Het voorkomen van prestatieverlies bij de koppeling met Web services

... en de rol van standaardisatie daarbij

Universiteit Twente

Pim Sierhuis (s0002577)

pim@sierhuis.com

Augustus 2007

Opdrachtgever: Huijsmans & Kuijpers Automatiseringsbureau

Begeleider opdrachtgever: Paul Lucassen

Begeleider universiteit: Maarten Fokkinga

Samenvatting

Na te hebben geconstateerd dat de koppeling met onafhankelijke systemen leidt tot uitdagingen met betrekking tot het bieden van garanties over de prestaties van het systeem, wordt in dit verslag een voorstel gedaan om deze prestatiegevolgen af te schermen. Hiertoe wordt aan de hand van een aantal cases uit de "echte wereld" een pakket van afbakenende aannames opgesteld. De belangrijkste aannames zijn dat het gaat om gegevensontsluitende Web services, die de gegevens structureren in de vorm van identificeerbare "records" en dat het geen probleem is de gebruiker niet de meest actuele gegevens te leveren.

Bij het ontwerp wordt eerst een universele interface naar gegevensontsluitende systemen gedefinieerd, zodat de oplossing algemeen wordt over de groep systemen die (al of niet met behulp van een "adapter") conformeren aan deze interface. Op basis hiervan wordt een optimalisatiesysteem ontworpen, dat het probleem onderverdeeld in twee categorieën:

- De categorie waarin zoekoperaties op de gegevens nodig zijn. Hierbij worden de problemen opgelost met een algemeen replicatieproces, zodat de zoekoperaties lokaal kunnen worden uitgevoerd.
- De categorie waarin de gegevens slechts worden gelezen aan de hand van hun identifier. Bij systemen van deze categorie worden de gegevens gecacht, zodat minder vaak met het gekoppelde systeem hoeft te worden gecommuniceerd.

Bij de invulling van het ontworpen systeem wordt gebruik gemaakt van algemene componenten, die zijn gedefinieerd aan de hand van de algemene interface. Hierbij kan worden gedacht aan een algemene projectie van de interface op XML en algemene componenten voor het lezen en schrijven van een SQL-database.

De twee belangrijkste conclusies zijn als volgt:

- Ook bij de kwestie van de afscherming van prestatiegevolgen biedt het begrip "standaardisatie" weer grote voordelen. Door de standaardisatie van de interface, kunnen de prestatiegevolgen met een *algemeen* systeem worden afgeschermd. Dit maakt het mogelijk niet bij elk project maatwerksoftware te hoeven implementeren om de steeds terugkerende prestatiegevolgen af te schermen.
- Een tweede belangrijke conclusie is de volgende: omdat Web services zo algemeen zijn, is het erg lastig hier algemene componenten voor te ontwikkelen. Daaruit blijkt de wenselijkheid van verdere standaardisatie van Web services, door middel van een extra niveau van specifiekere standaarden (zoals de in dit project gedefinieerde interface naar gegevensontsluitende services). Op het moment dat dergelijke standaarden zouden bestaan en worden gebruikt, kunnen namelijk ook hiervoor algemene componenten worden ontwikkeld, waardoor het applicatiespecifieke programmeerwerk verder wordt teruggebracht.

Voorwoord

In dit document wordt verslag gedaan van mijn afstudeerproject aan de Universiteit Twente, voor de opleiding Technische Informatica.

Mijn keuze voor Web services in combinatie met de prestatiegevolgen daarvan, was vrij vanzelfsprekend. In eigen projecten ben ik hier namelijk al een aantal keer mee geconfronteerd. Het intrigeerde me dat het probleem steeds in vrijwel dezelfde vorm terugkwam, wat toch de mogelijkheid tot een algemene oplossing zou moeten impliceren. Aangezien het doen van abstracties een soort automatisme en "hobby" van mij is, lag het voor de hand een dergelijk algemeen systeem te ontwerpen en implementeren. Helaas ontbrak hiervoor de tijd. Toen vervolgens duidelijk werd dat ook HKA met hun Bicat-systeem gebruik wilde gaan maken van Web services om centrale gegevens te benaderen, realiseerde ik me dat dit een goede kans was om van dit idee een afstudeerproject te maken.

Ik ben HKA (en in het bijzonder natuurlijk mijn begeleider daar, Paul Lucassen) dan ook dankbaar dat ze mij deze kans hebben geboden, terwijl een algemeen systeem voor hen wellicht minder nuttig is dan wat zou zijn bereikt als er meer in de richting van hun systemen was gewerkt.

Daarnaast gaat mijn dank uit naar mijn begeleider van de universiteit, Maarten Fokkinga, en alle anderen die mij hebben geholpen en gesteund bij dit project.

Pim Sierhuis, augustus 2007

Inhoudsopgave

1. Inleiding 1

1.1. Koppeling van systemen 1
1.2. Prestatiegevolgen 1
1.2.1. Voorbeeld 1
1.2.2. De probleemstelling 2
1.3. Aanleiding 2
1.3.1. Eigen ervaring 2
1.3.2. Opdrachtgever 2
1.4. Opbouw van het verslag 3

2. Analyse en afbakening 4

2.1. Gerelateerd werk 4
2.2. Aannames op voorhand 5
2.2.1. Web service techniek 5
2.2.2. Beïnvloedbaarheid van de Web service 5
2.2.3. Gegevensontsluiting 5
2.2.4. Zelfstandigheid van de Web service 5
2.3. Aspecten van gegevensontsluitende systemen 6
2.3.1. Lezen en schrijven 6
2.3.2. Structuur van gegevens 6
2.3.3. Type operaties op de gegevens 7
2.4. Concrete cases 7
2.4.1. Postcodegegevens 7
2.4.2. Koppeling met online Customer Relationship Management-systeem 8
2.4.3. Titelgegevens van een bibliotheekcollectie 8
2.5. Verdere afbakening 9
2.5.1. Records 9
2.5.2. Meerdere soorten records 9
2.5.3. Identificatie van records 10
2.5.4. Alleen lezen 10
2.5.5. Soorten selecties 10
2.5.6. Niet-synchrone gegevens 10
2.5.7. Meer leesoperaties dan wijzigingen op de gegevens 10
2.6. Overzicht afbakening 11

3. Programma van Eisen 12

3.1. Functionele eisen 12
3.1.1. Universele “gegevensontsluitende Web service” interface 12
3.1.2. Cache 12
3.1.3. Replicatie 13
3.2. Non-functionele eisen 14
3.2.1. Prestaties 14
3.2.2. Flexibiliteit 14

4. Universele interface naar gegevensontsluitende systemen – RecordIF 15

4.1. Gerelateerd werk 15
4.2. Definitie 15
4.2.1. Query 16
4.2.2. Resultaat 16
4.3. Timestamps 17
4.3.1. Eigenschappen 17
4.3.2. Replicatie 17
4.3.3. Implementatie notities 18
4.4. Verwijderingen 18
4.5. Definitie in Java 18
4.6. Projectie op XML 20
4.6.1. Query 21
4.6.2. Resultaat 22
4.7. Een algemene Web service 23

5. Ontwerp 25
5.1. Globaal ontwerp 25
5.1.1. Web service adapter 25
5.1.2. Aggregatie van de Web service – de Multiplexer 26
5.1.3. Alle opgevraagde records naar de cache schrijven – de CacheWriter 26
5.1.4. Een eenvoudige cache – de CachedRecordIF 26
5.1.5. Replicatie – de ReplicaManager 27
5.2. Algemene componenten op basis van RecordIF 27
5.2.1. Algemene interface naar SQL-databases 27
5.2.2. “Doorgeef” RecordIF die schrijft naar een SQL-database 27
5.2.3. Projectie van query en resultaat op XML 27
5.2.4. Parsen van query en resultaat uit XML 28
5.2.5. Algemene Web service / client 28
5.3. Invulling ontwerp met algemene componenten 29
5.3.1. Het optimalisatiesysteem 29
5.3.2. Adapters en gegevensontsluitende systemen 30
6. Implementatie in Java 33
6.1. Algemene interface naar SQL-databases 33
6.1.1. Lezen 33
6.1.2. Schrijven 34
6.2. Projectie op XML 34
6.2.1. Schrijven 35
6.2.2. Lezen 35
6.3. Algemene Web service / client 36
6.3.1. SOAP-berichten 36
6.3.2. Algemene Web service 37
6.3.3. Algemene client 37
7. Validatie 38
7.1. Centrale titelcatalogus 38
7.1.1. Implementatie 38
7.1.2. Verwijderingen 38
7.1.3. Replica initialisatie 38
7.1.4. Prestatiegevolgen 39
7.1.5. Overige kwesties 41
7.2. Adresgegevens uit gemeentelijk systeem 41
7.3. Adresgegevens aan de hand van postcodes 41
7.4. Salesforce 42
7.4.1. Implementatie adapter 42
7.4.2. Prestatieverbetering 42
7.5. Algemene evaluatie 42
7.5.1. Afscherming prestatiegevolgen 42
7.5.2. Hoeveelheid applicatiespecifiek programmeerwerk 43
8. Conclusie 44
8.1. Aanbevelingen 44
9. Bronnen 45
10. Bijlagen 47
10.1. XML-schema voor queries 47
10.2. XML-schema voor resultaten 48
10.3. WSDL-definitie van algemene Web service 49

1. Inleiding

In dit document wordt verslag gedaan van een onderzoek naar de prestatiegevolgen die optreden bij de (los)koppeling van systemen. Na deze kwestie te hebben onderzocht en op een relevante manier te hebben afgebakend, zal er een oplossing worden ontworpen en geïmplementeerd. Daarbij wordt een algemene Web service interface gedefinieerd, om de oplossing zo universeel mogelijk te maken. Vervolgens zal aan de hand van een aantal cases worden besproken hoe deze oplossing kan worden ingezet in de “echte wereld”.

Er wordt hierbij uitgegaan van een “client / server”-relatie tussen de deelsystemen, in de specifieke vorm van “Web Services”. Verder wordt verondersteld dat een dergelijke service bepaalde functionaliteit biedt, namelijk functionaliteit die kan worden omschreven als de “ontsluiting van gegevens”.

In de rest van deze inleiding zal in meer woorden worden beschreven wat hiermee wordt bedoeld.

1.1. Koppeling van systemen

In de architectuur van moderne bedrijfsapplicaties is vaak te zien dat het systeem is opgesplitst in onafhankelijke deelsystemen. In veel gevallen is het zelfs zo dat bepaalde deelsystemen zijn ontwikkeld door andere instanties. Voor zulke deelsystemen geldt verder dikwijls dat ze (grootschalig) zijn gecentraliseerd en door vele andere systemen worden gebruikt. Een voorbeeld hiervan is “DigiD”, het systeem waarmee de Nederlandse overheid burgers authenticeert voor verscheidene overheidszaken.

Dit koppelen van verschillende (deel-) systemen brengt uitdagingen met zich mee. Zo zullen verschillende systemen die zijn ontwikkeld met behulp van verschillende programmeertalen en draaien op verschillende besturingssystemen en hardware, gegevens ook op verschillende manieren opslaan en willen uitwisselen.

Op dit gebied is de afgelopen jaren grote vooruitgang geboekt. Hierbij kan worden gedacht aan de ontwikkeling van XML en de daarop gebaseerde Web services. Hiermee is het mogelijk om systemen een gestandaardiseerde, interoperabele “interface” te geven. Deze techniek wordt op steeds grotere schaal toegepast en lijkt de standaardmethode te worden om systemen aan elkaar te koppelen.

1.2. Prestatiegevolgen

Nu heeft deze behoefte om systemen op steeds grotere schaal aan elkaar te koppelen ook een keerzijde. Waar men bij een ouderwetse, “monolithische” architectuur de controle had over het gehele systeem, geldt dit bij een architectuur met onafhankelijke deelsystemen niet. Web services lossen het functionele deel van dit probleem aardig op, maar de prestatiekwesties die erbij komen kijken kunnen soms tot grote uitdagingen leiden.

1.2.1. Voorbeeld

Om dit te verduidelijken volgt nu een voorbeeld van een hypothetisch bibliotheeksysteem. Dit systeem heeft een op zichzelf staand deelsysteem voor het bijhouden van het ledenbestand. Het verkrijgt de gegevens over de leden van de bibliotheek via een op Web services gebaseerde interface van dit ledendeelsysteem. Deze manier van werken biedt de mogelijkheid dit deelsysteem te vervangen door een ander systeem, mits het nieuwe systeem dezelfde interface biedt.

Nu wordt vanuit de bibliotheekbranche besloten dat het nuttig zou zijn om een landelijk ledenbestand op te zetten, zodat mensen zich niet meer bij verschillende bibliotheken afzonderlijk hoeven in te schrijven. Functioneel gezien zou dit voor onze bibliotheek geen probleem zijn; aangenomen dat ook het landelijke systeem een compatibele interface heeft, kan simpelweg het oude ledendeelsysteem worden afgekoppeld en het nieuwe, landelijke systeem worden aangesloten.

Echter, het oude deelsysteem draaide op een lokale server die het niet erg druk had met het bijhouden van het ledenbestand. Bij het ontwerp van de rest van het systeem is hier (bewust of onbewust) rekening mee gehouden, zoals bijvoorbeeld door het direct (“blocking”) aanroepen van de Web services van het ledensysteem als de details van een bepaald lid aan de gebruiker moeten worden getoond. Het landelijke systeem heeft het echter een stuk drukker: een aanroep kan soms wel

een aantal seconden duren. Terwijl het systeem functioneel volledig correct functioneert, is het onbruikbaar geworden doordat de vertraging leidt tot grote irritatie van de gebruikers.

1.2.2. De probleemstelling

De oplossing van de kwesties die bestaan bij traditionele, monolithische systemen, namelijk een architectuur met zwak gekoppelde deelsystemen, brengt dus een potentieel prestatieprobleem met zich mee. Een direct gevolg van zwak gekoppelde systemen is immers (per definitie) dat ze weinig invloed op elkaar hebben, dus ook met betrekking tot de prestaties. Om deze reden kunnen er door een (deel)systeem dat gebruik maakt van een ander (deel)systeem geen aannames worden gedaan over de prestaties van het andere systeem. Als de ontwerper van zo'n systeem daarom enige performance wil bereiken, zal er daarom van uit moeten worden gegaan dat het andere systeem *slechte* prestaties levert. In sommige gevallen, zoals bij het interactieve deel van systemen, leidt dit tot uitdagingen. Deze uitdagingen zijn het onderwerp van dit project.

Er zal een systeem worden ontworpen dat het hierboven besproken probleem vermindert. Zo is gekomen tot de volgende probleemstelling:

> Het doel is om een systeem te ontwerpen dat de communicatie met een **gegevensontsluitende Web service** optimaliseert met als doel de gebruiker van deze service zo veel mogelijk af te schermen voor de **prestatiegevolgen** van deze koppeling.

Om tot een concrete oplossing te kunnen komen, wordt aangenomen dat de Web service voldoet aan de aannames die zullen blijken uit de verdere analyse van het probleem. Deze analyse zal worden behandeld in het volgende hoofdstuk. Verder kan er niet van uit worden gegaan dat de gebruiker *geheel* voor de prestatiegevolgen zal worden afgeschermd.

1.3. Aanleiding

De aanleiding voor dit project is tweeledig. Hieronder worden deze twee redenen apart besproken.

1.3.1. Eigen ervaring

Ten eerste leert de ervaring dat bij het ontwikkelen van systemen die een interactief deel hebben en gekoppeld worden met een ander systeem, de prestaties (zoals de responstijd) veelal worden beperkt door dit andere systeem. In veel gevallen wordt deze koppeling gelegd via Web services en wordt een dergelijke Web service gebruikt voor de betrekking van gegevens. Na dit diverse keren ervaren te hebben, kwam het besef dat dit een algemeen probleem is en de wens hier een algemenere oplossing voor te ontwikkelen.

1.3.2. Opdrachtgever

De opdrachtgever is Huijsmans & Kuijpers Automatiseringsbureau (HKA). HKA is de ontwikkelaar van Bicat, een systeem voor de automatisering van openbare bibliotheken. Bicat bestaat reeds ongeveer 18 jaar en wordt gebruikt bij meer dan 400 Nederlandse openbare bibliotheken. Het biedt functionaliteit voor het ondersteunen van de meeste bedrijfsprocessen, zoals lenersadministratie, beheer van de collectie, uitlening en financiën.

De tweede aanleiding voor dit project is het verschijnen van een rapport van de Vereniging van Openbare Bibliotheken (VOB), genaamd “Een informatiearchitectuur voor Openbare bibliotheken” [VOB2006]. Dit rapport is het product van een werkgroep van de VOB, die is opgezet na de constatering dat werd aangelopen tegen de beperkingen van de huidige ICT-voorzieningen binnen de branche. De werkgroep beoogt invloed uit te oefenen op de ontwikkelaars van bibliotheeksystemen (zoals Bicat).

In dit rapport worden concepten gegeven voor een gewenste architectuur van bibliotheeksystemen. Daarin worden deze systemen opgesplitst in verschillende deelsystemen, zoals de lenersadministratie en het collectiebeheer. Daarbij wordt gedacht aan het gebruik van Web services voor de koppelingen tussen de deelsystemen. Verder worden voor in de toekomst concepten als Service-Oriented

Architecture (SOA) en Enterprise Service Buses (ESBs) aangehaald[1]. Dit benadrukt de relevantie van Web services bij de in de toekomst gewenste architectuur.

Daarnaast wordt Bicat op dit moment herschreven van de huidige programmeertaal Foxpro naar Java. Daarom is dit een extra gunstig moment om nieuwe concepten als Web services te introduceren.

Ofschoon Bicat hier wel op voorbereid wordt door er rekening mee te houden in de architectuur, is er in de huidige fase van ontwikkeling nog geen sprake van een echte SOA. Er zijn echter wel verschillende plekken waar wordt gecommuniceerd met andere systemen of de wens bestaat om dit te doen. Hierbij kan worden gedacht aan: een landelijke database met titelgegevens, de betrekking van NAW-gegevens van leners uit de Gemeentelijke Basis Administratie (GBA) en het controleren van postcodes via externe partijen.

Omdat het hierbij gaat om niet-lokale systemen waarvan de snelheid niet kan worden gegarandeerd en deze worden gebruikt voor de betrekking van gegevens, past deze kwestie goed binnen dit project.

1.4. Opbouw van het verslag

Dit verslag is als volgt opgebouwd: In hoofdstuk 2 wordt een diepere analyse gemaakt van het probleem en een nauwkeurigere afbakening opgesteld. Vervolgens worden in hoofdstuk 3 de eisen besproken waaraan de oplossing moet voldoen. Daarna zal in hoofdstuk 4 een universele interface naar gegevensontsluitende systemen worden gedefinieerd, om de oplossing algemeen te maken. In hoofdstuk 5 wordt het ontwerp besproken en in hoofdstuk 6 de implementatie daarvan in Java. Ter validatie van het ontwerp, zal in hoofdstuk 7 worden beschreven hoe de oplossing kan worden ingezet bij de verschillende cases. Tenslotte worden in hoofdstuk 8 de nodige conclusies getrokken.

[1] Voor een uitleg over SOAs en ESBs, zie bijvoorbeeld [BAL2005]

2. Analyse en afbakening

In dit hoofdstuk zal het probleem, zoals dat in de inleiding is omschreven, verder worden geanalyseerd. Dit wordt gedaan om goed een onderbouwde probleemstelling en eisen te kunnen opstellen.

Hiertoe zal eerst aandacht worden besteed aan het gerelateerde werk in de literatuur. Daarna zal een aantal aannames worden gedaan, die op voorhand al kunnen worden gesteld. Vervolgens worden er aspecten omschreven van de daarmee afgebakende groep systemen (gegevensontsluitende Web services), aan de hand waarvan het probleem later verder kan worden afgebakend. Dan wordt een aantal concrete "cases" beschreven, waarbij in het bijzonder wordt gelet op de daarvoor omschreven aspecten. Op basis hiervan wordt vervolgens een gegronde verdere afbakening van het probleem gedaan, zodat deze kan dienen als afbakening voor de probleemstelling van de uiteindelijke oplossing.

2.1. Gerelateerd werk

Er zijn veel studies gedaan naar de prestaties van Web services [CHI2002], [JUS2003] en service composities [JIN2004], [GRU2006]. Hierbij wordt over het algemeen geconcludeerd dat Web services relatief slecht presteren, als gevolg van het feit dat er gebruik wordt gemaakt van XML. Zo wordt geconcludeerd dat Web services in termen van snelheid slechter presteren dan bijvoorbeeld CORBA en Java RMI [JUR2004]. Ook naar oplossingen voor dit type prestatieverlies is onderzoek gedaan, waarbij bijvoorbeeld kan worden gedacht aan slimmer parsen [JUN2006] en aan het slimmer routen van het netwerkverkeer.

Dit project richt zich echter niet op prestatieverlies dat inherent is aan Web services door de gebruikte technieken, maar op potentieel prestatieverlies doordat de Web service buiten de verantwoording ligt van het gebruikende systeem, waardoor geen aannames kunnen worden gedaan over de prestaties.

Onderzoeken die dit type prestatieverlies onder de loep nemen zijn een stuk schaarser. Deze vallen over het algemeen in twee categorieën:

- Het afspreken van garanties met betrekking tot de prestaties van een Web service [TIA2003]. Dit kan in sommige gevallen een goede oplossing zijn voor het probleem. In veel gevallen is het echter onwenselijk of praktisch onmogelijk om dergelijke garanties te bieden. Vooral in het geval van Web services die grote hoeveelheden verschillende gebruikende systemen bedienen is dit vaak onpraktisch. Daarnaast is het in veel gevallen de bedoeling om gebruik te maken van een reeds bestaande Web service, waar dergelijke functionaliteit ontbreekt en de aanbieder deze functionaliteit (ook in de toekomst) niet zal implementeren.
- De tweede categorie is het "standaard" cachen van berichten [ELB2004], [DEV2003]. Op het moment dat alle aanvragen en daarbij horende antwoorden worden opgeslagen, kan bij een reeds opgeslagen aanvraag het bijhorende antwoord worden teruggegeven zonder opnieuw met de Web service te communiceren. Hierbij worden de aanvragen en antwoorden niet geïnterpreteerd, om een zo algemeen mogelijke oplossing te creëren. Dit type oplossing introduceert echter een aantal kwesties:
    - Een specifieke aanvraag (XML-document) kan op verschillende manieren worden omgezet naar gegevens die over de netwerkverbinding worden verstuurd. Hierbij kan bijvoorbeeld worden gedacht aan witruimte (spaties / enters) en het op verschillende plekken introduceren van namespaces. Een oplossing hiervoor zou zijn om de aanvragen te canoniseren, voordat ze worden opgeslagen. [TAK2002]
    - Het kan zijn dat sommige gegevens in een aanvraag geen echte betekenis hebben. Een voorbeeld hiervan is een aanvraag waarin het tijdstip van versturen is opgenomen. Hierdoor zullen twee gelijke aanvragen toch geen cache-hit opleveren.
    - Een aanvraag heeft niet altijd de aard te kunnen worden gecacht. Zo kan een aanvraag betekenen dat er gegevens moeten worden gewijzigd. Als een dergelijke aanvraag zou worden gecacht, zouden latere wijzigingen met hetzelfde aanvraagbericht niet worden doorgevoerd. Een oplossing hiervoor is het per type aanvraag aangeven of deze wel of niet kan worden gecacht, zoals in [RAM2004]

- Verschillende aanvragen kunnen dezelfde gegevens bevatten. Zo zullen zowel een aanvraag voor een specifiek stuk gegevens, als een aanvraag voor alle gegevens, gedeeltelijk overlappende gegevens opleveren. Deze kwestie is inherent aan het niet interpreteren van de aanvragen en antwoorden.

2.2. Aannames op voorhand

Hieronder worden enkele belangrijke, algemene aannames gedaan. De aannames zijn zo gekozen dat ze een vrij algemene, vaak voorkomende groep van situaties beschrijven en toch zorgen voor een nuttige, concrete afbakening.

2.2.1. Web service techniek

Een mogelijke definitie voor "Web service" is: een software systeem dat is ontworpen om op een interoperabele manier machine-naar-machine-interactie over een netwerk mogelijk te maken en waarvan de interface in een machine-leesbaar formaat is gespecificeerd [W3C2004].

In dit project wordt daarnaast aangenomen dat gebruik wordt gemaakt van het Simple Object Access Protocol (SOAP [W3C2000]) voor de uitwisseling van gegevens en het Web Services Description Language (WSDL [W3C2001]) voor het specificeren van de beschikbaar gestelde interface. Dit is de meest gangbare vorm van het gebruik van web services en vereenvoudigt het probleem onder andere doordat hierbij per definitie gebruik wordt gemaakt van XML.

2.2.2. Beïnvloedbaarheid van de Web service

In sommige gevallen zal het mogelijk zijn om via de ontwikkelaars van een Web service bepaalde eigenschappen van een Web service te veranderen. Op die manier zou het bijvoorbeeld mogelijk zijn om een oplossing te ontwerpen waarbij er tussen de client en de Web service wordt "onderhandeld" over garanties met betrekking tot bepaalde prestatieaspecten.

Er wordt echter van uitgegaan dat in veel gevallen de prestatiekwesties evenredig zullen zijn met de mate van centralisatie van de Web service. Met de mate van centralisatie wordt in dit geval bedoeld hoeveel verschillende systemen er van de Web service gebruik maken. Verder wordt ervan uitgegaan dat de beïnvloedbaarheid omgekeerd evenredig is met de hoeveelheid verschillende systemen die gebruik maken van de Web service (omdat aanpassingen invloed zouden hebben op al deze systemen). Dit betekent dat juist de systemen waar prestatiekwesties een rol spelen geen gebruik kunnen maken van een oplossing die ervan uitgaat dat de Web service kan worden beïnvloed.

Om deze reden wordt dan ook aangenomen dat een Web service niet kan worden beïnvloed en vervalt dus de in de eerste alinea geschetste oplossing.

2.2.3. Gegevensontsluiting

De flexibiliteit van Web services heeft als gevolg dat aanroepen van alles kunnen betekenen en allerlei verschillende soorten neveneffecten kunnen hebben. Zo kan het zijn dat met een aanroep een bepaald proces in gang wordt gezet, een bepaalde gebeurtenis wordt gemeld, of een achterliggende database wordt geraadpleegd.

De prestatiekwesties zijn het meest nadrukkelijk aanwezig bij Web services die een interface bieden naar een database. Deze hebben namelijk in hun aard dat ze in een naïef ontworpen applicatie vaak worden aangeroepen om bijvoorbeeld gegevens aan de gebruiker te tonen.

Verder zijn de meeste denkbare services die een rol spelen bij de opdrachtgever van dit type en geldt dit ook voor de meeste services waarmee vóór dit onderzoek ervaring is opgedaan.

Om deze redenen wordt voor dit project uitgegaan van Web services die op één of andere manier gegevens ontsluiten en waarmee eventueel wijzigingen op deze gegevens doorgevoerd kunnen worden.

2.2.4. Zelfstandigheid van de Web service

Er wordt van uitgegaan dat de betreffende Web services, naast de eventueel daarvoor door henzelf geboden functionaliteit, ook zelfstandig de achterliggende gegevens wijzigen. Verder wordt aangenomen dat het systeem geen functionaliteit biedt om, in het geval van wijzigingen, belangstellenden hiervan op de hoogte te stellen ("Observer design pattern" [GOF1994]). Deze

aannames impliceren onder meer dat voor twee exact dezelfde aanvragen, zonder dat de aanvrager dit kán weten, verschillende resultaten kunnen worden teruggegeven. Dit betekent dat er bij een cache-achtige oplossing van het probleem inconsistenties kunnen ontstaan tussen de cache en de werkelijke gegevens.

2.3. Aspecten van gegevensontsluitende systemen

Nu het probleem is afgekaderd tot de koppeling van gegevensontsluitende systemen via Web services, kan onderzoek worden gedaan naar verdere aspecten van deze groep systemen. Deze worden hieronder besproken en later gebruikt om het probleem op een gegronde manier verder af te bakenen.

Al wordt er nog geen definitieve aanname gedaan over het soort oplossing, bij de omschrijving van sommige aspecten wordt rekening gehouden met een cache-achtige oplossing. Hiermee wordt een optimalisatie-laag bedoeld waardoor alle daadwerkelijke Web service aanroepen worden gedaan. Daarin worden dan de betreffende gegevens opgeslagen voor hergebruik bij aanroepen waarvan reeds een soortgelijke aanroep is langsgekomen.

2.3.1. Lezen en schrijven

Om het probleem voor een bepaalde Web service te kunnen oplossen, is het noodzakelijk om per type aanvraag aan te kunnen geven wat de aard is van de achterliggende operatie. Het gaat hierbij om het onderscheid tussen operaties die iets veranderen aan de toestand van het systeem en operaties waarbij dit niet het geval is. Specifieker voor dit project wordt hiermee bedoeld of een operatie gevolgen heeft voor het resultaat van latere operaties of niet. Hierna zal worden gesproken over respectievelijk schrijf- en leesoperaties.

Zoals gezegd, impliceert de aanname dat een systeem zelfstandig de onderliggende gegevens wijzigt, dat er bij een cache-achtige oplossing inconsistenties kunnen ontstaan tussen de cache en de werkelijke gegevens. Nu is dit altijd een probleem bij een dergelijke oplossing, waar rekening mee kan worden gehouden. De echte complexiteit ontstaat echter, als de client ook schrijfoperaties op de lokale gegevens uit gaat voeren, die vervolgens moeten worden gesynchroniseerd met de Web service. Er zal dan minstens bepaalde extra functionaliteit moeten worden geboden door de Web service en er zal gebruik moeten worden gemaakt van een “consistentie protocol” [TAN2002].

2.3.2. Structuur van gegevens

Ook al wordt aangenomen dat de gegevens worden gerepresenteerd door XML-documenten, er bestaat alsnog een onbegrensde hoeveelheid mogelijkheden om gegevens op XML te projecteren.

Bij een cache-achtige oplossing leidt dit tot een keuze, namelijk of de oplossing op de hoogte is van de structuur van de gegevens en deze dus kan interpreteren, of dat dit niet het geval is.

In het laatste geval kan de optimalisatielaag niet veel meer doen dan de aanroepen canoniseren, de aanroep met het antwoord opslaan en bij een volgende aanroep waarbij van de canonieke vorm reeds een antwoord is opgeslagen, dit antwoord teruggeven [TAK2002]. De wenselijkheid hiervan hangt mede af van de gewenste verdere operaties op de lokale cache. Zo is het op deze manier niet op een gangbare manier mogelijk om te zoeken binnen de cache. Verder zal het niet mogelijk zijn om op een nette manier schrijfoperaties toe te staan, omdat het voor de optimalisatielaag niet mogelijk zal zijn om te achterhalen welke invloed de schrijfactie zal hebben op de gecachte aanroepen.

In het geval dat de optimalisatielaag wel op de hoogte is van de structuur van de gegevens en dus in staat is deze te interpreteren, ontstaan er extra mogelijkheden met betrekking tot functionaliteit. Zo zal het in veel gevallen mogelijk zijn om afgebakende stukken informatie (hierna te noemen “records”) te onderscheiden die uniek kunnen worden geïdentificeerd met behulp van een identifier (“id”). Is dit het geval, dan kan gebruik worden gemaakt van extra functionaliteit die wordt geboden door veel van dergelijke Web services, zoals:

- Het ophalen van meerdere records tegelijk, die vervolgens als losse records in de cache kunnen worden opgeslagen. Dit is efficiënter dan het ophalen van één record per request en geldt zeker in het geval van grote databases waarbij zoekacties op de cache moeten kunnen worden uitgevoerd als voordeel.

- Het ophalen van de laatst gewijzigde records, zodat de consistentie van de cache met de Web service niet aannemelijk hoeft te worden gemaakt aan de hand van een maximale levensduur of aan de hand van het opnieuw uitvoeren van gecachte aanroepen.
- Het mogelijk maken van schrijfoperaties, omdat, door de kennis van de structuur van de gegevens, bij een schrijfactie tevens de gegevens in de cache kunnen worden gewijzigd of de betreffende leesoperatie opnieuw kan worden geconstrueerd aan de hand van het id. Overigens ontstaan er nieuwe kwesties bij het toestaan van schrijfoperaties, zoals de noodzaak om (de database achter) de Web service te kunnen locken en / of de afhankelijkheid van een notie van versies van records.

2.3.3. Type operaties op de gegevens

Bij sommige applicaties is het voldoende om gegeven een id, het bijbehorende record te kunnen opvragen. In dat geval kan een oplossing worden gebaseerd op een vrij rechttoe rechtaan ontwerp, waarin bij het gegeven id in de cache wordt gekeken of hier reeds gegevens over beschikbaar zijn, en anders de Web service wordt geraadpleegd.

In andere gevallen is het echter ook noodzakelijk om zoekoperaties te kunnen uitvoeren, zonder dat de betreffende Web service hier functionaliteit (of capaciteit) voor biedt. In dat geval is het nodig om de cache bij voorbaat te vullen met de records waarin gezocht moet kunnen worden. Dit betekent dat er of 1) functionaliteit beschikbaar moet zijn om alle records op te halen of 2) een lijst moet bestaan met de id's van alle records waarin gezocht moet kunnen worden. Een concreet voorbeeld van punt 2 is een ledenbestand met daarin per record lokaal-relevante gegevens (zoals een betaalsaldo) en een identifier (zoals een sofinummer). Met de identifier zouden dan bijvoorbeeld de NAW gegevens uit een gemeentelijk systeem kunnen worden verkregen. In dit geval kan het systeem automatisch een cache (of beter gezegd: replicadatabase) vullen met de NAW gegevens van alle leden.

2.4. Concrete cases

Hieronder wordt een aantal concrete situaties besproken, waarbij het beschreven probleem een rol speelt. De reden voor deze casusbeschrijvingen is dat het vervolgens mogelijk is om algemeen geldende aspecten van deze situaties te abstraheren, zodat wordt gekomen tot een zo algemeen mogelijke afbakening van het probleem.

2.4.1. Postcodegegevens

Een erg eenvoudige eerste case is die van postcodegegevens. Hiermee wordt in feite de redundantie in adresgegevens bedoeld, omdat slechts een postcode / huisnummer combinatie volstaat om een adres in Nederland uniek te identificeren. Dit betekent dat de straat- en plaatsnaam overbodig zijn en dus in feite niet in een database thuishoren.

Wat dan wel nodig is en dus vaak in systemen terug wordt gezien is een methode om, gegeven de postcode / huisnummer combinatie, de overige adresgegevens te achterhalen. In veel gevallen wordt deze functionaliteit via Web services betrokken van een externe, gespecialiseerde partij.

Hieruit volgt direct de relevantie voor dit project. Ten eerste zit hier de potentie voor het optreden van het prestatieprobleem. Ten tweede is het meestal zo dat de externe partij kosten in rekening brengt per opgevraagde postcode. Bij een naïef ontwerp zouden dus elke keer als er om wat voor reden dan ook adresgegevens moeten worden verkregen, kosten in rekening worden gebracht. Zelfs als het gaat om een adres waarvan de postcode al eerder is opgevraagd.

Hieronder volgen de voor dit project relevante eigenschappen van deze case:

- De postcode / huisnummer-combinatie kan worden gezien als een manier om een "record" (alle adresgegevens) uniek te identificeren.
- De operaties op de Web services hebben geen gevolgen voor latere aanroepen. Een opvraag van een bepaald adres heeft immers geen gevolgen voor latere aanroepen van (andere) adressen. Hierbij wordt geabstraheerd van mechanismen met "credits" voor een beperkte hoeveelheid aanroepen.
- Er kan van uit worden gegaan dat de structuur van de adresgegevens statisch is en deze gegevens dus op geautomatiseerde wijze kunnen worden verkregen. Als dit niet het geval zou

zijn, zou de Web service immers nutteloze functionaliteit bieden, omdat Web services juist bedoeld zijn voor machine-machine-interactie en niet voor mensen.

- Verder ingaande op het vorige punt is een realistische aanname dat de gegevens kunnen worden gerepresenteerd in een structuur van een vaste verzameling attributen (zoals “woonplaats”) met bijbehorende waarden (zoals “Enschede”).
- Er is hier geen sprake van schrijfacties, omdat juist de bedoeling is dat het beheer van het postcodebestand wordt uitbesteed.

2.4.2. Koppeling met online Customer Relationship Management-systeem

Sommige bedrijven maken gebruik van een online (op internet) Customer Relationship Management (CRM)-systeem. Dat het gunstig is om een CRM-systeem te centraliseren, komt onder meer door de aard van haar gebruikers. Zo zal een belangrijke gebruikersgroep, de verkoper, vaak op verschillende plaatsen zijn werk doen, bijvoorbeeld op locatie bij de klant. Het is dan erg nuttig om daarbij het klantenbestand beschikbaar te hebben, zodat bijvoorbeeld direct op locatie een offerte kan worden opgesteld.

Een concreet voorbeeld daarvan is Salesforce [SF]. Dit is een online CRM-oplossing die, naast de vele web-gebaseerde functionaliteiten, een Web service interface biedt tot alle gegevens die erin kunnen worden bijgehouden. Dit leidt tot vele mogelijkheden, zoals in de website van het bedrijf geïntegreerde functies voor het opvragen van offertes en dergelijke functionaliteit op de PDA van de verkoper.

Ook hier spelen de prestatiekwesties. Een voorbeeld hiervan is het moment dat een klant op de website een product wil selecteren voor op zijn offerte. Bij een naïeve implementatie zou hierbij direct de Web service worden aangeroepen om een lijst van producten te verkrijgen. Hierbij wordt de responstijd echter (mede) bepaald door de Web service, waarbij ervan uit moet worden gegaan dat deze waarden aanneemt die leiden tot irritatie van de gebruiker.

Relevante eigenschappen van deze case zijn:

- Een kwestie die bij dit voorbeeld speelt, is dat er geen vanzelfsprekende kandidaat is om gegevens te identificeren. In het geval van Salesforce wordt hiervoor een speciaal “id”-veld gebruikt. Als de gegevens nu opgevraagd moeten worden door een gekoppeld systeem, zal de Web service dus functionaliteit moeten bieden om records te verkrijgen aan de hand van andere kenmerken dan het id of om een lijst van alle (id’s van) records te verkrijgen. Zie ook het volgende punt.
- Salesforce biedt functionaliteit om alle records van een bepaald type tegelijkertijd op te halen. Op deze manier kan dan gelijk een link worden gelegd tussen de lokale gegevens en de uit Salesforce verkregen gegevens, via het Salesforce-id.
- Ook hier is de structuur van de gegevens statisch. Het is echter wel zo dat er meerdere soorten records bestaan (bedrijven, contactpersonen, offertes, etcetera) met verschillende structuren van gegevens.
- Verder geldt ook hier dat de gegevens kunnen worden gestructureerd als een verzameling attribuut / attribuutwaarde paren, waarbij de mogelijke verzameling attributen per soort record vastligt.
- Er wordt van uitgegaan dat er geen schrijfacties nodig zijn op de gegevens, of dat er via een andere weg wijzigingen kunnen worden uitgevoerd.

2.4.3. Titelgegevens van een bibliotheekcollectie

Titelgegevens zijn gegevens over een bepaalde titel in een bibliotheek. Dat wil zeggen, niet de gegevens over een specifiek exemplaar van een titel, maar alle gegevens (titel, auteur, ISBN, enzovoort), die hetzelfde zijn voor alle exemplaren van een bepaalde titel.

Titelgegevens zijn een interessante case voor dit project, vanwege de combinatie van de relatief grote hoeveelheden en de noodzaak om erin te kunnen zoeken. Verder bestaat er een sterk argument om een titeldatabase te centraliseren, namelijk de enorme overlap tussen de lokale databases. Het is namelijk zo dat er per bibliotheek enkele honderdduizenden tot maximaal één miljoen titels te vinden zijn. Het totaal aantal titels, gerekend over alle bibliotheken, overstijgt echter waarschijnlijk niet de

twee miljoen. Omdat al deze titels (en dus ook de grootschalige overlap) moeten worden beheerd, is er sprake van een behoorlijk efficiëntieverlies. Dit zou kunnen worden opgelost door de introductie van een centrale database van titelgegevens, waaruit alle bibliotheken de titelgegevens over hun collectie betrekken.

Omdat er nog geen centrale database met titelgegevens bestaat, is het lastig hier concrete eigenschappen van op te geven. Hieronder enkele relevante eigenschappen en aannames:

- Er wordt van uitgegaan dat een in een centrale database opgenomen titel op een eenvoudige wijze kan worden geïdentificeerd. Een kandidaat voor deze identificatie is het ISBN-nummer. Niet alle titels hebben echter een ISBN-nummer, zodat deze titels onder deze aanname niet in de database opgenomen zouden kunnen worden. Het is daarom de vraag of deze aanname realistisch is. Wellicht hangt dit samen met de reden dat er (nog) geen centrale database van titelgegevens bestaat. Deze problemen liggen niet binnen het bestek van dit project en daarom wordt toch aangenomen dat elke titel kan worden geïdentificeerd door één of andere “string”.
- Elke bibliotheek heeft een bepaalde collectie en dus een database met gegevens over de exemplaren in deze collectie. Er wordt aangenomen dat voor elk exemplaar de identificatiecode van de titel bekend is, reeds voordat de centrale database is geraadpleegd. Dit impliceert dat de centrale Web service in principe af zou kunnen met functionaliteit die gegeven een dergelijke code, de gegevens van die titel teruggeeft. Of dit laatste praktisch is, zal in de praktijk echter moeten blijken.
- Een centrale database van titelgegevens zou bestaan uit relatief veel records (orde van grootte twee miljoen) evenals het lokaal relevante deel ervan (orde van grootte maximaal één miljoen).
- Er moeten, minimaal op het lokaal relevante deel van de database, zoekacties kunnen worden uitgevoerd. Dit betekent dat of:
    - het deel van de database waar deze zoekacties op moeten kunnen worden uitgevoerd, lokaal beschikbaar moet zijn voordat er zoekacties mogelijk zijn, of:
    - tot het moment dat alle gegevens lokaal beschikbaar zijn, de zoekacties via de Web service moeten worden uitgevoerd.
- Ook hier zijn geen schrijfoperaties noodzakelijk, aangenomen dat de centrale database correct wordt beheerd.

2.5. Verdere afbakening

Gegeven bovenstaande cases, wordt hieronder een aantal verdere aannames gedaan. Deze aannames gelden algemeen voor de cases en het is de bedoeling dat deze een nog algemenere groep van cases beschrijven.

2.5.1. Records

Er wordt aangenomen dat er een eenheid van gegevens bestaat, die ondubbelzinnig en geautomatiseerd uit een respons van de Web service kan worden afgeleid. Een dergelijke eenheid zal hierna “record” worden genoemd.

Verder wordt aangenomen dat alle records kunnen worden geprojecteerd op dezelfde structuur. Hiervoor wordt een structuur gebruikt die erg universeel is. Deze structuur wordt als volgt gedefinieerd:

Een record kent een verzameling attributen en bestaat uit een toekenning van waardes aan elk van deze attributen.

Dergelijke records kunnen worden geprojecteerd op zeer verschillende applicatieniveau datastructuren, zoals Java POJO’s (“Plain Old Java Objects” – simpele Java objecten), tabellen in databases, XML-elementen, map’s (denk aan java.util.Map), enzovoorts.

2.5.2. Meerdere soorten records

Verder kan worden aangenomen dat één Web service meerdere soorten van dergelijke records kan onderscheiden. Een voorbeeld hiervan is de onderscheiding van bedrijf- en contactpersoonrecords in

een online CRM. De verzameling attributen is voor elk record van hetzelfde soort gelijk en verschillende soorten records kunnen verschillende verzamelingen attributen hebben.

Merk op dat deze aanname niet wegneemt dat sommige Web services slechts één soort records kennen. Een voorbeeld hiervan is de postcodeservice uit §2.4.1.

2.5.3. Identificatie van records

Voor de hierboven beschreven records wordt verder aangenomen dat voor elk soort record, de waarde van één van de attributen uniek is binnen alle waarden van dat attribuut van alle records. Deze uniciteit geldt slechts binnen hetzelfde soort records. Verder wordt aangenomen dat de waarde van dit identificerende attribuut, voor een gegeven record, nooit wijzigt.

Het is belangrijk te constateren dat deze records of 1) van tevoren bij de gebruiker bekend moeten zijn (zoals sofi- of ISBN-nummers) of 2) de Web service extra functionaliteit zal moeten bieden om deze (applicatie specifieke) identificatiewaarden te verkrijgen. Zie “Soorten selecties”, verderop.

2.5.4. Alleen lezen

De oplossing hoeft slechts de leesoperaties te optimaliseren. De reden hiervoor is dat het in alle cases vooral gaat om leesoperaties. Dit betekent niet dat het niet mogelijk is om schrijfoperaties uit te voeren, ze worden alleen niet direct ondersteund door het systeem.

Gegeven de aanname dat de Web service de achterliggende gegevens ook zelfstandig wijzigt, verandert er namelijk niets als ook schrijfoperaties door de gebruiker (buiten het systeem om) worden toegestaan. Het zou hoogstens zo kunnen zijn (als gevolg van onderstaande aanname “Niet-synchrone gegevens”), dat de gebruiker haar eigen wijzigingen niet direct doorgevoerd ziet worden. Gecombineerd met de aanname dat er veel minder schrijfacties gebeuren en het dus minder nodig is deze te optimaliseren, biedt dit een totale opsplitsing van het probleem lees- en schrijfoperaties. In dit project worden dus alleen de leesoperaties ondersteund, maar wordt (door deze aannames) de flexibiliteit geboden om later (via een andere weg) ook schrijfoperaties toe te staan.

2.5.5. Soorten selecties

Verschillende Web services zullen op verschillende manieren functionaliteit bieden om selecties uit de records te maken.

Er wordt van uitgegaan dat elke Web service tenminste de functionaliteit biedt om, gegeven het soort record en een id, het record terug te geven van de gegeven soort, met het gegeven id.

De meeste services zullen echter meer functionaliteit bieden voor de selectie van records. Hierbij kan bijvoorbeeld worden gedacht aan de selectie van alle records van een soort of alle records waar sinds een gegeven moment in de tijd wijzigingen op zijn doorgevoerd. Zo kunnen er specifiekere groepen van services worden gedefinieerd, zoals de groep van services die versiefunctionaliteit bieden en daarmee op een efficiënte manier replicadatabases mogelijk maken.

2.5.6. Niet-synchrone gegevens

Voor alle hierboven beschreven cases wordt aangenomen dat een gebruiker niet per se de real-time actuele gegevens nodig heeft. Met andere woorden: het mag zo zijn dat de Web service wijzigingen heeft doorgevoerd die nog niet bekend zijn in het optimalisatiesysteem en dus nog niet worden teruggegeven aan de gebruiker. Een logische (vanzelfsprekende) aanname is wel, dat er hoogstens met “oude” gegevens gewerkt mag worden en niet met willekeurige.

Deze aanname volgt min of meer uit de aanname dat een Web service grootschalig gecentraliseerd kan zijn en dus in het algemeen niet in staat zal zijn alle geïnteresseerden direct van alle wijzigingen op de hoogte te stellen.

2.5.7. Meer leesoperaties dan wijzigingen op de gegevens

Als laatste wordt aangenomen dat er vaker leesoperaties zullen worden uitgevoerd dan schrijfoperaties, wat in veel scenario’s het geval is. Deze aanname is nodig om het bijhouden van een replica van de ontsloten gegevens te verantwoorden.

2.6. Overzicht afbakening

Hieronder is een tabel weergegeven met een samenvatting van alle aannames die in dit hoofdstuk zijn gedaan.

2.1.1	De betreffende Web services conformeren aan de standaarden van het W3C
2.1.2	De Web service kan niet worden veranderd (bv. om prestatie eisen opleggen)
2.1.3	De Web service biedt een interface naar ("ontsluit") gegevens
2.1.4	De ontsloten gegevens kunnen tussen aanvragen zijn gewijzigd
2.5.1	De gegevens zijn onderverdeeld in "eenheden" van gegevens ("records"), die bestaan uit een toewijzing van een waarde aan elk van hun "attributen"
2.5.2	Er kunnen verschillende "soorten" records bestaan, alle records van hetzelfde soort hebben dezelfde verzameling attributen
2.5.3	Eén van de attributen heeft een waarde die uniek is voor alle records van dat soort (het "id")
2.5.4	Slechts het lezen van de gegevens hoeft te worden geoptimaliseerd
2.5.5	Verschillende Web services bieden verschillende manieren om selecties te doen op records
2.5.6	Het gebruik van (begrensde) verouderde gegevens is geen probleem
2.5.7	Gegevens worden vaker gelezen dan gewijzigd

Tabel 1: Overzicht van aannames

3. Programma van Eisen

Gegeven de analyse uit het vorige hoofdstuk, zal hier het programma van eisen worden besproken waaraan de te ontwerpen oplossing moet voldoen.

Allereerst mag worden aangenomen dat de situaties waarin het systeem dient te functioneren, voldoen aan de afbakening zoals gesteld in de analyse hierboven. Zie voor een samenvatting §2.6.

3.1. Functionele eisen

De functionaliteit van het systeem is onderverdeeld in drie categorieën. De eerste categorie is de definitie van een universele interface. De reden voor een dergelijke interface is dat deze het systeem algemeen maakt. De twee overige categorieën zijn caching en replicatie functionaliteit, die nodig zijn voor de oplossing voor het probleem in de twee verschillende gevallen waarin respectievelijk geen en wel ondersteuning voor zoekoperaties nodig is.

3.1.1. Universele “gegevensontsluitende Web service” interface

Om de eigenlijke oplossing zo algemeen mogelijk te maken, moet eerst een universele interface naar gegevensontsluitende Web services worden gedefinieerd. Gegeven deze interface, kan hiervoor vervolgens per applicatie (concrete Web service), een implementatie worden gerealiseerd. Door de rest van het systeem slechts via deze interface gebruik te laten maken van Web services, kunnen op deze manier flexibel nieuwe concrete situaties worden ondersteund.

Hieronder wordt de functionaliteit besproken, die deze interface in ieder geval moet bieden.

Het ondersteunen van alle mogelijke selecties die Web services toestaan

Omdat het onmogelijk en ongewenst is om alle mogelijke soorten selecties expliciet op te nemen in het ontwerp, zal er een notie moeten worden geïntroduceerd van een abstracte, universele selectie. Concrete soorten selecties kunnen vervolgens op een eenvoudige wijze, zonder te hoeven worden geïnterpreteerd, worden doorgegeven aan de applicatiespecifieke implementatie van de interface. Vervolgens kan deze implementatie de selectie interpreteren, uitvoeren en het resultaat teruggeven.

Het definiëren van standaard soorten selecties

Om de rest van de functionaliteit te implementeren, zal het nodig zijn bepaalde aannames te kunnen doen over de mogelijke soorten selecties. Het belangrijkste voorbeeld hiervan is de selectie van één specifiek record, gegeven het soort en het id van dat betreffende record. Deze dient door alle implementaties te worden ondersteund.

Bij het ontwerpen van het systeem kan het echter noodzakelijk of nuttig zijn om meer van dergelijke soorten selecties, die voor alle Web services dezelfde betekenis hebben, te definiëren. Hierbij kan bijvoorbeeld worden gedacht aan selecties die alle records selecteren of die, gegeven een tijdstip, alle records selecteren die zijn gewijzigd sinds dat tijdstip.

Itereren over de resultaten

Bij het uitvoeren van een selectie door de Web service implementatie, ontstaat er een lijst van records die aan de selectiecriteria voldoen. Omdat deze lijst potentieel erg lang is, moet het mogelijk zijn door deze lijst te lopen zonder deze noodzakelijkerwijs in zijn geheel in het geheugen te hebben.

3.1.2. Cache

De tweede categorie van functionaliteit betreft een cache. Met de universele notie van records en een universele interface om zulke records uit een ander systeem te verkrijgen, is het eenvoudig een cache bovenop deze interface te definiëren.

Hieronder wordt de gewenste functionaliteit van deze cache beschreven.

Het één op één opslaan van teruggegeven records

Op het moment dat er een selectie wordt gevraagd, wordt deze doorgegeven aan de onderliggende Web service interface. Alle daardoor teruggegeven records moeten vervolgens in een cache worden geplaatst. Reeds aanwezige records moeten worden geactualiseerd. Omdat een dergelijke selectie abstract is en dus niet universeel kan worden geïnterpreteerd, is het bijhouden van de cache-database de belangrijkste functie van deze categorie functionaliteit.

Het inzetten van de cache-database bij de standaardselectie

Het bijhouden van een dergelijke cache-database is, zonder verdere functionaliteit, niet nuttig. Daarom geldt als aanvullende eis dat deze cache de in §3.1.1 omschreven standaardselectie kan interpreteren (het ophalen van één record, gegeven het id). Op het moment dat er een dergelijke selectie wordt gevraagd, dient deze eerst op de cache-database uitgevoerd te worden. Levert dit geen resultaat, dan wordt de selectie op de gangbare manier afgehandeld.

Het "onderhouden" van de cache-database

De hierboven beschreven functionaliteit impliceert een oneindige levensduur van de records in de cache-database. Omdat ervan uit moet worden gegaan dat de gegevens wijzigen en deze wijzigingen zonder extra functionaliteit nooit zullen worden gedetecteerd, is dit een ongewenste situatie. Daarom dient functionaliteit te worden geboden om bepaalde records te kunnen "verversen" naar de laatste versie, zoals door de Web service aangeboden.

Bij nadere bestudering van bovenstaande functionaliteit kan worden geconstateerd, dat als men via andere selecties dan de standaardselectie gegevens opvraagt, deze gegevens nooit uit de cache-database zullen worden verkregen. Dit kan een manier zijn van cache-onderhoud. Zo kan bijvoorbeeld een speciale standaardselectie worden gedefinieerd, die exact hetzelfde doet als de standaardselectie, met als uitzondering dat er niet eerst in de cache-database wordt gekeken.

Een andere mogelijkheid is het bieden van functionaliteit die simpelweg records uit de cache-database verwijdert. Hiervoor is dus expliciet extra functionaliteit van de cache nodig.

De precieze functionaliteit voor het onderhouden van de cache-database mag in de ontwerpfase worden bepaald, met als voorwaarde dat het minimaal mogelijk moet zijn om specifieke records uit de cache te verwijderen of te verversen.

3.1.3. Replicatie

De derde categorie van functionaliteit bouwt voort op de hierboven omschreven cache-functionaliteit. Het betreft het bijhouden van een replica van de door de Web service ontsloten verzameling records.

Definitie

In dit geval wordt de term replica gedefinieerd als een volledige kopie van alle records die de Web service ontsluit. Het mag dus nooit zo zijn dat de replica bepaalde records niet bevat, op het moment dat de replica beschikbaar is voor de gebruiker. Een uitzondering hierop is de in §2.5.6 genoemde aanname, waarin wordt gesteld dat de gegevens niet altijd volledig actueel aan de gebruiker hoeven te worden geleverd. In dit geval dient dit te worden geïnterpreteerd als dat het is toegestaan om records niet in de replica op te nemen, als het de meest recentelijk toegevoegde records betreft.

Mogelijkheid tot zoekoperaties

De voornaamste reden om replica-functionaliteit toe te voegen, is dat een volledige replica het mogelijk maakt om zoekoperaties lokaal uit te voeren. Verder biedt het, als het proces van het bijhouden van de replica buiten beschouwing wordt gelaten, de mogelijkheid om totale afscherming te bieden voor de prestatiegevolgen van de koppeling. Het is waarschijnlijk ook de enige manier om de prestatiegevolgen van zoekoperaties af te schermen zonder de zoekoperaties te hoeven interpreteren.

Het bieden van een gestandaardiseerde interface naar de replica

Om de replica zo goed mogelijk tot zijn recht te laten komen, moet er een gestandaardiseerde interface naar deze replica worden geleverd. Afhankelijk van de ontwerpkeuzes kan dit bijvoorbeeld een SQL-interface zijn (gestandaardiseerd in Java als JDBC). Hierdoor wordt het voor de gebruiker mogelijk om alle functionaliteit van een dergelijke interface (zoals uitgebreide zoekoperaties) optimaal

te benutten. Een minimale eis aan deze interface is dan ook de uitgebreide ondersteuning van zoekoperaties, met booleaanse operatoren en het zoeken op meerdere attributen.

Het bieden van een configuratie-interface

Het replicatiesysteem zal door de gebruiker moeten kunnen worden geconfigureerd en afgestemd op de eisen die de concrete situatie met zich meebrengt.

De minimale eis is dat het interval waarmee de replicatie-slagen worden uitgevoerd (en dus de maximale ouderdom van gerepliceerde records wordt vastgelegd) kan worden ingesteld.

Een andere mogelijkheid is een ontwerp waarin het replicatiesysteem eenvoudig kan worden opgebouwd uit standaardcomponenten, zodat het kan worden geïntegreerd in het gebruikende systeem en zo ook alle instellingen kunnen worden bepaald.

3.2. Non-functionele eisen

Hieronder worden de non-functionele eisen besproken waar het te ontwerpen systeem aan dient te voldoen. Deze zijn onderverdeeld in drie categorieën.

3.2.1. Prestaties

De eis met betrekking tot de prestaties van het systeem is de belangrijkste. Het doel van het hele systeem is immers het voorkomen van prestatieverlies. Het is echter ook vrij lastig om hier een goede eis voor op te stellen, die bovendien realistisch is. Daarom wordt deze als volgt gesteld: Er moeten gevallen aan te wijzen zijn waarbij het systeem een aanzienlijke snelheidswinst oplevert. Een belangrijke aanname die hier kan worden gehanteerd is dat een aanroep van de Web service een veel lagere prestatie levert dan een lokale aanroep. Het systeem mag in sommige gevallen echter ook gelijke of zelfs als gevolg van overhead iets mindere prestaties (responstijd en throughput) vertonen dan zonder het te ontwerpen optimalisatiesysteem.

Een tweede eis met betrekking tot prestaties, is dat het systeem dient te blijven functioneren in het geval dat er grote hoeveelheden records worden verwerkt. Het systeem mag zelf geen beperkingen opleggen aan de hoeveelheid records; deze mogen slechts afhangen van de gebruikte (onderliggende) databasetechnologie en hardware waarop het systeem draait.

3.2.2. Flexibiliteit

Het systeem dient zo algemeen mogelijk inzetbaar en dus aanpasbaar aan nieuwe situaties te zijn. Daartoe moet het systeem uitbreidbaar zijn met nieuwe Web services. Deze eis wordt min of meer al geïmpliceerd door de functionele eis dat er een universele Web service interface moet worden gedefinieerd en dat de rest van het systeem slechts gebruik dient te maken van deze interface. Dit betekent dat het direct mogelijk is om nieuwe Web services te ondersteunen, namelijk door simpelweg voor deze concrete gevallen een implementatie te maken van de interface.

4. Universele interface naar gegevensontsluitende systemen – RecordIF

In de voorgaande hoofdstukken is steeds uitgegaan van de prestatiekwesties bij de koppeling met Web services. Omdat bij het ontwerpen van de beoogde oplossing bleek dat er geen gebruik werd gemaakt van aspecten die specifiek zijn voor Web services, is ervoor gekozen hiervan te abstraheren. Op deze manier wordt een algemenere oplossing gecreëerd. Daarom wordt vanaf hier niet meer gesproken van een "gegevensontsluitende Web service interface", maar van een "interface naar gegevensontsluitende systemen" (hierna te noemen "RecordIF").

In dit hoofdstuk wordt deze universele interface gedefinieerd, op basis van de eerdere aannames over gegevensontsluitende systemen en de eisen uit het vorige hoofdstuk. Deze definitie beslaat een apart hoofdstuk, om te benadrukken dat het in feite los staat van het ontwerp, zodat het ook kan worden gebruikt voor andere systemen.

4.1. Gerelateerd werk

Hieronder worden enkele technieken besproken, die gerelateerd zijn aan de in dit hoofdstuk gedefinieerde universele interface.

- SQL (Structured Query Language) is een taal om queries op relationele databases te beschrijven. In eerste instantie lijkt er wellicht een overlap te bestaan tussen SQL en de interface uit dit hoofdstuk. Er is echter een aantal belangrijke verschillen:
    - SQL is slechts een querytaal, waardoor elke database server met een eigen client / server-protocol een specifieke driver nodig heeft. In dit hoofdstuk wordt ook een eenvoudige algemene mapping van de interface op Web services vastgelegd.
    - Omdat SQL een querytaal is, ligt de nadruk daarbij op het opstellen van queries en het berekeningen van functies over de gegevens. Bij de interface in dit hoofdstuk wordt juist zo weinig mogelijk vastgelegd over (geabstraheerd van) mogelijke selecties.
    - Om efficiënt replicatie te ondersteunen, wordt in dit hoofdstuk een timestamp-begrip geïntroduceerd. SQL biedt geen gestandaardiseerde manier van replicatie.
- CORBA (Common Object Request Broker Architecture [CORBA]) is een gestandaardiseerde manier om communicatie tussen objecten mogelijk te maken in verschillende programmeertalen en via een computernetwerk. Net als bij Web services wordt gebruik gemaakt van een machine-leesbare interface specificatie. Er wordt hier geen gebruik gemaakt van CORBA, omdat het voor de functionaliteit van dit project onnodig ingewikkeld is om er implementaties voor te maken.
- Hoewel niet direct gerelateerd aan de in dit hoofdstuk beschreven interface, is het nog de moeite waard om het "POJO"-concept te noemen. POJO staat voor "Plain Old Java Object" en staat voor het gebruik van eenvoudige Java objecten om gegevens te representeren in plaats van speciale objecten die gebruik maken van een relatief ingewikkeld framework. Al lijkt dit vanzelfsprekend, toch wordt de eenvoud van dit concept pas enkele jaren gezien als belangrijk voordeel. Dit blijkt bijvoorbeeld uit de volgende uitspraak van Martin Fowler (één van de bedenkers van de term): "We wondered why people were so against using regular objects in their systems and concluded that it was because simple objects lacked a fancy name. So we gave them one, and it's caught on very nicely." [FOW2000]. Het is de bedoeling dergelijke eenvoud te laten terugkomen bij de definitie van de universele interface.

4.2. Definitie

Op het hoogste niveau koppelt een implementatie van de interface een zogenaamde query aan een resultaat. De query legt vast welke records de gebruiker opgeleverd wil hebben. De query en het resultaat worden hieronder uitvoerig besproken.

4.2.1. Query

Een query bestaat uit twee onderdelen: een string die het soort record identificeert (bijvoorbeeld “persoon”) en een “selectie”, die aangeeft welke deelverzameling van alle records van dat soort moet worden opgeleverd.

Er bestaat één speciaal soort record, de string “_supportedtypes”. Deze geeft in plaats van reguliere records, de ondersteunde soorten records terug. De teruggegeven records moeten precies één attribuut bevatten, die steeds als waarde de naam van het ondersteunde soort record bevat.

In het algemene geval is een selectie abstract, waarmee wordt bedoeld dat deze niet door tussenliggende systemen hoeft te kunnen worden geïnterpreteerd. Zo kan elke implementatie verschillende soorten selecties definiëren.

Omdat het in sommige gevallen gewenst is om selecties te gebruiken die voor alle mogelijke implementaties dezelfde betekenis hebben, worden hieronder enkele “algemene” selecties geïntroduceerd. Alle implementaties die deze selecties ondersteunen dienen onderstaande betekenissen te hanteren.

- All-selectie: Deze selectie levert alle records.
- ID-selectie: Gegeven een waarde van het id-attribuut (zie §2.5.3) levert deze selectie maximaal één record: het record met het gegeven id.
- ModifiedSince-selectie: Gegeven een timestamp, levert deze selectie alle records die zijn gewijzigd sinds of exact op deze timestamp. Timestamps worden verderop apart besproken.

4.2.2. Resultaat

Het resultaat dat door de implementatie wordt teruggegeven, bestaat minimaal uit twee onderdelen: een beschrijving van het soort record en informatie waarmee door de verzameling records kan worden gelopen die voldoen aan de query die aanleiding gaf tot dit resultaat. Verder kan het resultaat twee optionele onderdelen bevatten. Alle onderdelen worden hieronder apart besproken.

Beschrijving van het soort record

De beschrijving van het soort record (hierna te noemen “RecordType”), bevat de volgende informatie:

- Een string die de naam van het soort record aangeeft. Deze is gelijk aan de door de gebruiker in de query meegegeven waarde.
- Een lijst van strings: de namen van de attributen die elk record van dit soort heeft (zie §2.5.1)
- Een string die de naam van het attribuut aangeeft die het id van een record bevat. (zie §2.5.3)

De beschrijving van het soort record dient vóór de daadwerkelijke resultaat-records aan de gebruiker te worden teruggekoppeld, zodat de gebruiker zich hiermee kan voorbereiden op het verwerken van de teruggegeven records.

Een alternatief voor het meegeven van de beschrijving van het soort record met elk resultaat, is het opstellen van een apart type query om deze te verkrijgen. De reden dat hier niet voor is gekozen, is dat het soort record (bijvoorbeeld de lijst attributen) tussen twee queries kan wijzigen. Er zouden dan bijvoorbeeld nieuwe mechanismen moeten worden geïntroduceerd, om te garanderen dat dit niet is gebeurd. Dit zou de interface onnodig ingewikkeld maken.

Informatie waarmee door de verzameling records kan worden gelopen

Omdat in de eisen is gesteld dat het niet nodig mag zijn om alle opgeleverde records tegelijk in het geheugen te hebben, kan het resultaat niet simpelweg een lijst van al die records bevatten. Er zal dus slechts informatie kunnen worden geboden om door de lijst opgeleverde records heen te lopen. Een voorbeeld hiervan is een stuk programmatuur dat bij het uitvoeren hiervan steeds het volgende record in die lijst oplevert. Hierbij kan worden gedacht aan een implementatie van de Java “java.util.Iterator” interface. Zo wordt het mogelijk om bijvoorbeeld steeds één record van een netwerkverbinding te lezen, zonder dat alle opgeleverde records al over die verbinding zijn gestuurd.

Optionele onderdelen: Timestamp en verwijderingen

Er zijn twee optionele onderdelen van het resultaat. Het gaat hier om onderdelen die nodig zijn om efficiënte replicatie van de records mogelijkheid te maken.

Ten eerste kan er een timestamp worden teruggegeven die aangeeft tot (niet: tot en met) welk moment de in dit resultaat aanwezige gegevens mimimaal actueel zijn. Dit hoeft dus niet te betekenen dat het resultaat *alle* wijzigingen in de gegeven timestamp bevat; er kunnen bij een volgende selectie met dezelfde timestamp, extra records worden opgeleverd. In combinatie met de ModifiedSince-selectie maakt dit replicatie mogelijk. Hierop wordt in de volgende paragraaf dieper ingegaan.

Ten tweede kan worden teruggegeven welke records zijn verwijderd. De verwijderingen worden teruggegeven in de vorm van een lijst van id's van de verwijderde records. Ook hierop wordt verderop dieper ingegaan.

Zowel de timestamp als de verwijderingen hoeven pas bekend te zijn nadat over alle records van het resultaat heen is gelopen. De reden hiervoor is dat de berekening van deze onderdelen dan gebaseerd kan zijn op de resultaten zelf. Er wordt hierbij dus aangenomen dat deze gegevens niet eerder nodig zijn dan na het doorlopen van alle gegevens. Bij verwijderingen is dit beter, omdat zo wordt geforceerd de verwijderingen pas te verwerken na eventuele wijzigingen aan verwijderde records (als dit andersom zou worden gedaan, zouden er immers wijzigingen worden aangebracht op verwijderde records). Ook de timestamp is pas weer nodig bij de volgende ModifiedSince-selectie en dus niet voordat de records van het huidige resultaat zijn opgehaald.

4.3. Timestamps

In deze paragraaf wordt het timestamp-begrip besproken, zoals bedoeld in het kader van de universele interface. Het begrip wordt op twee plekken gebruikt: om aan te geven tot op welk moment de teruggegeven resultaten actueel zijn en bij de ModifiedSince-selectie om aan te geven vanaf welk moment de wijzigingen opgevraagd worden.

4.3.1. Eigenschappen

De gebruikte waarden van timestamps moeten voldoen aan de volgende eigenschappen:

- Er moet een volgorde inzitten, zoals bij getallen en tijdstippen.
- Als een record wordt gewijzigd, moet de timestamp die wordt gebruikt om aan te geven op welk moment dit record is gewijzigd groter of gelijk zijn aan de maximale waarde van alle timestamps waarop andere records van dit soort zijn gewijzigd.

4.3.2. Replicatie

Op het moment dat een implementatie van de interface zowel een timestamp bij de resultaten oplevert als de ModifiedSince- en All-selecties ondersteunt, is efficiënte replicatie mogelijk. Hieronder wordt een eenvoudig algoritme beschreven om een replica bij te houden, zoals bedoeld in §3.1.3. Er wordt van uitgegaan dat alle teruggegeven records worden weggeschreven in een replicadatabase.

- Stap 1: Verkrijg alle records met een All-selectie en onthoud de teruggegeven timestamp.
- Stap 2: Gegeven deze timestamp, verkrijg alle wijzigingen sindsdien met een ModifiedSince-selectie. Onthoud de teruggegeven timestamp voor de volgende ModifiedSince-selectie.
- Stap 3: Wacht het gewenste interval en herhaal stap 2. Het interval (plus de verwerkingstijd van stap 2) bepaalt de maximale "ouderdom" van de gerepliceerde records.

Zoals het hierboven beschreven algoritme aangeeft, hoeft het programma dat het uitvoert de timestamps niet te interpreteren. Dit maakt het mogelijk om een algemeen replicatiesysteem te creëren.

Bij grote hoeveelheden gegevens kan het onwenselijk zijn dat de timestamp niet kan worden geïnterpreteerd, omdat bij het "herstarten" van het algoritme "stap 1" opnieuw moet worden uitgevoerd. Een oplossing hiervan kan zijn om, na een herstart, stap 1 over te slaan en bij de eerste iteratie van stap 2 de timestamp te initialiseren op de maximale waarde van alle timestamps van alle records in de replica. Om dit mogelijk te maken, moeten deze per-record timestamps natuurlijk wel door de implementatie van de interface worden gecommuniceerd (bijvoorbeeld als attribuut van de

records). Daarnaast moeten de records kunnen worden geïnterpreteerd (om de maximale waarde te kunnen berekenen).

Een andere, eenvoudigere mogelijkheid is de timestamp op te slaan in niet-vluchtig geheugen. Dit vereist echter een manier om de willekeurige (abstracte) timestamp op te slaan en in dezelfde vorm op te halen.

4.3.3. Implementatie notities

Het implementeren van timestamps is in een aantal opzichten uitdagend. Hieronder wordt daarom een aantal implementatie-"hints" gegeven.

Omdat de gebruikte timestamps altijd groter worden naar mate de tijd verstrijkt, kan het handig zijn om tijdstippen te gebruiken als timestamps. Dit brengt echter extra uitdagingen met zich mee, omdat tijd door mensen wordt gedefinieerd in niet eenvoudig ondubbelzinnig vast te stellen grootheden.

Een eerste valkuil hierbij is het ingewikkelder maken dan nodig is. Hierbij kan bijvoorbeeld worden gedacht aan het introduceren van begrippen die in principe slechts interessant zijn voor mensen, zoals dagen, zomertijd, schrikkeljaren, enzovoort. Een bekende oplossing hiervoor is het uitdrukken van tijd in het aantal (milli)seconden sinds een bepaald moment.

Een tweede kwestie die ontstaat bij het gebruik van tijdstippen, is dat de gebruikte "tijdbron" nooit kan worden teruggezet, als er niet expliciet gecontroleerd wordt of er wel wordt voldaan aan de gestelde eisen. Als hiervoor bijvoorbeeld de klok van het onderliggende operating system wordt gehanteerd, betekent dit dat er op dit niveau goed moet worden stilgestaan bij deze kwestie. Hiervoor zijn meerdere oplossingen denkbaar, die buiten het bestek van dit project vallen.

Een andere mogelijkheid is het niet rechtstreeks definiëren van de timestamps in termen van tijdstippen. Zo kan simpelweg de timestamp van de laatste wijziging worden gebruikt, plus één. Het nadeel van deze oplossing is dat de "huidige" timestamp op één enkele plek moet worden bewaard. Dit introduceert vervolgens weer concurrency-kwesties.

Een belangrijke observatie is dat de timestamps zo zijn gedefinieerd dat er meerdere records mogen worden gewijzigd tijdens hetzelfde timestamp-"slot". Dit impliceert dat de enige reden om de timestamp te verhogen het opdelen van wijzigingen in hanteerbare blokken is. Dit betekent echter ook dat zolang de timestamp niet wordt verhoogd (en dus de door het resultaat teruggegeven timestamp gelijk kan zijn aan de in de ModifiedSince-selectie meegegeven timestamp), dezelfde gewijzigde records meerdere keren kunnen worden teruggegeven. Indien dit niet gewenst is, kan ervoor worden gekozen om de huidige timestamp te verhogen na elke wijziging of na elke query.

4.4. Verwijderingen

Het ondersteunen en terugkoppelen van verwijderingen is een bijzonder geval, omdat het niet vanzelfsprekend is dat na het verwijderen van een record nog bekend is dat het record ooit heeft bestaan. Daarom is het wenselijk de ondersteuning van verwijderingen optioneel te maken in de interface; anders wordt immers de algemeenheid van de interface aangetast.

Indien verwijderingen worden ondersteund, moeten deze in ieder geval worden teruggegeven bij de ModifiedSince-selectie. Het betreft hier dan slechts de verwijderingen die in de door deze selectie vastgelegde tijdsperiode hebben plaatsgevonden. De ModifiedSince-selectie is tevens de enige algemene selectie waarbij verwijderingen mogen worden teruggegeven, bij de All- en ID-selectie heeft dit immers weinig zin. Verwijderingen mogen wel worden teruggegeven bij applicatiespecifieke selecties.

Bij het doorvoeren van verwijderingen tijdens een replicatieslag, is het belangrijk eerst de wijzigingen te hebben doorgevoerd. Het kan namelijk voorkomen dat het resultaat zowel een wijziging aan een record bevat, als een indicatie dat het record is verwijderd.

4.5. Definitie in Java

In deze paragraaf wordt de universele interface gedefinieerd in Java met behulp van Java interfaces en eenvoudige Java klassen. Op deze manier wordt de definitie formeler en ondubbelzinniger.

Al wordt de definitie gegeven in Java, implementaties in andere programmeertalen kunnen zich eveneens conformeren aan de gestelde Java definitie, omdat er geen typen worden gebruikt die niet

in andere programmeertalen te vinden zullen zijn. Zo worden er slechts waarden gebruikt van het type String (die in elke programmeertaal terug te vinden zal zijn) en het type Object, dat aangeeft dat er een object van een willekeurig type kan worden opgeleverd. Er wordt wel van uitgegaan dat een dergelijk object ondubbelzinnig is te representeren als een string en onveranderd weer uit die representatie kan worden geïnterpreteerd, zodat deze kan worden getransporteerd (over bijvoorbeeld een netwerk).

Hieronder is de definitie weergegeven in de vorm van een diagram met daarin de gebruikte interfaces en klassen. De gekleurde vlakken stellen interfaces of klassen voor, met daaronder de ondersteunde methodes. Alleen de selectiehiërarchie linksonder zijn concrete klassen, de rest zijn interfaces. De lijnen tussen interfaces en methodes zijn om te verduidelijken dat de methode een object oplevert of als argument heeft die de betreffende interface implementeert. Pijlen worden gebruikt om aan te geven dat de interface of klasse waar de pijl vandaan komt een subklasse of -interface is van de ander.

Het diagram is enigszins gestructureerd van links naar rechts: links staan de klassen en interfaces die te maken hebben met de query, in het midden de algemene en rechts die te maken hebben met het resultaat. De centrale interface is RecordIF, de eigenlijke universele interface naar gegevensontsluitende systemen. Zoals te zien is de enige methode getRecords, die een RecordIFQuery als argument verwacht en een RecordIFResult oplevert.

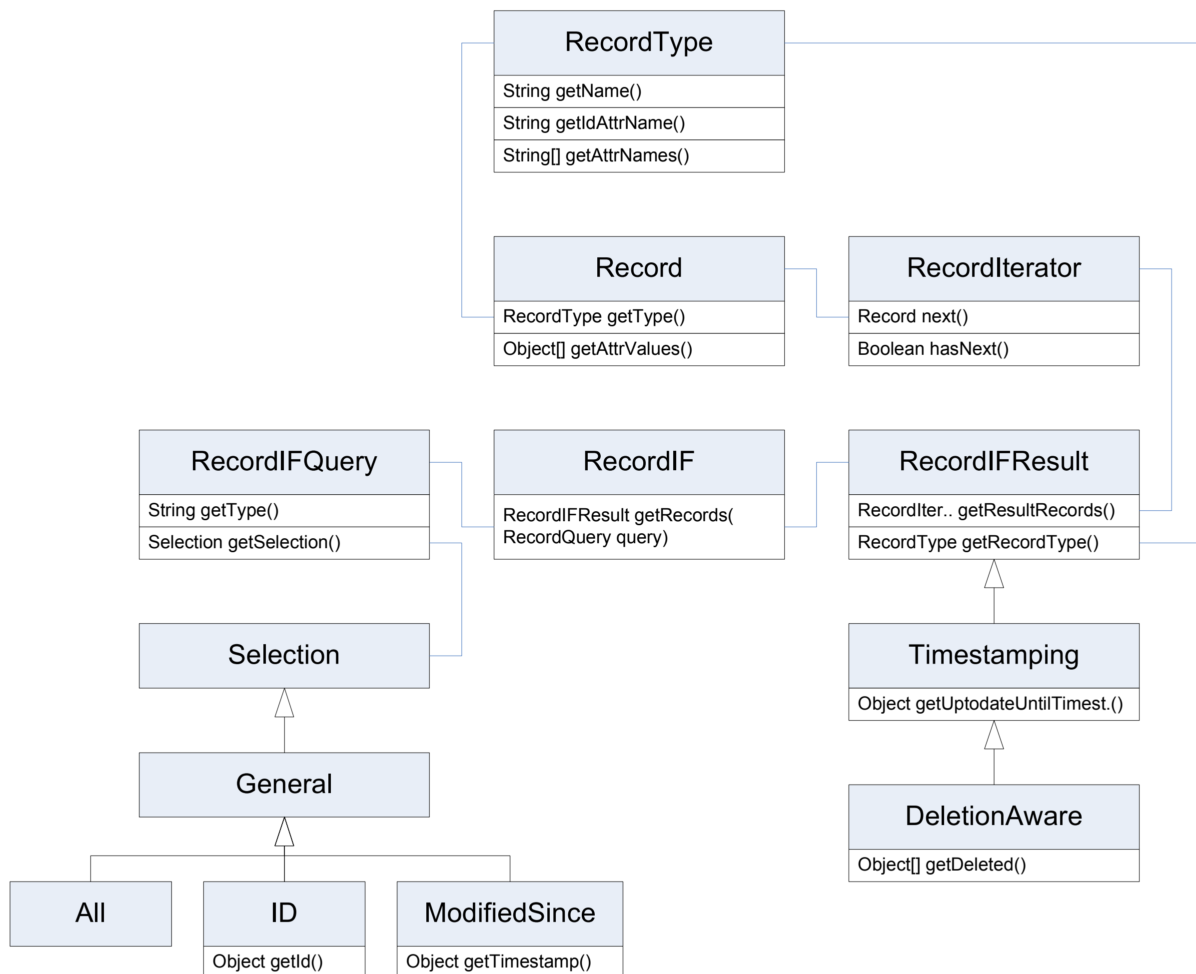
De query bevat het soort record, geïdentificeerd door een string. Daarnaast bevat de query de gevraagde selectie. De Selection klasse is de abstracte selectie, waar alle selecties een subklasse van moeten zijn. De GeneralSelection (afgekort voor compactheid) is de abstracte klasse waar alle algemene selecties subklassen van zijn. Applicatiespecifieke selecties zouden dus een subklasse moeten zijn van Selection en niet van GeneralSelection.

Het resultaat bevat een verwijzing naar RecordType, die een verdere beschrijving geeft van het soort record, met de attribuutnamen en de naam van het attribuut dat het id van een record bevat. De reden hiervan is, dat de gebruiker zo direct uit het resultaat de beschrijving van het soort record kan verkrijgen, zonder eerst records zelf op te vragen. Verder bevat het resultaat een RecordIterator, waarmee met de next() methode steeds het volgende Record in het resultaat kan worden verkregen.

Het resultaat kan verder de Timestamping- of DeletionAwareRecordIFResult interfaces implementeren, om aan te geven dat timestamps of verwijderingen worden ondersteund.

De array van attribuutwaarden die door de getAttrValues() methode van Record wordt opgeleverd, moet in dezelfde volgorde staan als de array van attribuutnamen die wordt opgeleverd door de getAttrNames() methode in RecordType. Er is hier gekozen voor arrays, omdat dit 1) efficiënt is en 2) erg algemeen is.

Er kan worden opgemerkt dat ondanks het feit dat het vooral gaat om interfaces, de meeste hiervan eenvoudige, algemene basisimplementaties kunnen hebben. Zo zal in het algemeen kunnen worden volstaan met slechts het implementeren van de RecordIF en de RecordIterator interfaces. Voor de overige interfaces kunnen dan standaard implementaties worden gebruikt.



Figuur 1: Klassendiagram van de universele interface

4.6. Projectie op XML

Naast de definitie in Java, is het verder erg nuttig om te standaardiseren hoe de query en het resultaat van de interface moeten worden geprojecteerd op XML. Dit wordt gedaan in deze paragraaf.

Op het moment dat de query en het resultaat kunnen worden geprojecteerd op XML, is het eenvoudig een "service" te specificeren die past bij de interface. Dit zou namelijk een systeem zijn dat een XML-document accepteert dat een query bevat en vervolgens als uitvoer een XML-document levert dat het bijbehorende resultaat bevat.

De projectie op XML is vrij triviaal, omdat het gaat om een relatief eenvoudige datastructuur. Het belangrijke is echter de standaardisatie hiervan, die het mogelijk maakt dat implementaties van verschillende partijen direct met elkaar kunnen communiceren.

De gebruikte namespace is "http://pimsierhuis.nl/unirec/".

4.6.1. Query

De query kan vrij eenvoudig worden geprojecteerd op XML. Deze projectie zal worden besproken aan de hand van de DTD en een voorbeeld query als XML-document. De XML-schema variant van de DTD (die bijvoorbeeld nodig is bij de definitie van een Web service) is opgenomen in de bijlagen.

```
<!DOCTYPE recordIFQuery [
  <!ELEMENT recordIFQuery         (recType, selection)>

  <!ELEMENT recType                (#PCDATA)>

  <!ELEMENT selection              (general, type, arg*)>
  <!ELEMENT general                (#PCDATA)>
  <!ELEMENT type                   (#PCDATA)>
  <!ELEMENT arg                    (#PCDATA)>
]>
```

Figuur 2: DTD voor queries

```
<recordIFQuery xmlns="http://pimsierhuis.nl/unirec/">
    <recType>persoon</recType>
    <selection>
        <general>true</general>
        <type>ModifiedSinceSelection</type>
        <arg>12344</arg>
    </selection>
</recordIFQuery>
```

Figuur 3: Voorbeeld query in XML

Zoals hierboven te zien, is de projectie van een query op XML erg eenvoudig. De enige ingewikkeldheid zit in het projecteren van de selectie. Zoals in het voorbeeld is aangegeven, moet bij het type selectie worden aangegeven of het een algemene of applicatie-specifieke is. Indien het gaat om een algemene, moet het “general” element de waarde “true” bevatten. Daarna volgt het “type” element met een van de waarden “ModifiedSinceSelection”, “AllSelection” of “IDSelection”. Als het gaat om een applicatie-specifieke selectie, moet het “general” element de waarde “false” bevatten. Verder moet de waarde van het “type” element dan starten met een internet domeinnaam die onder het beheer is van degene die de applicatie heeft ontwikkeld. Deze moet worden genoteerd zoals ook onder Java gangbaar is om packages te noteren, dus met de door punten gescheiden onderdelen in omgekeerde volgorde (zoals “nl.utwente”). De tekst daarachter is vrij invulbaar, zodat (bijvoorbeeld) de mogelijkheid ontstaat deze string direct te laten corresponderen met bijvoorbeeld een “fully qualified” Java klasse naam.

Een selectie kan nul of meer argumenten bevatten, zoals bij de in het voorbeeld aangegeven ModifiedSinceSelection. De volgorde van deze argumenten is van belang, omdat bij meerdere argumenten de positie aangeeft om welk argument het gaat.

4.6.2. Resultaat

Het resultaat is een iets ingewikkeldere datastructuur. Hieronder weer de DTD en een voorbeeld.

```
<!DOCTYPE recordIFResult [
  <!ELEMENT recordIFResult         (header, recs, footer)>

  <!ELEMENT header                 (recType, idAttr, recAttrs)>
  <!ELEMENT recType                (#PCDATA)>
  <!ELEMENT idAttr                 (#PCDATA)>
  <!ELEMENT recAttrs               (n+)>
  <!ELEMENT n                      (#PCDATA)>

  <!ELEMENT recs                   (rec*)>
  <!ELEMENT rec                    (a+)>
  <!ELEMENT a                      (#PCDATA)>

  <!ELEMENT footer                 (uptodateUntilTimestamp?, deleted?)>
  <!ELEMENT uptodateUntilTimestamp (#PCDATA)>
  <!ELEMENT deleted                (id*)>
  <!ELEMENT id                     (#PCDATA)>
]>
```

Figuur 4: DTD voor resultaten

```
<recordIFResult xmlns="http://pimsierhuis.nl/unirec/">

  <header>
    <recType>persoon</recType>
    <idAttr>id</idAttr>
    <recAttrs>
      <n>id</n>
      <n>voornaam</n>
      <n>achternaam</n>
    </recAttrs>
  </header>

  <recs>
    <rec>
      <a>1</a>
      <a>Pim</a>
      <a>Sierhuis</a>
    </rec>
    <rec>
      <a>2</a>
      <a>Jantje</a>
      <a>Smit</a>
    </rec>
  </recs>

  <footer>
    <uptodateUntilTimestamp>12345</uptodateUntilTimestamp>
    <deleted>
      <id>3</id>
      <id>4</id>
    </deleted>
  </footer>

</recordIFResult>
```

Figuur 5: Voorbeeld resultaat in XML

Op het hoogste niveau bestaat een resultaat uit een kop ("header"), een middenstuk ("recs") en een staart ("footer").

De header bevat de informatie die nodig is om het RecordType op te leveren. Door deze in een los element aan het begin van het document te plaatsen, wordt het mogelijk voor een parser hierop te "wachten" voordat het resultaat wordt teruggekoppeld. Zo kan een parser eerst het RecordType uit het document lezen, vervolgens een parser construeren die de records "streaming" (één voor één op het moment dat de gebruiker erom vraagt) uit het middenstuk leest en deze twee dan samen als resultaat opleveren.

De attributen die elk record zal dragen, worden vastgelegd in het recAttrs element. De volgorde is hierbij erg belangrijk, omdat in het middenstuk de records slechts de lijst van attribuutwaarden bevatten, zonder dat de namen daar terugkomen. Hierbij legt de positie in het XML-document dus vast om welk attribuut het gaat.

De records zelf staan als gezegd in het middenstuk van het resultaat, het "recs" element. Dit element bevat voor elk opgeleverde record een "rec" subelement. Deze bestaan vervolgens weer uit een lijst van "a" elementen, die de waarden van de attributen van het betreffende record representeren. Hierbij wordt gebruik gemaakt van de aanname dat de waarden van de attributen op strings te projecteren moeten zijn.

Er wordt expres niet in het XML-bestand aangegeven van welk "type" (integer, datum, string) de waarden zijn, zodat de algemeenheid niet wordt beperkt. De gebruiker kan er vervolgens voor kiezen de waarden als strings te behandelen of (via een andere weg verkregen) kennis te hebben van de typen en de waarden naar deze typen te converteren.

Wel kan worden aangegeven dat een waarde ongedefinieerd (oftewel "null") is. Hiertoe wordt gebruik gemaakt van het "nil" attribuut uit de "http://www.w3.org/2001/XMLSchema-instance" namespace. Bij een null-waarde moet dit attribuut (met de gegeven namespace) de waarde "true" krijgen en mag er geen text inhoud in het element worden gegeven. Merk op dat het verstandig is om de benodigde namespace op het "recordIFResult" element te zetten, zodat het niet nodig is dit bij iedere null-waarde te doen.

De optionele onderdelen (timestamp en verwijderingen) staan in een apart "footer" element, dat na de opgeleverde records komt. De reden hiervoor is dat in §4.2.2 is gesteld dat deze waarden pas bekend hoeven te zijn nadat door alle resultaten is gelopen. Door deze in de projectie op XML eveneens onderaan te plaatsen, blijft deze algemeenheid behouden. Zo wordt het bijvoorbeeld mogelijk een component te ontwikkelen, die de resultaten van een willekeurige RecordIF op XML-documenten projecteert.

4.7. Een algemene Web service

Gegeven de projectie op XML, is het nu relatief eenvoudig een "service" te specificeren die queries aan resultaten koppelt op basis van een willekeurige onderliggende RecordIF implementatie.

Er bestaat reeds een gestandaardiseerde methode om dit te doen, namelijk de groep specificaties van het W3C die worden gebruikt om het begrip "Web services" te definiëren. Het gaat hierbij om SOAP, een protocol om XML-documenten "in te pakken" tot gestandaardiseerde berichten en WSDL, een taal om zulke berichten te combineren tot een specificatie van zo'n Web service.

Zo definieert WSDL een manier om twee typen SOAP-berichten te combineren tot een "Remote Procedure Call", precies wat nodig is om de hierboven beschreven service te specificeren. Hieronder is het WSDL-bestand afgebeeld met daarin deze combinatie. Deze wordt aansluitend besproken. Voor een exacte, volledige betekenis hiervan wordt verwezen naar de WSDL- en SOAP-specificaties [W3C2001], [W3C2000].

```
<wsdl:definitions
    ... namespace declaraties ...
    xmlns:unirec="http://pimsierhuis.nl/unirec/"
    targetNamespace="http://pimsierhuis.nl/unirec/">

    <wsdl:types>
        <xsd:schema targetNamespace="http://pimsierhuis.nl/unirec/">
            <xsd:import namespace="http://pimsierhuis.nl/unirec/"
                        schemaLocation="types.xsd"/>
        </xsd:schema>
    </wsdl:types>

    <wsdl:message name="getRecordsRequest">
        <wsdl:part name="request" element="unirec:recordIFQuery"/>
    </wsdl:message>

    <wsdl:message name="getRecordsResponse">
        <wsdl:part name="response" element="unirec:recordIFResult"/>
    </wsdl:message>

    <wsdl:portType name="unirecPortType">
        <wsdl:operation name="getRecords">
            <wsdl:input message="unirec:getRecordsRequest" />
            <wsdl:output message="unirec:getRecordsResponse" />
        </wsdl:operation>
    </wsdl:portType>

    <wsdl:binding name="unirecBinding" type="unirec:unirecPortType">
      .....
    </wsdl:binding>

</wsdl:definitions>
```

Figuur 6: WSDL-definitie van algemene Web service

In het “types” onderdeel wordt het XML-schema geïmporteerd dat de in de vorige paragraaf besproken projecties van queries en resultaten op XML vastlegt. De inhoud van dit bestand is de samenvoeging van het XML-schema voor queries en dat van resultaten. Deze zijn beide opgenomen als bijlage.

Vervolgens worden twee “berichten” gedefinieerd, één bericht voor het request onderdeel van de operatie en één voor het response deel van de operatie. Hierin wordt verwezen naar de XML-schema definitie van respectievelijk de query en het resultaat.

Daarna worden in het portType element de twee berichten gecombineerd tot een operatie, genaamd getRecords. Een portType is een interface specificatie van een Web service en bestaat, zoals hierboven te zien, uit een verzameling operatie definities.

Als laatste wordt in het binding element de daarboven gespecificeerde portType geprojecteerd op concrete SOAP-begrippen. Zo wordt vastgelegd dat er gebruik wordt gemaakt van HTTP als transportprotocol en dat de XML-berichten letterlijk worden overgenomen in de “Body” van de SOAP-berichten. Dit onderdeel is hier weggelaten voor overzichtelijkheid. Het gehele WSDL-bestand is te vinden in de bijlagen.

5. Ontwerp

In dit hoofdstuk wordt het ontwerp besproken van een mogelijke oplossing voor het probleem. Hierbij wordt gebruik gemaakt van de universele interface naar gegevensontsluitende systemen uit het vorige hoofdstuk ("RecordIF"), voor de communicatie tussen verschillende componenten.

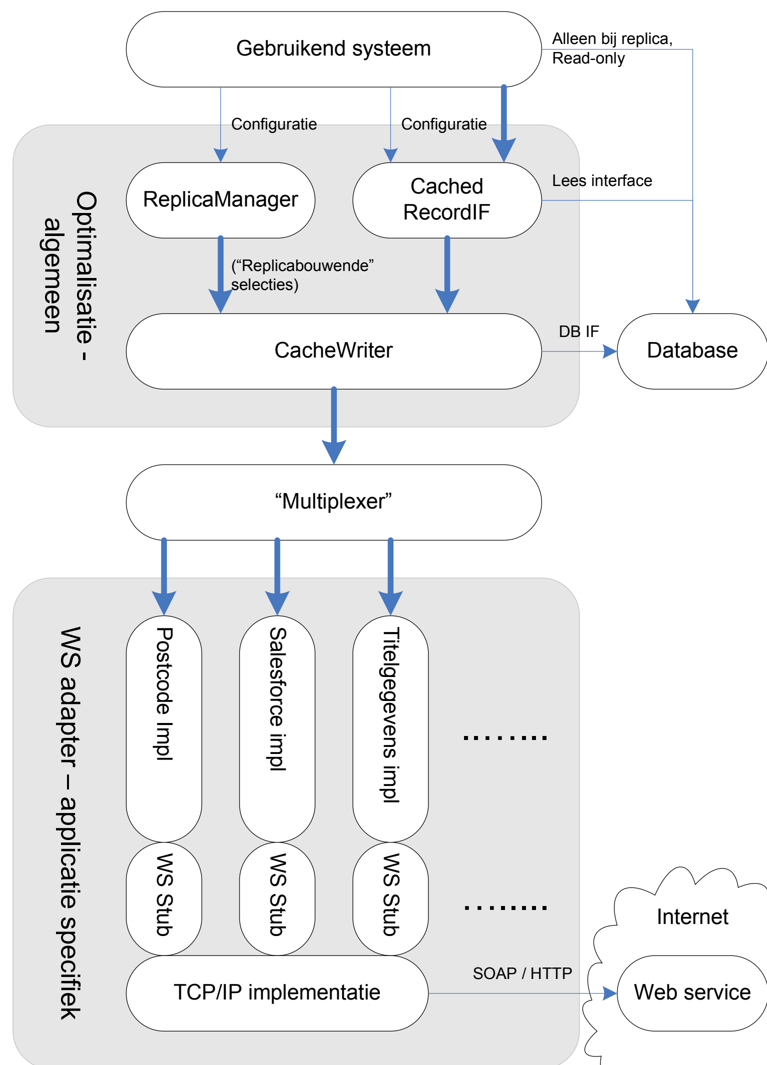
Hiertoe wordt eerst een abstract ontwerp opgesteld; daarbij wordt niet uitgegaan van een bepaalde implementatietaal of database technologie. Vervolgens wordt op basis van RecordIF een aantal algemene componenten gedefinieerd. Daarna wordt het abstracte ontwerp "ingevuld" met behulp van deze algemene componenten.

5.1. Globaal ontwerp

Hieronder is een diagram van het globaal ontwerp afgebeeld. Hierin geeft een pijl aan dat een deelsysteem een ander deelsysteem "gebruikt", oftewel afhankelijk is van dat deelsysteem. De richting van een pijl geeft dus niet de richting van de informatiestroom aan – er kan in beide richtingen informatie stromen. Een dikke pijl geeft aan dat op die plek gebruik wordt gemaakt van de universele interface.

Een implementatie van het ontwerp bestaat uit twee delen: 1) een algemeen deel dat de feitelijke afscherming van de prestatiegevolgen realiseert en 2) een deel dat specifiek is voor de applicatie, doordat het de "adapters" naar de universele interface implementeert voor de in die applicatie gebruikte Web services. ("adapter design pattern" [GOF1994]).

Het ontwerp zal hieronder uitgebreider per onderdeel worden besproken.



Figuur 7: Ontwerp oplossing

5.1.1. Web service adapter

De onderste laag bevat de "Web service adapters". Elke gekoppelde Web service heeft een dergelijke adapter nodig, om ervoor te zorgen dat de algemene RecordIF kan worden aangeboden aan de rest van het systeem. Op deze manier wordt dus geabstraheerd van de technische details van een Web service.

In het diagram is ter illustratie het geval van gangbare Web services gebruikt. Met "WS Stub" wordt de code bedoeld die voor een bepaalde Web service kan worden gebruikt om deze in een programmeertaal te benaderen. Deze wordt meestal automatisch gegenereerd door de Web service software. De stubs maken vervolgens gebruik van de internetprotocollen om de Web

service op het Internet te benaderen.

Normaal gesproken zullen deze stubs een laag-niveau manier voor het benaderen van de Web service aanbieden, bijvoorbeeld in de vorm van methodes die worden geprojecteerd op operaties van de Web service. Deze methodes leveren vervolgens bijvoorbeeld een XML-document of een Web service-specifiek object op. Deze gegevens moeten worden vertaald naar de algemene vorm die wordt geïmpliceerd door de RecordIF. Dit is de taak van de eigenlijke adapters (afgebeeld boven de stubs).

Deze abstractie bepaalt een belangrijk deel van de kracht van het systeem. Hierdoor is het mogelijk een universeel systeem voor de afscherming van de prestatiegevolgen te definiëren.

5.1.2. Aggregatie van de Web service – de Multiplexer

Omdat elke implementatie van de RecordIF (de adapter) een eigen verzameling soorten records ondersteunt, is het mogelijk meerdere van zulke adapters samen te voegen (te “aggregeren”) tot één implementatie van de RecordIF. Dit wordt gedaan in een triviaal deelsysteem dat, gegeven de lijst ondersteunde adapters, onthoudt welke soorten records deze individuele adapters ondersteunen en aanvragen voor een dergelijk soort naar de betreffende adapter “doorstuurt”.

Het nut hiervan is dat de bovenliggende deelsystemen kunnen abstraheren van het feit dat er meerdere Web services worden ondersteund, dan wel dat slechts één instantie van de bovenliggende deelsystemen hoeft te bestaan.

Als er slechts gebruik wordt gemaakt van één Web service, is dit onderdeel niet nodig. Indien één of meerdere adapters niet de mogelijkheid bieden om terug te koppelen welke soorten records worden ondersteund (omdat deze bijvoorbeeld algemeen is), is gebruik van dit onderdeel niet mogelijk. In dat geval zal er dus een instantie van het optimalisatiesysteem per dergelijke adapter moeten bestaan.

5.1.3. Alle opgevraagde records naar de cache schrijven – de CacheWriter

Om zowel de gevraagde cache- als replicatiefunctionaliteit te ondersteunen met dezelfde databaseimplementatie, is in het ontwerp het schrijven van de cache losgekoppeld van het lezen ervan.

De CacheWriter fungeert daartoe als “doorgeefluik” van RecordIF-aanvragen aan een onderliggende implementatie, met als eis dat er daarbij voor wordt gezorgd dat de opgeleverde records worden weggeschreven in een database.

Daarnaast dient de CacheWriter ervoor te zorgen, indien dit door de gebruiker wordt aangegeven, dat er een extra attribuut aan alle records wordt toegevoegd, met het huidige tijdstip. De naam van dit attribuut moet instelbaar zijn en het gebruik optioneel. Dit maakt het voor een gebruiker (zoals de CachedRecordIF) mogelijk te controleren hoe lang een record reeds in de database aanwezig is.

Het type database is een keuze, die wordt overgelaten aan de implementatie. Er dient wel rekening te worden gehouden met de eis uit §3.1.3, dat er (buiten het systeem om) een interface naar de database kan worden verkregen en dat er via deze interface zoekoperaties mogelijk moeten zijn op de database.

5.1.4. Een eenvoudige cache – de CachedRecordIF

De meest eenvoudige applicaties vragen de meest eenvoudige functionaliteit. Deze wordt geboden door de gecachte RecordIF. Omdat alle uit de Web service opgevraagde records via de CacheWriter worden opgevraagd, zullen al deze records in de cache-database worden opgeslagen.

Daarnaast heeft de CachedRecordIF een lees interface naar deze database. Op het moment dat er een selectie wordt gevraagd op id, wordt deze database geraadpleegd of het betreffende record reeds in de database aanwezig is. Is dit het geval, wordt dit record teruggegeven.

Omdat de CacheWriter tevens functionaliteit biedt om per record aan te geven sinds wanneer deze in de database aanwezig is, is het mogelijk een maximale “ouderdom” van het gecachte record aan te geven.

5.1.5. Replicatie – de ReplicaManager

Applicaties waarbij de prestatiegevolgen van de koppeling moeten worden afgeschermd voor meer dan alleen de ID-selectie, vereisen een uitgebreider systeem. Het gaat hier om de replicatiefunctionaliteit, zoals beschreven in §3.1.3.

Het replicatiesysteem maakt gebruik van de CacheWriter om de database te vullen, door de eis dat de resultaten van alle selecties die worden opgevraagd naar de database zullen worden geschreven. De meest eenvoudige manier van het bijhouden van een replica, zou dan zijn om zo nu en dan een All-selectie uit te voeren. Na deze selectie zouden alle records immers in de database aanwezig moeten zijn.

Nu is deze manier van repliceren niet efficiënt voor grote databases waarvan de records minder vaak wijzigen dan dat ze worden gelezen. Daarom kan het replicatiesysteem een stuk efficiënter worden geïmplementeerd met behulp van specifieke functionaliteit.

Deze functionaliteit wordt beschreven in §4.3.2. Het doel van de ReplicaManager is het uitvoeren van het algoritme, dat wordt beschreven in die paragraaf. Op die manier ontstaat een algemeen systeem voor het bijhouden van een replica.

5.2. Algemene componenten op basis van RecordIF

Hieronder wordt een aantal algemene componenten beschreven, die gebruik maken of een implementatie realiseren van de algemene interface, RecordIF. Op basis van deze componenten kunnen dan eenvoudig nieuwe systemen in elkaar worden gezet.

In dit hoofdstuk wordt slechts vastgelegd wat dergelijke componenten zouden kunnen, niet hoe deze worden geïmplementeerd. Dit laatste is het onderwerp van hoofdstuk 6.

5.2.1. Algemene interface naar SQL-databases

Veel databases maken gebruik van SQL om de onderliggende gegevens te ontsluiten. Het ligt daarom voor de hand om een “adapter” te creëren, die gegeven een (configuratie voor een) SQL-interface, een RecordIF interface levert naar de gegevens die daardoor worden ontsloten.

Omdat de recordstructuur van RecordIF zo gekozen is dat deze overeenkomt met die van relationele databases, is dit vrij eenvoudig. RecordTypes corresponderen dan met tabellen, records met regels uit een tabel, attributen met kolommen en het id-attribuut is de primary key van de betreffende tabel.

Voor het ondersteunen van iets uitgebreidere selecties, wordt een applicatiespecifieke selectie gecreëerd bij deze adapter. Het gaat hier om de “AttrValueConstraint” selectie, die gegeven een attribuutnaam, soort beperking (zoals “<” voor kleiner dan) en een constante, alle records teruggeeft waarvoor de opgegeven beperking geldt op de waarde van het betreffende attribuut en de constante.

Een probleem ontstaat echter op het moment dat wordt gewerkt met een “composite” primary key. Dit wordt niet ondersteund. Een oplossing hiervoor is om niet met composite keys te werken, maar met een enkelvoudige, kunstmatige primary key met daarnaast een “unique index” over de attributen die deel uit zouden maken van de composite key.

5.2.2. “Doorgeef” RecordIF die schrijft naar een SQL-database

Een tweede mogelijkheid, is een component die dient om records weg te schrijven naar een dergelijke SQL-database. Het idee hierbij is een component die op basis van een SQL-interface en een andere RecordIF, deze RecordIF één op één aanbiedt aan de gebruiker, met als toevoeging dat alle records die langskomen naar de SQL-database worden geschreven.

Verder wordt functionaliteit geboden die ervoor zorgt dat het tijdstip van schrijven aan elk record wordt toegevoegd als extra toegevoegd attribuut. Deze functionaliteit moet configureerbaar zijn.

5.2.3. Projectie van query en resultaat op XML

Bij de definitie van de universele interface hoort de gestandaardiseerde projectie van een query en een resultaat op XML. Dit maakt het mogelijk een component te creëren, die gegeven een willekeurige query of resultaat, deze op de juiste wijze projecteert op een dergelijk XML-document.

Dit XML-document kan vervolgens worden verstuurd over een netwerkverbinding of worden weggeschreven naar een bestand. Naast de eerder genoemde voordelen hiervan, impliceert dit ook de mogelijkheid om een dergelijk bestand fysiek naar een andere locatie te verplaatsen. Als nut hiervan kan worden gedacht aan het initialiseren van een replica uit een dergelijk bestand, indien het gaat om een te grote hoeveelheid gegevens om te versturen over een netwerk.

5.2.4. Parsen van query en resultaat uit XML

Naast het projecteren van een query en resultaat op XML, kan worden gedacht aan een component die een dergelijke XML-representatie interpreteert en weer omzet naar een query of resultaat object zelf.

Het parsen van een query-representatie is (mits de implementatietaal hier functionaliteit voor biedt), betrekkelijk eenvoudig. Het parsen van het resultaat kan tot een grotere uitdaging leiden, omdat hierbij niet vanzelfsprekend is dat het hele document tegelijk beschikbaar is. Hierbij kan worden gedacht aan het moment dat het document van een netwerkverbinding gelezen wordt en te groot is om in het interne geheugen van de computer te laden. Er zal dan een stuk “RecordIterator” programmatuur moeten worden gecreëerd dat de records één voor één uit het document leest (bij het aanroepen van de next() operatie), zonder het gehele document beschikbaar te hebben.

5.2.5. Algemene Web service / client

Verder is het mogelijk een algemene Web service te creëren die, gegeven een willekeurige RecordIF, deze aanbiedt als Web service op een computernetwerk. Daarmee kan dan op een interoperabele manier en op afstand gebruik worden gemaakt van de betreffende RecordIF. Een dergelijke Web service dient te conformeren aan de WSDL-specificatie uit §4.7.

Het opbouwen van een dergelijke Web service bestaat, naast de eerder besproken projectie op XML, uit twee onderdelen, die hieronder worden besproken (zie ook §4.7).

SOAP-berichten

Het “inpakken” van een XML-element tot SOAP-bericht is zeer eenvoudig. Dit komt neer op het plaatsen van twee XML-elementen met de SOAP-namespace om het in te pakken XML-element heen. Het parsen van SOAP-berichten komt dus neer op het parsen van deze zelfde twee elementen. Voor het schrijven en parsen van SOAP-berichten is daarom eenvoudig een algemeen component te ontwikkelen.

Web service: koppeling van request aan response SOAP-bericht

De implementatie van een algemene Web service komt nu neer op het samenvoegen van bovenstaande componenten, op het accepteren en onderhouden van de HTTP-verbindingen na. Hieronder wordt het proces dat de Web service implementeert in stappen beschreven.

- Wacht op HTTP-verbinding van client
- Accepteer nieuwe verbinding
- Ontvang SOAP-bericht met request
- Verwerk SOAP-request tot query
    - Parse / verwijder start SOAP-XML-elementen
    - Parse RecordIFQuery uit tussenliggend XML-document
    - Parse / verwijder eind SOAP-XML-elementen
- Roep onderliggende RecordIF aan met deze query, verkrijg daarbij het resultaat
- Verstuur SOAP-bericht met resultaat
    - Verstuur start SOAP-XML-elementen
    - Verstuur resultaat als XML-document
    - Verstuur eind SOAP-XML-elementen
- Sluit verbinding

Algemene Web service client

Een algemene Web service client zorgt ervoor dat de door de Web service aangeboden onderliggende RecordIF kan worden aangeroepen als een lokale implementatie van de RecordIF. Hieronder wordt op soortgelijke wijze als hierboven het proces van een dergelijke algemene client beschreven. Dit proces correspondeert met bovenstaand proces, met als verschil dat het resultaat direct wordt teruggekoppeld met daarin een “RecordIterator”, die de records één voor één uit de verbinding kan lezen.

- Wacht op “getRecords” aanroep van gebruiker, verkrijg daarbij de query
- Maak HTTP-verbinding naar Web service
- Verstuur SOAP-bericht met request
    - Verstuur start SOAP-XML-element
    - Verstuur query als XML-document
    - Verstuur eind SOAP-XML-element
- Start ontvangst SOAP-bericht met resultaat
- Verwerk SOAP-bericht tot resultaat
    - Parse / verwijder start SOAP-XML-elementen
    - Parse RecordIFResult met teruggegeven records uit tussenliggend XML-document
        - Parse RecordType uit “header” element
        - Construeer RecordIterator
        - Koppel RecordIFResult met RecordType en RecordIterator terug aan gebruiker
        - Gebruiker loopt door resultaat, door steeds de next() operatie van RecordIterator aan te roepen
        - Parse optionele onderdelen (timestamp en verwijderingen) uit “footer” element, voeg deze toe aan RecordIFResult
    - Parse / verwijder eind-SOAP-XML-tags. (Nadat de gebruiker door alle records is heengelopen en de optionele onderdelen zijn geparst, wordt hiertoe een “callback” van de SOAP-parser aangeroepen door de RecordIterator)
- Server sluit verbinding

5.3. Invulling ontwerp met algemene componenten

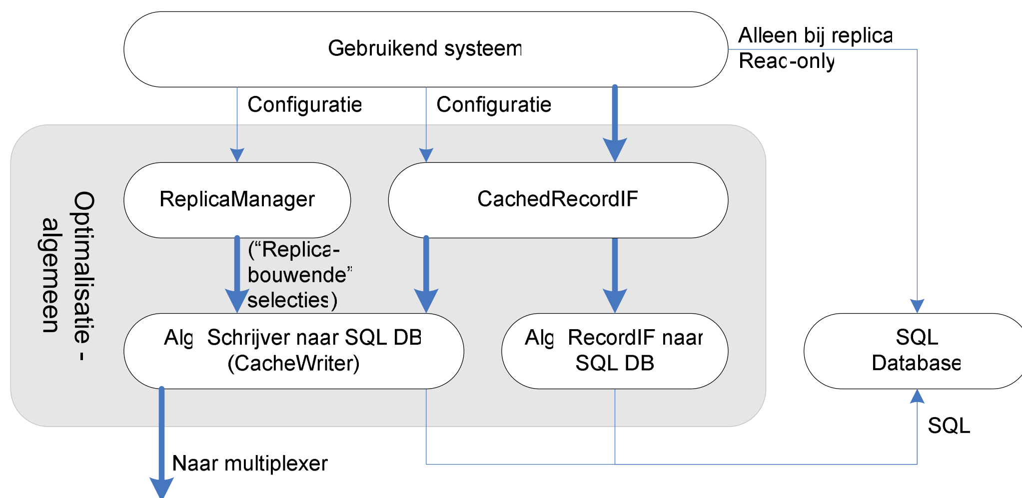
Hieronder wordt het ontwerp uit §5.1 ingevuld met algemene componenten uit de vorige paragraaf. Hiertoe is het ontwerp opgesplitst in twee hoofdonderdelen: 1) het optimalisatiesysteem zelf en 2) de achterliggende gegevensontsluiters: de daadwerkelijke systemen en haar clients / adapters.

Er wordt hierbij op sommige plekken aangenomen dat gebruik wordt gemaakt van een SQL-database. Door de standaardisatie van de interface waarmee gegevens worden ontsloten, kan hiervoor zonder al teveel moeite een alternatief worden gekozen. Dit zou neerkomen op de implementatie van een lees- en schrijfimplementatie van RecordIF naar de betreffende database.

5.3.1. Het optimalisatiesysteem

Het optimalisatiesysteem zelf is in feite niet erg ingewikkeld. Dit komt doordat, vanuit haar oogpunt, de gegevens op eenvoudige wijze (via de RecordIF) worden aangeleverd.

In het diagram hieronder is het ontwerp ingevuld door gebruik te maken van de algemene lees- en schrijfimplementatie van RecordIF naar SQL-databases. De ReplicaManager en CachedRecordIF zijn hierin de enige nog te implementeren, maar vrijwel triviale onderdelen. Ze worden hierna apart besproken.



Figuur 8: Invulling ontwerp: Optimalisatie systeem

CachedRecordIF

De CachedRecordIF wordt door de gebruiker geconfigureerd met een maximale levensduur van gecachte records.

Vervolgens kan de gebruiker met de geboden RecordIF selecties doen. Deze selecties worden één op één doorgegeven aan de CacheWriter, die ze op zijn beurt doorgeeft aan de onderliggende implementatie. Dit geldt echter niet voor de ID-selectie; deze wordt eerst doorgegeven aan de RecordIF naar de cache-database. Als deze het gevraagde record oplevert, wordt dit record teruggegeven. Is dit niet het geval, wordt de selectie behandeld als alle andere, door de CacheWriter.

Het enige niet-triviale in dit onderdeel is het controleren van de maximale levensduur van de records. De CacheWriter dient namelijk zo ingesteld te worden dat bij elk attribuut een extra record wordt toegevoegd met het tijdstip van schrijven. Deze moet bij het opvragen uit de cache-database worden vergeleken met het huidige tijdstip. Liggen deze twee verder uit elkaar dan de door de gebruiker aangegeven maximale waarde, wordt de selectie alsnog via de CacheWriter behandeld.

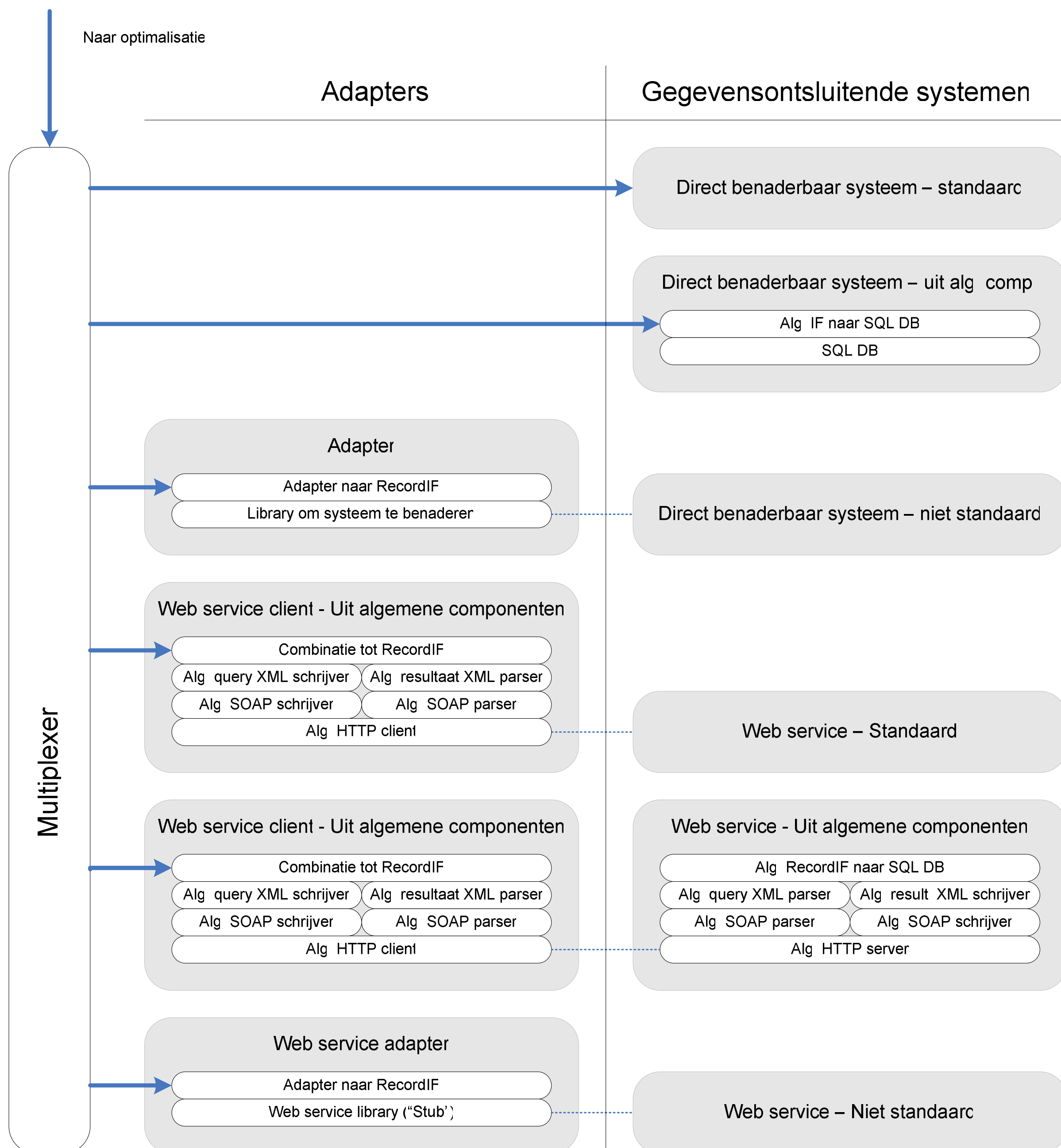
Verder biedt de CachedRecordIF een "applicatiespecifieke" selectie "NotCachedIDSelection". Dit is een subklasse van de ID-selectie, zonder dat deze daarbij wordt uitgebreid. Het enige verschil met de standaard ID-selectie is dat de CachedRecordIF daarbij de cache-database niet zal raadplegen.

ReplicaManager

Replicatiefunctionaliteit is eveneens vrijwel triviaal te implementeren. Deze wordt geconfigureerd met een interval waarmee de replicatieslagen moeten worden uitgevoerd. Vervolgens kan met de CacheWriter als onderliggende RecordIF het algoritme van §4.3.2 worden uitgevoerd. Indien gewenst kan de timestamp worden onthouden of bij het herstarten uit de database worden verkregen, zoals eveneens in die paragraaf wordt besproken.

5.3.2. Adapters en gegevensontsluitende systemen

De eigenlijke kern en kracht van het systeem ligt in de gestandaardiseerde interface. In deze paragraaf wordt daarom ingegaan op de vraag hoe bestaande systemen kunnen worden aangekoppeld, welke kwesties daarbij een rol spelen en hoe nieuwe gegevensontsluiters uit algemene componenten worden opgebouwd. Dit zal worden gedaan aan de hand van onderstaande figuur, die correspondeert met het "applicatiespecifieke" deel van Figuur 7. Ook hier geven de dikke pijlen aan dat op die plek gebruik wordt gemaakt van de RecordIF.



Figuur 9: Invulling ontwerp - adapters en gegevensontsluitende systemen

Direct benaderbare systemen

Met een direct benaderbaar systeem wordt een systeem bedoeld dat direct in dezelfde programmeertaal als de client kan worden benaderd. In de figuur hierboven zijn de eerste drie voorbeelden van dit type.

Het eerste voorbeeld is een direct benaderbaar systeem dat een interface biedt die conformeert aan de RecordIF. Dit systeem is daarom zonder programmeerwerk aan het optimalisatiesysteem te koppelen.

Het tweede voorbeeld betreft eveneens een systeem dat zich conformeert aan de standaard. Dit keer is echter bekend hoe het systeem is geïmplementeerd. Het is namelijk opgebouwd uit algemene componenten. In het voorbeeld betreft het een systeem dat de gegevens uit een SQL-database ontsluit. Aangenomen dat er een algemeen component (zoals dat uit §5.2.1) bestaat om de SQL-interface om te zetten naar de RecordIF, kan dit systeem worden opgebouwd zonder te programmeren.

Het derde voorbeeld is een systeem dat niet direct aan RecordIF conformeert. In dit geval zal er een adapter moeten worden gecreëerd, die RecordIF aanroepen hierop transformeert naar aanroepen op de software library om dit systeem te benaderen. Het kan ook zijn dat er geen library nodig is om het systeem te benaderen. In veel gevallen is het creëren van een adapter betrekkelijk eenvoudig, omdat er door RecordIF geen ingewikkelde functionaliteit wordt geëist van de implementatie.

Web services

In veel gevallen zal een gegevensontsluitend systeem slechts benaderbaar zijn via een Web service. Ook hiervan zijn drie voorbeeldgevallen gegeven.

In het eerste geval gaat het om een systeem dat zich conformeert aan de standaardmanier om een RecordIF als Web service beschikbaar te stellen. In dit geval hoeft er weer niet geprogrammeerd te worden, omdat de adapter (of "client") die nodig is om die interface om te zetten in een RecordIF kan bestaan uit een aantal algemene componenten. Het gaat hier om de algemene componenten om te fungeren als HTTP-client, om XML-berichten in te pakken in en uit te pakken uit SOAP-berichten en om XML-projecties van queries te verkrijgen en XML-projecties van een resultaat te parsen. In §5.2.5 is reeds uitgelegd hoe een dergelijke adapter zou werken.

In het tweede geval is weer de implementatie van het gegevensontsluitende systeem zichtbaar gemaakt. Het betreft hier een Web service die is opgebouwd uit algemene componenten, op soortgelijke wijze als de client uit het vorige voorbeeld. Ook de werking van een dergelijke Web service is uitgelegd in §5.2.5. Als voorbeeld is hier gekozen om een SQL-database te ontsluiten met behulp van de algemene RecordIF naar SQL-databases. Merk op dat er hier voor kan worden gekozen om een willekeurig onderliggend systeem om te zetten in een Web service, mits het zich houdt aan de RecordIF of er een adapter voor beschikbaar is.

Het laatste voorbeeld betreft een Web service die zich niet houdt aan de standaardprojectie. Dit geval is min of meer gelijk aan het derde voorbeeld van direct benaderbare systemen, omdat er een adapter moet worden geschreven die (eventueel) gebruik maakt van een software library.

6. Implementatie in Java

In dit hoofdstuk wordt de implementatie van de eerder besproken componenten behandeld. Er is gekozen voor Java als implementatietaal, omdat zo direct kan worden aangesloten op het systeem van de opdrachtgever en met deze taal de meeste ervaring is opgedaan.

Alle hieronder besproken klassen zijn niet meer dan enkele tientallen regels code. Dit betekent dat het duidelijke, eenvoudige en makkelijk op fouten controleerbare deelproblemen zijn.

6.1. Algemene interface naar SQL-databases

In deze paragraaf worden de componenten besproken die gebruik maken van een onderliggende SQL-database. Om te verbinden naar de onderliggende SQL-database, wordt gebruik gemaakt van JDBC [JDBC], de standaard manier om in Java SQL-databases te benaderen. Dit biedt eveneens de mogelijkheid om alle “merken” SQL-databases waar een JDBC-driver beschikbaar voor is op één uniforme manier te benaderen en zo al deze databases in één keer te ondersteunen.

De configuratie die aangeeft welke database moet worden gebruikt, is opgenomen in een aparte klasse. Het gaat hier om een eenvoudige klasse, die slechts de vier configuratiewaarden bevat. Deze waarden zijn: driverClass (de te gebruiken driver, bijvoorbeeld de klasse die de driver voor MySQL gebruikt), URL (die aangeeft hoe naar de database moet worden verbonden) en de gebruikersnaam en wachtwoord voor de betreffende database.

Ook de methode om een verbinding naar de database te verkrijgen is in een aparte klasse opgenomen. Op deze manier kan deze code worden gedeeld door de lees en schrijf implementaties en kan de code later worden aangepast om de verbinding bijvoorbeeld op een geavanceerde manier te verkrijgen (uit een “pool” van verbindingen bijvoorbeeld). Naast de methode om een verbinding te verkrijgen, bevat deze klasse een “utility” methode om de tabellen met hun primary keys te verkrijgen. Een object van deze klasse kan worden geconstrueerd met behulp van een databaseconfiguratie uit de vorige alinea.

Hieronder worden apart de implementaties van de lees- en schrijfcomponenten besproken.

6.1.1. Lezen

Een JdbcReaderRecordIF (zoals de implementatieklasse genoemd is), wordt geconstrueerd met een database configuratie object. Hiermee wordt naar de database verbonden en worden de aanwezige tabellen verkregen, zodat bekend is welke soorten records (deze zijn gelijk aan de tabelnamen) kunnen worden ondersteund.

Op het moment dat nu de getRecords methode wordt aangeroepen, wordt de selectie vertaald naar een SQL “WHERE” expressie. Bijvoorbeeld: “WHERE <id-attribuut>=<in selectie opgegeven waarde van id>” voor een ID-selectie. Het id-attribuut is verkregen bij het ophalen van de aanwezige tabellen.

Nu de selectie is vertaald naar een WHERE-expressie, kan deze worden uitgevoerd. Daartoe wordt de volgende SQL-query uitgevoerd:

```
SELECT * FROM <soort record uit query> WHERE <naar sql vertaalde selectie>
```

Daarbij wordt een ResultSet verkregen, die tevens de meta-data (zoals de kolomnamen) van de gevraagde query bevat. Aan de hand van deze meta-data wordt het voor het resultaat benodigde RecordType object geconstrueerd.

Daarnaast wordt er aan de hand van de ResultSet een speciale implementatie van RecordIterator gecreëerd, die samen met het RecordType als resultaat wordt opgeleverd. Deze RecordIterator kan vervolgens simpelweg volstaan met elke keer dat de gebruik de next() methode aanroept, de volgende rij uit de ResultSet te verkrijgen en aan de hand van de waarden daaruit een Record te creëren.

Replicatie

Standaard wordt door deze RecordIF implementatie geen replicatiefunctionaliteit ondersteund. Hier is immers meer informatie voor nodig. Als bekend is dat de timestamps van elk record onder een

bepaald attribuut worden opgeslagen en deze waarde door de standaard relationele operators van SQL kunnen worden vergeleken, kan deze functionaliteit echter vrij eenvoudig worden gecreëerd.

Een voorbeeld hiervan is gegeven als de “TimestampAttributeJdbcReaderRecordIF” klasse. Dit is (zoals de naam al aangeeft) een subklasse van de JdbcReaderRecordIF en wordt geconstrueerd met een extra configuratiewaarde: de naam van het attribuut waar de timestamp in wordt bijgehouden.

Op het moment dat nu de getRecords methode wordt aangeroepen, wordt gekeken of het gaat om een ModifiedSince-selectie. Is dit het geval, dan wordt deze omgezet in de in §5.2.1 besproken AttrValueConstraint selectie, gegeven het timestamp attribuut, de “>=” operator en de gevraagde timestamp-waarde uit de selectie. Zo worden dus alle records teruggegeven waarvan de timestamp groter is dan de gevraagde waarde, precies wat de ModifiedSince-selectie zou moeten doen.

Om tevens de voor replicatie benodigde timestamp “tot welk moment de teruggegeven records actueel zijn” te kunnen teruggeven, wordt een andere methode toegepast. Hiervoor wordt de stelling gebruikt dat als er een record in de database voorkomt met timestamp X, de database wel actueel moet zijn tot deze timestamp X (mits gebruik wordt gemaakt van het replicatieproces uit §4.3.2). Dit geldt, omdat “actueel zijn tot” in §4.2.2 is afgezwakt, door te stellen dat niet noodzakelijk *alle* wijzigingen van die timestamp hoeven te zijn teruggegeven (oftewel: de timestamp geeft aan *tot* welk moment de gegevens up-to-date zijn, en niet *tot en met*). Gegeven deze stelling, is de database dus minimaal actueel tot de maximale waarde van alle timestamps van alle records van het betreffende soort record. Deze waarde kan eenvoudig worden verkregen met de SQL “max()” functie. Aangenomen dat de juiste index op dit attribuut aanwezig is, kan deze waarde tevens efficiënt worden verkregen.

Verwijderingen worden niet door deze algemene component ondersteund, omdat er geen algemene en vanzelfsprekende manier is om deze bij te houden. Dit vergt extra ondersteuning van het systeem dat de gegevens in de database bijhoudt, waar de adapter van op de hoogte zou moeten zijn.

6.1.2. Schrijven

De algemene SQL-schrijfcomponent is een stuk eenvoudiger, omdat deze zelf geen selectie functionaliteit hoeft te ondersteunen. Het is immers zo, dat de resultaten van de onderliggende (“te schrijven”) RecordIF één op één kunnen worden doorgegeven aan de gebruiker.

Het enige dat de component doet is, na het doorgeven van de query aan de onderliggende RecordIF, het “inpakken” van de RecordIterator uit het teruggegeven resultaat met een eigen implementatie. Deze implementatie roept de onderliggende iterator aan, maar daarnaast schrijft deze het record weg naar de database. Hiertoe wordt, bij het construeren van de nieuwe iterator, een SQL-statement klaargemaakt (“prepared”):

```
INSERT INTO <recordtype> (attr1, ... , attrN)
VALUES <?, ?, ...>
ON DUPLICATE KEY UPDATE
<attr1>=values(<attr1>), ... , <attrN>=values(<attrN>)
```

Bij verwerking van het volgende record (bij aanroepen van de next() methode), worden de vraagtekens (placeholders) ingevuld met de attribuutwaarden van het betreffende record, waarna het statement kan worden uitgevoerd. Op het moment dat de insert mislukt omdat er reeds een record bestaat met dezelfde primary key, wordt dit record geactualiseerd met dezelfde waarden.

De “ON DUPLICATE KEY UPDATE” maakt het mogelijk de communicatie met de database te beperken tot één statement per record. Deze functionaliteit is echter specifiek voor de MySQL databaseserver. Andere soorten databaseservers kunnen als volgt worden ondersteund: doe eerst een UPDATE statement en controleer op hoeveel rijen dit statement betrekking had. Is dit aantal nul, doe dan een INSERT met dezelfde waarden. Hierbij ontstaat echter de noodzaak om gebruik te maken van transacties, omdat theoretisch na de (mislukte) UPDATE wellicht al een ander proces de INSERT heeft gedaan, waardoor de INSERT van dit proces mislukt. Bij de MySQL-specifieke functionaliteit hoeft hier geen rekening mee te worden gehouden, omdat deze atomair is.

6.2. Projectie op XML

In deze paragraaf wordt de implementatie behandeld van de algemene projectie van queries en resultaten op XML, zoals besproken in §4.6, §5.2.3 en §5.2.4.

De lees- en schrijfimplementaties werken op respectievelijk Input- en OutputStreams, die abstraheren van de manier waarop in een dergelijk object de gegevens worden ontvangen / verstuurd. Hierdoor is het bijvoorbeeld mogelijk uit / naar zowel een netwerkverbinding als een bestand op harde schijf te lezen / schrijven.

Omdat een RecordIF neerkomt op een operatie, kan deze niet worden vastgelegd op een XML-document. Het is dus (zoals in hoofdstuk 4 besproken) slechts mogelijk om afzonderlijke query en resultaat objecten op XML te projecteren. Hieronder zal dus worden besproken hoe RecordIFQuery en RecordIFResult objecten worden geschreven naar of geparst uit een XML-document.

Omdat wordt gewerkt met een standaard Java library ("Streaming API for XML" of "StAX") voor het lezen en schrijven van XML, is character-encoding (ondersteuning van speciale tekens en de projectie daarvan op bytes) geen kwestie. Dit wordt automatisch afgehandeld door deze library. Er wordt daarbij gebruik gemaakt van (standaard) unicode en UTF-8.

6.2.1. Schrijven

Eerst wordt het schrijven van een XML-projectie van een query of resultaat behandeld. Dit is namelijk een stuk eenvoudiger dan lezen (parsen), omdat 1) bij schrijven niet geldt dat er willekeurige invoer verwacht moet worden en 2) schrijven niet asynchroon werkt (met een iterator waar de gebruiker op elk moment een Record uit kan willen lezen).

Om ervoor te zorgen dat de XML-documenten makkelijk kunnen worden "ingepakt" tot een nieuw XML-document (zoals een SOAP-bericht) door hetzelfde stuk programmatuur, implementeren de schrijf componenten een algemene interface, genaamd XMLWriter. Deze interface bevat de methode "void doWrite(XMLStreamWriter writer)", die de XML-projectie van het betreffende object (bij de constructor meegegeven RecordIFResult of RecordIFQuery) naar de XMLStreamWriter schrijft. Deze kan vervolgens door een "bovenliggende" schrijver worden aangeroepen om het document in te pakken.

Query

Het ingewikkelde aspect van het schrijven van een query is de selectie. De selectie kan namelijk een willekeurig aantal argumenten hebben. Een selectie is echter zo geïmplementeerd dat de argumenten als array kunnen worden opgevraagd. Zo kan eenvoudig de lijst van "arg" XML-elementen worden geschreven.

Verder moet worden uitgezocht of het een applicatie-specifieke of een algemene selectie betreft. Dit wordt gedaan met de "isAssignableFrom" methode van de "Class" klasse. Hiermee kan worden bepaald of de selectie een subklasse is van de "GeneralSelection" klasse, de klasse waar alle algemene selecties een subklasse van zijn. In dit geval is de string-representatie (die nodig is in het "type" XML-element) gelijk aan de klassenaam van het selectie object. In het geval van een applicatie-specifieke selectie wordt gebruik gemaakt van de fully qualified klassenaam (de klassenaam inclusief de packagenaam).

Resultaat

De programmatuur voor het schrijven van het resultaat is uitgebreider dan die van de query, maar niet ingewikkelder. Ook dit gaat erg rechttoe rechtaan, gegeven de specificatie uit §4.6.2.

Ook hier zit één aspect dat dit niet-triviaal maakt. Het gaat hier om attribuutwaarden die geen waarde bevatten, de zogenaamde "null" waarde. Deze worden gedetecteerd door te controleren of een attribuutwaarde gelijk is aan "null" (attribuutwaarde == null). In dit geval zal het "nil" attribuut (uit de "XMLSchema-instance" namespace) van het representerende "a" XML-element de waarde "true" worden gegeven. Zie ook de specificatie in hoofdstuk 4.

6.2.2. Lezen

Het parsen van XML-documenten is in het algemeen lastiger dan het schrijven ervan. Er bestaan hier meerdere manieren voor, zoals DOM (Domain Object Model), waarbij het document in één keer wordt geïnterpreteerd en daarbij als datastructuur in de programmeertaal te lezen valt. Dit is een eenvoudige manier, maar niet geschikt voor het parsen van een resultaat van een RecordIF, omdat deze niet altijd in één keer in het geheugen kan worden opgeslagen.

Een andere manier is SAX (Simple API for XML), die dit probleem oplost door het geïnterpreteerde document niet in het geheugen op te slaan, maar bij "gebeurtenissen" tijdens het parsen (zoals het tegenkomen van een begin-tag van een element) "callback" methodes van de gebruiker aan te roepen. De gebruiker kan dan zelf beslissen wat hiermee te doen. Bij een eerste implementatie van de algemene resultaat parser, bleek een implementatie met behulp van SAX echter verre van triviaal. Dit kwam, doordat de gebruiker van de RecordIF eveneens op willekeurige momenten een nieuw record opvraagt (met de next() methode van de RecordIterator). Dit impliceerde de introductie van een buffer, een aparte thread om te parsen en de daarbij behorende synchronisatiekwesties.

Na implementatie hiervan bleek er een (zeer) nieuwe, eenvoudigere manier van parsen te bestaan, genaamd StAX (Streaming API for XML [STAX]). Deze is vergelijkbaar met SAX, maar in plaats van de "onhandige" callbacks, wordt hier gebruik gemaakt van een iterator-achtige interface. De next() methode van de StAX-parser zoekt binnen het document naar een volgende "gebeurtenis" en vervolgens zijn met andere methoden de details van die gebeurtenis op te vragen. Op die manier kan bij het door de gebruiker aanroepen van de next() methode op de RecordIterator, direct (synchroon) het volgende record uit het XML-document worden geparst. Dit reduceert het probleem van "behoorlijk ingewikkeld" tot "betrekkelijk eenvoudig".

Om dezelfde reden als bij de schrijfcomponenten, implementeren ook de lezers een algemene interface, genaamd XMLParser. Deze bevat de methode "Object doParse(XMLStreamReader parser)", die een bovenliggende parser kan aanroepen om het ingepakte document te verkrijgen. In het geval van de query parser is het Object een RecordIFQuery en in het geval van het resultaat parser een RecordIFResult.

Query

Het implementeren van de parser voor queries is triviaal. Het enige ingewikkelde aspect is weer het interpreteren van de selectie tot een selectie-object. Dit wordt gedaan door de type-string om te zetten in een klassenaam (zoals al bij het schrijven in omgekeerde volgorde is beschreven), van deze klasse een nieuw object te instantiëren (met de "Reflection API" [REFL]) en vervolgens de gevonden argumenten in dit object te plaatsen.

Resultaat

Zoals hierboven al besproken is het parsen van het resultaat een stuk ingewikkelder, omdat dit "streaming" moet gebeuren. Met StAX is dit echter geen probleem: vóór het opleveren van het resultaat object wordt het "header" element met de gegevens over het RecordType geparst. Vervolgens wordt, gegeven de StAX-parser, een speciale implementatie van RecordIterator gecreëerd, die bij het aanroepen van de next() methode direct het volgende record uit het document interpreteert en oplevert.

Het enige probleem dat nu nog blijft, is het interpreteren van de optionele onderdelen van een resultaat, die aan het eind van het document in het "footer" element staan. Omdat deze in het RecordIFResult object moeten worden geplaatst en dit object op het moment van parsen reeds aan de gebruiker is teruggekoppeld, moet dit worden gedaan in de RecordIterator. Dit wordt gedaan door in de implementatie van de next() methode, na het parsen van het volgende record, te controleren of dit het laatste aanwezige record was. Is dit het geval, dan worden de optionele gegevens uit het footer element geïnterpreteerd en in het RecordIFResult object geplaatst.

6.3. Algemene Web service / client

In deze paragraaf wordt de implementatie beschreven van de algemene Web service en de "adapter" voor een dergelijke service terug naar een RecordIF.

6.3.1. SOAP-berichten

Zoals eerder aangegeven, is het "inpakken" van een XML-document tot een SOAP-bericht in dit geval erg eenvoudig. Het gaat slechts om het omsluiten van het XML-document (wat vervolgens volgens de definitie zelf dus geen XML-document meer zal zijn) door twee speciale XML-elementen met een SOAP-namespace.

Schrijven

Omdat de schrijfimplementatie voor zowel queries als voor resultaten dezelfde interface implementeren, kunnen met één stuk programmatuur beide soorten XML-documenten in SOAP-berichten worden ingepakt. Hiertoe schrijft de SOAP-schrijver de begin-tags van de SOAP-elementen, roept de doWrite() methode van de onderliggende query of resultaat schrijver aan en als deze klaar is schrijft hij de eind tags van de SOAP-elementen.

Parsen

Het parsen van de SOAP-elementen gebeurt op een soortgelijke wijze: eerst worden de begin-tags van de SOAP-elementen geparst, vervolgens wordt de doParse() methode van de onderliggende parser aangeroepen. Dit levert een RecordIFQuery of RecordIFResult object op. Vervolgens kunnen de eind-tags van de SOAP-elementen worden geparst.

Bij de resultaat-parser is dit laatste echter niet zonder meer mogelijk, omdat op het moment dat het RecordIFResult is opgeleverd, het ingepakte document nog niet helemaal geparst is. Dit wordt immers pas gedaan op het moment dat de gebruiker de RecordIterator gebruikt om de records in te lezen. Dit is opgelost door de RecordIterator, op het moment dat door het gehele resultaat is heengelopen, een "callback" methode van de SOAP-parser te laten aanroepen. In deze methode worden dan de SOAP-eind-tags geparst.

6.3.2. Algemene Web service

Nu er algemene componenten zijn geconstrueerd voor het interpreteren en schrijven van queries en resultaten uit SOAP-berichten, kunnen deze worden gecombineerd tot een algemene Web service.

De laatste techniek waarvoor hierbij nog een implementatie moet worden geïmplementeerd, is het HTTP-protocol. Hiervoor wordt gebruik gemaakt van een reeds bestaande techniek voor de afhandeling van dit protocol aan de server-kant, namelijk servlets [SERV]. Een servlet is een klasse die kan worden geplaatst in een "servlet container". Deze container zorgt voor de afhandeling van het HTTP-protocol en op het moment dat een (te configureren) URL door een client wordt opgevraagd, wordt een standaard methode van de servlet aangeroepen. Hierbij worden twee objecten als argument meegegeven: een "request" en een "response" object. Deze objecten vertegenwoordigen respectievelijk de door de client verstuurde aanvraag en het door de servlet te genereren resultaat van deze aanvraag.

Aangezien via deze objecten een InputStream (om de letterlijke aanvraag uit te lezen) en een OutputStream (om het resultaat naar te kunnen schrijven) kunnen worden verkregen, kan nu de implementatie van de Web service in elkaar worden gezet. De benodigde XMLStreamReader en -Writer kunnen namelijk eenvoudig worden geconstrueerd aan de hand van een Input- of OutputStream.

Omdat de Web service algemeen is en de servlet die de requests afhandelt door de servlet container wordt geconstrueerd, is speciale functionaliteit nodig om de onderliggende RecordIF implementatie te creëren. Dit is opgelost door de servlet te definiëren als "abstract", met een abstracte methode "RecordIF createRecordIF(ServletConfig config)". Een concrete implementatie van de servlet (bijvoorbeeld een servlet die een interface naar een SQL-database biedt), kan deze methode vervolgens implementeren en aan de hand van de servlet configuratie de RecordIF implementatie construeren. Deze methode wordt vervolgens bij het initialiseren van de servlet aangeroepen en het opgeleverde RecordIF onthouden voor het afhandelen van verzoeken.

6.3.3. Algemene client

Ook de algemene client kan nu, met de complementerende componenten, in elkaar worden gezet. De enige moeilijkheid zit in het opzetten van de HTTP-verbinding. Hiervoor is een standaard component in Java beschikbaar, waarmee een verbinding kan worden opgezet en vervolgens een InputStream en OutputStream kan worden verkregen. De algemene client kan nu de RecordIF implementeren door bij een aanroep van getRecords de query als SOAP-bericht over de OutputStream te versturen en vervolgens het resultaat uit de InputStream te lezen. De geboden RecordIF zou nu gelijk moeten zijn aan de bij de Web service onderliggende RecordIF.

7. Validatie

Om het systeem te valideren, zal in dit hoofdstuk worden besproken hoe het kan worden ingezet bij de verschillende cases. Daarbij wordt beschreven in hoeverre het probleem voor de betreffende case wordt opgelost en welke onderdelen daartoe nog handmatig zullen moeten worden geïmplementeerd.

7.1. Centrale titelcatalogus

Zoals eerder besproken is de centrale titelcatalogus van het type waarbij zoekoperaties plaatsvinden en de prestatieproblemen dus zullen moeten worden opgelost met een replicatiesysteem.

De titelcatalogus betreft een nog te ontwikkelen systeem. Om deze reden hoeft er geen adapter te worden ontwikkeld. Het systeem kan namelijk (afhankelijk van de noodzaak tot de ondersteuning van verwijderingen) volledig uit de eerder besproken standaard componenten worden opgebouwd.

7.1.1. Implementatie

Het betreft hier het tweede geval van de Web services uit §5.3.2, ofwel de uit algemene componenten opgebouwde Web service en client.

De Web service heeft een onderliggende RecordIF implementatie nodig, die als Web service zal worden aangeboden. Omdat ervoor is gekozen om gebruik te maken van een (My)SQL-database, kan hiervoor de algemene SQL-adapter worden ingezet.

7.1.2. Verwijderingen

Verwijderingen zijn een speciaal geval, zo ook in deze case. Omdat de individuele bibliotheeksystemen namelijk lokaal-wijzigende gegevens hebben die verwijzen naar gegevens uit de replica (zoals de aanwezigheid van exemplaren van titels), kan een record niet zomaar worden verwijderd uit de centrale catalogus. Hiervoor zou eerst moeten worden gecontroleerd of geen enkel systeem naar de betreffende titel verwijst. Het is daarom niet vanzelfsprekend dat verwijderingen überhaupt door de centrale catalogus zullen worden ondersteund.

Mocht dit toch gewenst zijn en hier een oplossing voor worden gevonden, zou er dus een speciale versie van de algemene RecordIF naar SQL-databases moeten worden geïmplementeerd. Deze kan gebruik maken van de algemene implementatie en slechts de functionaliteit voor het teruggeven van verwijderingen implementeren.

7.1.3. Replica initialisatie

Omdat het gaat om een behoorlijk grote database, is het initialiseren van de replica's een kwestie. Wegens de grootte, kunnen de resultaten van de daarmee gepaard gaande All-selecties namelijk niet zomaar via de internetverbinding van een lokale bibliotheek worden opgehaald.

Voor deze kwestie worden hieronder twee oplossingen besproken, met elk zijn eigen voor- en nadelen. Beide oplossingen gaan uit van het opslaan van de betreffende gegevens in bestanden, die vervolgens via een daarvoor geschikt medium naar de lokale systemen kunnen worden gebracht ("Adidas netwerk").

Het gebruik van de standaardprojectie op XML

De eerste mogelijkheid is de All-selecties te doen vanaf een locatie die een snellere verbinding heeft met de centrale catalogus, zoals vanaf het systeem waar de service op draait zelf. De resultaten hiervan (in het gestandaardiseerde XML-formaat uit §4.6.2) worden hierbij dan naar bestanden geschreven. Deze bestanden kunnen vervolgens op de gewenste manier worden gekopieerd naar een lokaal systeem, dat moet worden geïnitialiseerd.

Vervolgens zou bijvoorbeeld gebruik kunnen worden gemaakt van een aangepaste versie van de algemene Web service client, die in het geval van een All-selectie in plaats van een InputStream van een HTTP-verbinding, een InputStream uit het betreffende bestand aan de gebruikte parser levert. Dit is een erg eenvoudig te implementeren oplossing, die bovendien gebruik maakt van de standaard manier om resultaten op XML te projecteren. Het nadeel hiervan is dat de gegevens moeten worden

geïnterpreteerd door de Java programmatuur, die minder goede prestaties (throughput) levert dan de onderstaande oplossing. In hoeverre dit een probleem is, zal de praktijk moeten uitwijzen.

Het gebruik van de MySQL-specifieke projectie op databasebestanden

De tweede mogelijkheid is het kopiëren van bestanden met een MySQL-specifieke projectie van de *onderliggende* database. Hierbij kan worden gedacht aan het uitvoeren van het "mysqldump" commando, wat in staat is een kopie van alle gegevens uit een gehele database naar een bestand te schrijven. Dit bestand zou vervolgens moeten worden gekopieerd naar het lokale systeem en daar met behulp van het complementaire "mysqlimport" commando worden ingevoerd.

Omdat deze oplossing gespecialiseerd is en in een lager-niveau programmeertaal is geïmplementeerd, is deze oplossing een stuk sneller.

Er ontstaat hierbij wel een nieuwe kwestie, namelijk het verkrijgen van de timestamp voor de volgende ModifiedSince-selectie in het replicatieproces. Omdat niet het resultaat van de All-selecties is opgeslagen (waar deze timestamp in is verwerkt), maar slechts de onderliggende database, is deze timestamp onbekend. Een oplossing hiervoor zou daarom in ieder geval speciale functionaliteit aan de ReplicaManager toevoegen om deze te verkrijgen. Een voorbeeld van dergelijke functionaliteit, is het verkrijgen van de timestamp door middel van het bepalen van de maximale waarde van alle timestamps van alle aanwezige records (zoals in §6.1.1). Een (theoretisch) nadeel hiervan is dat de timestamps niet meer abstract zijn voor de client, omdat deze hiervoor 1) aanwezig moeten zijn in de records en 2) moeten worden geïnterpreteerd (door de SQL-database).

7.1.4. Prestatiegevolgen

In deze paragraaf wordt een kwestie besproken die het gevolg is van de centralisatie van de titelgegevens in Bicat. Deze staat in principe los van het algemene deel van dit project, maar wordt toch behandeld, wegens de relevantie voor de opdrachtgever.

Een replica volgens het optimalisatiesysteem van dit project is gedefinieerd als een replica van de gehele database. Het is echter in het geval van Bicat niet altijd nodig om op alle systemen een gehele replica te hebben. Op dit moment is het namelijk zo dat elk systeem slechts de gegevens bevat van alle titels waarvan een exemplaar aanwezig is binnen de bibliotheek die door het betreffende systeem wordt bediend. De vraag is dus of het praktisch mogelijk is om in de situatie met een centrale catalogus, dit te veranderen en op alle systemen een replica van de gehele catalogus te hebben.

Als alle systemen dezelfde (complete) replica hebben, heeft dit een aantal voordelen. Ten eerste levert dit de mogelijkheid tot het lokaal zoeken in de complete catalogus, zodat gebruikers direct ook de catalogus van "alle" bibliotheken kunnen raadplegen. Ten tweede maakt dit het ontwerp eenvoudiger, doordat niet ten tijde van repliceren hoeft te worden uitgezocht of bepaalde gegevens lokaal-relevant zijn.

Het voornaamste nadeel is het potentiële prestatieverlies. Deze kwestie speelt vooral bij de kleinere bibliotheken, waarbij de hoeveelheid gegevens dan werkelijk een andere orde van grootte aanneemt. Een eis zou zijn dat de gehele catalogus op een systeem met de minimale systeemvereisten zou kunnen draaien en de zoekoperaties niet wezenlijk trager zouden worden.

Hiernaar is onderzoek gedaan en hieruit bleek dat dit met de juiste indices op de gegevens en juist geconstrueerde queries waarschijnlijk geen problemen zal opleveren. Hieronder wordt dit toegelicht aan de hand van de prestatietechnisch meest gevoelige functionaliteit.

Voorbeeld

Een belangrijk deel van de functionaliteit is het zoeken in de catalogus. Hiertoe kan de gebruiker een zoekterm opgeven, waarnaar dan wordt gezocht in alle mogelijke zoektermen. Vervolgens moet een lijst worden getoond van alle zoektermen die beginnen met de door de gebruiker opgegeven waarde, met voor elk van deze zoektermen het aantal titels waar die term betrekking op heeft. Als vervolgens een dergelijke zoekterm wordt geselecteerd, moet een lijst worden getoond van alle titels waar die zoekterm betrekking op heeft.

Om deze functionaliteit te implementeren bestaan er twee tabellen in de database: "titelszk", de zoektermen en "titels", de daadwerkelijke titels. Titelszk heeft een foreign key naar titels, die aangeeft dat een zoekterm betrekking heeft op de gekoppelde titel. Verder heeft titelszk een attribuut met de

zoekterm. In het geval dat er meerdere titels voorkomen waarop eenzelfde zoekterm betrekking heeft, bestaan er dus meerdere titelszk-records met dezelfde zoekterm.

Door nu een unique index te creëren op de titelszk tabel, bestaande uit de zoekterm en de foreign key naar titels, kan de eerste functionaliteit zeer efficiënt worden geïmplementeerd. Door de uniciteit van de index hoeft er dan namelijk geen join te worden uitgevoerd met de titels tabel. Dit komt, doordat het aantal opgeleverde regels waarvoor de zoekterm gelijk is, dan gelijk is aan het aantal verschillende titels. De volgende query zou deze functionaliteit leveren:

```
SELECT zoekterm, count(*)
FROM titelszk
WHERE zoekterm LIKE '<waarde waarnaar gebruiker zoekt>%'
GROUP BY zoekterm
```

Deze query wordt met een grote database (8000000 zoektermen) in verwaarloosbaar korte tijd uitgevoerd, doordat alleen gebruik wordt gemaakt van de index.

Om te zoeken naar titels die slechts in de betreffende bibliotheek beschikbaar zijn, kan een extra beperking aan het WHERE deel worden toegevoegd, zoals:

```
AND EXISTS (
      SELECT exemplaren.titel
      FROM exemplaren
      WHERE exemplaren.titel=titelszk.titel
)
```

Deze subquery verlaagt uiteraard de snelheid, maar niet op grote schaal, mits er ook een index aanwezig is op het titelattribuut van de exemplarentabel.

De tweede stap, gegeven een specifieke zoekterm een samenvatting weergeven van alle titels waar die zoekterm betrekking op heeft, is iets minder triviaal. De reden hiervoor is dat er een join moet worden gedaan met de titels tabel en de corresponderende titels records moeten worden opgehaald om hier een samenvatting van weer te kunnen geven. Omdat deze records van willekeurige locaties uit de database zullen worden verkregen, moet er een vrij grote hoeveelheid (potentieel niet-sequentiële!) diskblokken worden opgehaald. Dit levert bij grote hoeveelheden titels in het resultaat (orde van grootte duizend) vrij lange wachttijden op.

Nu is het niet noodzakelijk om de gebruiker in één keer alle resultaten te tonen. Dit kan in groepjes van enkele tientallen titels, die elk in korte tijd uit de database op te halen zijn. Een voor de hand liggende manier om dit te doen is de SQL “LIMIT” functionaliteit. Een query die deze functionaliteit zou leveren wordt dan:

```
SELECT t.titel, ....
FROM titels t, titelszk tz
WHERE tz.zoekterm='<gevraagde zoekterm>' AND tz.titel=t.id
LIMIT 1000,25
```

Hierbij wordt met het “LIMIT 1000,25” aangegeven dat moet worden gestart bij het 1000e resultaat en er vervolgens slechts 25 resultaten moeten worden opgeleverd.

Deze query levert echter niet de gewenste snelheid. De reden daarvoor is waarschijnlijk dat de LIMIT-functionaliteit van de (in dit geval gebruikte) databaseserver werkt op het uiteindelijke resultaat en dus de eerste 1000 resultaten alsnog worden opgehaald.

Een manier om dit te voorkomen is het opsplitsen van de functionaliteit in twee queries, namelijk het deel dat alle titel-id’s waarop de zoekterm betrekking heeft ophaalt en het deel dat bij een groepje van deze titels de samenvatting ophaalt. Hierbij kan worden gedacht aan een query als:

```
SELECT t.titel, ....
FROM titels t
WHERE t.id IN (
    SELECT tz.titel
    FROM titelszk tz
    WHERE tz.zoekterm="<gevraagde zoekterm>"
    LIMIT 1000,25
)
```

Deze combinatie van LIMIT in een subquery waar gebruik wordt gemaakt van “IN”, wordt echter niet door de gebruikte databaseserver (MySQL) ondersteund.

Een andere oplossing is deze queries in de gebruikende software los van elkaar uit te voeren. Hierbij zouden dan dus eerst alle titel-id’s worden opgehaald die aan de gevraagde zoekterm voldoen en vervolgens 25 losse queries die de titelgegevens ophalen. Dit biedt voldoende hoge prestaties.

7.1.5. Overige kwesties

Hieronder worden enkele overige kwesties besproken die buiten het bestek van dit project vallen.

Wijzigingen

Wijzigingen aan de titelcatalogus gaan buiten het systeem uit dit project om. Het replicatieproces detecteert wijzigingen slechts doordat timestamps worden verhoogd.

Een belangrijk probleem dat bij wijzigingen optreedt, ontstaat als er meerdere gebruikers tegelijk dezelfde gegevens willen wijzigen. Allereerst zullen zij de huidige gegevens ophalen om daar wijzigingen op aan te kunnen brengen. Zodra ze vervolgens beiden de gegevens opslaan, zullen de wijzigingen van degene die dit als eerste heeft gedaan, verloren gaan. Een mogelijke oplossing zou zijn, de timestamp van het in de database aanwezige record te vergelijken met de timestamp van het gewijzigde record, zoals deze door de gebruiker wordt aangeboden. Deze zouden gelijk moeten zijn, anders zijn de gegevens in de tussentijd gewijzigd door een andere gebruiker.

Het genereren van timestamps

Omdat dit ook geldt voor wijzigingen, valt ook het genereren van nieuwe timestamps bij wijzigingen buiten het bestek van dit project. Hiervoor is eerder al een aantal implementatie-hints gegeven (zie §4.3.3).

7.2. Adresgegevens uit gemeentelijk systeem

Het verkrijgen van adresgegevens uit een gemeentelijk systeem zal waarschijnlijk van de categorie zijn, waarbij niet kan worden aangenomen dat volledige replicatiefunctionaliteit wordt geboden. Het zal dan dus slechts mogelijk zijn om caching functionaliteit te bieden.

Hiertoe zou een adapter moeten worden ontwikkeld, die de interface van het gemeentelijke systeem omzet in een RecordIF. Vervolgens zou het gebruikende systeem de adresgegevens moeten benaderen via de algemene CachedRecordIF. Daarbij zouden de id’s van de betreffende records in het systeem bekend moeten zijn.

Een mogelijkheid waar hierbij aan kan worden gedacht, is het via ID-selecties bijhouden van een niet-efficiënte “deelreplica”. Op het moment dat de ID-selectie wordt gedaan bij iedere toegevoegde klant, zal een replica ontstaan met alle aanwezige adres records. Deze zal vervolgens echter niet automatisch actueel worden gehouden. Dit kan gebeuren door automatisch aan de hand van de ouderdom van de records in de cache steeds de betreffende ID-selectie opnieuw uit te voeren, zodat deze gegevens worden bijgewerkt en de ouderdom zo in de hand wordt gehouden. Verder kan worden gedacht aan het handmatig verversen van de records, bijvoorbeeld als een klant aan de balie zijn adresgegevens zegt te willen wijzigen.

In de praktijk zal er ongetwijfeld meer komen kijken bij de implementatie van deze case. Zo zal het wellicht wenselijk zijn functionaliteit te bieden waarmee een extra adres naast het woonadres van een klant kan worden opgegeven. Verder kan het vaststellen van het “id” waarmee de adresgegevens kunnen worden opgehaald (het sofinummer zou hiervoor een kandidaat zijn) een probleem zijn.

7.3. Adresgegevens aan de hand van postcodes

Bij het ophalen van de exacte straat- en plaatsnaam van een adres, worden vaak postcodetabellen gebruikt. Deze case is in feite vrijwel gelijk aan de vorige, in die zin dat het erg waarschijnlijk is dat er geen replicatiefunctionaliteit zal worden geboden.

Het betreft hier weer een bestaand systeem, deze keer in de vorm van een Web service. Hiervoor kan vrij eenvoudig een adapter worden ontwikkeld, waarna de caching functionaliteit direct kan worden gecreëerd met behulp van de algemene onderdelen. Hoe dit in zijn werk gaat is reeds besproken in hoofdstuk 5.

7.4. Salesforce

De laatste case betreft Salesforce, de online CRM-oplossing. Deze biedt een vrij geavanceerde interface naar de onderliggende gegevens, in de vorm van een Web service. Voor dit systeem is een RecordIF-adapter geïmplementeerd die volledige replica functionaliteit biedt, inclusief ondersteuning voor verwijderingen. Op deze manier kan zonder extra programmeerwerk een replicatiesysteem voor Salesforce in elkaar worden gezet.

7.4.1. Implementatie adapter

De gegevens die worden ontsloten zijn in de vorm van records, met attributen en een id-attribuut. Deze zijn dus direct te projecteren op RecordIF records. Verder wordt één van deze attributen steeds gewijzigd op het moment dat er een wijziging op dat record wordt doorgevoerd. Omdat verder records kunnen worden geselecteerd met SQL-achtige functionaliteit, kan zo de ModifiedSince-selectie worden geïmplementeerd.

Om een TimestampingRecordIFResult terug te kunnen geven is het noodzakelijk te bepalen tot welke timestamp de teruggegeven resultaten actueel zijn. Dit wordt gedaan door gebruik te maken van speciale functionaliteit van de Web service om de huidige tijd van het systeem te verkrijgen. Omdat de timestamps eveneens in de vorm zijn van tijdstippen, is dit probleem op deze manier opgelost.

Ook wordt functionaliteit geboden om de verwijderingen uit een bepaalde periode op te halen. Deze worden apart opgehaald na het ophalen van de wijzigingen.

Op deze manier moet er voor elke ModifiedSince-selectie drie keer met de Web service worden gecommuniceerd. Dit is geen groot probleem, omdat dan wel in één keer alle wijzigingen worden opgehaald. Als de maximale toelaatbare ouderdom van de gegevens bijvoorbeeld vijftien minuten bedraagt, hoeft slechts iedere vijftien minuten drie keer met de Web service te worden gecommuniceerd. Dit is zeer haalbaar en levert geen capaciteitsproblemen.

7.4.2. Prestatieverbetering

Om aan te geven dat het systeem inderdaad een prestatieverbetering realiseert, wordt in deze paragraaf besproken wat de werkelijke prestatieverbetering is.

Bij directe benadering van de Salesforce Web service, duurt het ophalen van een enkel record per query een halve tot een hele seconde. Hierbij wordt aangenomen dat reeds is ingelogd op de Web service en het gebruikende systeem in staat is nieuwe queries uit te voeren zonder opnieuw te hoeven inloggen. Inloggen kost namelijk ook nog enkele seconden. Deze snelheid betekent dat het onwenselijk is de Web service direct aan te roepen bij interactieve applicaties. De gebruiker zal dan immers elke keer dat gegevens uit Salesforce moeten worden getoond tot één seconde moeten wachten. Als er meerdere queries of grotere hoeveelheden gegevens moeten worden opgehaald, zal deze tijd verder oplopen en de responstijd van de applicatie voor de meeste situaties ontoelaatbare waarden aannemen. Daarbij komt dat, omdat de Web service niet binnen hetzelfde beheer als dat van de applicatie ligt, de prestaties kunnen variëren, wat eveneens tot irritatie van gebruikers zou leiden.

Indien de Salesforce-gegevens zijn gerepliceerd naar een lokale database, zullen de prestaties fors toenemen. Doordat deze database lokaal is en slechts hoeft te worden benaderd door de applicatie zelf, levert dit een aanzienlijke prestatiewinst. Bij een lokale database zal de responstijd eerder in de orde van grootte van milliseconden liggen.

7.5. Algemene evaluatie

Omdat de beschreven oplossing algemeen is, kan er een algemene evaluatie worden gegeven. Deze is onderverdeeld in twee delen, die hieronder apart worden beschreven.

7.5.1. Afscherming prestatiegevolgen

De mate waarin de prestatiegevolgen met het ontworpen systeem kunnen worden afgeschermd, hangt af van de door het gekoppelde systeem geboden functionaliteit.

Replicatie

Op het moment dat het gekoppelde systeem functionaliteit biedt om efficiënte replicatie mogelijk te maken, kunnen de lees-prestatiegevolgen volledig worden afgeschermd. Voorbeelden hiervan zijn de cases uit §7.1 en §7.4. Hiermee wordt bedoeld dat de prestatie niet meer mede wordt bepaald door het gekoppelde systeem, maar door de gebruikte replicadatabase. Op deze manier komen de prestatiegevolgen weer binnen de verantwoordelijkheid te liggen van de ontwerper van het systeem zelf.

Hierbij moet worden opgemerkt dat de situatie dient te voldoen aan de aannames uit hoofdstuk 2, waarbij met betrekking tot de prestaties vooral moet worden gedacht aan de aanname dat het geen probleem is als het systeem werkt met (beperkt) verouderde gegevens.

Caching

Op het moment dat het gekoppelde systeem geen replicatiefunctionaliteit biedt, kunnen de prestatiegevolgen slechts gedeeltelijk worden afgeschermd. Het gaat hierbij om cache-functionaliteit, zoals bij de cases uit §7.2 en §7.3. Hierbij worden de prestatiegevolgen dus beperkt tot de eerste keer dat elk record wordt opgehaald, of de maximale levensduur van het betreffende record in de cache is verstreken.

Om deze laatste prestatiegevolgen te voorkomen, kan een strategie worden toegepast waarbij alle records die nodig kunnen zijn (bijvoorbeeld alle adresgegevens van de personen uit de lokale database), bij voorbaat worden opgehaald en dus altijd in de cache-database aanwezig zullen zijn op het moment dat ze worden gelezen. Daarnaast kan ervoor worden gekozen om deze records periodiek te "verversen" om de actualiteit van de gegevens te waarborgen.

Zo ontstaat een deelreplica, die op minder efficiënte wijze wordt bijgehouden dan de hiervoor besproken volledige replica. Of dit efficiënt genoeg is, zal per geval moeten blijken en hangt onder meer af van de schrijf-frequentie van de gegevens en de hoeveelheid gebruikte records.

Voorbeelden van niet-opgeloste gevallen

Aangezien de situatie moet voldoen aan de eisen uit hoofdstuk 2, zullen niet in alle willekeurige situaties de prestatiegevolgen kunnen worden afgeschermd met de oplossing uit dit project. Enkele voorbeelden hiervan zijn:

- De situatie waarin het gebruikende systeem altijd de meest actuele gegevens dient te worden geleverd. Denk bijvoorbeeld aan een reserveringssysteem, waarbij tickets kunnen worden geboekt.
- De situatie waarbij gebruik wordt gemaakt van een Web service waarmee zoekoperaties worden uitgevoerd, zonder replicatiefunctionaliteit te bieden.
- Een te koppelen systeem, dat niet-identificeerbare gegevens levert.

7.5.2. Hoeveelheid applicatiespecifiek programmeerwerk

Door de standaardisatie van de interface naar de te koppelen systemen, is de mogelijkheid gecreëerd algemene componenten te introduceren, zoals gedaan in §5.2. Dankzij deze componenten ontstaat de mogelijkheid om grote delen van het optimalisatiesysteem vrijwel zonder programmeerwerk op te bouwen.

Indien het te koppelen systeem direct conformeert aan de standaard (doordat het direct is te benaderen of gebruik maakt van de standaard Web service interface), zal het systeem zonder programmeerwerk kunnen worden aangekoppeld. Ook een Web service zelf kan volledig uit algemene componenten worden opgebouwd, mits er een RecordIF interface bestaat naar de te ontsluiten gegevens.

Als een systeem of de onderliggende gegevenslaag geen RecordIF interface bieden, kan een adapter worden gecreëerd om dit alsnog te bereiken. Op die manier kunnen de gewenste systemen (Web service, optimalisatiesysteem) alsnog met algemene componenten worden opgebouwd. Zo wordt de hoeveelheid applicatie specifiek programmeerwerk beperkt tot het creëren van een adapter.

8. Conclusie

De eerste conclusie van dit project is dat ook hier weer de kracht van standaardisatie en eenvoud naar voren komt. Standaardisatie zorgt ervoor dat het mogelijk is om algemene componenten te creëren en deze in te zetten op plekken waar anders maatwerksoftware zou moeten worden geïmplementeerd. Dit betekent een behoorlijke efficiëntiewinst bij het implementeren van nieuwe systemen, in dit specifieke geval bij het koppelen met andere systemen en het creëren van Web services uit onderliggende gegevensontsluiters. Ook het begrip "eenvoud" speelt een belangrijke rol bij het definiëren van de interface (en andere zaken). Door te streven naar een zo eenvoudig mogelijke oplossing, verzekert men zich ervan het probleem terug te brengen tot de kern. Hierdoor wordt een krachtige, efficiënte en betrouwbare oplossing gecreëerd. Hierbij kan worden verwezen naar Dijkstra (bijvoorbeeld [EWD1304]), die een belangrijke voorstander was van eenvoud en "elegantie".

Een tweede belangrijke conclusie is dat bij het standaardiseren van Web services wellicht een verdere standaardisatie nodig is voor bepaalde, specifieke vormen van deze services. Omdat Web services zo algemeen zijn, is het bijzonder ingewikkeld om hier bepaalde algemene zaken voor te ontwikkelen. Een voorbeeld hiervan is dit project, waarin werd beoogd een algemene oplossing te creëren voor de kwestie van prestatiegevolgen als gevolg van de koppeling met Web services. Het is waarschijnlijk onmogelijk dit te doen voor Web services in het algemeen, vandaar dat eerst een pakket van aannames is opgesteld. Daarom zou kunnen worden geconcludeerd dat het wenselijk is om bovenop het concept van Web services, een extra "laag" van gestandaardiseerde soorten Web services te definiëren, zoals de gegevensontsluitende Web services uit dit project. Een illustrerende analogie die hierbij kan worden gelegd is die met de lagenstructuur van andere netwerkprotocollen: zoals het lastig zou zijn zonder de standaardisatie van TCP gebruik te maken van IP (iedereen zou dan immers "eigen", incompatibele transportprotocollen moeten definiëren), is het ongunstig slechts de huidige Web services te hebben gestandaardiseerd. Er zijn meer, specifiekere standaarden nodig die voortbouwen op Web services.

8.1. Aanbevelingen

In dit project is door middel van een afbakening onderzoek gedaan naar een deel van de situaties waarbij de prestatiegevolgen een rol spelen. Een aanbeveling voor vervolgonderzoek zou zijn om het gehele probleem op te delen in meer van dergelijke deelgebieden, om zo een totaalbeeld te krijgen van de prestatiegevolgen en mogelijke oplossingen daarvoor. Een voorbeeld hiervan is de optimalisatie van schrijfoperaties. Omdat hierbij allerlei andere kwesties een rol spelen (zoals consistentie en synchronisatie), is dit op zijn minst een even interessant deelprobleem.

Een voorbeeld van een oplossing voor de afscherming van de prestatiegevolgen van schrijfoperaties is een lokale buffer. Daarin zouden schrijfacties dan tijdelijk kunnen worden geplaatst, voordat ze worden uitgevoerd. Vervolgens worden ze dan "in de achtergrond" en op een later moment daadwerkelijk uitgevoerd. Op deze manier kan het gebruikende systeem direct na het bufferen van een schrijfoperatie verder gaan met uitvoeren. Uiteraard spelen hierbij legio andere kwesties, zoals de situatie waarin de daadwerkelijke schrijfoperatie mislukt, terwijl het programma al verder is gegaan met uitvoeren.

9. Bronnen

Alls bronnen waarbij naar websites wordt verwezen zijn voor het laatst geraadpleegd op 1 augustus 2007.

- ALO2004: Gustavo Alonso, Fabio Casati, Harumi Kuno, Vijay Machiraju (2004) – Web Services: concepts, architectures and applications - Springer
- BAL2005: Naveen Balani (IBM) – Model and build ESB SOA frameworks - http://www-128.ibm.com/developerworks/web/library/wa-soaesb/
- CHI2002: Kenneth Chiu, Madhusudhan Govindaraju, Randall Bramley (2002) - Investigating the Limits of SOAP Performance for Scientific Computing
- CORBA: Catalog of OMG CORBA / IIOP Specifications - http://www.omg.org/technology/documents/corba_spec_catalog.htm
- DEV2003: Kiran Devaram and Daniel Andresen (2003) - SOAP Optimization via Client-side Caching
- ELB2004: Kamal Elbashir (2004) - Transparent Caching of Web Services for Mobile Devices
- EWD1304: Edsger W. Dijkstra - The end of Computing Science? (2000) - http://www.cs.utexas.edu/users/EWD/ewd13xx/EWD1304.PDF
- FOW2000: Martin Fowler (2000) - POJO - http://www.martinfowler.com/bliki/POJO.html
- GOF1994: Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides (1994) - Design Patterns: Elements of Reusable Object-Oriented Software - Addison-Wesley
- GRU2006: John Grundy, John Hosking, Lei Li, Na Liu (2006) - Performance engineering of service compositions
- JDBC: Sun Microsystems - Java Database Connectivity (JDBC) - http://java.sun.com/javase/technologies/database/
- JIN2004: Jingwen Jin and Klara Nahrstedt (2004) - On Exploring Performance Optimizations in Web Service Composition
- JUN2006: Wei Jun, Hua Lei, Niu Chunlei (2006) - Speed-up SOAP Processing by Data Mapping Template
- JUR2004: Matjaz B. Juric, Bostjan Kezmah, Marjan Hericko, Ivan Rozman, Ivan Vezocnik (2004) - Java RMI, RMI Tunneling and Web Services Comparison and Performance Analysis
- JUS2003: Kai S. Juse, S. Kounev, A. Buchmann (2003) - Petstore-WS: Measuring the performance implications of Web services
- RAM2004: Venugopalan Ramasubramanian, Douglas B. Terry (2004) - Caching of XML Web Services for Disconnected Operation
- REFL: Sun Microsystems – The Reflection API (tutorial) - http://java.sun.com/docs/books/tutorial/reflect/
- SERV: Sun Microsystems – Java Servlet Technology - http://java.sun.com/products/servlet/
- SF: Salesforce – http://www.salesforce.com
- STAX: Streaming API for XML - http://stax.codehaus.org/
- TAK2002: Toshiro Takase, Yuichi Nakamura, Ryo Neyama, Hiroaki Eto (2002) – A Web Services Cache Architecture Based on XML Canonicalization – IBM Research
- TAN2002: Andrew S. Tanenbaum, Maarten van Steen (2002) – Distributed Systems – Prentice Hall
- TIA2003: M. Tian, A. Gramm, T. Naumowicz, H. Ritter, J. Schiller (2003) - A Concept for QoS Integration in Web Services

- VOB2006: Vereniging van Openbare Bibliotheken (2006) – Een informatiearchitectuur voor Openbare bibliotheken
- W3C2000: W3C (2000) - Simple Object Access Protocol - http://www.w3.org/TR/soap/
- W3C2001: W3C (2001) - Web Services Description Language (WSDL) - http://www.w3.org/TR/wsdl
- W3C2004: World Wide Web Consortium (2004) - Web Services Glossary - http://www.w3.org/TR/2004/NOTE-ws-gloss-20040211

10. Bijlagen

10.1. XML-schema voor queries

```
<schema targetNamespace="http://pimsierhuis.nl/unirec/"
        xmlns="http://www.w3.org/2001/XMLSchema">
  <element name="recordIFQuery">
    <complexType>
      <sequence>
        <element name="recType" type="string" />
        <element name="selection">
          <complexType>
            <sequence>
              <element name="general" type="string" />
              <element name="type" type="string" />
              <element name="arg" type="string" minOccurs="0"
                       maxOccurs="unbounded" />
            </sequence>
          </complexType>
        </element>
      </sequence>
    </complexType>
  </element>
</schema>
```

10.2. XML-schema voor resultaten

```
<schema targetNamespace="http://pimsierhuis.nl/unirec/"
        xmlns="http://www.w3.org/2001/XMLSchema">
  <element name="recordIFResult">
    <complexType>
      <sequence>

        <element name="header">
          <complexType>
            <sequence>
              <element name="recType" type="string" />
              <element name="idAttr" type="string" />
              <element name="recAttrs">
                <complexType>
                  <sequence>
                    <element name="n" type="string"
                             maxOccurs="unbounded" />
                  </sequence>
                </complexType>
              </element>
            </sequence>
          </complexType>
        </element>

        <element name="recs">
          <complexType>
            <sequence>
              <element name="rec" minOccurs="0" maxOccurs="unbounded" >
                <complexType>
                  <sequence>
                    <element name="a" type="string"
                             maxOccurs="unbounded" />
                  </sequence>
                </complexType>
              </element>
            </sequence>
          </complexType>
        </element>

        <element name="footer">
          <complexType>
            <sequence>
              <element name="uptodateUntilTimestamp" type="string" />
              <element name="deleted">
                <complexType>
                  <sequence>
                    <element name="id" type="string" minOccurs="0"
                             maxOccurs="unbounded" />
                  </sequence>
                </complexType>
              </element>
            </sequence>
          </complexType>
        </element>

      </sequence>
    </complexType>
  </element>
</schema>
```

10.3. WSDL-definitie van algemene Web service

```
<wsdl:definitions
    xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
    xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
    xmlns:http="http://schemas.xmlsoap.org/wsdl/http/"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    xmlns:unirec="http://pimsierhuis.nl/unirec/"
    targetNamespace="http://pimsierhuis.nl/unirec/">

    <wsdl:types>
        <xsd:schema targetNamespace="http://pimsierhuis.nl/unirec/">
            <xsd:import namespace="http://pimsierhuis.nl/unirec/"
                        schemaLocation="types.xsd"/>
        </xsd:schema>
    </wsdl:types>

    <wsdl:message name="getRecordsRequest">
        <wsdl:part name="request" element="unirec:recordIFQuery"/>
    </wsdl:message>

    <wsdl:message name="getRecordsResponse">
        <wsdl:part name="response" element="unirec:recordIFResult"/>
    </wsdl:message>

    <wsdl:portType name="unirecPortType">
        <wsdl:operation name="getRecords">
            <wsdl:input message="unirec:getRecordsRequest" />
            <wsdl:output message="unirec:getRecordsResponse" />
        </wsdl:operation>
    </wsdl:portType>

    <wsdl:binding name="unirecBinding" type="unirec:unirecPortType">
        <soap:binding style="rpc"
         transport="http://schemas.xmlsoap.org/soap/http"/>
        <wsdl:operation name="getRecords">
            <soap:operation style="rpc" soapAction=""/>
            <wsdl:input>
                <soap:body use="literal"/>
            </wsdl:input>
            <wsdl:output>
                <soap:body use="literal"/>
            </wsdl:output>
        </wsdl:operation>
    </wsdl:binding>


</wsdl:definitions>
```