true){
        int readyChannels = selector.select(10000);
        if(readyChannels == 0){
                continue;
        }

        for(SelectionKey key : selector.selectedKeys()){
                if(key.isAcceptable()){

                } else if (key.isConnectable()){

                } else if (key.isReadable()){

                } else if (key.isWriteable()){

                }
        }
        selector.selectedKeys().clear();
}
```

Voorbeeld: `SocketChannel`

```
// aanmaken
SocketChannel socketChannel = SocketChannel.open();
socketChannel.connect(
        new InetSocketAddress("localhost", 9999)
);

// lezen
ByteBuffer buf = ByteBuffer.allocate(48);
int bytesRead = socketChannel.read(buf);

// schrijven
String newData = "New String to write to file...";
buf.clear();
buf.put(newData.getBytes());
buf.flip();
while(buf.hasRemaining()){
        socketChannel.write(buf);
}

// afsluiten
socketChannel.close();
```

Voorbeeld: `ServerSocketChannel`

```
ServerSocketChannel serverSocketChannel
        = SocketServerSocketChannelChannel.open();
serverSocketChannel.socket().bind(
        new InetSocketAddress(9999)
);

while(true){
        SocketChannel socketChannel = serverSocketChannel.accept();
        // ...
}

serverSocketChannel.close();
```

6.4 Java IO vs Java NIO

6.4.1 Streams vs Buffers

NIO is buffergeoriënteerd terwijl IO streamgeoriënteerd is. Een stream laat enkel toe om sequentieel te lezen, en kan niet meer teruggegaan worden, tenzij dit opgeslagen wordt in een buffer. Het voordeel van een buffer is dus dat het niet sequentieel afgelopen moet worden.

6.4.2 Blocking vs Non-blocking IO

De traditionele IO zijn blocking. Een thread moet wachten vooraleer de blocking methoden klaar zijn. De non-blocking manier van buffers laat toe om enkel de huidige beschikbare data van een

buffer op te halen, en zal niet blokkeren in het geval dat er nog data verwacht wordt. Dit heeft natuurlijk als gevolg dat een thread meerdere kanalen kan beheren.

6.4.3 Selectors

Dit is het direct gevolg van de asynchroniteit van buffers/kanalen. Een selector is de implementatie die toelaat om meerdere kanalen te monitoren met slechts één thread.

6.4.4 Verwerken van data

Stel dat we volgende inhoud willen lezen:

```
Name: Anna
Age: 25
Email: anna@mailserver.com
Phone: 1234567890
```

Bij een streamgeoriënteerde methode moet elke lijn één voor één ingelezen worden. De opeenvolging van `readLine()` methoden verzekert dat de vorige `readLine()` aanroep met succes voltooid is.

```
InputStream input = ...;
BufferedReader reader = new BufferedReader(
        new InputStreamReader(input);
)
String name = reader.readLine();
String age = reader.readLine();
String email = reader.readLine();
String phone = reader.readLine();
```

Bij een buffergeoriënteerde methode is de verwerking wat complexer. Aan elke buffer moet een vooropgelegde aantal bytes toegekend worden die de buffer kan bevatten. Het probleem dat hierbij komt is dat het onmogelijk wordt om te weten wat er in de buffer zit, en je telkens de buffer moet inspecteren.

```
ByteBuffer buffer = ByteBuffer.allocate(48);
int bytesRead = inChannel.read(buffer);
```

Het kan zijn dat de buffer slechts 'Name: An' bevat.

6.4.5 Overige topics

- **Change notifiers:** De `WatchService` klasse laat toe om eenvoudig wijzigingen in directory of bestand te monitoren. Dit is een verbetering dan het pollingsmechanisme dat gebruikt wordt door traditionele IO.
- **Selectors:** De I/O van NIO is eventgebaseerd. Een selector wordt pas wakker wanneer een bepaald event optreedt.
- **Channels:** Het gebruik van kanalen is niet altijd sneller. Een eenvoudig bestand inlezen kan twee tot drie maal performanter zijn met de traditionele IO operaties.

- **Memory Mapping:** NIO heeft een mechanisme dat ervoor zorgt dat een bestand eigenlijk gezien wordt als een blok geheugen, met als gevolg dat I/O nu even snel is als traditioneel geheugen uitlezen, in plaats van schijfoperaties. Het maakt het ook mogelijk om meerdere readers en writers op ditzelfde geheugen te laten werken.
- **Character encoding en zoeken:** De laatste feature van NIO zijn character encodings via de klasse `Charset`. Het laat eenvoudig conversie tussen charsets toe. Het is meer flexibel dan de traditionele `getBytes()`-methode, en is zeer bruikbaar bij het zoeken in teksten die niet in het Engels geschreven zijn (met rare tekens).

Hoofdstuk 7

Java Message Service

Java Message Service (JMS) is een asynchrone, betrouwbare berichtenservice. Verschillende componenten kunnen berichten sturen en ontvangen. Er zijn twee soorten berichtstijlen:

- **Queue:** Niet te verwarren met een effectieve queue datastructuur. Een messaging queue laat slechts één enkele subscriber toe en één enkele producent. Het is een point-to-point verbindingen waarbij de enige producer berichten op de queue zet, en de consumer (de enige mogelijke subscriber) deze berichten zoals een queue leest, first in first out.
- **Topic:** Een topic laat toe om meerdere subscribers te hebben. Een publisher publiceert een bepaald bericht en elke subscriber (meerdere mogelijk) kan dit bericht lezen. Dit is in feite gewoon het publisher-subscriber model. Elke subscriber bevat nog altijd een wachtrij van nog te lezen berichten, maar is fundamentaal anders dan als de Messaging Queue.

Om een connectie te maken met een JMS provider wordt de klasse `ConnectionFactory` gebruikt. Deze klasse wordt niet zelf geïnitialiseerd, maar met behulp van annotaties (cfr. 7.1).

```
@Resource(lookup = "java:comp/DefaultJMSConnectionFactory")
private static ConnectionFactory connectionFactory;
```
(7.1)

Hoe dat een `Queue` en `Topic` aangemaakt moeten worden, is gezien in de labo's. Een voorbeeld:

```
create-jms-resource --restype javax.jms.Queue customQueue
```

De queue (of topic) wordt ook met een annotatie geïnitialiseerd (cfr. 7.2).

```
@Resource(lookup = "jms/customQueue")
private static Queue queue;
```
(7.2)

Een producer kan een bericht op de queue zetten:

```
try (JMSContext context = connectionFactory.createContext();) {
        context.createProducer().send(queue, "een bericht");
} catch (JMSRuntimeException e) {
        //...
}
```

en een consumer kan dit bericht lezen:

```
try (JMSContext context = connectionFactory.createContext();) {
        context.createConsumer(queue);
        while(true){
                Message m = consumer.receive(1000); // om de seconde
                if(m != null && m instanceof TextMessage){
                        System.out.println(m.getBody(String.class));
                }
        }
} catch (JMSRuntimeException e) {
        //...
}
```