Een snel identificatie algoritme voor XML berichten

Auteur: Walter Hoolwerf
Begeleider: Rinus Plasmeijer
Scriptienummer: 602

29 maart 2009

Inhoudsopgave

1 Inleiding **5**

2 Achtergrond **6**
2.1 Communication Channel . . . 7
2.2 Adapter Module . . . 7
2.3 Sender Agreement . . . 7
2.4 Receiver Determination . . . 7
2.5 Interface Determination . . . 8
2.6 Interface Mapping . . . 8
2.7 Receiver Agreement . . . 8
2.8 Performance . . . 8
2.9 Correctheid . . . 9

3 Probleemstelling **10**
3.1 Doelen . . . 11
3.1.1 Identificeren . . . 11
3.1.2 Performance . . . 11
3.2 Probleemstelling . . . 12

4 Relevant onderzoek **13**
4.1 Parsen . . . 13
4.2 Classificeren van berichten . . . 13

5 Aanpak **15**

6 Casus **16**
6.1 Patiënt . . . 16
6.2 Labuitslag . . . 17
6.3 Labuitslag-nieuw . . . 17
6.4 Labuitslag wijzig . . . 18
6.5 Labuitslag verwijder . . . 19
6.6 Gebruik casus in verdere secties . . . 20

7 Extensible Markup Language (XML) **21**
7.1 Well formed . . . 21
7.2 XML Schema . . . 22
7.2.1 XML Schema Definition (XSD) . . . 22
7.2.2 Valide . . . 23
7.3 Volgorde . . . 23
7.4 XPath . . . 24

8 XI Adapter and Module Framework **26**

9 Identificatie van berichten **28**
9.1 Valideren bericht . . . 28
9.2 Kleinste aantal eigenschappen . . . 29
9.3 Identificerende eigenschap met kleinste diepte in boom . . . 30
9.4 Identificerende eigenschap met kleinste diepte in parser . . . 30
9.5 Identificerende eigenschap met kleinste diepte in XML-stream . . . 31

10 Parser 33
10.1 DOM parser . . . 33
10.2 Lazy parser . . . 33
10.3 Double lazy parser . . . 35
10.4 SAX parser . . . 36
10.5 Identificerende parser . . . 38

11 Implementaties 40
11.1 Parsers . . . 40
11.1.1 DOM parser . . . 40
11.1.2 Lazy XML Parser . . . 41
11.1.3 SAX Parser . . . 41
11.2 Identificerende eigenschappen . . . 42
11.3 Identificatie . . . 44
11.3.1 Valideren bericht . . . 44
11.3.2 XPath . . . 45
11.3.3 Kortste in XML stream . . . 47

12 Benchmark 50
12.1 Testopstelling . . . 50
12.2 Metingen . . . 50
12.3 Resultaten . . . 51
12.3.1 Validator . . . 51
12.3.2 XPath . . . 51
12.3.3 SAX . . . 51

13 Conclusie 53

Bibliography54

1 Inleiding

In deze scriptie worden verschillende methoden om XML berichten te identificeren voorgesteld en met elkaar vergeleken, om zo een meest optimale methode vast te stellen. Om te beginnen wordt de achtergrond waarin onze probleemstelling zich afspeelt beschreven, met daarna een exacte specificatie van onze probleemstelling.

In de direct daarop volgende secties wordt de aanpak, de doelen, en een casus beschreven. Hierna zal aandacht worden besteed aan verschillende stukken achtergrondinformatie zoals XML en XI, de message broker van SAP. Deze onderwerpen zijn niet direct onderwerp van het onderzoek, maar voor het begrijpen en oplossen van het probleem, is enige kennis over deze zaken wel nodig.

De twee opvolgende secties beschrijven verschillende manieren om berichten te identificeren, en verschillende manieren om berichten te parsen. Uiteindelijk komen deze twee onderwerpen samen in een proof of concept implementatie, waarop metingen worden verricht om de onderlinge snelheid te bepalen.

Als laatste rest dan nog de conclusie, waarin we kort bespreken in hoeverre de beoogde doelstelling zijn behaald met de verschillende voorgestelde methoden.

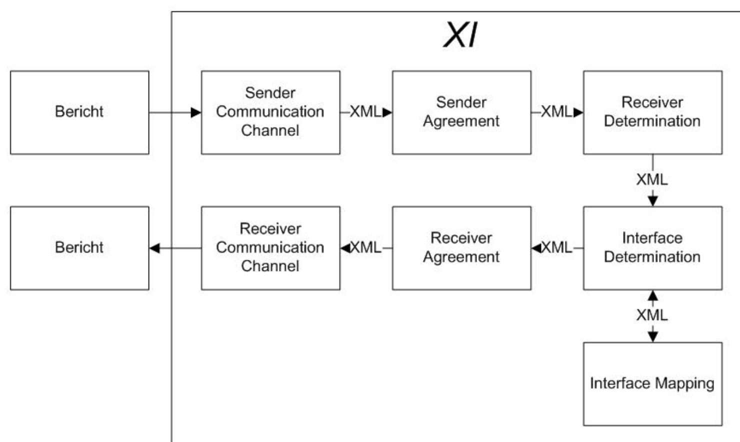
2 Achtergrond

In een modern software landschap staan meerdere softwaresystemen, welke onderling met elkaar informatie uitwisselen. Zo heeft een ziekenhuis doorgaans een centraal ziekenhuis informatie systeem (ZIS), maar draait het laboratorium binnen zo'n ziekenhuis vaak op een gespecialiseerd laboratorium informatie systeem (LIS). Deze systemen communiceren onderling om patiënt informatie, labaanvragen en meetresultaten uit te wisselen. Wanneer het om slechts enkele software systemen gaat, die elkaars "taal" (berichten standaard) spreken, dan kan deze koppeling direct opgezet worden.

In de praktijk gaat het echter om meer dan enkele systemen, welke vaak niet dezelfde standaard, of een andere variant/dialect/versie van dezelfde standaard ondersteunen. Wanneer dit het geval is, wordt vaak een Message Broker ingezet.

Een Message Broker ontvangt een bericht van een extern systeem, kan dit bericht zo nodig vertalen naar een ander berichtformaat en stuurt dit door naar het externe doelsysteem [25]. Deze manier van werken heeft enkele voordelen. In de eerste plaats hoeven systemen elkaars standaarden niet te ondersteunen: de Message Broker kan berichten vertalen. Verder kan een Message Broker een bericht dat binnen komt van een extern systeem, doorsturen naar meer dan een doelsysteem. Een bericht vanuit een ZIS waarin een nieuw ingeschreven patient wordt vermeld, is immers van belang voor alle patient gerelateerde systemen binnen een ziekenhuis. Als laatste wordt het onderhouden en monitoren van de communicatie een stuk gemakkelijker: omdat alle communicatie via een centrale Message Broker gaat, is de monitoring van de berichtenstroom binnen alleen die Message Broker vaak voldoende om problemen snel op te merken.

De grootste software leverancier binnen de Enterprise Resource Planning (ERP) markt is SAP. Een SAP landschap bestaat uit verschillende systemen, bijvoorbeeld een basis ERP systeem, een speciaal Human Resource (HR) systeem, een Business Warehousing (BW) systeem, maar uiteraard ook systemen van andere leveranciers. Binnen een SAP landschap, of een landschap waar SAP software een grote rol vervult, wordt vaak voor de SAP Message Broker , Exchange Infrastructure (XI) genaamd [26] [13], gekozen. Nieuwere versies van deze Message Broker heten Process Integration (PI), maar de versie die wij gebruiken heet nog XI. XI stuurt een bericht wat het ontvangt door naar een ander systeem aan de hand van een aantal stappen:



Figuur 1: Grafische weergave van de verwerking van een bericht binnen XI

2.1 Communication Channel

Een Communication Channel binnen XI implementeert het feitelijke communicatie protocol met met een extern systeem. Standaard zijn er Communication Channels voor email (smtp, imap, pop3), ftp, filesystem, http, soap, SQL etcetera voorhanden. Een Communication Channel kan, gezien vanuit het externe systeem, een Sender Communication Channel of een Receiver Communication Channel zijn. Een Sender Communication Channel is een Communication Channel waarbij het externe systeem een bericht verstuurt. Dit is dus een bericht dat bij XI binnen komt en verwerkt moet worden. Een Receiver Communication Channel is een Communication Channel waarbij het externe systeem een bericht ontvangt. Dit bericht wordt dus door XI naar het externe systeem opgestuurd.

Wanneer over een Communication Channel een ander formaat dan XML [4] (zoals bijvoorbeeld bij een email Communication Channel of een file Communication Channel het geval kan zijn) wordt gecommuniceerd, dan is de Communication Channel verantwoordelijk voor het vertalen van dit formaat van en naar XML. Een Sender Communication Channel is ook verantwoordelijk voor het bepalen van het bericht type van een inkomend bericht.

2.2 Adapter Module

Een Communication Channel kan qua functionaliteit uitgebreid worden middels een of meerdere Adapter Modules. Zo'n Adapter Module krijg het bericht aangeboden door de Communication Channel en kan hier een bewerking op uitvoeren, bijvoorbeeld het vertalen van of naar XML. In het geval van een Sender Communication Channel krijgt een Adapter Module het bericht ter bewerking aangeboden voor het door de Communication Channel verzonden wordt. Bij een Receiver Communication Channel krijgt de Adapter Module het bericht nadat het door de Receiver Communication Channel ontvangen is.

Er kunnen ook meerdere Adapter Module's aangeroepen worden vanuit een Communication Channel. De volgorde kan door de ontwikkelaar worden bepaald.

2.3 Sender Agreement

Binnen XI wordt vastgesteld welke berichten externe systemen mogen versturen. Een Sender Agreement voor extern systeem X en bericht A geeft aan dat extern systeem X bericht A mag versturen. Bij een Sender Agreement wordt ook aangegeven op welke Sender Communication Channel dit verstuurd mag worden. XI accepteert alleen berichten op een Sender Communication Channel waarvoor een Sender Agreement voorhanden is.

2.4 Receiver Determination

Wanneer XI een bericht ontvangt op een Sender Communication Channel en er is voor dit bericht een geldige Sender Agreement voorhanden, dan moet XI bepalen naar welke extern systeem het betreffende bericht doorgestuurd moet worden.

Een Receiver Determination wordt aangemaakt voor een versturende partij en een berichttype. Een Receiver Determination bepaalt dus dat wanneer er van extern systeem X een bericht van type A wordt ontvangen, dit naar extern systeem Y doorgestuurd moet worden. Een Receiver Determination kan ook bepalen dat het bericht naar meerdere partijen doorgestuurd moet worden.

2.5 Interface Determination

Wanneer een Receiver Determination heeft bepaald dat een bericht van A doorgestuurd moet worden naar systeem Y, dan kan het zijn dat systeem Y berichten van type A niet ondersteunt. Een Interface Determination hangt samen met een Receiver Determination en bepaalt aan de hand van een berichttype en een ontvangende partij naar welk berichttype het bericht vertaald moet worden. Wanneer het bericht vertaald moet worden, dan gebeurt dit middels een Interface Mapping.

2.6 Interface Mapping

Wanneer een doelsysteem een bericht in een ander formaat (interface) moet ontvangen dan een bronsysteem het bericht verstuurt, moet het bericht vertaald worden. Dit gebeurt middels een Interface Mapping. Bij een interface mapping geef je op wat de broninterface en de doelinterface is. Verder geef je uiteraard op hoe de vertaling (mapping) plaats moet vinden. Dit gebeurt middels een grafische tool waarin beide (broninterface, doelinterface) XML structuren naast elkaar getoond worden en waarin je aan kunt geven welke waarden vanuit de bronstructuur waar in de doelstructuur terecht moeten komen. Verder kun je simpele bewerkingen op de data doen, zoals het wijzigen van het datumformaat, berekingen etc.

2.7 Receiver Agreement

Een Receiver Agreement is hetzelfde als een Sender Agreement, maar wordt gebruikt om aan te geven welke berichten een extern systeem allemaal kan ontvangen. Binnen een Receiver Agreement geef je op dat extern systeem X een bericht van type A kan ontvangen. Verder geef je ook op over welke Receiver Communication Channel systeem X dit bericht kan ontvangen.

2.8 Performance

Bij een Message Broker zijn twee zaken van belang bij performance: verwerktijd per bericht en doorvoer snelheid in totaal. Deze belangen kunnen tegenstrijdig zijn. Een Communication Channel plaatst een bericht, nadat deze correct ontvangen is van het externe systeem, op een interne queue. Hierna wordt het bericht weer van de interne queue gelezen om verder verwerkt te worden (aan de hand van Receiver Determination, Interface Determination etc).

Er is binnen XI een mogelijkheid om berichten niet per stuk, maar in een batch van meerdere berichten te verwerken. Deze feature heet binnen XI “message packaging”. Omdat je hierbij minder overhead hebt per bericht, verbetert de doorvoer. Echter, omdat het eerste binnengekomen bericht moet “wachten” tot de batch vol is en gedraaid wordt, is de verwerk tijd per bericht gemiddeld hoger.

Bij asynchrone communicatie is doorgaans voornamelijk de doorvoer snelheid van belang. Bij synchrone communicatie, waarbij vaak een gebruiker zit te wachten op resultaat van een query, is verwerktijd veel meer van belang.

Voor XI zijn geen duidelijke waarden beschikbaar waarin wordt gesteld hoeveel doorvoer en welke verwerktijd per bericht gehaald kan worden. Dit komt omdat dit van veel factoren afhankelijk is. Echter, SAP heeft wel een Sizing and Performance Guide [14], waarin de performance van XI onder verschillende omstandigheden wordt gemeten. Hier zijn waarden uitgekomen met betrekking tot totale doorvoersnelheid en verwerktijd per bericht, waarvan SAP stelt dat deze waarden representatief zijn voor reële situaties.

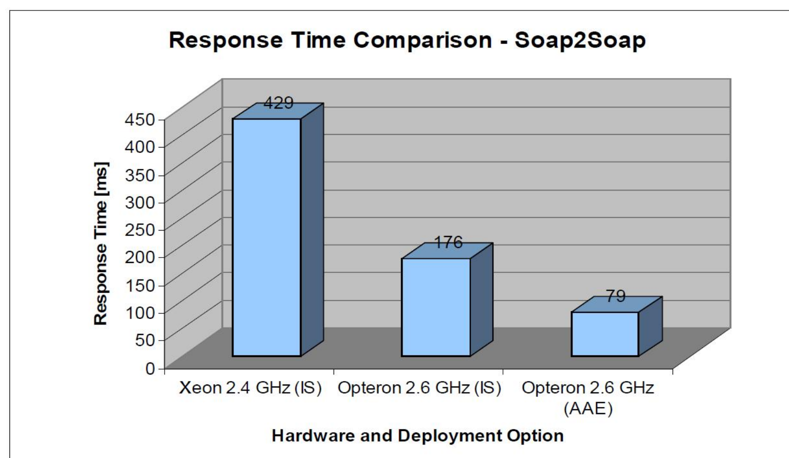
In deze metingen komt naar voren dat bij kleine berichten een doorvoer snelheid van 220 berichten per seconde en een datadoorvoer van 8.7GB per uur gehaald kan worden. Bij grote

berichten loopt de doorvoer snelheid terug naar 9 berichten per seconde, maar de datadoorvoer loopt dan wel op naar 78GB per uur. Al deze waarden zijn bij een voor doorvoer geoptimaliseerd XI systeem door gebruik te maken van message packaging.

Message size	11 KB	32 KB	245 KB	2.37 MB
Corresponding number of line items	2	10	100	1000
Expected number of messages with 66% system usage (as calculated by Quicksizer)	200,000 ~ 56 m/s	180,000 ~ 50 m/s	110,000 ~ 31 m/s	21,500 ~ 6 m/s
Calculated possible number of messages with 100% usage (without message packaging)	300,000 ~84 m/s	270,000 ~75 m/s	155,000 ~66 m/s	32,000 ~9 m/s
Number of messages with message packaging set on Integration Server to 10 seconds, 100 messages, and 1 MB maximum package size	790.000 = 220 m/s = 8,7 GB/h	630.000 = 175 m/s = 20,2 GB/h	207.000 = 58 m/s = 50,7 GB/h	
Number of messages with message packaging set on Integration Server to 10 seconds, 100 messages, and 5 MB maximum package size				Up to 33.000 = 9 m/s = 78 GB/h
Improvement factor with message packaging compared to expected values from Quicksizer	2.6	2.33	1.34	

Figuur 2: Meting van doorvoer afhankelijk van berichtgrootte [14]

Verder kan, door gebruik te maken van andere technieken, de response tijd van een synchrone call teruggebracht worden naar 79ms. Hierbij is de doorvoersnelheid zowel in berichten per seconden als in datadoorvoer in GB per uur niet gemeten, maar deze waarden zullen flink lager liggen dan bij voorgaande metingen waar message packaging aan stond.



Figuur 3: Meting van response tijd bij synchrone calls [14]

2.9 Correctheid

Belangrijk bij een Message Broker is dat de status van berichten altijd correct is en dat berichten niet verdwijnen of corrupt raken. Wanneer een bericht correct verwerkt is, dan moet de status van dat bericht ook op verwerkt staan, om dubbel verwerken te voorkomen. Andersom ook: wanneer een bericht niet correct verwerkt is, moet dit ook in de status terug te vinden zijn.

3 Probleemstelling

XI kan alleen van een Sender Communication Channel waarover een vast, van tevoren afgesproken, berichtenstandaard wordt gebruikt bepalen van welke type een inkomend bericht is. Voorbeelden van zulke Sender Communication Channel typen zijn iDoc en SOAP Communication Channels. Omdat over zo'n Sender Communication Channel een vooraf gedefinieerde standaard wordt gecommuniceerd, is in de Sender Communication Channel ingeprogrammeerd waar de Sender Communication Channel aan kan zien welk type bericht binnen is gekomen.

Wanneer echter gebruik wordt gemaakt van bijvoorbeeld een filesystem Sender Communication Channel, is dit niet mogelijk. Het enige dat bij het bouwen van XI bekend is, is dat het bericht van een filesystem afgehaald wordt. Welke inhoud in het bericht te verwachten is en hoe de Sender Communication Channel aan deze inhoud kan zien welk type bericht het is, is bij het bouwen van XI niet bekend. Je kunt een Sender Agreement aanmaken voor zo'n Sender Communication Channel, maar wanneer je een tweede probeert aan te maken voor dezelfde Sender Communication Channel, krijg je een foutmelding.

Figuur 4: Foutmelding bij aanmaken tweede sender agreement

XI geeft hier aan dat hij geen tweede Sender Agreement ondersteunt voor dezelfde Sender Communication Channel van dit type Communication Channel. Wat in feite gebeurt is dat XI alle inkomende berichten op het betreffende Sender Communication Channel identificeert als zijnde van het type bericht waarvoor de Sender Agreement is aangemaakt. Als deze Sender Agreement voor berichttype A is en er komt een bericht van type B binnen dan zal XI toch, foutief, het bericht als zijnde van type A identificeren. Dit levert uiteraard in de verdere verwerking van het bericht problemen op.

De reden voor dit merkwaardige gedrag is dat XI eigenlijk niet weet hoe het een bericht moet identificeren als zijnde van een bepaald berichttype. Bij veel Communication Channel types zoals SOAP, iDoc etc. wordt in de communicatie al verteld welk berichttype XI gaat ontvangen. Hierdoor hoeft XI dus zelf het bericht niet meer te identificeren. Echter, bij file channels, maar ook mail channels, JMS channels etcetera wordt alleen het bericht verstuurd, maar geen melding om wat voor type bericht het gaat. Verder is het ook nog goed mogelijk dat via zo'n Communication Channel een ander formaat dan XML verstuurd wordt, wat later door een Adapter Module pas vertaald wordt naar XML.

Een belangrijk punt bij XI en bij message brokers in het algemeen, is de reactiesnelheid bij synchrone calls. Om deze reactiesnelheid niet al te veel te beïnvloeden, is er een doel gesteld om berichten binnen 40ms te kunnen identificeren.

3.1 Doelen

3.1.1 Identificeren

Het identificeren zelf zal plaatsvinden in een Adapter Module. Een Adapter Module is een stukje software wat binnen een Communication Channel gedraaid kan worden. In het geval van bijvoorbeeld een file Communication Channel kan het zo zijn dat het bestand wat opgepakt wordt door de Communication Channel niet in XML formaat is. Het is dan nodig dit naar XML te vertalen voor het verder verwerkt kan worden. Het is handig hier een Adapter Module voor te gebruiken, omdat een of meer Adapter Modules naar keuze en in gewenste volgorde aangeroepen kunnen worden nadat een Communication Channel het bericht heeft ontvangen, maar voordat het bericht doorgegeven wordt aan de Sender Agreement. Voor iedere identificatie methode zal een Message Broker geschreven worden. Meer inhoudelijke details over Adapter Modules worden gegeven in het hoofdstuk “XI Adapter and Module Framework”.

Deze adapter modules zijn “read-only”. Dit betekent dat ze zowel de vorm als de inhoud van het te identificeren bericht niet mogen wijzigen. Het doel van deze modules is immers om het XI systeem te vertellen van welk berichttype een binnengekomen bericht is, niet om iets aan het bericht te wijzigen.

Verder moeten de modules altijd een uitkomst hebben: ook wanneer een bericht niet geïdentificeerd kan worden, moet dit terug gemeld worden.

3.1.2 Performance

Zoals eerder gezien behaalt XI een kortste verwerktijd van 79ms [14]. Hierbij is het dus van belang hoe het systeem geoptimaliseerd is (response tijd of throughput). Het is belangrijk dat onze identificatie module niet teveel afbreuk doet aan performance, voornamelijk op gebied van response-tijd. De grote hoeveelheid berichten per uur doorvoer wordt behaald door meerdere berichten tegelijk van een queue af te halen en parallel te verwerken en zo dus overhead te minimaliseren. Deze queue wordt gevuld door de Communication Channel. Omdat we hier een module ontwikkelen die op de Communication Channel functioneert, hebben we met deze message packaging niets van doen.

De verwerktijd binnen onze module moet echter wel opgeteld worden bij de totale verwerktijd van een bericht. Omdat deze verwerktijd minimaal 79ms is (bij optimalisatie voor response-tijd), is het doel om onze adapter module in onze testopstelling in maximaal 40ms zijn werk als identificator te laten doen. Dit is namelijk maximaal de helft van de “native” performance, waardoor de performance niet al te veel wordt beïnvloed.

3.2 Probleemstelling

Hoe kan er een automatische en dynamische manier ontwikkeld worden om zonder van tevoren (bij het bouwen van een Communication Channel) te weten wat voor berichttypen er verstuurd kunnen worden naar een XI systeem, toch runtime berichten correct en binnen de 40ms te identificeren.

4 Relevant onderzoek

Om de probleemstelling beter te begrijpen is het altijd zinnig om naar vergelijkbare problemen en bijbehorende literatuur te kijken.

Het lijkt voor de hand te liggen dat onze probleemstelling in de categorie van parsers ligt. Echter, nader onderzoek naar parsers leerde ons, zoals te zien in de sectie *Parsen*, dat een parser toch wezenlijk anders blijkt dan ons algoritme en daardoor de oplossing niet in bestaande parsers gezocht kon worden.

Een ander gebied van onderzoek wat interessant kan zijn, is het identificeren of classificeren van berichten. Dit gebeurt voor uiteenlopende doeleinden. De grootste zijn spam- en virusfilters, zoekmachines, maar ook spionage of gerelateerde doeleinden. De overeenkomsten en verschillen van deze onderzoeksgebieden in relatie tot onze probleemstelling, wordt beschreven in de sectie "classificeren van berichten".

4.1 Parsen

Parsers zijn er in veel soorten en maten en er is veel onderzoek naar gedaan. Alle parsers hebben echter gemeen dat ze een bericht of document ontrafelen aan de hand van een grammatica. Een grammatica in de Engelse taal geeft aan welke woorden op welke manier gecombineerd mogen worden.

Een boek welke een goed overzicht geeft van verschillende parsers is *New Developments in Parsing Technology* [9]. In dit boek worden allerlei parsers behandeld en gecategoriseerd.

Bij het parsen van een XML document is de grammatica uiteraard ook al bekend: deze staat netjes beschreven in de XML specificatie. XML staat echter voor Extensible Markup Language. Ook al is de basis van een XML document beschreven in een vaste grammatica (<element> </element>), is er ook nog een tweede laag grammatica, namelijk de naam van de elementen en hoe deze zich onderling verhouden.

Normaal bij het parsen van een XML document ga je er vanuit dat ook deze tweede laag grammatica bekend is. Deze staat namelijk normaal gesproken beschreven in een XSD document. Echter, in ons probleem is het vinden van het bijbehorende XSD document, danwel bijbehorende grammatica, danwel het juiste berichttype, juist de problematiek.

Waar een normale parser als input een document en een grammatica heeft en als output de gestructureerde inhoud van het document, heeft onze parser als input een document en een aantal mogelijk grammatica's, en als output de bijbehorende grammatica. Alle parsers die in het boek [9] beschreven staan of die in andere artikelen die ik heb gelezen beschreven staan zijn "normale" parsers die een document een grammatica verwachten en de inhoud van het document als resultaat opleveren. Deze parsers en de achterliggende technieken bleken daarmee ook niet toepasbaar op ons probleem.

4.2 Classificeren van berichten

Er is nog een ander onderzoeksgebied wat zich veel bezighoudt met het identificeren, of beter gezegd het classificeren, van informatie, namelijk information retrieval. Het vakgebied van information retrieval wordt veel gebruikt door zoekmachines, spamfilters en ook spionage instanties. Information retrieval richt zich op het zo goed mogelijk classificeren van informatie, aan de hand van eigenschappen in deze informatie.

Hoe hoger de score, hoe hoger de waarschijnlijkheid dat een bericht tot een bepaalde klasse behoort. Echter, in ons onderzoek was niet het classificeren van berichten, maar het met zekerheid

identificeren van berichten het doel. Waar bij een classificatie zoveel mogelijk classificerende kenmerken (sleutelwoorden, structuren, etc) bij elkaar opgeteld worden, is het bij het identificeren van XML berichten mogelijk om aan de hand van slechts één identificerende eigenschap met zekerheid te stellen dat een bericht van een bepaald type is. Zo'n identificerende eigenschap is daarbij een element, of een combinatie van elementen, welke maar in één van de mogelijke berichttypen voor komt.

Verderop in deze scriptie zullen we namelijk zien dat we een eigen algoritme ontwikkeld hebben die uit een set van grammatica's (XSD documenten) voor iedere grammatica de identificerende eigenschappen kan bepalen. Omdat dit mogelijk is, hoeven we dus niet meer te classifeceren, maar hoeven we enkel nog maar te kijken of we bij een bericht één van de identificerende eigenschappen aantreffen. Omdat deze eigenschap identificerend is (dat wil zeggen, hij komt slechts in één grammatica voor) kunnen we dus met zekerheid zeggen welke grammatica bij het binnen gekomen bericht hoort.

De methoden om scores toe te kennen aan bepaalde eigenschappen in een bericht, iets wat bij classificeren heel belangrijk is, zijn voor onze probleemstelling dan ook niet relevant. De manier van werken, namelijk het zoeken naar bepaalde eigenschappen in een bericht, hebben we echter wel overgenomen in onze oplossing.

5 Aanpak

Omdat XI intern met XML [4] werkt, is het nodig om eerst een goed beeld van XML en de mogelijkheden en onmogelijkheden van deze berichtenstructuur te krijgen. Daarna kijken we naar welke standaardmogelijkheden er in parsers ingebouwd zitten die we kunnen gebruiken voor het identificeren van een berichttype en waarom deze niet al gebruikt worden binnen XI.

Nadat we de standaard aanwezige mogelijkheden bekeken hebben, proberen we op verschillende manieren een betere methode te ontwerpen om berichten op een zo snel mogelijke wijze nauwkeurig te identificeren. Hiervoor ontwikkelen we een methode welke deze identificerende eigenschappen kan bepalen aan de hand van de verschillende berichttype specificaties en ontwikkelen we een methode welke op een zo efficiënt mogelijk manier zoekt naar identificerende eigenschappen in een willekeurig XML bericht. Aan de hand van deze methoden bouwen we een parser welke zo efficiënt mogelijk berichten identificeert.

De verschillende mogelijkheden worden in voorbeelden uitgewerkt aan de hand van een casus, welke zich in een ziekenhuis afspeelt. We beschrijven eerst deze casus, daarna de XML standaard, dan de verschillende manieren van identificeren en uiteindelijk de parser die hieruit voortkomt.

De parser wordt vergeleken met de bestaande mogelijkheden die we eerder hadden beschreven. De complexiteitsgraad van de verschillende oplossingen worden met elkaar vergeleken en de door ons ontwikkelde parser wordt in een praktische benchmark vergeleken met de standaard oplossingen.

6 Casus

We nemen voor deze casus een ziekenhuisomgeving. In deze ziekenhuisomgeving is een Ziekenhuis Informatie Systeem (ZIS) geïmplementeerd waar alle algemene informatie binnen het ziekenhuis in geadministreerd wordt. Voorbeelden van zulke algemene informatie zijn patiëntgegevens, arts gegevens, locatie gegevens binnen het ziekenhuis, de planning of agenda van artsen en medische gegevens van de patiënten. Alle medewerkers maken gebruik van dit ZIS om patiëntinformatie op te vragen of in te voeren, maar ook om uitslagen van onderzoeken te raadplegen, of om nieuwe onderzoeken aan te vragen.

Veel vooral medische informatie komt echter niet uit dit centrale ZIS, maar uit gespecialiseerde randsystemen. Zo maakt het laboratorium van dit ziekenhuis gebruik van een labsysteem genaamd LIS. Met LIS is het laboratorium grotendeels geautomatiseerd: een bloedmonster wordt in een machine gezet en de machine doet alle benodigde testen op dit monster. LIS analyseert automatisch de resultaten en stuurt deze daarna door naar het centrale ZIS om opgeslagen te worden.

Patiënten die vanuit een huisarts direct naar een lab voor bloedonderzoek worden doorverwezen staan nog niet altijd in het ZIS ingeschreven. In het LIS system kunnen patiënten ook ingeschreven worden. Het LIS stuurt direct alle patiënt informatie door naar het ZIS, zodat het daar centraal opgeslagen kan worden. De patiënt informatie is dan direct in het hele ziekenhuis beschikbaar en niet alleen binnen het lab.

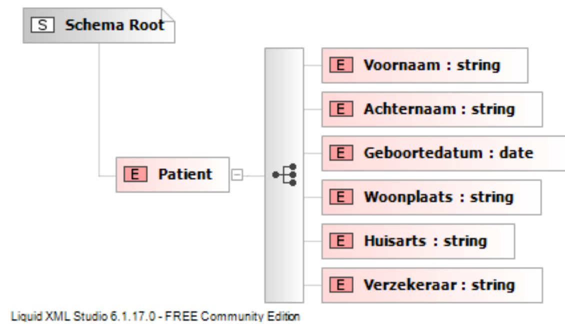
Het LIS verstuurt twee typen berichten, namelijk Patiënt of Labuitslag. Van alle berichtvarianten die we hieronder zien is zowel het XML Schema (XSD [15]) als een grafische weergave gemaakt met Liquid XML Studio [21] gegeven. De XSD is voor de volledigheid afgedrukt (XML schema wordt in detail behandeld in "Extensible Markup Language (XML)"), de grafische weergave is om makkelijker een goede indruk van de structuur te krijgen. Verderop in dit document geven we telkens alleen nog maar de XSD variant, of simpelweg een voorbeeld XML met een simpele tekstuele beschrijving.

6.1 Patiënt

Het bericht *Patiënt* is relatief eenvoudig. Het kent maar een variant welke simpelweg alle patiëntgegevens bevat. Elke bericht van het type *Patiënt* bevat gegevens over exact een patiënt.

Listing 1: XSD voor berichttype *Patiënt*

```
<?xml version="1.0" encoding="utf-8" ?>
<xs:schema elementFormDefault="qualified"
                  xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:element name="Patient">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="Voornaam" type="xs:string" />
        <xs:element name="Achternaam" type="xs:string" />
        <xs:element name="Geboortedatum" type="xs:date" />
        <xs:element name="Woonplaats" type="xs:string" />
        <xs:element name="Huisarts" type="xs:string" />
        <xs:element name="Verzekeraar" type="xs:string" />
      </xs:sequence>
    </xs:complexType>
  </xs:element>
</xs:schema>
```



Figuur 5: Grafische weergave van het berichttype *Patiënt*

Deze berichten bevatten wel alle patiënt informatie, maar geen patiëntID. Dit unieke nummer wordt alleen door het ZIS bij inschrijving toegekend. Het LIS stuurt dus de patiënt gegevens op, waarna het ZIS een patiëntID toekent en de patiënt inschrijft.

6.2 Labuitslag

Berichttype *Labuitslag* is een stuk complexer dan berichttype *Patiënt*. In feite zijn er drie varianten van *Labuitslag*: *Nieuw*, *Wijzig*, *Verwijder*. Elke variant heeft dezelfde root-tag, namelijk labuitslag. Ze bevatten echter wel alle drie andere elementen. Verder is aan de root-tag een attribuut gekoppeld, waarin staat om welke variant het gaat. Dit attribuut heet "variant".

Wanneer een nieuwe labuitslag beschikbaar komt, stuurt het LIS deze uitslag automatisch door naar het ZIS. Dit gebeurt middels een *Labuitslag-nieuw* bericht. Wanneer een ander systeem (bijvoorbeeld een analyse systeem) deze uitslagen nodig heeft, dan vraagt hij deze op bij het ZIS. Het ZIS is namelijk de centrale opslagplaats voor alle patiënt gerelateerde informatie. Wanneer echter een extern systeem alle uitslagen tot 1-1-2008 heeft van een bepaalde patiënt, dan zal dit systeem bij een volgende analyse alleen labuitslagen vanaf die datum opvragen.

Het komt echter voor dat er op het lab een fout ontdekt wordt. Dit kan een menselijke fout zijn, een systeemfout of een software fout. In dit geval kunnen labuitslagen achteraf gewijzigd worden of zelfs verwijderd worden. Ook deze *wijzig* en *verwijder* opdrachten worden naar het ZIS doorgestuurd.

Echter, wanneer het externe analyse systeem deze uitslagen al heeft opgevraagd bij het ZIS, zal deze die niet nog een keer opvragen. Het zal hem dus nooit bekend worden dat er wijzigingen of zelfs verwijderingen gedaan zijn en de foutieve gegevens blijven in de database zitten en gebruikt worden voor analyses. Het is daarom ook van belang dat *wijzig* en *verwijder* berichten door de Message Mapping niet alleen naar het ZIS, maar ook naar alle andere systemen doorgestuurd worden. Omdat dit slechts sporadisch (alleen in het geval van fouten) voorkomt, is dit qua performance geen probleem.

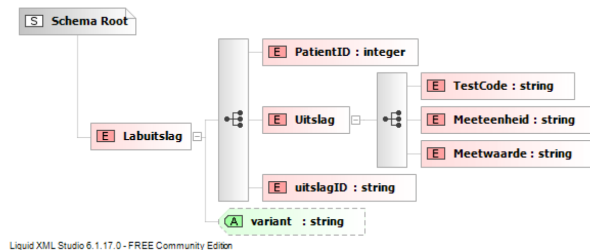
Dit betekent echter wel dat de Message Mapping onderscheid moet kunnen maken tussen *nieuw*, *wijzig* en *verwijder* berichten.

6.3 Labuitslag-nieuw

Het *Labuitslag* bericht variant *nieuw* bevat een root-tag *labuitslag*, met daaronder een element met het *patiëntID* waarvoor de uitslag bestemd is. Verder is er de *testcode* (welke bloedtest uitgevoerd is), een uniek nummer waarmee de uitslag later weer aangeduid kan worden (*uitslagID*), de meetwaarde en de meeteenheid. Het *uitslagID* nummer is uniek voor de patiënt.

Listing 2: XSD voor berichttype *labuitslag-nieuw*

```
<?xml version="1.0" encoding="utf-8" ?>
<xs:schema elementFormDefault="qualified"
                  xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:element name="Labuitslag">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="PatientID" type="xs:integer" />
        <xs:element name="Uitslag">
          <xs:complexType>
            <xs:sequence>
              <xs:element name="TestCode" type="xs:string" />
              <xs:element name="Meeteenheid" type="xs:string" />
              <xs:element name="Meetwaarde" type="xs:string" />
            </xs:sequence>
          </xs:complexType>
        </xs:element>
        <xs:element name="uitslagID" type="xs:string" />
      </xs:sequence>
      <xs:attribute name="variant" type="xs:string" />
    </xs:complexType>
  </xs:element>
</xs:schema>
```



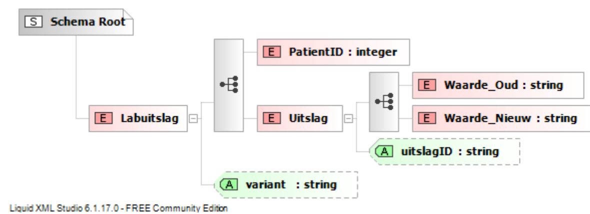
Figuur 6: Grafische weergave van het berichttype *labuitslag-nieuw*

6.4 Labuitslag wijzig

Een *Labuitslag* variant *wijzig* bericht bevat het *patientID* voor wie de uitslag origineel bedoeld was, de *uitslagID* van de uitslag waar het om gaat, de oude waarde en de nieuwe, geupdate waarde.

Listing 3: XSD voor berichttype *labuitslag-wijzig*

```
<?xml version="1.0" encoding="utf-8" ?>
<xs:schema elementFormDefault="qualified"
                  xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:element name="Labuitslag">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="PatientID" type="xs:integer" />
        <xs:element name="Uitslag">
          <xs:complexType>
            <xs:sequence>
              <xs:element name="Waarde_Oud" type="xs:string" />
              <xs:element name="Waarde_Nieuw" type="xs:string" />
            </xs:sequence>
            <xs:attribute name="uitslagID" type="xs:string" />
          </xs:complexType>
        </xs:element>
      </xs:sequence>
      <xs:attribute name="variant" type="xs:string" />
    </xs:complexType>
  </xs:element>
</xs:schema>
```



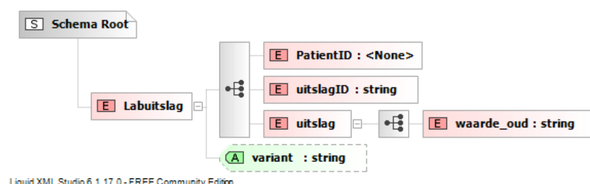
Figuur 7: Grafische weergave van het berichttype *labuitslag-wijzig*

6.5 Labuitslag verwijder

Labuitslag verwijder bevat alleen de *patientID* van de patiënt en de *uitslagID* van de uitslag die verwijderd moet worden.

Listing 4: XSD voor berichttype *labuitslag-verwijder*

```
<?xml version="1.0" encoding="utf-8" ?>
<xs:schema elementFormDefault="qualified"
                  xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:element name="Labuitslag">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="PatientID" />
        <xs:element name="UitslagVerwijderID" type="xs:string" />
      </xs:sequence>
      <xs:attribute name="variant" type="xs:string" />
    </xs:complexType>
  </xs:element>
</xs:schema>
```



Figuur 8: Grafische weergave van het berichttype *labuitslag-verwijder*

6.6 Gebruik casus in verdere secties

De voorbeelden in deze casus zijn erg uitgebreid. Dit is goed voor de beeldvorming, maar verderop gebruiken we versimpelde varianten die alleen de kern van het probleem wat op dat moment wordt behandeld illustreren:

Listing 5: Versimpelde casus voorbeelden

```
<root>
        <a></a>
        <b></b>
</root>
```

Om verderop in deze scriptie de verschillende onderwerpen zo goed mogelijk te illustreren zullen we telkens een zo goed passend voorbeeld gebruiken, zodat in een zo'n klein mogelijk XML of XSD document goed te zien is hoe een bepaalde eigenschap werkt. Hierdoor zullen de voorbeelden per keer verschillen. Echter, de in deze casus voorgestelde berichttypen bevatten alle eigenschappen die in onze voorbeelden van belang zijn. In ieder voorbeeld zijn dus ook de berichttypen zoals in deze casus beschreven te gebruiker en door de lezer zelf naar wens in te denken.

7 Extensible Markup Language (XML)

Extensible Markup Language (XML) [4] is een door het World Wide Web Consortium (W3C) [23] opgestelde standaard voor het opstellen van gestructureerde gegevensstructuren. Het is een markup language, wat betekent dat het een taal is welke met sleutelwoorden aangeeft hoe informatie geïnterpreteerd dient te worden. Het is een Extensible Markup Language, omdat deze sleutelwoorden zelf te bepalen en uit te breiden zijn.

In dit hoofdstuk beschrijven we niet de complete XML [4] standaard. Daarvoor verwijzen we naar de website van het W3C en verschillende andere pagina's op het internet met informatie en specificatie van XML. We behandelen hier alleen de basis van XML en de specifieke eigenschappen die voor ons onderzoek van belang zijn.

Een voorbeeld XML document ziet er als volgt uit:

Listing 6: Voorbeeld XML document

```
<?xml version="1.0" encoding="UTF-8" ?>

<afdeling naam="directie">
  <medewerker naam="Piet Buursema">
    <functie>directeur</functie>
    <jaarsalaris>150.000</jaarsalaris>
  </medewerker>
  <medewerker naam="Jolien Jansen">
    <functie>secretaresse</functie>
    <jaarsalaris>40.000</jaarsalaris>
  </medewerker>
</afdeling>
```

De eerste regel van het document geeft aan dat het om een XML document gaat, welke opgesteld is volgens de 1.0 versie van de standaard en dat er gebruik is gemaakt van de UTF-8 karaktertabel. Op regel 3 wordt de eerste tag, genaamd *afdeling*, geopend. Deze tag heeft een attribuut *naam*, met als waarde “directie”.

De buitenste tag in een XML document, in dit geval de *afdeling* tag, heet de root-tag van een document. Ieder XML document heeft exact één root-tag. Wanneer je in bovenstaand voorbeeld informatie over twee afdelingen op wilt slaan, heb je een nieuwe root-tag nodig, bijvoorbeeld *bedrijf*, met daaronder twee *afdeling*-tags.

7.1 Well formed

XML documenten moeten aan bepaalde voorwaarden voldoen [24]. Documenten die aan deze voorwaarden voldoen, heten well formed. Well formed zegt alleen maar iets over de vorm waarin informatie wordt opgeschreven, niet over welke informatie wordt opgeschreven.

De belangrijkste eis aan een well formed XML document is dat iedere tag die geopend wordt, ook gesloten wordt. Verder moeten tags die geopend worden in omgekeerde volgorde weer gesloten worden. Een element wat alleen attributen heeft en verder geen waarde of subelementen, kan direct gesloten worden bij het openen: *<persoon naam=“piet” achternaam=“geluk” />*.

Listing 7: Voorbeeld van een niet well formed XML document

```
<?xml version="1.0" encoding="UTF-8" ?>
<afdeling naam="directie">
  <medewerker naam="Piet␣Buursema">
    <functie>directeur</functie>
    <jaarsalaris>150.000</jaarsalaris>
</afdeling>
```

In bovenstaande voorbeeld wordt de *medewerker* tag wel geopend, maar niet afgesloten. Dit document is dus niet well formed.

Listing 8: Voorbeeld niet well formed XML document

```
<?xml version="1.0" encoding="UTF-8" ?>
<afdeling naam="directie">
  <medewerker naam="Piet␣Buursema">
    <functie>directeur</functie>
    <jaarsalaris>150.000</jaarsalaris>
  </afdeling>
</medewerker>
```

In bovenstaand document worden zowel de *afdeling* als de *medewerker* tag afgesloten, echter in de verkeerde volgorde. Ook dit document is dus niet well formed.

Een document wat geen well formed XML bevat, is officieel geen XML document en zal door de meeste XML parsers ook niet verwerkt kunnen worden.

7.2 XML Schema

Een XML Schema [15] beschrijft welke elementen voor mogen komen in een XML document. Elementen kunnen verplicht zijn, of optioneel zijn. Verder kan opgegeven worden hoe vaak een bepaald element voor mag komen, bijvoorbeeld precies één keer, 1 tot n keer of 0 tot n keer. Ook geef je in een XML Schema aan van welke type elementen moeten zijn, zoals bijvoorbeeld een *string*, *integer* of *datum*.

7.2.1 XML Schema Definition (XSD)

Een XML Schema Definition (XSD) bevat de definitie van het schema. Een XSD moet zelf ook weer well formed XML zijn.

Zoals gezegd geef je binnen een XML Schema op welke elementen voor moeten en mogen komen en van welk type die elementen zijn. Elementen kunnen bijvoorbeeld van een type *string*, *integer* of *datum* zijn, maar elementen kunnen ook andere elementen (subelementen) bevatten. Zulke samengestelde elementen zijn van het type *complextype*.

Een *complextype* is een elementtype, waaronder zich een reeks (*sequence*) van andere elementen bevind. In ons voorbeeld is het element *medewerker* van het type *complextype*. De waarde van dit element is immers niet atomair, zoals een *integer* of *datum*, maar een combinatie van twee andere elementen, namelijk *functie* en *jaarsalaris*. Ook een element wat maar één element als subelement heeft, is van het type *complextype*.

Een XSD document voor ons voorbeeld van een *afdeling* is:

Listing 9: XSD voor het voorbeeld XML document

```
<?xml version="1.0" encoding="utf-8" ?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:element name="afdeling" type="afdelingType" />
  <xs:complexType name="afdelingType">
    <xs:sequence>
      <xs:element minOccurs="0" maxOccurs="unbounded" name="medewerker"
                                                    type="medewerkerType" />
    </xs:sequence>
  </xs:complexType>
  <xs:complexType name="medewerkerType">
    <xs:sequence>
      <xs:element name="functie" type="xs:string" />
      <xs:element name="jaarsalaris" type="xs:string" />
    </xs:sequence>
  </xs:complexType>
</xs:schema>
```

In bovenstaand schema zie je dat zowel *afdelingType* als *medewerkerType complextype*'s zijn en dat zowel *functie* als *jaarsalaris* van het type *string* zijn. Verder zie je dat in *afdelingType* het element *medewerker* minimaal 0 keer voor moet komen, zonder maximum. Standaard moet ieder element exact één keer voor komen, tenzij anders aangegeven, zoals bij het element *medewerker*.

7.2.2 Valide

Een document is *valide* volgens een standaard wanneer het alle en alleen in die standaard voorgeschreven elementen bevat [2]. Een voorwaarde voor een *valide* XML document is dat het een well formed document is.

Volgens voorbeeld 9 is ons voorbeeld XML document 6 valide.

7.3 Volgorde

In een XSD document kun je de volgorde van elementen vastleggen. Zo kun je beschrijven dat voorbeeld 10 valide is en 11 niet.

Listing 10: Voorbeeld van een valide XML document met juiste volgorde in elementen

```
<?xml version="1.0" encoding="UTF-8" ?>
<afdeling naam="directie">
  <medewerker naam="Piet␣Buursema">
    <functie>directeur</functie>
    <jaarsalaris>150.000</jaarsalaris>
  </medewerker>
</afdeling>
```

Listing 11: Voorbeeld van een niet valide XML document met verkeerde volgorde in elementen

```
<?xml version="1.0" encoding="UTF-8" ?>
<afdeling naam="directie">
  <medewerker naam="Piet␣Buursema">
    <jaarsalaris>150.000</jaarsalaris>
    <functie>directeur</functie>
  </medewerker>
</afdeling>
```

Het bijbehorde XSD document wat deze volgorde afdwingt is als volgt:

Listing 12: Voorbeeld XSD met vaste volgorde in elementen

```
<?xml version="1.0" encoding="utf-8" ?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:element name="afdeling" type="afdelingType" />
  <xs:complexType name="afdelingType">
    <xs:sequence>
      <xs:element minOccurs="0" maxOccurs="unbounded" name="medewerker"
                                                   type="medewerkerType" />
    </xs:sequence>
  </xs:complexType>
  <xs:complexType name="medewerkerType">
    <xs:sequence>
      <xs:element name="functie" type="xs:string" />
      <xs:element name="jaarsalaris" type="xs:string" />
    </xs:sequence>
  </xs:complexType>
</xs:schema>
```

Omdat het gebruik van een xs : *sequence* veel voorkomt, ligt ook de volgorde waarin elementen voorkomen van tevoren bij de meeste XML documenten vast. Het nut en onnut hiervan met betrekking tot dit onderzoek wordt later besproken.

7.4 XPath

Om gegevens in een XML document makkelijk te benaderen, is *XPath* ontwikkeld. Met *XPath* beschouw je het XML document als een boom-structuur. Met een *XPath expressie* kun je gegevens uit deze boom ophalen.

Bij het XML document *afdeling* van hieronder, kunnen we een *XPath expressie* gebruiken:

```
<?xml version="1.0" encoding="UTF-8" ?>
<afdeling naam="directie">
  <medewerker naam="Piet␣Buursema">
    <jaarsalaris>150.000</jaarsalaris>
    <functie>directeur</functie>
  </medewerker>
</afdeling>
```

XPath: $/afdeling/medewerker/jaarsalaris$. Hiermee krijgen we het *jaarsalaris* van iedere *medewerker*, in een lijst. In dit geval dus maar één waarde, maar gebruiken we dezelfde expressie op het volgende bericht:

```
<?xml version="1.0" encoding="UTF-8" ?>
<afdeling naam="directie">
  <medewerker naam="Piet␣Buursema">
    <jaarsalaris>150.000</jaarsalaris>
    <functie>directeur</functie>
  </medewerker>
  <medewerker naam="Klaas␣Janssen">
    <jaarsalaris>200.000</jaarsalaris>
    <functie>CEO</functie>
  </medewerker>
</afdeling>
```

Dan krijgen we twee waarden terug (150.000 en 200.000). We kunnen ook het gemiddelde opvragen: $avg(/afdeling/medewerker/jaarsalaris)$. Dit levert de waarde 175.000 op.

XPath expressies lijken heel veel op de manier waarop een directory structuur in een Unix/Linux filesystem benaderd wordt, maar met de toevoeging van handige functies zoals *sum*, *avg*, *min*, *max*, maar ook zoekfuncties: $//a$ zoekt alle elementen met de naam a op, ongeacht waar deze element zich in de boom bevindt en wat zijn parent- of childelementen zijn. Door deze toevoegingen is *XPath* een krachtige taal om een informatie behoefte uit te drukken.

8 XI Adapter and Module Framework

De verantwoordelijkheid voor het identificeren van het type bericht ligt bij de communication channel ("Achtergrond"). Deze communication channel is opgebouwd uit een *adapter*, en optioneel uit een of meerdere *modules*. De *adapter* is verantwoordelijk voor de communicatie van het bericht over een specifiek protocol (FTP, SOAP, etc) en de *modules* zijn verantwoordelijk voor eventuele bewerkingen op de inhoud van het bericht, zoals bijvoorbeeld het vertalen van een flatfile bericht naar XML.

Omdat identificeren gebeurt op de inhoud van het bericht, zullen we deze logica in een *module* implementeren. Deze *module* kan dan in iedere communication channel, onafhankelijk van het protocol, gebruikt worden. De basis van een module ziet er als volgt uit:

Listing 13: De basis van een module

```
public class DummyModuleBean implements SessionBean, Module {
        public static final String VERSION_ID ="1.0";
        private static final Trace TRACE = new Trace(VERSION_ID);

        public void ejbRemove() {}
        public void ejbActivate() {}
        public void ejbPassivate() {}
        public void setSessionContext(SessionContext context) {
                myContext = context;
        }
        private SessionContext myContext;
        public void ejbCreate() throws CreateException {
        }

        public ModuleData process(
                ModuleContext moduleContext,
                ModuleData inputModuleData)
                throws ModuleException {

                return inputModuleData;
        }
}
```

Een *module* implementeert de interfaces *Module* en *SessionBean*. De interface *SessionBean* schrijft voor dat de functies *ejbActivate*, *ejbPassivate*, *ejbRemove* en *setSessionContext* geïmplementeerd moeten worden. Deze functies worden gebruikt door XI om aan de module door te geven in welke staat deze zich moet bevinden. Zo kan een *module*, na het starten van de server en dus ook het initialiseren van de *module* de opdracht krijgen om op status inactief te gaan staan, bijvoorbeeld wanneer de bijbehorende Communication Channel op non-actief wordt gezet. De interface *Module* schrijft de functie *process* voor en definieert gegevenstypen zoals *ModuleData* en *ModuleContext*.

De uiteindelijke functie die aangeroepen wordt wanneer de module gevraagd wordt zijn werk te doen, is de functie *process*. Deze functie krijgt de *context* mee, waarin metadata te vinden is en gewijzigd kan worden in het bericht zelf. Dit bericht is van het type *ModuleData* en is te vinden onder de parameter *inputModuleData*. Een bericht kan meerdere *payloads* bevatten (bijvoorbeeld als je een email bericht verwerkt, waar meerdere bijlagen bij zitten). Iedere *payload* kan los opgevraagd worden. Verder kunnen *payloads* toegevoegd worden aan het bericht, of verwijderd. De inhoud van een *payload* kan opgevraagd worden als een stream van het type *ModuleStream*.

Om te garanderen dat een *module* goed samenwerkt met andere *modules* en overal gebruikt kan worden, geeft de module eenzelfde *stream* terug als hij aangeboden krijgt. De inhoud van de *stream* is dan uiteraard wel gewijzigd, of op zijn minst gelezen (zoals bij ons identificatie probleem

voldoende is). De eenvoudigste implementatie van een *module* is er een die de *datastream* volledig ongewijzigd doorgeeft.

Listing 14: Meest simpele vorm van een adapter module

```
public ModuleData process(
        ModuleContext moduleContext,
        ModuleData inputModuleData)
        throws ModuleException {
          return inputModuleData;
        }
```

Vanuit de inputModuleData kunnen we het feitelijke bericht opvragen:

```
(Message) message = Message msg = (Message) inputModuleData.getPrincipalData();
TextPayload payload = msg.getPayload();
payload.getModuleStream();
```

We vragen het bericht op aan het *ModuleData* object. Hierna kunnen we aan het bericht (*message*) de feitelijke *payload* opvragen. Deze *payload* kunnen we uitlezen als een *stream*, namelijk een *ModuleStream*. Dit bericht heeft naast een inhoud (*payload*) ook metagegevens, zoals richting (*inboud/outbound*), maar ook de interfacenaam. Deze metagegevens kunnen we opvragen, maar ook wijzigen:

```
String messageinterface = msg.getMessageProperty(``FromInterfaceName'');
msg.setMessageProperty(``FromInterfaceName'', ``MI_mijn_interface_naam'');
```

In de eerste regel wordt het berichttype waarin het bericht aangeleverd wordt opgevraagd. In de tweede regel wordt hetzelfde berichttype ingesteld.

9 Identificatie van berichten

Wanneer je naar de verschillende XML berichten uit onze casus kijkt, zie je dat ze verschillende en overeenkomstige elementen hebben.

Elementen van variant *Nieuw* van *Labuitslag*:

- labuitslag
- labuitslag @variant
- labuitslag → patientID
- labuitslag → uitslag
- labuitslag → uitslag → TestCode
- labuitslag → uitslag → Meeteenheid
- labuitslag → uitslag → Meetwaarde
- labuitslag → uitslagID

Het → teken staat hier voor een subelement en het @ teken voor een attribuut van het element.

Wanneer je *Patiënt* tegenkomt als root element, dan weet je zeker dat je met een bericht van het type *Patiënt* te maken hebt. Kom je echter als root element *Labuitslag* tegen, dan weet je niet met welke variant je te maken hebt. *Patiënt* is dus een identificerende eigenschap, maar *Labuitslag* niet.

Er kunnen echter meerdere identificerende eigenschappen zijn en identificerende eigenschappen kunnen uit meerdere eigenschappen bestaan. *Patiënt* is dus een identificerende eigenschap, maar *Patiënt → Voornaam* net zo goed. De combinatie van *Patiënt* en *Patiënt → Voornaam* is ook net zo goed een identificerende eigenschap, omdat de combinatie van deze twee elementen in geen van de andere berichten voor komen.

Labuitslag is geen identificerende eigenschap, omdat deze in alle drie de varianten voor komt. Echter, *labuitslag → uitslag → meeteenheid* is een identificerende eigenschap voor de variant *nieuw* van *Labuitslag*, omdat dit element alleen in die variant voor komt.

Wanneer je kijkt naar de variant verwijder dan heeft die variant geen enkel element wat exclusief in die variant voor komt. Echter, de combinatie van *Patiënt → UitslagID* en *Patiënt → Uitslag → Waarde_Oud* komt in geen van de andere varianten voor. Deze combinatie van twee eigenschappen is wel een identificerende eigenschap te noemen voor de variant verwijder.

9.1 Valideren bericht

De makkelijkste en meest veilige manier om een bericht te identificeren, is om het bericht te valideren aan het bijbehorende XSD document. Via deze weg controleer je dat alle verplichte elementen en attributen in het bericht aanwezig zijn, maar ook dat er geen elementen of attributen aanwezig zijn welke niet toegestaan zijn. Wanneer een bericht aan beide voorwaarden voldoet, dan is het een valide bericht van het type wat het XSD beschrijft. Wanneer het bericht niet valide volgens een XSD is, moet een volgende mogelijke XSD geprobeerd worden, totdat alle mogelijkheden geprobeerd zijn.

Om dit mogelijk te maken moet het gehele XML bericht geparsed worden naar een geheugen structuur. Hierna moet deze volledige geheugen structuur doorgelopen en gecontroleerd worden.

Het voordeel hiervan is dat je zeker weet dat je met een correct bericht van een bepaald type te maken hebt.

Normaal gesproken is het valideren van een bericht iets wat de parser zelf al kan doen. Nog voor het parsen kun je aangeven om welke type bericht het gaat, waarna de parser tijdens het parsen het bericht valideert. Dit is sneller en overzichtelijker dan eerst parsen en dan valideren. Echter, omdat wij willen proberen te valideren met meerdere schema's, is deze oplossing niet toereikend. Immers, je moet van tevoren al aangeven om welk type bericht het gaat, iets wat juist nog niet bekend is.

In onze probleemstelling is de validiteit van een bericht echter geen doel. Het enige doel is het bepalen van welk berichttype uit een set van mogelijke berichttypen (beschreven door XSD's) een bericht is. Hiervoor is voldoende om te bepalen dat een bericht een eigenschap bevat welke alle andere berichttypen uitsluit. Validiteit van het bericht volgens het berichttype is iets wat hierna gecontroleerd kan worden.

We gaan dus op zoek naar alternatieven voor bericht validatie, waar we met de kleinst mogelijke identificerende eigenschap uit de voeten kunnen.

9.2 Kleinste aantal eigenschappen

In plaats van alle eigenschappen te controleren, zoals bij het valideren van een bericht, kunnen we ook uit zoeken welke eigenschappen identificerend zijn voor een bepaald berichttype. Neem bijvoorbeeld de volgende twee XML definities:

```
<root>
        <a></a>
        <b></b>
</root>
```

```
<root>
        <a></a>
        <c></c>
</root>
```

Wanneer we naar de eigenschap */root/a* kijken, dan komt deze in beide berichten voor. Echter, */root/b* komt alleen in het eerste bericht voor en */root/c* alleen in het tweede. Dit betekent dat als we willen kijken of een willekeurig bericht tot het eerste type behoort, dan hoeven we alleen te kijken of */root/b* voorkomt als element. Immers, in het tweede bericht kan dit niet voorkomen.

Elke eigenschap die je moet controleren, kost tijd. Hoe minder eigenschappen, hoe minder tijd je nodig hebt om te controleren of een bericht van dat type is. Voor iedere identificerende eigenschap tellen we het aantal eigenschappen wat gecontroleerd moet worden en kennen dat als score toe aan de identificerende eigenschap. De identificerende eigenschap met de laagste score is de "kleinste identificerende eigenschap".

<table>
<tr><td>labuitslag → uitslag → testcode</td><td>1</td></tr>
<tr><td>labuitslag → uitslag → meetwaarde</td><td>1</td></tr>
<tr><td>labuitslag → uitslag → meetwaarde
labuitslag → uitslag → testcode</td><td>2</td></tr>
</table>

Tabel 1: Scores Identificerende Eigenschappen *Labuitslag* variant *nieuw*

Zoals in bovenstaande (overigens incomplete) tabel te zien is, zijn zowel *labuitslag* → *uitslag* → *testcode* en *labuitslag* → *uitslag* → *meetwaarde* een kleinste identificerende eigenschap. De

combinatie van beiden is wel een identificerende eigenschap, echter geen kleinste identificerende eigenschap.

Voor deze methode moet het XML bericht geparsed worden. Er wordt in deze methode geen rekening gehouden met geheugen of reken capaciteit.

9.3 Identificerende eigenschap met kleinste diepte in boom

In plaats van het kijken naar hoeveel eigenschappen we moeten controleren, kunnen we ook per eigenschap kijken hoeveel moeite het kost om deze eigenschap te controleren en hoeveel moeite een identificerende eigenschap dan in totaal kost.

Een manier om deze moeite uit te drukken, is het aantal stappen dat in een boom geïtereerd moet worden om een eigenschap te bereiken. In feite tellen we het aantal $\rightarrow$ en @ tekens in de eigenschap. Met deze methode komen we op de volgende tabel uit:

<table>
<tr><td>$root \rightarrow uitslag \rightarrow nieuw$</td><td>2</td><td>2</td></tr>
<tr><td>$root \rightarrow uitslag \rightarrow nieuw \rightarrow waarde$</td><td>3</td><td>3</td></tr>
<tr><td>$root \rightarrow uitslag \rightarrow nieuw$</td><td>2</td><td rowspan="2">5</td></tr>
<tr><td>$root \rightarrow uitslag \rightarrow nieuw \rightarrow waarde$</td><td>3</td></tr>
</table>

Tabel 2: Scores Identificerende Eigenschappen Uitslag-Update

9.4 Identificerende eigenschap met kleinste diepte in parser

Bij XML kost het parsen van een XML document naar een interne geheugen structuur vaak het meeste werk. Wanneer gebruik gemaakt wordt van een zogeheten lazy xml-parser, wordt de XML boom alleen geparsed voor zover dat nodig is. Wanneer je dus de eigenschap $root \rightarrow Uitslag \rightarrow Nieuw$ opvraagt, wordt er minder werk verricht dan wanneer je $root \rightarrow Uitslag \rightarrow Nieuw \rightarrow Waarde$ opvraagt.

Tot zover komt de score nog overeen met de vorige score. Wanneer je een eigenschap uit een XML boom opvraagt, dan itereer je de boom opnieuw vanaf de wortel. Echter, wanneer je een deel van een boom opvraagt, waarvan eerder al een deel geparsed is, dan wordt alleen het resterende deel geparsed.

Wanneer je dus eerst $root \rightarrow Uitslag \rightarrow Nieuw$ opvraagt en daarna $root \rightarrow Uitslag \rightarrow Nieuw \rightarrow Waarde$, dan is de score aan de hand van de diepte van de boom 5. Immers, je itereert twee keer vanaf de wortel van de boom, een keer 2 diep en een keer 3 diep. Wanneer je kijkt naar een lazy parser, dan parsed deze de eerste keer 2 diep en zal de bij de tweede eigenschap nog maar een stap verder hoeven parsen: de eerste twee stappen staan immers al in het geheugen. De score is nu dus maar 3, in plaats van 5.

De tabel komt er hiermee zo uit te zien:

<table>
<tr><td>$root \rightarrow Uitslag \rightarrow Nieuw$</td><td>2</td><td>2</td></tr>
<tr><td>$root \rightarrow Uitslag \rightarrow Nieuw \rightarrow Waarde$</td><td>3</td><td>3</td></tr>
<tr><td>$root \rightarrow Uitslag \rightarrow Nieuw$</td><td>2</td><td rowspan="2">3</td></tr>
<tr><td>$root \rightarrow Uitslag \rightarrow Nieuw \rightarrow Waarde$</td><td>3</td></tr>
</table>

Tabel 3: Scores Identificerende Eigenschappen Uitslag-Update

In bovenstaande tabel geldt dus dat zowel $root \rightarrow uitslag \rightarrow nieuw \rightarrow waarde$ als $root \rightarrow uitslag \rightarrow nieuw$ samen met $root \rightarrow uitslag \rightarrow nieuw \rightarrow waarde$ even grote identificerende

factoren zijn. Dit is onder de aanname dat het opvragen van gegevens uit een XML boom in tijd verwaarloosbaar is ten opzichte van het opbouwen van deze boom.

Let wel, dat wanneer je een score voor een identificerende eigenschap wilt berekenen met twee (of meer) eigenschappen waarbij het eerste deel van de boom niet overlapt, dit niet geldt. Je telt dus alleen overlappende boomdelen slechts één keer, maar niet overlappende delen worden gewoon bij elkaar opgeteld.

9.5 Identificerende eigenschap met kleinste diepte in XML-stream

Het XML document wordt binnen XI als ModuleStream aangeboden aan een module. Wanneer we vooraan in de stream al kunnen zien dat een bericht van een bepaald type is, dan kunnen we de rest van de stream ongemoeid laten. Dit zou betekenen dat we niet het volledige XML document hoeven te parsen en zo dus performance winst boeken.

Eerdere oplossingen parsen het hele XML document en zoeken dan in het geheugen naar identificerende eigenschappen. Het nadeel hiervan is dat je het hele document moet parsen. Wanneer een identificerende eigenschap helemaal in het begin van een XML stream al gevonden kan worden, hoeft de rest van het document niet verder geparsed te worden.

Omdat in veel XML files de volgorde van de elementen vast ligt, zou je kunnen bepalen welk element je het eerste tegen zal komen. Echter, bij optionele elementen loop je het risico dat deze niet voorkomen. Wanneer deze toevallig wel voorkomt, kan hij wel identificerend zijn. Voor de zekerheid moet je een verplicht element als identificerende eigenschap nemen, maar het kan zijn dat deze pas veel verder in de stream voorkomt dan het optionele veld, waardoor deze weer minder optimaal is.

Handiger is het dus om een methode te ontwikkelen welke meerdere eigenschappen per berichttype kan gebruiken en reeds bij de allereerste die hij tegenkomt de identificatie voltooit. Hierbij maakt het dan ook niet meer uit of dit een optioneel of verplicht veld is. Verderop, bij de beschrijving van de implementatie, zullen we zelfs zien dat het niet meer uitmaakt hoeveel identificerende eigenschappen we onthouden voor onze validatie en dat we dus alle identificerende eigenschappen gewoon opslaan en de volgorde niet meer uitmaakt.

In ons voorbeeld bevat ieder bericht van het type *Patiënt* slechts informatie over één patiënt. Echter, wanneer we een bericht type hebben waarbij een bericht meerdere patiënten kan bevatten, krijgen we een voorbeeld als volgt:

Listing 15: Een voorbeeld XML met meerdere patiënten

```
<?xml version="1.0" encoding="UTF-8" ?>
<patiënten>
  <patiënt>
    <naam>Walter Hoolwerf</naam>
  </patiënt>
  <patiënt>
    <naam>John Doe</naam>
  </patiënt>
  <patiënt>
    <naam>Jane Doe</naam>
  </patiënt>
</patiënten>
```

Als in dit voorbeeld de eigenschap *patiënten* → *patiënt* een identificerende eigenschap is, dan is het duidelijk dat na de derde regel al gestopt kan worden met het parsen van het document. De andere 9 regels kunnen dus genegeerd worden.

Bij voorgaande methoden was het algoritme voor het identificeren van berichten vrij eenvoudig: na het parsen van een XML document naar een geheugenstructuur kun je met behulp van XPath eenvoudig identificerende eigenschappen opvragen. Wanneer je resultaten terug krijgt, zijn de eigenschappen aanwezig, en wanneer je (afhankelijk van de XPath implementatie) niets of een fout terug krijgt, zijn de eigenschappen niet aanwezig.

Bij deze methode proberen we zoveel mogelijk het parsen te beperken. Dit betekent dat we zo min mogelijk van het XML document willen parsen en dat deel uiteraard maar een keer. Wanneer we een tag tegenkomen, moeten we deze dus vergelijken met alle mogelijke identificerende eigenschappen. Hiervoor moet het algoritme in de parser zelf ingebouwd worden.

Nog meer dan bij de voorgaande methoden is de parser en de manier waarop deze gebruikt wordt van belang voor de snelheid van het algoritme.

10 Parser

Er zijn verschillende soorten XML parsers, met verschillende voor- en nadelen. Alle standaard parsers controleren of een XML document well-formed is. Verder kunnen de meeste parsers een document ook valideren aan de hand van een XSD document.

In dit hoofdstuk bespreken we verschillende XML parsers, met verschillende eigenschappen. In *DOM parser* bespreken we een parser die het gehele XML document naar een geheugenstructuur parsed, zodat we informatie uit het document kunnen raadplegen. In *Lazy parser* beschrijven we een variant op een DOM parser, welke wel het hele document in het geheugen laadt, maar pas wanneer gegevens geraadpleegd worden, dat deel van het XML document parsed. Het hoofdstuk *Double lazy parser* beschrijft een parser welke nog een extra stap doet, door niet alleen het gedeelte wat geraadpleegd wordt te parsen, maar ook alleen dat gedeelte in te lezen. In *SAX parser* beschrijven we een parser welke direct bij het parsen al de informatie doorgeeft aan de aanroepende applicatie, in plaats van eerst het hele XML document in het geheugen te parsen.

10.1 DOM parser

Een DOM [22] parser leest het hele XML document en parsed deze naar een geheugenstructuur, het Document Object Model [22] (DOM) genaamd. Deze geheugen structuur is in feite een boomstructuur waar iedere knoop een arbitrair aantal onderliggende knopen heeft (*n-boom*), waarin elk element een child-node of leaf is en attributen in de node zelf opgeslagen worden.

Voor identificatie methode *Validatie* is een DOM parser nodig. Bij het valideren van een XML document bij methode *Validatie* moeten alle elementen in een XML structuur gecontroleerd worden op validiteit. Dit betekent dat alle informatie beschikbaar moet zijn in een geheugen structuur.

Deze parser kost zowel qua geheugen gebruik en rekenkracht de meeste capaciteit, omdat de hele inhoud van het document verwerkt en opgeslagen in het geheugen moet worden. De hoeveelheid tijd die het kost is sterk afhankelijk van de grootte van het document.

10.2 Lazy parser

Een lazy parser parsed alleen dat deel van een XML document wat gevraagd wordt [5]. Zo kun je een lazy parser vragen maar tot maximaal twee nivo's diep te parsen. De rest wordt als tekst element onder de diepst geparsde node gehangen. Wordt er toch iets opgevraagd wat dieper in de boom ligt dan wat er tot dan toe geparsed wordt, dan kan de parser het tekst element parsen om een diepere boom in het geheugen op te bouwen.

Wanneer maar een beperkt gedeelte van een XML document benaderd wordt, levert deze manier van parsen snelheidswinst op. Wanneer echter het hele document benaderd wordt, of het grootste deel van het document, dan is de overhead van deze parser een vertragende factor. Omdat deze parser niet alles parsed, maar alleen datgene parsed wat opgevraagd wordt en nog niet eerder is geparsed, zal deze parser elke keer bij het afdalen van de boomstructuur bij iedere node eerst moeten controleren of de onderliggende structuur reeds geparsed is, voor hij verder de boom in kan dalen.

Wanneer de parser tot de conclusie komt dat de onderliggende structuur die hij wil opvragen nog niet geparsed is, zal hij deze alsnog moeten parsen.

Belangrijk om op te merken bij een lazy parser is dat wel het hele XML document doorgelezen moet worden door de parser. Voor iedere tag die in een XML document geopend wordt, moet ook een sluiting voorkomen. Om deze sluiting te vinden zal de parser wel het hele document door moeten zoeken.

Een voorbeeld van de werking van een lazy parser. We hebben het volgende XML document welke 4 nivo's diep is:

Listing 16: Voorbeeld document voor lazy parser met diepte 4

```
<?xml version="1.0" encoding="UTF-8" ?>
<ROOT>
  <A>
    <B>
      <C>X</C>
    </B>
  </A>
</ROOT>
```

De lazy parser parsed in eerste instantie alleen de root-tag van het XML document. Omdat deze parser zowel de opening van de tag root (<root>) moet vinden, maar ook de sluiting (</root>) moet de parser wel het hele document doorlopen. Hij hoeft echter geen parser berekeningen te doen op de gegevens welke hij tegenkomt. Hier ligt de snelheids winst.

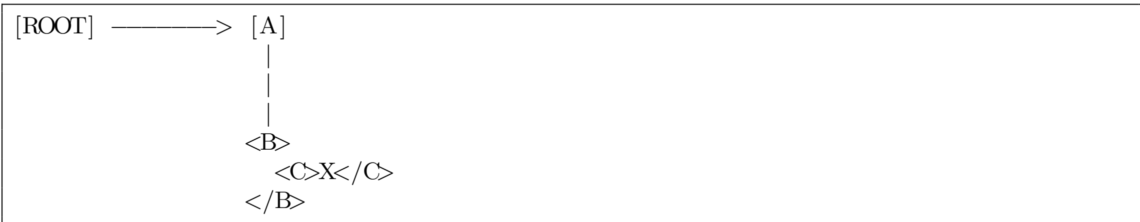
De gegevens welke hij tegenkomt, zonder dat hij ze parsed, worden onder de root-node in het geheugen opgeslagen.

Listing 17: Voorbeeld van een lazy parser: stap 1

```
[ROOT]
   |
   |
   |
  <A>
    <B>
      <C>X</C>
    </B>
  </A>
```

Wanneer je nu gegevens uit de XML boom opvraagt (bijvoorbeeld $root \rightarrow A \rightarrow B \rightarrow C$) begint de parser bij root. Vanaf ROOT wordt de subnode A geprobeerd te benaderen. Omdat de subnodes van ROOT nog niet zijn geparsed gaat dit niet. De parser zal dus het XML segment wat onder ROOT is opgeslagen moeten parsen. We krijgen dan de volgend geheugen structuur:

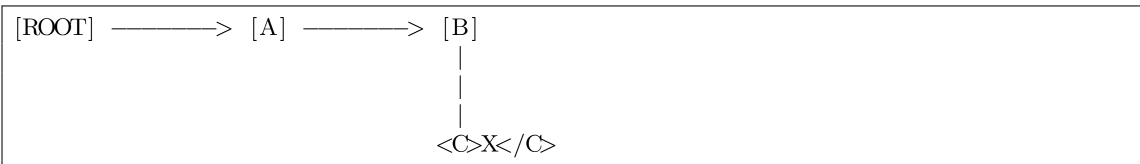
Listing 18: Voorbeeld van een lazy parser: stap 2



```
[ROOT] ---------> [A]
                   |
                   |
                   |
                  <B>
                    <C>X</C>
                  </B>
```

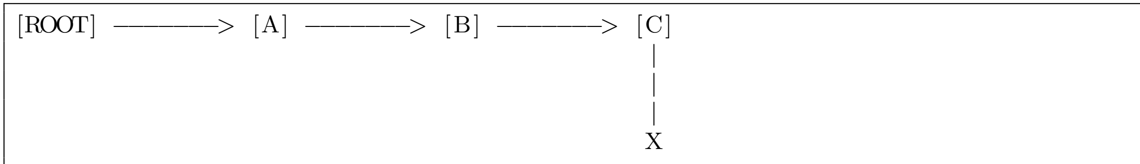
Waarna het doorgaat:

Listing 19: Voorbeeld van een lazy parser: stap 3



```
[ROOT] ---------> [A] ---------> [B]
                                  |
                                  |
                                  |
                                <C>X</C>
```

Listing 20: Voorbeeld van een lazy parser: stap 4



Pas in de laatste stap kan de waarde (X) gevonden worden onder element C. In elke stap zal het XML segment wat geparsed moet worden, helemaal doorlopen moeten worden. Het segment <C>X</C> is dus 4 keer helemaal doorgelezen om uiteindelijk op het eindresultaat te komen. Wanneer we gebruik hadden gemaakt van een normale parser, dan had deze maar een keer doorgelezen hoeven te worden.

Wanneer je te maken hebt met grote en complexe XML documenten, met veel verschillende elementen en daaruit heb je maar een paar elementen voor je informatie nodig, dan levert een lazy parser wel degelijk winst op [10]. In ons geval, waar we een zo klein mogelijk aantal elementen willen bekijken om een berichttype te identificeren, is een lazy parser dus een goede keuze.

We gebruiken de lazy parser voor methode 2, 3 en 4. We zien hierbij ook meteen dat methode 2 en 3 hierbij het beste passen omdat deze de diepte van het parsen proberen te beperken. Het meeste voordeel is te behalen bij identificatie methode 3, omdat hierbij gedeelten die reeds geparsed zijn niet meer meegenomen worden in de score.

Een groot nadeel van de lazy parser is dat hij nog steeds het hele document in het geheugen laadt. Wat een lazy parser niet doet, is het volledig verwerken van het in het geheugen geladen document, zoals boven beschreven. Dit scheelt wel in tijd en ook in geheugen, echter, wanneer een document erg lang is, wordt het document nog steeds van begin tot einde gelezen om het einde van de root-tag te vinden.

10.3 Double lazy parser

In [5] wordt de standaard lazy parser beschreven en ook de double lazy parser. Een normale parser leest dus nog steeds het document van begin tot einde door om het einde van de roottag te vinden. Hiervoor moet alsnog het hele document in het geheugen gelezen worden (de preprocessing stage). De double lazy parser daarentegen is ook in de preprocessing stage lazy.

Dit wordt gedaan door het bericht in segmenten op te knippen, zodat alleen segmenten ingelezen hoeven te worden. Hierna hoeft de parser alleen het XML segment in te lezen wat op dat moment nodig is.

Wanneer we weer kijken naar het document wat we bij de lazy parser ook bekeken:

```
<ROOT>
  <A>
    <B>
      <C>X</C>
    </B>
  </A>
</ROOT>
```

Dan kunnen we dit in kleine segmenten opknippen:

```
<ROOT>
        Pointer -> A
</ROOT>

<A>
        Pointer -> B
</A>

<B>
        Pointer -> C
</B>

<C>X</C>
```

Ieder segment is apart te benaderen. Het artikel beschrijft dat ze in losse files opgeslagen worden op het filesystem. In ons geval staat het XML document reeds in het geheugen, dus kan dit gebeuren middels pointers naar het begin en einde van het segment.

Wanneer nu het root-element opgevraagd wordt, hoeft alleen het eerste segment geparsed te worden. Wanneer de subelementen van deze root-tag opgevraagd worden, hoeft alleen het tweede element geparsed te worden. Telkens is dit maar 3 regels. In vergelijking, wanneer bij een normale lazy parser de root-tag opgevraagd wordt, zal 7 regels geparsed moeten worden, voordat het einde van deze root-tag gevonden wordt. De tussenliggende elementen worden niet verwerkt, maar dus wel gelezen.

Het artikel beschrijft dat de snelheid van opvragen van gegevens uit het XML document met deze double lazy parser beduidend sneller is dan bij een normale lazy parser, terwijl het geheugen gebruik net zo laag blijft. Echter, wat in het artikel weinig aandacht krijgt is de tijd die het kost om de initiële segmentering te doen. Omdat wij een document precies één keer parsen en er waarschijnlijk maar één gegeven per mogelijke berichttype uit opvragen, zal dit segmenteren in verhouding dus veel tijd kosten, terwijl de snelheidswinst van gegevens opvragen betrekkelijk minder relevant is.

Om bovenstaande performance reden, samen met het gegeven dat er geen implementatie beschikbaar is van deze double lazy parser, betekent dat we deze parser verder niet meer gebruiken voor het oplossen van onze identificatie problematiek. De gedachte van zo min mogelijk karakters hoeven parsen om de gegevens te vinden die we zoeken, zullen we echter wel op een andere manier proberen vorm te geven.

10.4 SAX parser

Een DOM parser parsed eerst het hele XML document naar een geheugen structuur en geeft je pas dan de mogelijkheid om de informatie uit het XML document te verwerken [27]. Een SAX parser geeft je reeds tijdens het parsen alle informatie door. Bij ieder begin van een element en ieder eind van een element geeft de parser aan door de ontwikkelaar geschreven functies de benodigde informatie door: naam van element, eventuele inhoud, parameters etc.

Het grote voordeel is dus dat je reeds tijdens het parsen alle informatie krijgt. Dit is sneller (het opbouwen van de geheugen structuur kost tijd), maar scheelt ook enorm in het geheugen (alleen de informatie die je bij de verwerking zelf in het geheugen opslaat kost ruimte, maar alles wat je negeert om dat je het niet nodig hebt, verdampt).

Het grote nadeel van een SAX parser is dat random access niet mogelijk is. Wanneer je later besluit dat je extra gegevens uit je XML document nodig hebt (bijvoorbeeld wanneer access control gegevens opgeslagen staan in een XML document en het systeem per actie van een gebruiker hierin

controleert of die gebruiker dat mag), terwijl je die op het moment van parsen niet in het geheugen opgeslagen hebt, dan zul je het hele XML document opnieuw moeten parsen.

Wanneer we het volgende voorbeeld XML document middels een SAX parser parsen, dan krijgen we de informatie in de volgende volgorde door:

```
<root>
        <a>
        </a>
        <b>
        </b>
</root>
```

```
begin: root
begin: a
end: a
begin: b
end: b
end: root
```

Uiteraard kan het random-access probleem opgelost worden door alle informatie die je tegenkomt op te slaan in een dynamische geheugen structuur. Dit is dus precies wat een DOM parser doet. Een DOM parser is dus niets meer dan een SAX parser waar de implementatie van de functies die de aangeleverde informatie verwerken een dynamische structuur opbouwen. Hiermee is ook meteen bepaald dat de performance van een DOM parser per definitie lager moet liggen dan die van een SAX parser met triviale functies.

10.5 Identificerende parser

Hoewel een SAX parser het meest in de buurt komt van het gewenste effect (gegevens worden niet in het geheugen opgeslagen en al tijdens het parsen krijgen we de gegevens ter verwerking aangeboden), is een SAX parser nog steeds een parser welke het volledige document parsed. Echter, onze doelstelling is om zo snel mogelijk een bericht te identificeren en we zijn daarbij niet geïnteresseerd in de inhoud van het betreffende bericht. Hiervoor hebben we een eigen methode ontwikkeld, welke in deze sectie beschreven staat.

Onze methode is er speciaal op gemaakt om zo min mogelijk te parsen. De oplossing is om tijdens het parsen de identificatie al te doen. Hiervoor wordt dan ook een SAX parser gebruikt. Wanneer de identificatie gelukt is (omdat er een identificerende eigenschap gevonden is) wordt het parsen afgebroken, door de SAX parser de opdracht te geven om te stoppen.

```
<labuitslag>
  <b>3</b>
  <a>test</a>
</labuitslag>
```

Wanneer we op zoek zijn naar de identificerende eigenschap *labuitslag* $\rightarrow$ *b*, dan hoeven we minder dan de helft van het XML document te parsen. Het is dus zaak om identificerende elementen te zoeken welke zo veel mogelijk aan het begin van een binnenkomend XML bericht te vinden zijn.

Er zijn hierbij twee mogelijkheden denkbaar. De eerste mogelijkheid is het XML document te parsen en bij ieder element te kijken of het overeenkomt met een van de identificerende eigenschappen. Hierbij vind je altijd het identificerende element wat het meest aan het begin van de XML structuur opgeslagen zit. Het nadeel is dat ieder element controleren aan iedere identificerende eigenschap veel werk kan zijn.

Zoals beschreven in sectie *XML schema* wordt de volgorde waarin elementen voor komen vastgelegd in het XSD document. Dit betekent dat we kunnen bepalen welke eigenschappen we, gezien vanaf het begin van een XML stream, als eerste tegen zullen komen. Het is dan voldoende om slechts de allereerste identificerende eigenschap die je zult tegenkomen op te slaan, de rest kun je vergeten. Het algoritme wat we in sectie *Identificerende Eigenschappen* zullen zien, legt automatisch dus alle elementen vast op de volgorde dat je ze in het XML bericht tegen zult komen.

Stel dat in bovenstaand XML bericht *labuitslag* $\rightarrow$ *a* een identificerende eigenschap is voor berichttype A en we willen bepalen of bovenstaand bericht inderdaad van type A is. We lezen het bericht als een XML stream, in plaats van als een boom-structuur. Dit betekent dat we gegevens tegen komen in de volgorde dat ze in het document zijn opgeslagen. Als eerste komen we dus *<labuitslag>* tegen. We kunnen hiermee de expressie */labuitslag* mee vaststellen. We controleren of dit een identificerende eigenschap is, maar dat is deze expressie niet. Als volgende komen we ** tegen, maar ook */labuitslag/b* is geen identificerende eigenschap. Hierna komen we ** tegen, dus breken we de expressie af en komen we weer op */labuitslag* uit.

Als volgende maken we expressie */labuitslag/a* en na controle blijkt dit inderdaad een identificerende eigenschap te zijn, namelijk voor berichttype A. Omdat we nu al weten tot welk berichttype het bericht A behoort, hoeven we de rest van het bericht niet meer te parsen. We hebben dus een succesvolle identificatie bewerkstelligd, door maar de helft van het XML bericht te parsen.

Deze methode van identificeren is met name interessant in berichten waar in de root-tag al te zien is tot welk berichttype een bericht behoort (een situatie die vaak voorkomt). Deze methode hoeft dan alleen de allereeste regel van het document te parsen, waar voorgaande methoden alsnog het hele bericht moeten parsen.

In de worst-case scenario, waarbij het bericht niet geïdentificeerd kan worden, zal deze methode overigens wel het hele bericht parsen. Immers, je kunt pas helemaal aan het einde van het bericht

concluderen dat je geen identificerende eigenschappen kunt vinden. In sectie *Implementaties* zullen we zien op welke manier we proberen deze methode van implementeren zo snel mogelijk te krijgen, door de methoden voor het opbouwen en afbreken van de expressie en het controleren van de expressie ten opzichte van de identificerende eigenschappen zoveel mogelijk te optimaleren. Verder zullen we in sectie *Benchmark* zien hoe deze methode zich verhoudt ten opzichte van andere methoden, zowel in de normale situatie waarin identificatie succesvol is, alsook in de worst-case scenario waarin identificatie niet succesvol is.

11 Implementaties

De verschillende identificatie methoden zoals beschreven in *Identificatie van berichten* worden geïmplementeerd in Adapter Modules. Voor een binnenkomend bericht kunnen we dan alle verschillende adapter modules aanroepen en zo van iedere module meten hoe lang de verschillende modules, en dus identificatie methoden, in beslag nemen. Door ook de volgorde van de adapter modules te variëren, sluiten we uit dat de volgorde van invloed is.

Iedere adapter module zorgt bij het initiëren van de module voor het aanmaken van herbruikbare structuren. Zo leest de standaard *identificator* bij het initialiseren alle XML schema's in het geheugen in. Deze structuren kunnen dan voor ieder te identificeren bericht opnieuw gebruikt worden.

Iedere *AdapterModule* meet zelf de tijd die het identificeren in beslag neemt. Hierbij wordt alleen het identificeren gemeten. Het initialiseren van de herbruikbare structuren hoeft maar een keer te gebeuren en wordt daarom niet in de meting meegenomen. Het resultaat van de identificatie, alsmede de tijd die het in beslag heeft genomen, wordt aan het bericht als een payload gehangen. Deze payload staat volledig op zichzelf en beïnvloedt dus verdere identificaties niet.

Wanneer er bij het opvragen van identificerende informatie of het valideren van een bericht aan een XML Schema een fout optreedt, dan gaan we er in deze test opstelling vanuit dat dit komt omdat de opgevraagde informatie niet beschikbaar is of omdat het bericht niet gevalideerd kan worden. We houden geen rekening met factoren zoals het niet kunnen alloceren van geheugen, fouten in aangeleverde XML bestanden, fouten in XML Schema's en dergelijken. Dit zou in een productie implementatie uiteraard wel van toepassing zijn, maar in onze test opstelling kunnen we deze factoren voldoende beheersen. Verder sluiten we hiermee uit dat meetresultaten beïnvloed worden door ingewikkelde foutafhandeling routines, welke met de feitelijke meeting weinig van doen hebben.

Op deze manier hebben we een makkelijk en uniforme manier om de verschillende implementaties te testen op dezelfde invoerdata en onder dezelfde omstandigheden.

11.1 Parsers

11.1.1 DOM parser

Voor zowel de DOM parser als de SAX parser maken we gebruik van de standaard Java libraries. Voor de DOM parser is dit de *javax.xml.parser.DocumentBuilder* [17] library. Het gebruik van een DOM parser werkt in feite in twee stappen. De eerste stap is het parsen van een XML document naar een geheugen structuur. De tweede stap is het raadplegen van data.

Het parsen van een XML document gebeurd door een *Parser* object. Dit object wordt verkregen middels het *JavaFactoryModel* [11].

```
DocumentBuilder parser =
            DocumentBuilderFactory.newInstance().newDocumentBuilder();

Document document = parser.parse(new File("adt_a03_sample.xml"));
DOMSource domSource = new DOMSource(document);
```

Het document object parsed het XML bestand tot een boom structuur. De $root-node$ van deze boom kunnen we daarna opvragen door een nieuw *DOMSource* object aan te maken gebaseerd op de gemaakte geheugen structuur.

Hierna kunnen we de gegevens raadplegen. Van iedere *node* in de *boom* (een element in de XML structuur) kunnen we de eigenschappen, maar ook de *childnodes* (subelementen) opvragen:

```
NodeList nodes = domSource.getChildNodes();
for (Node n : nodes) {
  System.out.println(n.getNodeName());
}
```

In bovenstaande code wordt de naam van ieder subelement van het root-element op het scherm gezet.

11.1.2 Lazy XML Parser

De meest gebruikte lazy XML parser is de Xerces 2.0 J parser [6]. Het gebruik van deze parser werkt ruwweg hetzelfde als de standaard parser van Java. Het aanmaken van een parser object werkt niet middels het factory model, de rest werkt echter op een vergelijkbare manier.

```
DOMParser parser = new DOMParser();
parser.setFeature(
        "http://apache.org/xml/features/dom/defer-node-expansion",
        true);
InputSource is = new InputSource(new FileInputStream(xml_file));
parser.parse(is);
```

Met de *setFeature* methode vertellen we de parser dat we de lazy methode willen gebruiken [7]. Hierna laten we hem het XML bestand parsen.

Het blijkt echter dat deze XML implementatie standaard geen XPath expressies kan evalueren. Het is niet moeilijk om zelf zo'n evaluator te schrijven, echter, metingen van deze parser duiden reeds aan dat, hoewel hij sneller is dan de standaard XML parser, hij ook niet snel genoeg is voor onze doeleinden. We kiezen er dan ook voor om ter referentie in de benchmark wel de standaard DOM parser te implementeren en meten, maar om geen tijd te steken in het schrijven van een XPath evaluator op deze lazy parser. Bij de benchmark laten we de lazy parser dan ook buiten beschouwing.

11.1.3 SAX Parser

Als SAX parser maken we ook weer gebruik van de standaard Java 1.5 libraries. Deze keer gaat het met name om de *javax.xml.parsers.SAXParser* [18] library. Zoals gezegd worden bij het parsen van een XML document door een SAX parser enkele door de ontwikkelaar geschreven functies aangeroepen. Deze functies moeten geschreven worden volgens een bepaalde interface en voor het parsen van een XML document moet aan de SAX parser duidelijk gemaakt worden welke functies hij moet aanroepen.

De functies worden in een speciale klasse geprogrammeerd, waarin de interface *DefaultHandler* [16] wordt geïmplementeerd.

```
private class SaxHandler extends DefaultHandler {

        public void startElement(String uri, String name, String qName,
                        Attributes attributes) throws SAXException {
                        System.out.println ("Begin: " + qName);
        }

        public void endElement(String uri, String name, String qName) {
                System.out.println("End: " + qName        };
        }

        public void beginDocument() {
                System.out.println("begin document");
        }

        public void endDocument() {
                System.out.println("end document");
        }
}
```

Een nieuwe SAX parser wordt aangemaakt middels het standaard *FactoryModel* [11]. Deze parser wordt verteld welke functies hij aan moet roepen en welk XML document hij moet parsen. Wanneer het volgende XML document wordt geparsed met onze voorbeeld functies, dan krijgen we de output die daaronder staat:

```
<root>
        <a>
                <b>
                </b>
        </a>
</root>
```

```
begin document
begin: root
begin: a
begin: b
end: b
end: a
end: root
end document
```

11.2 Identificerende eigenschappen

Voor berichten geïdentificeerd kunnen worden, moeten van iedere berichttype de identificerende eigenschappen worden bepaald. Ieder berichttype wordt beschreven door een XSD, welke we dan ook gebruiken voor het bepalen van de identificerende eigenschappen.

Als eerste stap is het nodig om per berichttype B alle identificerende eigenschappen I_b te bepalen. Een XSD is een well-formed XML document [28], dus deze parsen we dan ook als zodanig. Alle eigenschappen beschrijven we als $XPath - expressies$.

```
DocumentBuilder parser = DocumentBuilderFactory.newInstance()
                                        .newDocumentBuilder();
FileInputStream fis = new FileInputStream(schemaFile);
this.documentParsed = parser.parse(fis);
this.documentParsedSource = new DOMSource(this.documentParsed);
this.BuildXpath(documentParsed.getChildNodes(), "");
```

De laatste stap roept een functie aan welke alle mogelijke $XPath-expressies$ genereert. Deze functie krijgt een lijst van elementen ($lstElementen$) mee en de $XPath - expressie$ zoals die tot zoverre is opgebouwd ($xpathExpr$). Verder is er een globale (nu nog lege) lijst van mogelijk $XPath - expressies$ ($pathExprLijst$) aanwezig, welke door deze functie gevuld worden.

```
parameters:
        xpathExpr
        lstElementen
voor alle elementen E in elementenlijst:
        als E van type "element"
                xpathExpr += elementnaam
                voeg xpathExpr toe aan xpathExprLijst
                herhaal functie recursief met alle subelementen van E
                            als lstElementen en xpathExpr als xpathExpr
        als E van type "referentie"
                zoek definitie van element E, $E_def$, op in XSD
                herhaal functie recursief met alle subelementen van
                      $E_def$ als lstElementen en xpathExpr
                      (ongewijzigd) als xpathExpr
```

Dit algoritme berekent alle singleton eigenschappen van een berichttype. Alle complexe eigenschappen ($/root/A$ en $/root/B$) kunnen bepaald worden door de machtsverzameling van deze verzameling singleton eigenschappen te nemen. Bij de eerste experimenten bleek echter, dat bij verschillende op de praktijk gebaseerde test-sets van HL7 berichten, er altijd een singleton identificerende eigenschap te bepalen was. Verder is bij de implementatie het veel makkelijker om een hoge performance te halen wanneer gebruik gemaakt wordt van singelton identificerende eigenschappen, omdat er gebruik gemaakt kan worden van een structuur met O(1) lookup (zie sectie *Identificitie*). Er is dan ook voor gekozen om complexe identificerende eigenschappen buiten beschouwing te laten en ons te richten op singleton identificerende eigenschappen.

Wanneer dit algoritme voor iedere berichttype is uitgevoerd, dan worden de onderlingen lijsten van eigenschappen vergeleken, om de niet-identificerende eigenschappen eruit te filteren. Dit gebeurd in twee stappen. De eerste stap is een lijst opstellen van alle eigenschappen die in meerdere berichttypen voorkomen. Deze lijst heet *OverlappendeElementen*. De tweede stap is in elke lijst met eigenschappen deze dubbel voorkomende eigenschappen te verwijderen, zodat alleen de identificerende eigenschappen overblijven.

```
parameters:
  - Berichttypen =  lijst van berichttypen met per
                    berichttype alle elementen
  - OverlappendeElementen = {lege lijst}
voor alle berichttypen T1:
  voor alle berichttypen T2:
```

De manier waarop alle eigenschappen ($expressies$) nu bepaald worden en daarna de niet-identificerende eigenschappen eruit gefilterd worden is niet optimaal. Door gebruik te maken van $stringbuffers$ in plaats van normale strings (zie sectie *Identificerende Parser*), kan deze routine een stuk sneller gemaakt worden.

Belangrijk echter is om te realiseren dat deze routine alleen bij het initialiseren uitgevoerd wordt. Wanneer je in een server omgeving kijkt, waar de bedoeling is dat servers vele dagen achter elkaar draaien zonder een herstart, dan is een routine die bij initialisatie enkele minuten duurt, maar daarna geen enkele performance impact meer heeft, geen issue wat opgelost moet worden.

Omdat wij nu hetzelfde object gebruiken voor meerdere manieren van identificeren, worden de expressies wel op verschillende manieren opgeslagen (lijst van expressie, lijst van gecompileerde XPath-objecten, globale Map). Uiteraard hoeft er maar een opslag methode behouden te blijven wanneer de meest optimale identificatie methode gekozen is en voor productie wordt geïmplementeerd.

11.3 Identificatie

11.3.1 Valideren bericht

Bij het valideren van het binnenkomende bericht wordt gebruik gemaakt van de standaard Java XML implementatie. Bij het initialiseren van de module worden alle XML schema's ingelezen en wordt voor ieder schema een itValidator object aangemaakt. Een *Validator* object is een geheugen representatie van een XSD schema, net zoals een DOM-tree een geheugen representatie van een XML document is. Een *Validator* object kan gebruikt worden om een XML document mee te valideren. Deze *Validator* objecten worden in een centrale Vector opgeslagen, zodat deze later gebruikt kunnen worden.

Een binnenkomend XML bericht wordt geparsed tot een DOM tree en daarna gevalideerd met ieder $Validator$ object. Wanneer het bericht correct gevalideerd wordt met een $Validator$ object is de identificatie voltooit en wordt het resultaat terug gegeven.

Inlezen alle XML Schema's De XML Schema's staan allemaal opgeslagen in een directory. Deze directory wordt uitgelezen en alle aanwezige XML Schema's worden ingelezen. Voor ieder schema wordt een object aangemaakt en toegevoegd aan een centrale $Vector(validatorList)$.

```
public void ejbCreate() throws javax.ejb.CreateException {
  this.validatorList = new Vector<CustomValidator>();

  File schemaDirectory = new File("/data/sap-share/pi/vaag-zis/schema");
  File[] schemaListing = schemaDirectory.listFiles();
  SchemaFactory schemaFactory =
          SchemaFactory.newInstance(XMLConstants.W3C_XML_SCHEMA_NS_URI);
    for (File schemaFile : schemaListing) {
    try {
      Schema schema = schemaFactory.newSchema(schemaFile);
      validatorList.add(new CustomValidator(schema.newValidator(),
                                              schemaFile.getName()));
    } catch (SAXException e) {
      e.printStackTrace();
    }
  }
}
```

Dit $CustomValidator$ object slaat zowel de naam van het Schema op, als de $Validator$ welke het werk object is van dat Schema.

Valideren inkomend bericht Eerst wordt een binnengekomen bericht geparsed naar een DOM tree.

```
Message msg = (Message) inputModuleData.getPrincipalData();
Payload pl = msg.getMainPayload();
ByteArrayInputStream document = new ByteArrayInputStream(pl
                .getContent());
try {
        parser = DocumentBuilderFactory.newInstance().newDocumentBuilder();
        Document documentParsed = parser.parse(documentStream);
        DOMSource documentDomSource = new DOMSource(documentParsed);
} catch (Exception ex) {
        ex.printStackTrace();
}
```

Hierna wordt een functie aangeroepen welke deze DOM tree gebruikt om het bericht te identificeren:

```
private String identifyMessage(InputStream documentStream) {
        DocumentBuilder parser;
        try {
                for (CustomValidator validator : this.validatorList) {
                        if (validator.validate(documentDomSource) ) {
                                return validator.getSchemaName();
                        }
                }
                } catch (ParserConfigurationException e) {
                        e.printStackTrace();
        } catch (SAXException e) {
                        e.printStackTrace();
        } catch (IOException e) {
                        e.printStackTrace();
        }
        return "Not␣Identified!";
}
```

De functie *validate* van een *CustomValidator* object doet niets anders dan de standaard Java Validator de DOM tree laten valideren aan de hand van het XML Schema. Wanneer hier geen excepties bij optreden, is het bericht correct gevalideerd en wordt de waarde *true* terug gegeven. Wanneer hier wel een exceptie optreedt, dan kan het bericht om wat voor reden dan ook niet gevalideerd worden. Omdat we alleen geïnteresseerd zijn in identificatie, is deze reden van niet kunnen valideren voor ons niet interessant. We geven dan ook een *false* als waarde terug en doen verder niets met de inhoudelijke exceptie.

```
public Boolean validate (DOMSource ds) {
        try {
                this.validator.validate(ds);
        } catch (Exception ex) {
                return false;
        }
        return true;
}
```

11.3.2 XPath

We hebben eerder verschillende manieren besproken om, in plaats van alle eigenschappen van een bericht te controleren (het valideren), slecht een identificerende eigenschap te controleren. Wanneer

er meerdere identificerende eigenschappen voor een bericht zijn, hebben we verschillende criteria benoemd om de meest gunstige te selecteren, zoals diepte in de boom. Voor de implementatie maakt het echter niet uit welke identificerende eigenschap we gebruiken. Daarom gaan we er in dit voorbeeld voor het gemak vanuit dat de expressie $root \rightarrow A \rightarrow B$ de meest optimale identificerende eigenschap is voor berichttype A.

Wanneer we slechts een validerende eigenschap willen controleren, dan is het voldoende om deze eigenschap in een $XPathExpressie$ te vertalen en te kijken of deze $XPathexpressie$ meer dan 0 resultaten terug geeft. Immers, wanneer een eigenschap opgevraagd wordt in de boom en er komen een of meerdere resultaten terug, dan is deze eigenschap dus aanwezig. In de vorige sectie is te zien hoe alle identificerende eigenschappen per berichttype bepaald worden. Deze eigenschappen worden opgebouwd in de $XPath$ vorm in een string: $/root/A/B$. Voor iedere berichttype is een identificatie object aangemaakt, welke allemaal opgeslagen staan in de lijst *identificatorList* van het type *Vector<CustomIdentificator>*.

Voordat we $XPathExpressie$ uit kunnen voeren moet het te identificeren bericht eerst geparsed worden. Dit gebeurt middels een DOM parser, zodat er een geheugen structuur wordt opgebouwd waar we later mee kunnen werken. Het maakt hierbij voor de implementatie niet uit of we een normale of een lazy DOM parser gebruiken. De voorbeeld implementatie code is exact hetzelfde als bij het valideren van het bericht en dus ook in die sectie terug te vinden. We verkrijgen uiteindelijk een variabele $documentDomSource$.

Per $CustomIdentificator$ object moeten we vragen of deze het bericht als zijnde van zijn type kan identificeren:

```
for (CustomIdentificator identificator : identificatorList) {
        if (identificator.identify(documentDomSource)) {
                return identificator.getSchemaName();
        }
}
return "Not␣Identified!";
```

De functie $identify$ neemt de meest optimale identificerende eigenschap ($/root/A/B$), maakt hier een *XpathExpression* object van en laat deze los op de DOM-tree:

```
XPathFactory factory = XPathFactory
             .newInstance(XPathFactory.DEFAULT_OBJECT_MODEL_URI);
XPathExpression XPathExpr =
              factory.newXPath().compile(this.optimalExpression);

try {
  XPathExpr.evaluate(documentDomSource);
} catch (Exception ex) {
  return false;
}
return true;
```

De functie *evaluate* van een $XPathExpression$ object geeft alle gegevens terug welke middels de $XPath-expressie$ gevonden kunnen worden. Wanneer een $XPath-expressie$ niet uitgevoerd kan worden, dan wordt een exceptie opgeworpen. Wanneer we dus een exceptie afvangen, is deze identificerende eigenschap niet aanwezig. Wanneer we geen exceptie vangen, worden er blijkbaar gegevens gevonden bij de identificerende eigenschap als $XPath - expressie$ en is deze eigenschap dus aanwezig in het bericht. Het bericht is nu succesvol geïdentificeerd, dus kan een *true* waarde als resultaat van de functie gegeven worden.

11.3.3 Kortste in XML stream

Een bijzonder criterium om de meest optimale identificerende eigenschap te bepalen, was om te kijken welke identificerende eigenschap op de kortste afstand vanaf het eerste karakter van het XML bericht zit. Hierbij wordt dus niet gekeken naar een DOM-tree, maar naar het bericht nog voordat het geparsed is naar een DOM tree.

Voor deze methode van identificatie hebben we een *streamparser* nodig. Een SAX parser is zo'n *streamparser*. Tijdens het parsen van het bericht kunnen we telkens de identificerende eigenschappen controleren. Bij de eerste identificerende eigenschap die we tegenkomen is de validatie voltooid en kunnen we de parser stoppen.

Het nadeel van deze methode van identificeren is dat er heel veel routines heel vaak uitgevoerd worden. Namelijk, bij iedere opening van een element $< A >$ wordt gecontroleerd of hiermee een identificerende expressie gevonden kan worden. Deze methode parsed dus wel een zo klein mogelijk gedeelte van het bericht, echter, afhankelijk van hoeveel tijd de functies die per element uigevoerd worden kosten, kan het alsnog zijn dat het totaal langer duurt dan de andere identificatie methoden. Het is dus zaak om de functies die vaak aangeroepen worden zoveel mogelijk te optimaliseren. Om het effect van optimalisaties te kunnen bepalen, gebruiken we een voorbeeld situatie met bruikbare getallen:

- We hebben 3 mogelijke berichttypen
- Per berichttype hebben we 10 identificerende eigenschappen
- We proberen een binnenkomend bericht met 100 elementen te identificeren
- Het bericht kan niet worden geïdentificeerd. Dit leidt tot het worst-case scenario: het hele bericht moet worden geparsed.

Voor iedere berichttype is een lijst met identificerende eigenschappen aanwezig, in de XPath-expressie vorm ($/root/A/B$). Wanneer we door het eerste XML voorbeeld heenlopen, kunnen we soortgelijke expressies opbouwen. Wanneer we het begin van het root-element ($< root >$) tegenkomen kunnen we de expressie $/root$ opbouwen en in het geheugen opslaan. Verderop komen we dan het begin van de A-element tegen ($< A >$) en kunnen we de expressie uitbreiden tot $/root/A$. Wanneer we dan daarna het einde van het A-element tegenkomen, moeten we deze A er weer af halen en blijven we over met $/root$. Zo gaan we door tot we aan het einde weer met een lege expressie overblijven.

Voor de implementatie gebruiken we een standaard SAX parser en schrijven een eigen klasse om de SAX events af te handelen. Alleen de functie *startElement* en *endElement* worden geïmplementeerd. Voor het opbouwen van de expressie tijdens het parsen kunnen we geen standaard string gebruiken, omdat de expressie ook weer afgebroken moet worden, wanneer we het einde van een element tegen komen. Verder zijn string-operaties in Java erg duur, omdat een string immutable is. Dit betekent dat wanneer je een string $s2$ toe wil voegen aan een bestaande string $s1$, $s1$ niet uitgebreid wordt, maar er een volledig nieuwe string opgebouwd wordt waarin beide strings gekopieerd worden.

In plaats daarvan gebruiken we een $stringbuffer$, wat in feite hetzelfde is als een string, maar dan wel mutable. Verder gebruiken we een *stack* om de lengtes van de element-namen die we toevoegen aan de *stringbuilder* in volgorde in op te slaan, zodat we de expressie ook weer kunnen afbreken.

Alle variabelen die genoemd zijn, zijn globale variabelen:

```
private Stack<Integer> expressionBuilderLengths;
private StringBuilder expressionBuilder;
```

De functies die de expressie opbouwen en afbreken:

```
public void startElement (String elementNaam) {
  this.expressionBuilder.append("/");
  this.expressionBuilder.append(elementNaam);
  this.expressionBuilderLengths.push(elementNaam.length() + 1); //+1 voor "/"
  if (this.isIdentifying(this.expressionBuilder.toString())){
        stopParser();
  }
}
```

```
public void endElement(String elementName) {
        int iMax = this.expressionBuilder.length();
        int iElementLength = this.expressionBuilderLengths.pop();
        this.expressionBuilder.delete(i - iElementLength, i);
}
```

Telkens controleren we of de tot dan toe opgebouwde expressie een identificerende eigenschap voor een van de mogelijke berichttypen is. Dit doen we middels de functie $isIdentifying$. Deze functie werkt als volgt:

```
public boolean isIdentifying(expressie)
voor elke berichttype:
  voor elke identificerende expressie als identExpr binnen berichttype:
    Als expressie gelijk aan identExpr:
      return true
return false
```

Bovenstaande functie wordt 100 keer aangeroepen (één per element). Verder zijn er 3 berichttypen en per berichttype 10 identificerende eigenschappen. In totaal komen we hiermee uit op $100 * 3 * 10 = 3000$ operaties.

In plaats van de identificerend eigenschappen op te slaan in een lijst, kunnen we ze ook opslaan in een *hashmap*. Een *hashmap* is een structuur waarin waarden gekenmerkt worden door een sleutel. Zo kunnen woonplaatsen van personen als volgt opgeslagen worden:

```
HashMap map = new HashMap();
map.put("Piet", "Amsterdam");
map.put("John", "New York");
map.put("Nancy", "Groningen");

map.get("John");
map.containsKey("John");
```

Zowel de *map.get* als de *map.containsKey* functies vinden plaats in constante tijd en zijn erg snelle operaties. De eerste functieaanroep levert de string “New York” op, de twee de booleaanse waarde “true”.

We bouwen de *hashmap* door de identificerende eigenschappen als sleutels te gebruiken en lege waarden te houden. Hiermee kunnen we de routine aanpassen naar de volgende:

```
public boolean isIdentifying(expressie)
voor elke berichttype:
  als expressie in hashmap van berichttype:
    return true
return false
```

We moeten in ons voorbeeld voor ieder element (100 in totaal) in 3 hashmap structuren controleren of de identificerende eigenschap daarin voorkomt. Deze controle kan in $O(1)$ tijd gedaan worden, waardoor het niet meer uitmaakt dat hier 10 identificerende eigenschappen in zitten. Hiermee komen we op een totaal van $100*3 = 300$ operaties uit, wat ten opzichte van de originele 9000 reeds een hele verbetering is. We hebben nu echter nog een *hashmap* per berichttype. Verder zijn de waarden achter de sleutels nu nog leeg. Een verdere optimalisatie is om een centrale hashmap te maken en deze te vullen met alle identificerende sleutels van alle berichttypen als waarden. Als waarden achter de sleutels hangen we dan de naam van de berichttype waarvoor de betreffende sleutel identificerend is. Deze centrale *hashmap* noemen we *XPathExpressionMap_Overall*.

Hiermee kan de routine nog verder versimpeld worden tot:

```
public boolean isIdentifying(expressie)
return (XPathExpressionMap_Overall.hasKey(expressie))
```

Deze functie wordt in het worst-case scenario waarbij het bericht niet geïdentificeerd wordt nog steeds 100x (een keer voor ieder element) aangeroepen, waardoor het totaal aantal operaties nu op 100 uitkomt.

Behalve optimalisatie in tijd alleen, is het grote voordeel nu dat we een aantal belangrijke factoren hebben kunnen wegstrepen als het aan komt op tijd die nodig is om een bericht te identificeren. Het aantal identificerende eigenschappen (wat weer afhankelijk is van de complexheid van de verschillende berichttypen) en het aantal mogelijke berichttypen zijn nu niet meer van invloed op de performance. Dit betekent dat er geen enkele belemmering meer is om een groot aantal complexe berichttypen toe te staan op dezelfde Communication Channel.

Verder is de grootte van een bericht met deze methode alleen van toepassing in een worst-case scenario waarbij het bericht niet geïdentificeerd kan worden. In een goed doordacht landschap met correct functionerende systemen mag je verwachten dat systemen alleen berichten naar elkaar sturen welke ook door het andere systeem ontvangen kan worden.

Wanneer bij het parsen reeds bij het 10de element het bericht geïdentificeerd kan worden, dan wordt de parser gestopt. Dit betekent dat het voor deze identificatie methode niet uitmaakt of een te identificeren bericht 20, of 20000 elementen bevat.

12 Benchmark

12.1 Testopstelling

We maken gebruik van de HL7 [3] standaard als testset. De HL7 standaard is een veel gebruikte standaard voor het uitwisselen van zorg gerelateerde informatie. Zo kunnen patiënt gegevens uitgewisseld worden, maar ook meetresultaten, verrichtingen, gerelateerde financiële informatie, orders, et cetera.

De HL7 standaard is gebaseerd op flat-file formaat en dus niet op het XML formaat. Echter, dit flat-file formaat is één-op-één te vertalen naar een gelijkwaardig XML formaat. Hier zijn verschillende converters voor op de markt. We hebben twee HL7 berichten gekozen, namelijk ADT_A03 en ORU_R01. Deze formaten komen vaak voor in de communicatie tussen zorg systemen. Verder komen ze voor een deel overeen en voor een deel verschillen ze ook. Alle HL7 berichten verschillen van elkaar.

De berichten verschillen ook van elkaar qua eigenschappen. Zo is het ORU bericht beduidend complexer dan het ADT bericht: dit betekent dat het ORU bericht een XSD heeft van 66k, terwijl het ADT bericht een XSD heeft van 30k en ook dat het voorbeeld ORU bericht 4400k groot is, terwijl het ADT bericht slechts 423k groot is.

Voor onze test-opstelling maken we geen gebruik van een XI server. De reden hiervoor is dat een XI server geoptimaliseerd is voor het parallel verwerken van berichten. Echter, in onze test-opstelling maken we geen gebruik van parallel verwerking van de berichten. De XI server verstoort onze metingen teveel om handig als test-omgeving te functioneren.

In plaats van de modules op de XI server te laten draaien, is ervoor gekozen om een aantal test-classes te schrijven. Deze test-classes laden XML bestanden vanuit een directory en geven deze door aan de verschillende identificatie modules. Voor ieder berichttype is een representatief test-bestand gegenereerd. Ieder bestand wordt 10 keer geïdentificeerd door iedere module, om een goede meting te krijgen.

Wat we niet meenemen in de meting is het ontsluiten van het bericht vanuit de meegeleverde structuur (moduleData) en het weergeven van het resultaat van het identificeren. Wanneer we een van deze identificatie methoden rechtstreeks in de adapter implementeren, dan zijn deze overhead stappen namelijk helemaal niet meer nodig. De tijd die deze stappen kosten komen daarmee ook te vervallen en het is dus niet logisch om deze tijd dan in de meting mee te nemen.

De benchmark computer heeft ten tijde van de benchmark geen andere taken te voldoen en heeft voldoende werkgeheugen beschikbaar. De identificatie modules worden los van elkaar getest, om onderlinge invloeden (zoals een garbage collector die nog objecten van de vorige module aan het verwijderen is) te voorkomen. We middelen de tijden die gemeten worden.

12.2 Metingen

Alle metingen zijn in onderstaande tabel opgenomen. De minimale tijd (snelste identificatie), maximale tijd (langzaamste identificatie) zijn hier apart in opgenomen, net als de gemiddelde tijd. Alle metingen zijn in milliseconden (ms).

	1	2	3	4	5	6	7	8	9	10	min	max	avg
Validator: ADT A03	296	172	110	172	93	172	94	156	110	156	93	296	153
Validator: ORU R01	1140	1016	1031	1172	1094	1078	1063	1093	1078	1079	1016	1172	1084
XPath: ADT A03	157	109	109	47	110	47	93	63	94	46	46	157	87
XPath ORU R01	719	578	578	672	641	625	656	703	672	656	578	719	650
SaxParser: ADT A03	16	0	0	0	0	0	0	0	0	0	0	16	1,6
SaxParser: ORU R01	16	0	0	0	0	0	0	0	16	0	0	16	3,2

12.3 Resultaten

12.3.1 Validator

De validator identificatie methode is duidelijk de langste. Wanneer we met een runtime-analyser de performance meten, zien we dat bij het ADT bericht ongeveer 50% van de tijd gaat zitten in het parsen van het XML bericht en de rest in het uitvoeren van de XSD validatie. Het uitvoeren van de XSD validatie is een tijdrovende taak, omdat twee kanten op alles gecontroleerd moet worden. Eerst zal van alle elementen in de XML structuur gecontroleerd moeten worden (aan de hand van de XSD structuur) of deze elementen voor mogen komen. Daarna zal voor ieder verplicht elementen wat beschreven staat in de XSD gecontroleerd moeten worden of deze ook daadwerkelijk voor komt in de XML structuur. Zowel de grootte van het XML bericht als de grootte en complexiteit van het XSD document is van invloed op de tijd die het identificeren kost met deze methode.

12.3.2 XPath

Wanneer we naar de tijden van de XPath-identificatie methode gaan kijken, is het vanzelfsprekend dat deze methode sneller is. Het parsen van het XML bericht kost nog steeds even lang als bij de validatie methode, echter kost het uitvoeren van een XPathexpressie beduidend minder tijd dan het uitvoeren van een XSD validatie. In plaats van een bidirectionele alles-met-alles vergelijking, vragen we nu maar een element op uit de XML structuur. Wanneer we ook bij de XPath methode met een runtime-analyser de performance meten, dan zien we dat bij zowel het kleine ADT bericht als bij het grote ORU bericht het uitvoeren van de XPath expressie ongeveer even lang duurt. Het verschil in tijd zit hem dus met name in het parsen van het XML bericht. De grootte en/of complexiteit van het bijbehorende XSD bericht is echter niet meer van invloed op de snelheid van identificatie.

Bij beide methoden (zowel bij identificatie middels validatie als middels een XPathexpressie) variëren de tijden enigszins. Dit heeft met name te maken met het gedrag van de garbage collector van de java runtime engine.

12.3.3 SAX

Bij de SaxParser identificatie methode zien we dat de tijden afwisselen tussen 0ms en 15/16ms. Ook dit kan verklaard worden door de garbage collector die af en toe op de achtergrond zijn werk doet. Het wisselt wanneer dit gebeurt. In deze test-run blijkt dat de garbage collector bij het ADT berichten 2 keer intreedt en bij de ORU berichten 3 keer. Hierdoor lijkt het of de identificatie van de ORU berichten net iets langer duurt dan die van de ADT berichten. Voeren we de test echter meerdere malen uit, dan wisselt dit.

Bij de SaxParser vallen dus twee zaken op. In de eerste plaats is deze methode van identificeren veel sneller. In de tweede plaats valt op dat er geen verschil meer lijkt te zijn tussen een groot bericht zoals het ORU bericht en een klein bericht, zoals het ADT bericht.

De snelheid van deze methode van identificeren valt te verklaren aan de hand van het aantal elementen dat verwerkt moet worden. Onze test bestanden bestaan uit een groot aantal elementen. Het ADT bericht bevat ongeveer 9000 elementen, waar het ORU bericht ongeveer 75000 elementen bevat. Echter, beide berichten worden bij benadering in dezelfde tijd geïdentificeerd. De reden hiervoor ligt in het feit dat deze identificator stopt met parsen zo gauw als de identificatie is geslaagd. Blijkbaar kunnen beide berichten na ongeveer evenveel elementen te parsen geïdentificeerd worden, ongeacht hoeveel elementen daar nog achteraan komen.

Opvallend is de snelheid van de SaxParser, maar met name opvallend is dat het identificeren van een heel groot bericht, zoals het ORU bericht, niet langer duurt dan het identificeren van een

veel kleiner bericht zoals het ADT bericht. Dit is bij de andere twee manieren van identificeren wel het geval. Deze andere twee methoden moeten echter eerst het hele XML bericht parsen, voordat het identificeren kan beginnen. De tijd die het parsen kost is lineair afhankelijk van de grootte van het XML bericht. Echter, bij de SAX methode stoppen we zo gauw als het identificeren is gelukt. Stel dat dit het geval is bij het 30ste element. Het maakt hierna niet meer uit of het resterende deel 30 of 3000 elementen lang is.

In onze praktijk voorbeelden heeft het ADT bericht 8542 elementen en het ORU bericht heeft 75828 elementen. Identificatie vindt echter plaats na 9 elementen bij het ORU bericht en 12 elementen bij het ADT bericht. Dit is dus maar een fractie van het aantal elementen wat er is. Hierna kan het parsen van het bericht gestopt worden en de overige elementen worden dus genegeerd. Dit verklaart ook waarom het identificeren van beide berichten in < 1 ms kan gebeuren.

Worst case scenario Bij deze SAX Parser krijgen we dus voordeel van het feit dat al vrij vroeg in het XML bericht gezien kan worden om welk bericht het gaat. In het geval dat we het bericht niet kunnen identificeren lopen we echter wel tegen de situatie aan dat we het hele bericht van voor tot achter middels de SAX parser doorlopen voordat we kunnen concluderen dat we geen identificerende eigenschap kunnen vinden in het bericht. Ook deze worst-case scenario's zijn gemeten, door de root-tag van beide XSD berichten aan te passen, waardoor iedere expressie ineens ongeldig wordt. Hierbij werd gemeten dat het niet succesvol identificeren van een ADT bericht gemiddeld 32ms in beslag neemt en van een ORU bericht gemiddeld 230ms. Hierbij zien we dus dat de grootte wel degelijk van invloed is, wat ook te verwachten is.

Hoewel deze worst-case scenario's beduidend meer tijd in beslag nemen dan de succesvolle identificatie methoden, zijn er een paar zaken in overweging te nemen. In de eerste plaats kun je, zoals eerder al besproken, er vanuit gaan dat in een goed ontworpen en ingericht landschap een systeem alleen berichten aangeboden krijgt welke hij ook kan verwerken. Systemen sturen namelijk niet "random" berichten naar elkaar: het is van tevoren ingesteld dat systemen berichten naar elkaar versturen. Uiteraard stel je alleen berichten in om te versturen , die het ontvangende systeem ook kan ontvangen.

Verder is het worst-case scenario van deze SAX parser identificatie methode nog steeds beduidend sneller dan een normale identificatie met de XPath methode, namelijk 32ms worst-case SAX parser vs. 87 ms. voor de XPath methode met een ADT bericht en 230 ms vs. 650 ms voor een ORU bericht. De reden waarom deze methode ook nog sneller is wanneer het hele bericht geparsed moet worden (net zoals bij de XPath methode) ligt in het feit dat bij de XPath methode het bericht tijdens het parsen opgeslagen wordt in een geheugen structuur welke opgebouwd moet worden. Dit levert een substantiële overhead op, welke we bij deze SAX parser identificatie methode niet hebben, omdat we niets opslaan.

13 Conclusie

Uit de meetresultaten komt duidelijk naar voren dat de identificerende parser de snelste methode van identificeren is. Ook al is geheugen gebruik hier niet gemeten, is met zekerheid te stellen dat deze methode ook het minste geheugen zal gebruiken. Het XML document wordt immers niet geheel in het geheugen geladen als datastructuur, omdat we gebruik maken van een SAX parser.

Wel blijkt uit de meetresultaten dat er bij de identificerende parser een zeer groot verschil zit tussen best-case en worst-case scenario's. Echter, ook in het worst-case scenario is deze parser niet (veel) trager dan de alternatieven. Eerder is al beargumenteerd dat het worst-case scenario in een stabiele en goed ingerichte omgeving nooit voor zal komen.

Verder komt uit de tests naar voren dat de identificerende parser niet afhankelijk is van de grootte van het binnenkomende bericht, maar puur afhankelijk van hoever in het bericht gekeken moet worden om een succesvolle identificatie uit te voeren.

Omdat de identificerende parser bij een succesvolle identificatie significant sneller is dan zijn alternatieven en ook veel beter schaalbaar is naar veel grotere berichten, is te concluderen dat deze parser het best presteert onder alle voorkomende omstandigheden. Verder voldoet deze parser als enige aan de eerder gestelde eis van maximaal 40ms.

Referenties

[1] Mehmet Altinel and Michael J. Franklink. Efficient filtering of xml documents for selective dissemination of information. *internet*, -.

[2] XML Blueprint. Well-formed and valid xml documents. *internet*, -.

[3] HL7 Consortium. About health-level-7. *http://hl7.org/about*, -.

[4] W3 Consortium. Xml. *internet*, -.

[5] Raju Rangaswami Fernando Farfán, Vagelis Hristidis. Beyond lazy xml parsing. *internet*, -.

[6] Apache Foundation. Xerces j 2.0. *internet*, -.

[7] Apache Foundation. Xerces j 2.0: Howto use lazy parser. *internet*, -.

[8] Gnome. Acceptable response times. *http://library.gnome.org/devel/hig-book/stable/feedback-response-times.html.en*, -.

[9] Giorgio Satta Harry Bunt, John Carroll. New developments in parsing technology. -, -.

[10] Welf Lowe Markus L. Noga, Steffen Schott. Lazy xml processing. -, 2002.

[11] Gopalan Suresh Raj. The factory method (creational) design pattern. *http://gsraj.tripod.com/design/creational/factory/factory.html*, -.

[12] SAP. Sap netweaver platform. *internet*, -.

[13] SAP. Sap pi. *internet*, -.

[14] SAP. Xi sizing and performance guide. *internet*, -.

[15] W3 Schools. Xml schema. *internet*, -.

[16] Sun. Api documentation: Documentbuilder. *http://java.sun.com/j2se/1.5.0/docs/api/javax/xml/parsers/DocumentBuilder.html*, -.

[17] Sun. Api documentation: Javax documentbuilder. *internet*, -.

[18] Sun. Api documentation: Sax parser. *http://java.sun.com/j2se/1.5.0/docs/api/javax/ xml/parsers/SAXParser.html*, -.

[19] Sun. Java se. *internet*, -.

[20] Toshiro Takase, Hsashi MIYASHITA, Toyotaro Suzumura, and Michiaki Tasubori. An adaptive, fast, and safe xml parser based on byte sequences memorization. *internet*, -.

[21] Liquid Technologies. Liquid xml studio. *http://www.liquid-technologies.com/XmlStudio/XmlStudio.aspx*, -.

[22] W3C. Document object model (dom). *internet*, -.

[23] W3C. W3 consortium. *internet*, -.

[24] W3C. Xml well-formedness. *internet*, -.

[25] Wikipedia. Message broker. *internet*, -.

[26] Wikipedia. Sap pi. *internet*, -.

[27] Wikipedia. Sax parser. *internet*, -.

[28] Wikipedia. Xml schema. *internet*, -.