Inhoudsopgave

Inleiding

Binnen bedrijven bevinden zich zeer veel applicaties. Vaak ontstaat nadien de behoefte om deze, afzonderlijk ontwikkelde, applicaties te laten samenwerken. Dit is echter een groot probleem aangezien de meeste van die applicaties onderling incompatibel zijn. Als reactie op deze problemen zijn in het verleden middleware systemen en Enterprise Application Integration systemen ontwikkeld. Deze twee systemen zorgen voor de integratie van verschillende heterogene applicaties en worden besproken in hoofdstuk 1.

De nieuwe eisen die nadien aan de applicaties gesteld werden, vooral communicatie over verschillende netwerken en tussen verschillende bedrijven konden door deze systemen echter niet ingewilligd worden. Web Services profileren zich als "de oplossing" voor al deze problemen. In hoofdstuk 2 behandelen we daarom de basisprincipes van web services bestaande uit een Service Oriented Architecture, standaarden en een herdefinitie van bestaande middleware protocollen.

In de hoofdstukken 3, 4 en 5 worden de drie belangrijkste standaarden, SOAP, WSDL en UDDI, nader toegelicht.

Hoofdstukken 6 en 7 geven meer uitleg omtrent de herdefinitie van bestaande protocollen.

De wereld van web service ontwikkeling is verdeeld in twee kampen: het java-gebaseerde J2EE-platform en het Microsoft .NET framework. In hoofdstuk 8 wordt een zeer eenvoudige web service met behulp van Apache Axis geïmplementeerd. Apache Axis heeft gekozen voor de java aanpak maar een C++ versie is momenteel in ontwikkeling.

Hoofdstuk 9 tot slot bevat de conclusie.

Hoofdstuk 1

Ontstaansgeschiedenis van web services

1.1 Evolutie van de architectuur van informatiesystemen

Een informatiesysteem kan qua functionaliteit in 3 layers opgedeeld worden:

1. Presentation layer
2. Application logic layer
3. Resource management layer

De presentation layer zorgt voor de communicatie met externe entiteiten. Deze entiteiten kunnen vertolkt worden door mensen maar ook door andere computers. De presentation layer zorgt enerzijds voor het presenteren van informatie en anderzijds voor de communicatie tussen externe entiteiten en het informatiesysteem.

De application logic layer bevindt zich tussen de presentation layer en de resource management layer. Opdrachten afkomstig van de presentation layer worden hier verwerkt en indien nodig wordt hiervoor data vanuit de resource management layer gebruikt. Het is in deze layer dat zich de zogenaamde services van het informatiesysteem bevinden vandaar ook dat deze layer soms ook wel de server genoemd wordt.

De Resource management layer bevat de concrete data en stelt deze beschikbaar aan de hoger gelegen lagen en in de eerste plaats aan de application

logic layer. Standaard componenten die behoren tot deze layer zijn onder andere database management systemen en andere externe systemen die informatie kunnen leveren zoals bijvoorbeeld andere informatiesystemen. Op deze manier kunnen informatiesystemen recursief opgebouwd worden.

1.1.1 One-tier

In de eerste informatiesystemen werden deze drie layers geïntegreerd in één mainframe machine. Beperkte interactie was mogelijk via een vast aantal terminals. Dergelijke systemen worden one-tier genoemd. De voordelen van deze configuratie zijn dat alle componenten zich op één centraal punt bevinden en dat de layers door hetzelfde team en in dezelfde taal ontwikkeld kunnen worden. Hierdoor is communicatie tussen de verschillende layers uiterst snel en efficiënt. Een nadeel is natuurlijk het gebrek aan flexibiliteit en de hoge kostprijs van dergelijke systemen.

1.1.2 Two-tier

Om aan de wens naar meer flexibiliteit te voldoen diende een opsplitsing van de drie basislayers zich aan. Door de komst van Local Area Networks (LANs) en de groei van de PC- en workstation-markt bleek dit nu ook praktisch haalbaar. Dit heeft geleid tot het onstaan van de zogenaamde two-tier systemen.

Two-tier systemen splitsen de layers in twee delen: het eerste deel bestaat uit de presentation layer en het tweede deel uit een combinatie van de application logic layer en de resource management layer waarbij de presentation layer zich op een aparte “client” PC bevindt. Hier zijn twee voordelen aan verbonden: ten eerste kan je voor de presentation layer gebruik maken van extra kracht geleverd door de PC en ten tweede is het nu mogelijk om de presentation layer aan te passen en verschillende versies aan te bieden zonder dat hierdoor extra complexiteit in de andere twee layers geïntroduceerd wordt.

Two-tier systemen zijn zeer populair als client/server architectuur waarbij de presentation layer dienst doet als client en de andere twee layers de rol van server op zich nemen. Daarnaast werden verschillende belangrijke concepten ontwikkeld rond two-tier systemen zoals bijvoorbeeld Remote Procedure Calls (RPC’s) en Application Programmer Interfaces (APIs).

Toen men het potentieel van deze clients begon in te zien en er bijgevolg steeds meer programmatuur in de presentation layer geïntegreerd werd leidde dit tot steeds complexere, moeilijk te hanteren clients en een verdere opsplitsing bleek noodzakelijk.

1.1.3 Three-tier

Dit heeft geleid tot de geboorte van three-tier systemen. De eerste tier bevat de presentation layer en is net zoals bij two-tier systemen terug te vinden aan de client zijde. De tweede tier bestaat deels uit middleware en deels uit application logic (application logic layer). De middleware zorgt ervoor dat applicaties uit de resource management layer, die overeenkomt met de derde tier, met elkaar kunnen samenwerken. De application logic maakt gebruik van de middleware om nieuwe functionaliteiten, opgebouwd uit bestaande applicaties, aan te bieden. In de resource management layer kunnen zich zowel one-tier, two-tier als three-tier systemen bevinden.

Three-tier systemen worden vooral gebruikt als integratie platform en hebben geleid tot belangrijke concepten zoals o.a. standaard interfaces voor resource managers. Voorbeelden hiervan zijn Open Database Connectivity (ODBC) en Java Database Connectivity (JDBC).

De voordelen van three-tier systemen zijn in de eerste plaats een grotere flexibiliteit, een betere scalability (application layer kan uit een set van geclusterde computers bestaan) en de mogelijkheid om applicatie software te schrijven die minder gebonden is aan de onderliggende resource management layer, dankzij gestandaardiseerde APIs. Dit laatste komt overeen met een betere overdraagbaarheid naar andere platformen en een betere ondersteuning voor hergebruik.

Het nadeel t.o.v. de two-tier systemen is een toename in complexiteit wat betreft de communicatie tussen de application logic layer en de resource management layer wat tevens een mindere efficiëntie impliceert. Een bijkomend nadeel is het feit dat middleware systemen zich omwille van hun interne architectuur niet lenen tot gebruik over internet. (zie hoofdstuk 2)

1.1.4 N-tier

Tot slot bestaan er ook N-tier systemen. De N-tier architectuur is een veralgemening van de three-tier architectuur. We spreken over N-tier systemen in de volgende situaties:

- Een three-tier systeem wordt N-tier genoemd als in de resource management layer volledige two-tier of three-tier systemen opgenomen zijn.
- Een three-tier systeem wordt N-tier genoemd als er in de presentation layer een soort van web server voorzien is om het systeem via internet toegankelijk te maken.

N-tier systemen kunnen dus wel over Internet communiceren maar dit gebeurt dan via omslachtige en niet gestandaardiseerde methoden en talen.

Over het algemeen kan gesteld worden dat hoe meer layers er gebruikt worden hoe groter de flexibiliteit wordt. De keerzijde van de medaille is dan weer dat dit ook extra complexiteit oplevert alsook een inboeting op het vlak van de efficiëntie.

Wat uit deze evolutie ook duidelijk blijkt is het steeds toenemende belang van integratie van heterogene componenten en het afschermen van de interne werking door het definiëren van interfaces.

1.1.5 Web services

Web services vormen de volgende stap in deze evolutie. Web services zijn een antwoord voor problemen die niet opgelost kunnen worden met three-tier en n-tier architecturen. Ze zorgen ervoor dat systemen met elkaar communiceren en interageren over zowel LANs als het Internet, en dit op een gestandaardiseerde manier.

1.2 Middleware

Middleware vereenvoudigt en beheert de interacties tussen applicaties over heterogene platformen. Het is een oplossing voor het integratieprobleem van een collectie van servers en applicaties onder een gemeenschappelijke service interface.

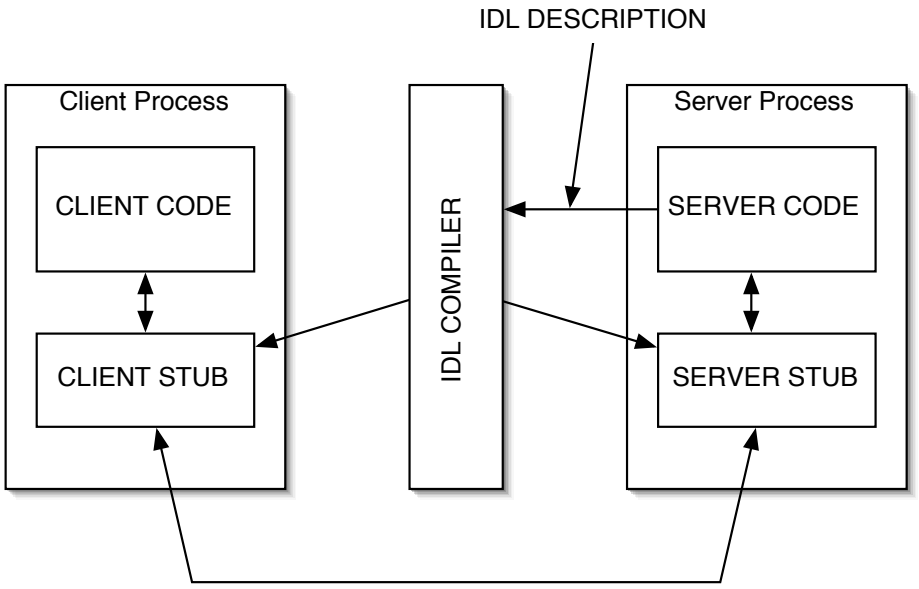
Middleware is een abstractie waarmee de complexiteit van het ontwikkelen van gedistribueerde applicaties ten dele wordt afgeschermd. Verder worden ook transacties en andere functionaliteiten aangeboden. Dit gebeurt allemaal in de middleware zonder dat de gebruiker zich daar iets van moet aantrekken.

Er bestaan verschillende vormen en soorten van middleware: RPC-gebaseerde systemen, TP-Monitors, Object Brokers, Object monitors, Message-oriented middleware en Message Brokers.

1.2.1 RPC

RPC is de meest algemene vorm van middleware die gebruikt wordt als basis voor verschillende soorten middleware. Het voorziet de technische structuur die nodig is om gewone procedure oproepen om te zetten naar remote procedure oproepen. Het doel van RPC is de complexiteit van remote oproepen af te schermen door te doen alsof het om een locale oproep gaat. Zowel statische als dynamische binding is mogelijk. Dit laatste gebeurt d.m.v. een naam- en directory-service.

De algemene werkwijze voor statische binding kan beschreven worden aan de hand van figuur 1.1:



Figuur 1.1: RPC gebaseerde systemen

Om een procedure geschikt te maken voor remote oproepen moet er eerst een interface voor gedefinieerd worden. Dit gebeurt met de Interface Definition Language (IDL). Aan de hand van deze beschrijving worden met de IDL compiler twee stubs gecreëerd, de client stub en de server stub. Een stub is een stuk code dat meegecompileerd moet worden hetzij met de client code hetzij

met de server code. Als de client de betreffende service wil oproepen gebeurt dit d.m.v. een call naar een locale procedure aanwezig in de client stub. De client stub localiseert de server en brengt de data in het juiste formaat alvorens de oproep verder te verzenden. Nadien krijgt de client stub het resultaat binnen en geeft dit na omzetting terug door aan de client. De werking van de server stub is analoog. Een gelijkaardig principe zal later ook gebruikt worden voor het oproepen van web services aan de hand van WSDL documenten.

In het geval van dynamische binding komt er een extra component in de vorm van een naam- en directory-service bij in de bovenstaande figuur. Deze service zorgt voor een loskoppeling van de client en de server waardoor clients at runtime op zoek kunnen gaan naar bepaalde services. Extra flexibiliteit kan dan bekomen worden door de name en directory service te voorzien van bijkomende functionaliteiten zoals load balancing en dergelijken. Als een client de geschikte service gevonden heeft verloopt de rest van de communicatie op analoge wijze als bij de statische binding.

1.2.2 TP Monitors

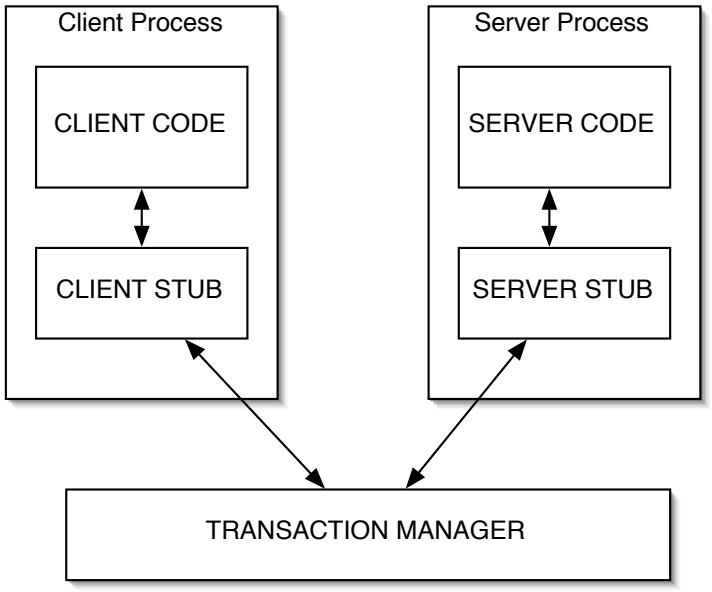
TP monitor staat voor Transaction Processing monitor. Het doel van TP monitors is het ondersteunen van remote transacties. TP monitors implementeren de zogenaamde Transactional RPC's (TRPC) hetgeen een combinatie is van: een set van RPC's, een transaction manager en een Two-Phase-Commit Protocol (2PC). [28, 25]

De algemene werkwijze van TP Monitors wordt duidelijk gemaakt aan de hand van figuur 1.2:

De client definieert een groep van RPC's als een transactie door ze te groeperen binnen een "Begin of Transaction" (BOT) en een "End of Transaction" (EOT). De client begint zijn transactie door het versturen van een BOT naar de client stub. Deze merkt dat het om een transactie gaat en stuurt dit door naar de transaction manager waarop deze als antwoord een transactioncontext terugstuurt. Deze context wordt vanaf nu, totdat we EOT tegenkomen, opgenomen in elke uitgaande call. Voor elke RPC die de client bij een nieuwe server uitvoert worden de volgende stappen gezet:

1. De server stub merkt dat het om een transactie gaat dankzij de transaction context en meldt zich bij de transaction manager aan.

2. De server stub geeft de oproep door aan de server.



Figuur 1.2: TP Monitors

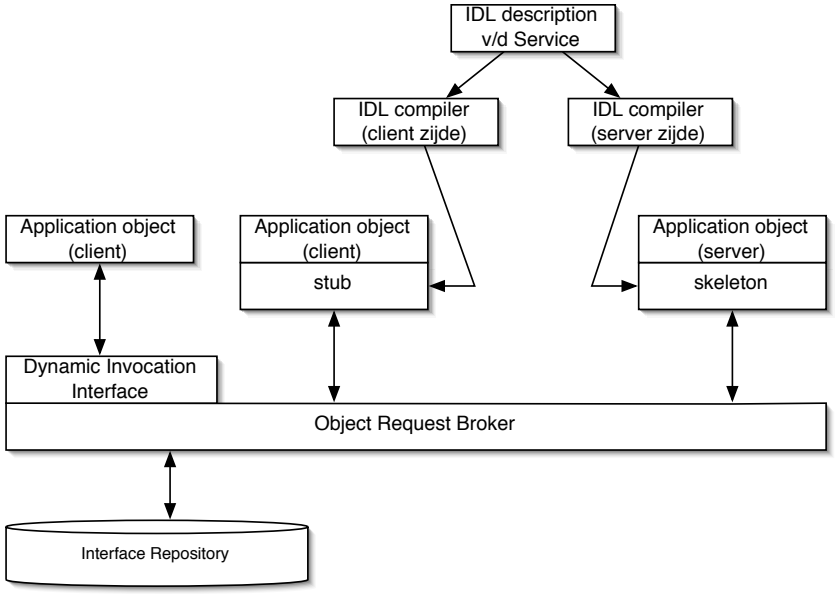
3. De server voert de operatie uit en eventuele responses worden teruggestuurd.

Na verloop van tijd heeft de client zo een aantal RPC's bij verschillende servers uitgevoerd, die zich allen bij de transaction manager geregistreerd hebben. De client stub ontvangt dan een EOT van de client ten teken dat de transactie beëindigd moet worden. Hij geeft dit door aan de transaction manager die daarop het 2PC protocol start met alle geregistreerde servers.

1.2.3 Object brokers

RPC's blijven beperkt tot eenvoudige functie oproepen. Object brokers breiden RPC's uit naar de object georiënteerde wereld. Een bekend voorbeeld van object brokers is de Common Object Request Broker Architecture (CORBA). Naast statische binding is ook dynamische binding mogelijk wat in feite eigen is aan de object georiënteerde wereld waarbij via inheritance eenzelfde oproep verschillende resultaten kan opleveren al naargelang de klasse van het object. Net zoals bij RPC worden ook hier data formaten omgezet naar algemene representaties wat CORBA programmeertaal - en platform - onafhankelijk maakt. Het verschil met RPC is dat bij CORBA standaarden gedefinieerd zijn betreffende de omzetting van data formaten.

De CORBA architectuur ziet er uit zoals in figuur 1.3:



Figuur 1.3: CORBA Architectuur [21]

Aan de hand van de IDL description worden enerzijds een client stub en anderzijds een server skeleton gegenereerd. Alle oproepen passeren via de Object Request Broker die instaat voor de vertaling van dataformaten en voor het localiseren van de services. Dynamic binding wordt voorzien door de Dynamic Invocation Interface in combinatie met een Interface Repository. Er kan gezocht worden op de naam van de service of de functionaliteit die de service voorziet. Aangezien deze beschrijvingen afhangen van interpretatie (semantiek die iemand er aan geeft) wordt dit in de praktijk weinig gebruikt. Zoals verder zal blijken wordt de dynamische binding ook bij Web Services weinig gebruikt.

1.2.4 Object Monitors

Een object monitor is een combinatie van een TP monitor met een Object broker.

1.2.5 Message-Oriented Middleware

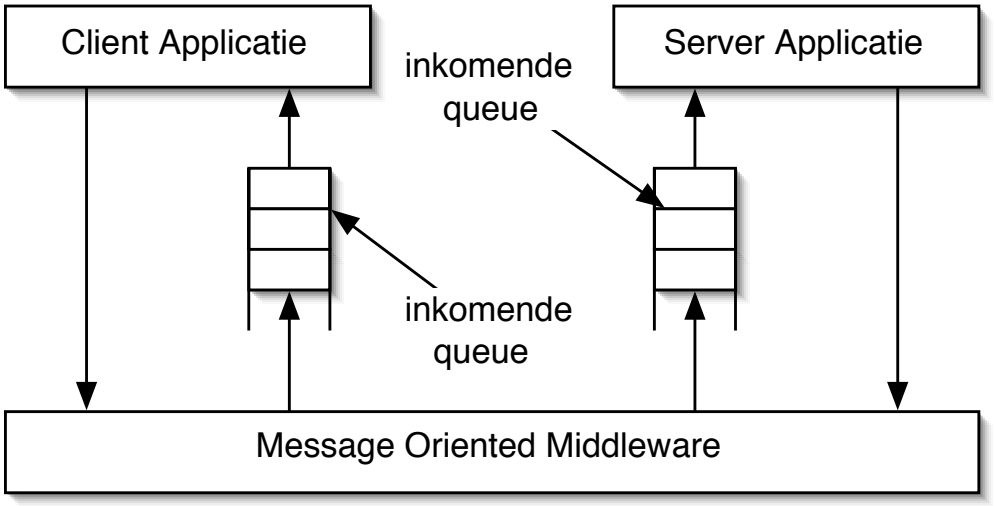
De tot nu toe besproken vormen van middleware waren vooral synchroon. Een interactie is synchroon als deze voldoet aan het volgende patroon:

1. De client applicatie doet een bepaald verzoek en wacht met de verdere uitvoering tot een antwoord binnenkomt.

2. De server krijgt een verzoek binnen en handelt dit onmiddellijk af.

Message-Oriented Middleware (MOM) richt zich meer op de assynchrone manier van interactie waarbij de client in afwachting op een antwoord van de server gewoon verder gaat met de normale verwerking. Tot de klasse van MOM behoren alle middleware applicaties die message gebaseerde interactie ondersteunen.

De algemene werkwijze van MOM wordt weergegeven in figuur 1.4:



Figuur 1.4: Message Oriented Middleware

Als de client een bepaalde service wenst uit te voeren stuurt hij een message met daarin de bestemming en de uit te voeren operatie naar de MOM die ervoor zorgt dat de message bij de juiste server terechtkomt. Meestal wordt dit dan gecombineerd met een systeem van input queues waarin de messages bewaard worden alvorens ze te verwerken.

De voordelen van deze queues zijn:

- Er kunnen prioriteiten aan de messages toegekend worden.
- Er kan aan load balancing gedaan geworden door meerdere servers per queue te voorzien.
- Servers moeten niet steeds online zijn.

Het grootste nadeel van de gewone message oriented middleware is het feit dat in elke message de bestemming vermeld moet worden en dat we dus op die manier enkel over point to point verbindingen beschikken. Message Brokers bieden hier een oplossing voor.

1.2.6 Message Brokers

Traditionele RPC en MOM gebaseerde systemen creëren point-to-point verbindingen tussen applicaties en zijn dus eerder statisch en niet flexibel. Message brokers zijn een speciale vorm van MOM systemen die deze beperkingen aanpakken door te handelen als een soort tussenpersoon. Een Message broker transporteert niet alleen de messages maar heeft ook controle over de routing, filtering en de verwerking van messages als ze doorheen het systeem bewegen. Daarnaast kunnen ze ook op een dynamische manier de bestemming van een message bepalen aan de hand van de inhoud van de message.

In feite is een message broker gewoon een queueing systeem waarin alle messages samenkomen en waarop software geïmplementeerd kan worden. Deze software staat dan in voor de routing, filtering en verwerking van de messages. Tot die software behoort onder meer het publish/subscribe mechanisme. Hierbij is het de bedoeling dat applicaties zich eerst "subscriben" voor bepaalde messages. Dit kan gebeuren a.d.h.v. het type message, de parameters van de message of de herkomst van de message. Applicaties die messages versturen moeten geen recipiënten specificeren, ze moeten gewoon de message "publishen", de message broker doet de rest. Zender en ontvanger zijn nu van elkaar losgemaakt: de zender weet niet wie de ontvanger zal zijn en de ontvanger weet niet van wie de message zal komen.

Daarnaast biedt de message broker ook nog tal van andere functionaliteiten aan zoals bijvoorbeeld transacties, workflow management,...

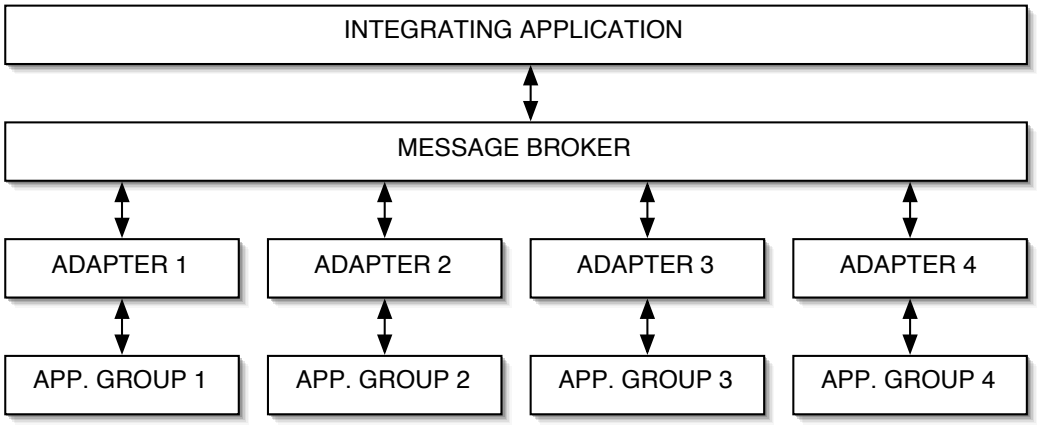
1.3 Enterprise Application Integration

Dankzij three-tier systemen en middleware was het mogelijk om de verschillende servers behorend tot de resource management layer te integreren.

Enterprise Application Integration (EAI) is een veralgemening van dit idee waarbij ook application logic layers van verschillende middleware systemen geïntegreerd kunnen worden. EAI tracht te komen tot een situatie waarbij de verschillende middleware platformen met elkaar kunnen communiceren.

De functionaliteiten geboden door message brokers tesamen met een assynchrone interactie zijn uiterst geschikt voor EAI systemen. Ze vormen dan ook de basis van de meeste EAI systemen van vandaag de dag. Naast Message Brokers gebruiken EAI systemen ook nog adapters voor het maskeren van de heterogeniteit van de onderliggende systemen.

De algemene architectuur van zo'n EAI systeem is te zien in figuur: 1.5



Figuur 1.5: Enterprise Application Integration Architectuur

In deze situatie hebben we dus vier groepen van applicaties met elk hun eigen middleware platform. Aangezien ze allen een ander middleware platform gebruiken is rechtstreekse communicatie niet mogelijk. Om dit probleem op te lossen werden de adapters in het leven geroepen. De adapters staan in voor de convertie van heterogene data formaten naar algemene data formaten en omgekeerd. Communicatie gebeurt dus zoals gezegd door het publishen en subscriben van en voor messages. Bovenop dit alles wordt een nieuwe layer gedefinieerd waarin zich software kan bevinden die op een algemene manier gebruik maakt van al de onderliggende applicaties.

1.3.1 Problemen van EAI

EAI systemen werken steeds met een soort centraal orgaan, de message broker, dat zorgt voor de interactie en de integratie van verschillende applicaties.

Zolang dit binnen één bedrijf over een LAN gebeurt verloopt alles prima. De problemen ontstaan pas wanneer we dit principe willen gebruiken voor business to business (B2B) interactie via het Internet. De grote moeilijkheid waarmee we dan geconfronteerd worden is dat het om verschillende redenen niet mogelijk is zo'n centraal orgaan op te richten (zie Hoofdstuk 2). Bijgevolg moet dit heel anders aangepakt worden. Web services leveren ons een oplossing voor dit en andere problemen.

Hoofdstuk 2

De principes van web services

Business to business (B2B) interacties gebeuren vandaag de dag veelal op manuele basis, d.m.v. telefoon, email of het invullen van web formulieren. Het hoeft niet gezegd te worden dat dit een weinig efficiënte en tijdrovende bezigheid is die beter geautomatiseerd zou moeten zijn.

Zoals we gezien hebben bij de bespreking van middleware in hoofdstuk 1 wordt er steeds gebruik gemaakt van een centraal orgaan dat instaat voor de communicatie tussen de verschillende applicaties. In sectie 1.3.1 haalden we reeds aan dat zo'n centraal orgaan niet kan gebruikt worden als het over B2B interactie gaat. De voornaamste redenen hiervoor zijn:

1. Gebrek aan vertrouwen tussen de bedrijven.
2. Probleem wat beteft de locatie van het centrale orgaan.

Meer in het bijzonder: De grote vraag is waar dit orgaan zich zou moeten bevinden. Er zijn verschillende mogelijkheden. Een eerste optie is om dit orgaan bij één van de aan de B2B interactie deelnemende bedrijven onder te brengen. Door een gebrek aan vertrouwen tussen de verschillende bedrijven is dit echter geen optie. Andere mogelijkheden zijn: de oprichting van één centraal orgaan voor alle B2B interacties, of de oprichting van één centraal orgaan per B2B interactie. Ook hier stoot men op problemen van vertrouwen.

Web services profileren zichzelf als de oplossing voor deze problemen. De oplossing bestaat uit 3 basisconcepten:

1. Service Oriented Architecture (SOA). (zie sectie 2.1).
2. Herontwerpen van allerhande middleware protocollen zoals transacties, 2PC, betrouwbare messaging,... (zie hoofdstuk 6 en 7)

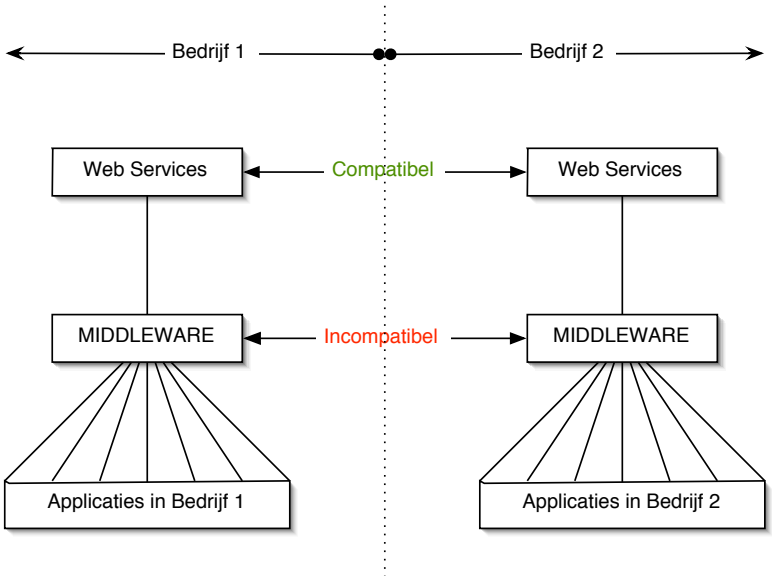
3. Standaardisatie: SOAP, WSDL, UDDI. (zie hoofdstuk 3, 4 en 5)

Het World Wide Web Consortium (W3C) definieert web services als volgt:

> A Web service is a software application identified by a URI, whose interfaces and bindings are capable of being defined, described, and discovered as XML artifacts. A Web service supports direct interactions with other software agents using XML based messages exchanged via Internet-based protocols. [11]

Deze definitie geeft aan dat web services beschreven en ontdekt moeten kunnen worden en dat hiervoor XML gebruikt wordt. Verder wordt ook aangegeven dat deze services kunnen gebruikt worden als bouwstenenen van grotere gedistribueerde systemen en dat communicatie mogelijk is via XML gebaseerde messages.

Grafisch kunnen we web services voorstellen zoals in figuur 2.1:



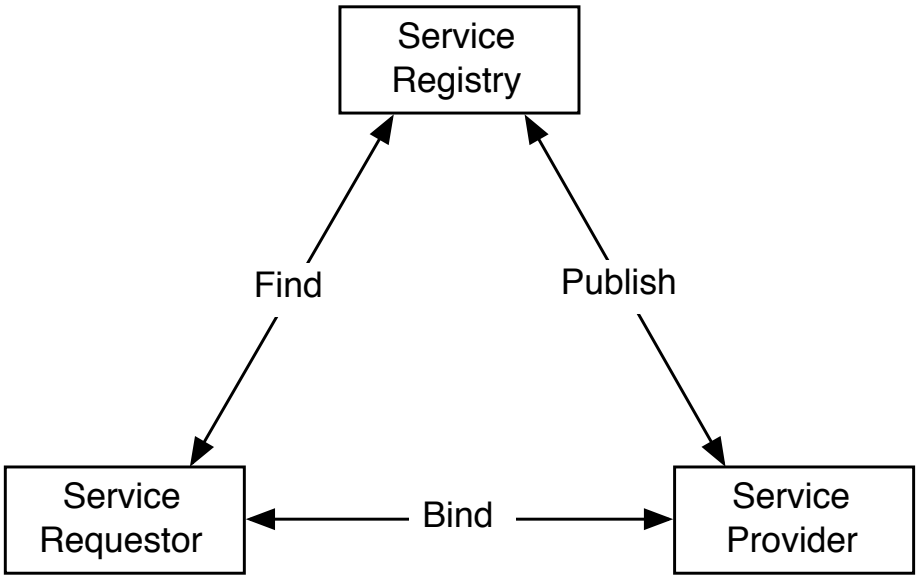
Figuur 2.1: Web Services

Links op de figuur bevinden zich alle applicaties van bedrijf 1 en rechts die van bedrijf 2. Elk bedrijf beschikt over een eigen middleware systeem dat ervoor zorgt dat de applicaties binnen het bedrijf met elkaar kunnen communiceren. De middleware van bedrijf 1 is echter niet compatibel met die van bedrijf 2 waardoor rechtstreekse communicatie niet mogelijk is. Aangezien

de interactie over internet moet gebeuren bieden ook de EAI systemen geen oplossing omwille van de eerder vermelde vertrouwens kwestie. Web services doen dit wel door het definiëren van een extra layer bovenop de bestaande middleware die ervoor zorgt dat de applicaties als web service kunnen opgeroepen worden. Communicatie en interactie tussen bedrijf 1 en bedrijf 2 is vanaf nu wel mogelijk via XML gebaseerde standaarden.

2.1 Service Oriented Architecture

Figuur 2.2 is een grafische voorstelling van de SOA architectuur:



Figuur 2.2: Service Oriented Architecture

De algemene werking van de SOA architectuur is als volgt. Een Service Provider publiceert beschrijvingen van de door hem aangeboden services bij de Service Registry. Dit kan bijvoorbeeld gebeuren met behulp van WSDL documenten waarover meer in hoofdstuk 4. Een Service Requestor die op zoek is naar een bepaalde service voert een zoekopdracht uit op de service descriptions verzameld door de Service Registry. Indien de Service requestor een geschikte beschrijving vindt, kan er een binding afgesloten worden tussen Service Requestor en Service Provider.

Het SOA model bestaat uit 3 hoofdactiviteiten:

1. Service description (sectie 2.1.1)
2. Service discovery (sectie 2.1.2)
3. Service integration (sectie 2.1.3)

2.1.1 Service description

Een service description moet minstens de volgende onderdelen beschrijven:

1. De interface.
2. De gebruikte business protocollen.
3. De properties en semantiek van de service.

Interfaces

Interface Definition Languages (IDL) vormen de basis van elk SOA systeem. Er zijn heel wat verschillen tussen de IDL in traditionele middleware en de IDL van web services. Het grootste verschil is dat web services geen impliciete context hebben, web services zijn eerder loosely coupled in tegenstelling tot traditionele middleware die eerder tightly coupled is. Dit brengt met zich mee dat bepaalde zaken, zoals de semantiek van de service, niet uit de context kunnen worden afgeleid. Naast de semantiek worden ook het onderliggende protocol en de locatie van de service in de service description opgenomen. De typische taal die hiervoor gebruikt wordt is de Web Service Description Language **(WSDL)**. (zie hoofdstuk 4)

Business protocollen

Web services bieden dikwijls verschillende operaties aan waarbij de volgorde van oproepen belangrijk is. De opeenvolging van oproepen en uitvoeren van operaties wordt een *conversatie* genoemd. Elke web service bepaalt aan de hand van een aantal constraints de toegelaten conversaties. Deze rules maken deel uit van het business protocol dat door de betreffende web service ondersteund wordt. Deze gegevens kunnen niet in WSDL beschreven worden. Business protocollen kunnen gedefinieerd worden met talen zoals: Web Service Conversation Language *(WSCL)* en Business Process Execution Language for Web Services *(BPEL4WS)* (zie hoofdstuk 7). Geen van deze talen is momenteel gestandaardiseerd.

Properties en semantiek

Het gaat hier over informatie die bepalend is voor de keuze van de web service maar die niet deel uitmaakt van de interface. De kostprijs van een bepaalde service, de naam van de ondernemening die de service aanbiedt of de Quality of Service (QoS) van een service zijn voorbeelden van dergelijke informatie. Deze info wordt meestal verstrekt door Universal Description, Discovery and Integration **(UDDI)**. (zie hoofdstuk 5)

2.1.2 Service discovery

De service descriptions van hierboven worden dan gepubliceerd in service directories zodat anderen deze kunnen raadplegen op zoek naar de geschikte service. **UDDI** service registries voorzien daarom in een reeks van API's voor het toevoegen, aanpassen, verwijderen van service descriptions, het identificeren van geschikte services,...

2.1.3 Service interaction

In de vorige twee punten hebben we het bindingprobleem behandeld. Het bindingprobleem komt overeen met de stappen die gezet moeten worden vooralleer de geschikte service gevonden wordt. Als er een binding is kunnen we over gaan naar de interactie.

Web Services definiëren op het gebied van service interactie:

De mogelijke transport protocollen: De communicatie gebeurt via een bepaald transport protocol. Meestal is dit *HTTP*.

De mogelijke message formaten: Er is een standaard formaat nodig voor het uitwisselen van informatie. Deze taak is weggelegd voor het Simple Object Access Protocol **(SOAP)**. (zie hoofdstuk 3)

Mechanismen voor het ondersteunen van business protocollen: Een onderdeel van de protocol infrastructuur is de software die instaat voor het ondersteunen van een bepaald business protocol. Dit kan bijvoorbeeld gebeuren door het bijhouden van een state voor elke conversatie.

Een aantal Middleware protocollen: Dit is in feite de herdefinitie van protocollen die reeds beschikbaar waren op tranditionele middleware systemen. Door het ontbreken van een centrale coordinatie en de asynchrone interactie is een nieuwe implementatie noodzakelijk. Deze

protocollen worden ook wel *horizontale protocollen* genoemd omdat ze zeer algemeen zijn en op verschillende web services van toepassing zijn. Een voorbeeld van zo een protocol is het protocol dat instaat voor het coördineren van een transactie. Dit is een zeer algemene functionaliteit die door verschillende services gebruikt kan worden.

2.2 Drie basisconcepten

Waar komen nu de drie basisconcepten van web services aan bod?

1. De SOA architectuur wordt verduidelijkt aan de hand van de service description, service discovery en service interaction.

2. De herdefinitie van de middleware protocollen dat zijn de horizontale protocollen in de service interaction.

3. De woorden in vette druk in bovenstaande uitleg (WSDL, UDDI en SOAP) komen overeen met de standaarden die we verderop zullen bespreken.

2.3 Web service compositie

Er bestaan twee soorten web services: basis en samengestelde web services. Een samengestelde (of in het Engels “composite”) web service is een service die zelf nieuwe services oproept. Indien er geen nieuwe services worden opgeroepen gaat het om een basis web service. Op het vlak van service composition zijn er nog geen standaarden maar meestal wordt gebruik gemaakt van het reeds eerder genoemde BPEL4WS. (zie hoofdstuk 7)

2.4 Web Services architectuur

Web services zijn een soort van wrappers voor de conventionele middleware, ze zorgen ervoor dat twee soorten middleware met elkaar kunnen communiceren. Ze kunnen gezien worden als een extra tier die ervoor zorgt dat middleware services kunnen opgeroepen worden als web services.

De architectuur van web service kan in 2 delen gesplitst worden. Enerzijds is er de interne architectuur die ervoor zorgt dat middleware services als web services opgeroepen kunnen worden. Anderzijds is er de externe architectuur die zorgt voor:

1. Web services die elkaar kunnen vinden en met elkaar kunnen communiceren.

2. De ondersteuning van diverse middleware protocollen.

3. De ondersteuning van service compositie.

De standaarden die we verder zullen bespreken zijn vooral terug te vinden in de externe architectuur.

Hoofdstuk 3

SOAP: Simple Object Access Protocol.

3.1 Wat is SOAP?

SOAP versie 1.2 [30, 31, 32] is een protocol dat ontwikkeld werd voor het uitwisselen van gestructureerde informatie in een gedecentraliseerde omgeving. SOAP is het resultaat van een samenwerking tussen Canon, IBM, Microsoft en Sun en is een W3C Recommandation gebaseerd op XML. Het is een messaging framework dat tegelijk eenvoudig en uitbreidbaar is en waarbij messages over verschillende onderliggende protocollen verstuurd kunnen worden (Protocol Bindings).

Het SOAP protocol specificeert onder andere:

1. Een **message formaat** voor one-way communicatie dat aangeeft hoe data verpakt moet worden in een XML structuur.

2. Een reeks conventies waarin staat hoe een RPC interactie patroon verwerkt kan worden in SOAP messages.

3. Een **processing model** met een reeks regels waaraan elke SOAP verwerkende node zich moet houden. Meer in het bijzonder gaat het hier over hoe aangegeven wordt welke elementen door welke SOAP nodes (zie definitie 3.1 voor de definitie van een SOAP node) verwerkt moeten worden.

4. **Fouten scenario's** waarin aangegeven is wat er moet gebeuren in geval van fouten.

5. Hoe SOAP messages verstuurd kunnen worden over verschillende onderliggende protocollen. Momenteel is enkel HTTP als onderliggende protocol gestandaardiseerd. Andere protcollen zoals SMTP zijn mogelijk maar zij behoren niet tot de standaard.

Elk van deze onderdelen wordt beschreven in één van de volgende secties.

Definitie 3.1. Een SOAP node volgens W3C [13]:

> The embodiment of the processing logic necessary to transmit, receive, process and/or relay a SOAP message, according to the set of conventions defined by this recommendation. A SOAP node is responsible for enforcing the rules that govern the exchange of SOAP messages. It accesses the services provided by the underlying protocols through one or more SOAP bindings.

Deze **SOAP nodes** komen echter vaak overeen met de verschillende "tiers" in de web service middleware. Het **SOAP message path** is de verzameling van SOAP nodes via dewelke de SOAP messages, vertrekkend van de zender, de ultimate receiver bereiken. De SOAP nodes tussen de zender en de uiteindelijke ontvanger worden **"intermediaries"** genoemd. Op dit punt moeten we toch even het routing- en adresserings-probleem aanhalen. Er is namelijk nog geen mechanisme voorzien om binnen een SOAP message het adres van de bestemming of het te volgen message path aan te geven. Zowel de adressering als de routing van de messages worden momenteel afgehandeld door het onderliggende transport protocol:

1. Het path van een SOAP message komt volledig overeen met de door het onderliggende transport protocol gevolgde route.

2. Het adres van de ultimate receiver wordt niet in de SOAP message opgenomen maar maakt deel uit van het onderliggende transport protocol. (meestal HTTP)

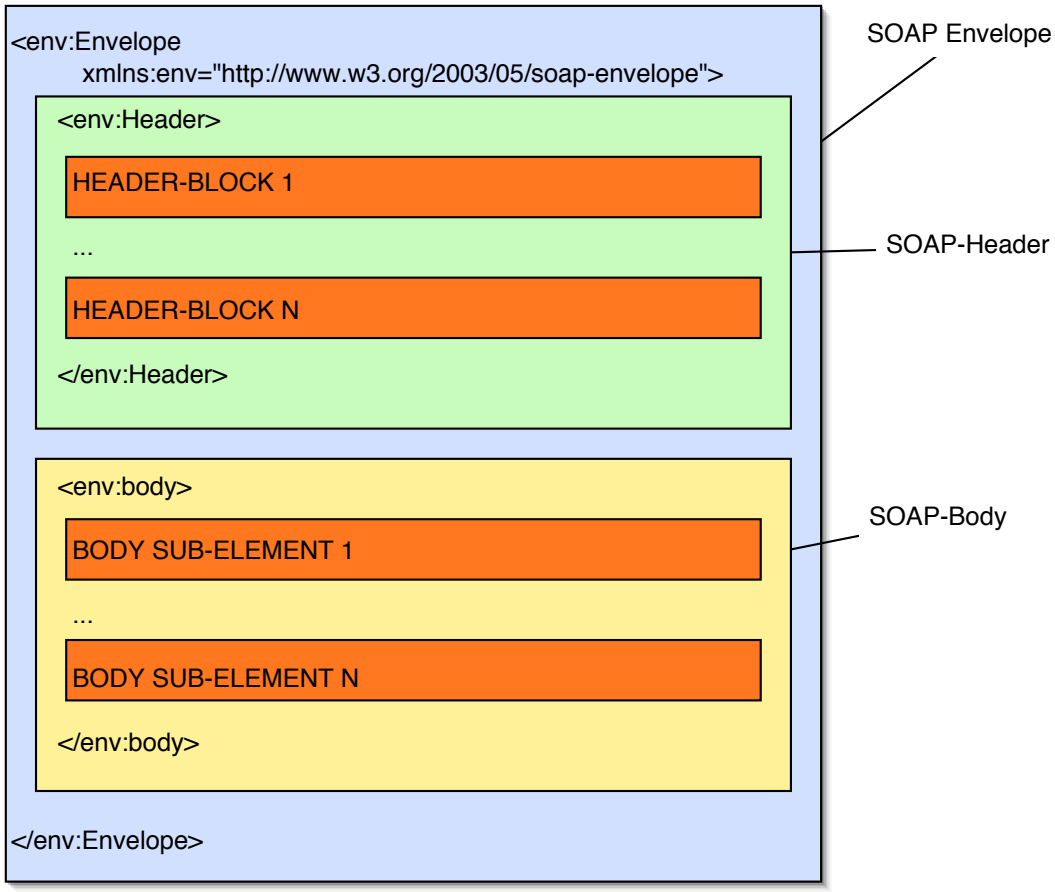
Op het vlak van routing en adressering zijn momenteel wel standaarden in ontwikkeling. (zie sectie 3.9.2)

3.2 Welke functie heeft SOAP binnen web-services?

SOAP messages worden gebruikt voor communicatie tussen clients en servers van web services. SOAP is in feite ontwikkeld voor het ondersteunen

van loosely coupled systemen die met elkaar communiceren door het uitwisselen van one-way assynchrone messages. Andere interactie patronen kunnen voorzien worden door SOAP te combineren met verschillende onderliggende protocollen of middleware. De SOAP specificatie definieert o.a. hoe het RPC interactiepatroon in SOAP messages ingebed kan worden. (zie sectie 3.3.1).

3.3 SOAP Message Formaat



Figuur 3.1: SOAP message structuur

Zoals te zien op figuur 3.1 bevat het "Envelope" element, dat tevens ook het root element van elke SOAP message is, twee subelementen:

- Een optioneel Header-element (env:header)

- Een verplicht Body-element (env:body)

De inhoud van deze elementen is applicatie-afhankelijk en wordt niet gedefinieerd als deel van de SOAP specificatie. Het SOAP Header-element is een uitbreidings mechanisme bedoeld voor het toevoegen van extra informatie aan de SOAP message. Het Body-element is de plaats waar de belangrijkste informatie bijgehouden wordt. Het grote verschil tussen data in de SOAP header en data in de SOAP body is dat informatie in de header door zowel de intermediaries als de ultimateReceiver kan verwerkt worden terwijl info in de body enkel voor de ultimateReceiver bestemd is.

Er zijn twee aspecten die de structuur van een SOAP message bepalen:

1. Interaction style. (zie sectie 3.3.1)
2. Encoding style. (zie sectie 3.3.2)

3.3.1 Interaction Style

SOAP kan gebruikt worden in combinatie met twee verschillende interactie stijlen:

1. **Document style**: de twee interagerende partijen komen overeen wat betreft de document structuur van de uit te wisselen messages.
2. **RPC style**: voorgedefinieerde message structuur.

De interactie stijl wordt niet expliciet vermeld maar beïnvloedt wel de structuur van het body element. Als er gekozen wordt voor de RPC interaction style dan ligt de structuur van de request en response messages vast. De structuur van de RPC request message ziet er dan uit zoals voorbeeld 3.1.

Voorbeeld 3.1.

```
<env:Body>
  <m:methodeNaam xmlns m="eenURI">
    <m:param1></m:param1>
    <m:param2></m:param2>

  </m:methodeNaam>
</env:Body>
```

De structuur van de RPC response message is te zien in voorbeeld 3.2.

Voorbeeld 3.2.

```
<env:Body>
  <m:methodeNaamResponse xmlns m="eenURI">
    <m:response1></m:response1>
    <m:response2></m:response2>

  </m:methodeNaamResponse>
</env:Body>
```

Als daarentegen voor de document interaction style gekozen wordt dan komen beide partijen zoals gezegd een bepaalde structuur overeen die nader gedefinieerd zal worden via de encoding style.

3.3.2 Encoding Style

De encoding style definieert hoe een bepaalde datastructuur wordt voorgesteld in XML. Als de client en de server met elkaar willen communiceren moeten ze een encoding style overeenkomen.

SOAP specificeert een eigen encodering nl. SOAP-encoding. Deze encoding definieert hoe basis types en complexe types gaande van integers tot arrays in XML structuren worden gegoten. Dit is echter geen verplichte encoding style, users kunnen een eigen encodering definiëren. Men spreekt in dat geval van "literal encoding". Een literal encoding is meestal gebaseerd op een overeengekomen XML Schema.

De encoding style wordt, in tegenstelling tot de interactie stijl, wel expliciet vermeld aan de hand van het encodingStyle attribuut dat in praktisch elk element kan voorkomen. Document style interactie wordt meestal gebruikt in combinatie met literal encoding terwijl RPC interactie meestal met SOAP-encoding gecombineerd wordt.

SOAP encoding

SOAP encoding is een encoding stijl waarbij objecten, voorgesteld door een gerichte graaf, worden omgezet in een XML representatie. De algemene werkwijze voor de zender is als volgt:

1. De zender wil een bepaald object uit een bepaalde programmeertaal zoals java of C versturen.

2. SOAP stelt van dit object een SOAP data model op. (dit is een gerichte graaf zoals in figuur 3.2)

3. Dit SOAP data model wordt dan via SOAP encoding omgezet naar XML.

Aan de ontvangst zijde gebeurt het omgekeerde proces:

1. De ontvanger krijgt een SOAP message binnen en ziet dat er gebruik werd gemaakt van SOAP encoding.

2. Het XML document wordt via SOAP decoding terug omgezet naar het SOAP data model.

3. Dit SOAP data model kan dan weer omgezet worden naar een Object in C, Java of een andere programmeertaal.

Het grote voordeel van deze manier van werken is dat objecten van de ene programmeertaal omgezet kunnen worden in objecten behorend tot een andere programmeertaal. Web services zijn bijgevolg onafhankelijk van de gekozen programmeertaal.

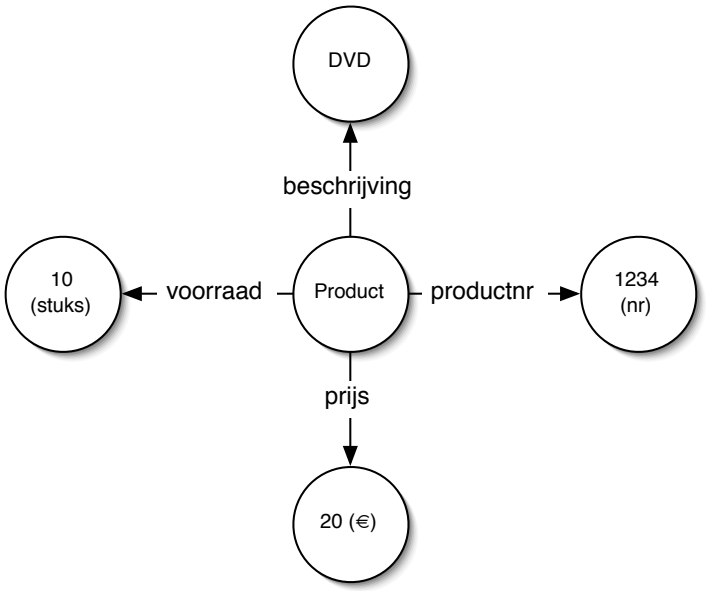
We verduidelijken het omzettingsproces aan de hand van het volgende voorbeeld waarbij we beginnen met het Java object in voorbeeld 3.3.

Voorbeeld 3.3.

```
Class Product
{
   String beschrijving;
   int productnr;
   int prijs;
   int voorraad;
}
```

Dit object wordt dan omgezet in het SOAP data model in figuur 3.2 waarbij de velden reeds ingevuld zijn. Deze omzetting kan ofwel handmatig geïmplementeerd worden ofwel wordt deze functionaliteit voorzien door een bestaand framework.

Zoals te zien is op figuur 3.2 is het SOAP data model een gerichte graaf waarbij complexe types, in dit geval de klasse Product, herkend kunnen worden doordat er edges uit vertrekken. Basis types zoals strings en integers zijn weergegeven als nodes met enkel inkomende edges. De labels van de edges komen overeen met de velden van het type, Product, en de labels van de nodes komen overeen met de waardes voor de overeenkomstige velden. Dit is in feite een heel eenvoudig voorbeeld, in de praktijk kunnen de subtypes



Figuur 3.2: SOAP Data Model

van een complex type opnieuw complexe types zijn en kunnen er ook lussen in de graaf ontstaan. Bij de omzetting naar XML wordt dit opgelost door gebruik te maken van ids en referenties.

De laatste stap is de omzetting van dit data model naar een XML representatie: zie voorbeeld 3.4.

Voorbeeld 3.4.

```
<Product soapenv:encodingstyle="http://www.w3.org/2003/05/soap-encoding">
   <beschrijving>DVD</beschrijving>
   <productnr>1234</productnr>
   <prijs>20</prijs>
   <voorraad>10</voorraad>
</Product>
```

De XML representatie stelt complexe types voor als elementen waarvan de subelementen overeenkomen met de veldnamen van het gemodelleerde complexe type. Deze subelementen bevatten op hun beurt de eigenlijke waarde voor het overeenkomstige veld.

Als encodingstijl wordt “http://www.w3.org/2003/05/soap-encoding” gebruikt om aan te geven dat het om soap-encoding gaat. Op deze manier weet de

ontvanger van de message welke decoding hij moet gebruiken.

Literal Encoding

Zoals gezegd wordt literal encoding meestal gecombineerd met document style interactie waarbij beide partijen een bepaalde message structuur overeenkomen. Als er gekozen wordt voor literal encoding dan wordt de structuur van het betreffende element aangegeven door een XML-Schema type.

3.4 SOAP processing Model

Het SOAP processing model geeft een beschrijving van wat er moet gebeuren als een SOAP node een message ontvangt. Bij het verwerken van een SOAP message wordt allereerst nagegaan of de message voldoet aan de SOAP syntax. De verdere verwerking van de message is afhankelijk van het role attribuut, het mustUnderstand attribuut en het relay attribuut.

3.4.1 Het role attribuut

Het role attribuut wordt gebruikt om aan te geven welke delen van de SOAP message door welke SOAP nodes verwerkt moeten worden. Elke headerblock kan een role attribuut bevatten waarbij een Uniform Resource Identifier (URI) gebruikt wordt als role identifier. SOAP specificeert drie standaard roles:

De none role:
"http://www.w3.org/2003/05/soap-envelope/role/none": Geen enkele intermediary vervult ooit de none role. Deze role is enkel bedoeld voor het opslaan van extra informatie waarnaar kan gerefereerd worden vanuit andere header blocks of het body element.

De next role:
"http://www.w3.org/2003/05/soap-envelope/role/next": Elke SOAP node voldoet aan de next role.

De ultimateReceiver role:
"http://www.w3.org/2003/05/soap-envelope/role/ultimateReceiver": Als het role attribuut de ultimateReceiver role aangeeft of indien het role attribuut afwezig is dan is de header block bedoeld voor de ultimateReceiver. Het body element heeft geen role attribuut en is bijgevolg steeds bedoeld voor de ultimateReceiver.

Tabel 3.1 geeft een overzicht van wanneer header blocks verwerkt worden in functie van de waarde van hun role attribuut. UR staat hierbij voor "Ultimate Receiver".

	Role gespecificeerd in header			
Node:	**Afwezig**	**None**	**Next**	**UR**
Intermediary	nee (1)	nee	ja (2)	nee
UR	ja	nee	ja	ja

Tabel 3.1: Verwerking in functie van het role attribuut

Tabel 3.1 moet op de volgende manier gelezen worden. Voor element (1): Indien de SOAP node die de message binnenkrijgt een intermediary is en het role attribuut is niet aanwezig in de betreffende header dan wordt deze header niet verwerkt door de intermediary. Voor element (2): De SOAP node die de message binnenkrijgt is opnieuw een intermediary maar dit keer is het role attribuut wel aanwezig en heeft het de next waarde. Aangezien elke SOAP node voldoet aan de next role wordt deze header block wel verwerkt door de betreffende intermediary. De andere entries in de tabel kunnen op dezelfde manier afgelezen worden.

3.4.2 Het mustUnderstand attribuut

Als een message een SOAP node bereikt moet dus eerst gecontroleerd worden of die message voldoet aan de SOAP syntax. Daarna wordt gecheckt welke header-blocks bedoeld zijn voor de huidige node, dit gebeurt aan de hand van het role attribuut zoals hierboven beschreven. In de volgende stap wordt gecontroleerd wat de waarde van het optionele mustUnderstand attribuut is.

Als het mustUnderstand attribuut ontbreekt of false is, en de role komt overeen met de huidige SOAP node, dan kan de node zelf kiezen of hij de headerblock al dan niet zal verwerken. In beide gevallen gaat de verdere verwerking gewoon door en zal de uitgaande message deze header-block niet meer bevatten tenzij er gebruik gemaakt wordt van het relay attribuut waarover meer in sectie 3.4.3.

Indien echter het mustUnderstand attribuut true is, dan is de node verplicht om de header-block te verwerken. Indien hij hierin niet slaagt wordt het verdere verwerkingsproces stopgezet en wordt een foutboodschap gegenereerd.

(zie sectie 3.5)

Het body element bevat geen mustUnderstand attribuut maar het is natuurlijk vanzelfsprekend dat de Ultimate Receiver de inhoud van het body element moet begrijpen.

Tabel 3.2 toont het verband tussen de verwerking van header-blocks en de waarde van het mustUnderstand attribuut in het geval waarbij de role overeenkomt. UR staat hierbij voor "Ultimate Receiver".

	MustUnderstand gespecificeerd in Header		
Node	**Afwezig**	**False**	**True**
Intermediary	Mag verwerken	Mag verwerken	Moet verwerken
UR	Mag verwerken	Mag verwerken (1)	Moet verwerken

Tabel 3.2: Verwerking in functie van het MustUnderstand attribuut

Tabel 3.2 moet op de volgende manier gelezen worden. Voor element (1): Stel dat de SOAP node die de message binnenkrijgt de ultimateReceiver is en dat het role attribuut hiermee overeenkomt. Als nu het mustUnderstand attribuut gelijk is aan "false" dan mag de ultimateReceiver de header block verwerken. Hij is echter niet verplicht om dit te doen. De keuze hangt meestal af van het feit of hij de header block begrijpt of niet. In het eerste geval zal hij wel verwerken terwijl in het tweede geval geen verwerking zal plaatsvinden.

3.4.3 Het relay attribuut

De standaard procedure bij het verwerken van een headerblock verwijdert de header block in de uitgaande message. Dit geldt ook voor messages die niet verwerkt worden maar toch voor de huidige node bedoeld waren (role komt overeen en mustUnderstand is false of afwezig). Om dit default gedrag naar eigen keuze aan te passen kan er gebruik gemaakt worden van het optionele relay attribuut.

De defaultwaarde voor het relay attribuut is false. Indien het relay attribuut echter true is dan wil dit zeggen dat de betreffende header sowieso in de uitgaande SOAP message aanwezig moet zijn, onafhankelijk van de verwerking van de header.

3.5 Fouten Scenario's

Gedurende het transport en de verwerking van SOAP messages kunnen heel wat fouten optreden. Daarom werd in de SOAP specificatie een model opgenomen voor het afhandelen van fouten. De grammatica in tabel 3.3, waarbij "?" staat voor optionele elementen en "+" voor elementen die 1 of meerdere keren voorkomen, geeft een overzicht van de structuur van fout messages.

<body>	::=	<fault>
<fault>	::=	<code> <reason> <detail>? <node>? <role>?
<code>	::=	<value> <subcode>?
<subcode>	::=	<value> <subcode>?
<reason>	::=	<text xml:lang>+

Tabel 3.3: structuur van een fout message

Een foutmessage bevindt zich steeds in het fault element wat op zijn beurt deel uit maakt van het body element van een SOAP message. Een fault element bevat twee verplichte elementen: code en reason, en drie optionele elementen: detail, node en role.

Het **code** element bevat steeds een **value** element, eventueel gevolgd door een optioneel subcode element. Het value element kan één van de volgende waardes aannemen:

- *Sender*: de fout is het gevolg van incomplete of foutieve data afkomstig van de zender. Om deze fout te verhelpen moet de zender de inhoud van de message corrigeren alvorens de message opnieuw te verzenden.
- *Receiver*: deze fout treedt op indien er iets misloopt tijdens de verwerking van de message bij de receiver. Er is in dit geval niet noodzakelijk een direct verband tussen de fout en de inhoud van de message. Daarom kan de fout in sommige gevallen verholpen worden door nadien dezelfde message nog eens te versturen.
- *VersionMismatch*: deze fout treedt op als de namespace van de SOAP envelope niet compatibel is met die van de receiver.
- *MustUnderstand*: Deze fout treedt op telkens wanneer een SOAP node een voor hem bedoeld element, hetzij een header hetzij de body, niet begrijpt terwijl van hem verwacht wordt dat hij dit wel begrijpt (mustUnderstand=true).

Het **subcode** element is recursief gedefinieerd en kan gebruikt worden om bijkomende, applicatie-afhankelijke, informatie omtrent de fout uit te drukken. Op deze manier kunnen verschillende niveau's van foutcodes gedefinieerd worden.

Het **Reason** element beschrijft de fout in een menselijk leesbaar formaat aan de hand van een aantal text sub-elementen. Ieder text element heeft een xml:lang attribuut waardoor de fout in meerdere talen kan gedefinieerd worden.

Het optionele **node** element identificeert aan de hand van een URI de SOAP node die de fout genereerde. Bij afwezigheid van dit element wordt per default aangenomen dat de fout door de ultimate receiver gegenereerd werd. Deze informatie kan eventueel nog aangevuld worden met het **role** element dat aangeeft welke role de node speelde op het moment dat de fout zich voordeed.

Het optionele **detail** element bevat extra applicatie-specifieke informatie.

Voorbeeld 3.5 is een voorbeeld van een foutboodschap als reactie op een verzonden SOAP message. Eén van de parameters voldoet hier niet aan de vereiste dat het een nummer moet zijn.

Voorbeeld 3.5.

```
<?xml version='1.0' ?>
<env:Envelope xmlns:env="http://www.w3.org/2003/05/soap-envelope"
              xmlns:calc="http://calculation.org">
  <env:Body>
   <env:Fault>
     <env:Code>
       <env:Value>env:Sender</env:Value>
       <env:Subcode>
        <env:Value>calc:BadArguments</env:Value>
       </env:Subcode>
     </env:Code>
     <env:Reason>
      <env:Text xml:lang="en-US">Not A Number</env:Text>
      <env:Text xml:lang="nl">Geen nummer</env:Text>
     </env:Reason>
     <env:Detail>
      <e:myFaultDetails
        xmlns:e="http://calculation.org/faults">
        <e:message>Term1 is not a number</e:message>
        <e:errorcode>004</e:errorcode>
```

```
      </e:myFaultDetails>
     </env:Detail>
   </env:Fault>
 </env:Body>
</env:Envelope>
```

In dit voorbeeld is het dus de zender die de fout veroorzaakt heeft (value element = env:Sender). Het subcode element bevat meer specifieke informatie en geeft aan dat er iets mis is met de meegegeven parameters (BadArguments). Het reason element bevat hier twee text elementen met een duidelijke omschrijving van het probleem in het Engels en het Nederlands. Zoals te zien in bovenstaande message bevat het detail element applicatie specifieke informatie met een eigen errorcode.

Errors tijdens de verwerking van header blocks kunnen ook uitgedrukt worden aan de hand van een fault element. Dit soort fouten treden op wanneer header blocks met mustUnderstand op true niet begrepen worden of indien er een fout optreedt tijdens het verwerken. Voorbeeld 3.6 bevat twee header files met mustUnderstand gelijk aan true. Voorbeeld 3.7 is een foutboodschap die aangeeft dat er iets is misgelopen bij de verwerking van uitbreiding1. De header-block waar het mis liep wordt steeds in de foutboodschap opgenomen.

Voorbeeld 3.6.

```
<?xml version='1.0' ?>
<env:Envelope xmlns:env="http://www.w3.org/2003/05/soap-envelope">
  <env:Header>
    <a:Uitbreiding1 xmlns:a="http://www.URL1.be"
          env:mustUnderstand="true"/>
    <b:Uitbreiding2 xmlns:b="http://www.URL2.be"
          env:mustUnderstand="true"/>
  </env:Header>
  <env:Body>
    . . .
  </env:Body>
</env:Envelope>
```

Voorbeeld 3.7.

```
<?xml version='1.0' ?>
<env:Envelope xmlns:env="http://www.w3.org/2003/05/soap-envelope">
  <env:Header>
    <env:NotUnderstood qname="a:Uitbreiding1"
          xmlns:a="http://www.URL1.be"/>
  </env:Header>
  <env:Body>
    <env:fault>
```

```
      <env:code>
        <env:value>env:mustUnderstand</env:value>
      </env:code>
      <env:reason>
        <env:Text xml:lang="nl">
          1 of meer verplichte SOAP header blocks niet begrepen
        </env:Text>
        <env:Text xml:lang="en">
          one or more mandatory SOAP header blocks not understood
        </env:Text>
      </env:reason>
    </env:fault>
 </env:Body>
</env:Envelope>
```

3.6 Message Exchange Patterns

Het messaging framework van SOAP 1.2 is zeer eenvoudig en beperkt zich tot het versturen van één message tussen zender en ontvanger. In meer interessante scenario's worden meerdere messages uitgewisseld. Voorbeelden van dergelijke scenario's zijn:

- Het request-response pattern.
- Het conversation pattern.

De SOAP specificatie geeft enkel aan hoe messages opgebouwd zijn, hoe ze verwerkt worden, wat er gebeurt in geval van fouten,... Elke verzonden message wordt aanzien als een losstaand feit. Het is de taak van de user en de applicatie om verbanden te zien tussen de messages. Messages kunnen bijvoorbeeld tot dezelfde conversatie behoren of de ene message kan later verzonden zijn dan de andere, dit zijn slechts enkele voorbeelden van verbanden tussen messages.

Deze verbanden kunnen ofwel eigenhandig in de SOAP message zelf bijgehouden worden ofwel worden ze standaard ondersteund door het onderliggende protocol. (zie sectie 3.7) Zo is het bijvoorbeeld mogelijk om voor elke conversatie een bepaald referentie-nummer en verzendingstijd in de message op te nemen. Typisch komt dergelijke informatie in een header van de SOAP message te staan.

De volgende twee SOAP messages maken deel uit van dezelfde conversatie. Dit wordt aangegeven door het reference element. (zie eerste header-block)

Verder is ook het tijdstip van verzenden opgenomen in beide messages zodat duidelijk wordt dat voorbeeld 3.9 later verstuurd werd dan voorbeeld 3.8.

Voorbeeld 3.8.

```
<?xml version='1.0'?>
<env:Envelope xmlns:env="http://www.w3.org/2003/05/soap-envelope">
 <env:Header>
  <m:reservation
        xmlns:m="http://travelcompany.example.org/reservation">
   <m:reference>123456</m:reference>
   <m:dateAndTime>2004-11-29T13:20</m:dateAndTime>
  </m:reservation>
  <n:passenger xmlns:n="http://mycompany.example.com/employees">
   <n:name>Jan Janssens</n:name>
  </n:passenger>
 </env:Header>
 <env:Body>
   . . .
 </env:Body>
</env:Envelope>
</env:Envelope>
```

Voorbeeld 3.9.

```
<?xml version='1.0'?>
<env:Envelope xmlns:env="http://www.w3.org/2003/05/soap-envelope">
 <env:Header>
  <m:reservation
      xmlns:m="http://travelcompany.example.org/reservation">
   <m:reference>123456 </m:reference>
   <m:dateAndTime>2004-11-29T13:35</m:dateAndTime>
  </m:reservation>
  <n:passenger xmlns:n="http://mycompany.example.com/employees">
   <n:name>Jan Janssens</n:name>
  </n:passenger>
 </env:Header>
 <env:Body>
   . . .
 </env:Body>
</env:Envelope>
```

3.7 Protocol Bindings

Een SOAP binding specificeert hoe SOAP messages van de ene SOAP node naar de andere worden doorgegeven gebruik makend van een bepaald onderliggend protocol.

Een **feature** is een specificatie van een bepaalde functionaliteit voorzien door een binding of een SOAP module. Een SOAP module is de implementatie van een feature d.m.v. SOAP header blocks.

Enkele voorbeelden van features zijn:

1. *Vaste aflevervolgorde* van de verzonden SOAP messages
2. *Beveiliging* van de verzonden SOAP message (verzekerd aan de hand van encryptie)
3. *Betrouwbaarheid* van aflevering (er mogen geen messages verloren gaan)

Features worden geïdentificeerd door middel van URIs alwaar de semantiek van de bewuste feature verduidelijkt kan worden. Indien de feature voorzien wordt door de binding dan moet de gebruiker zelf niets meer doen en handelt het onderliggende protocol alles af. In het andere geval moet je als user eigen headers gaan definiëren voor de implementatie van de betreffende feature.

Een SOAP binding bepaalt onder andere:

1. De te gebruiken serialisatie. Dit wordt mee bepaald door de gewenste features. Als de SOAP messages onleesbaar moeten zijn voor onbevoegden, dan kan er bijvoorbeeld gebruik gemaakt worden van encryptie.
2. Het mechanisme waarmee bepaalde features geïmplementeerd kunnen worden door gebruik te maken van de primitieven die behoren tot het onderliggende transport protocol.
3. Aan welke features de binding voldoet.
4. De Message Exchange Patterns (MEPs) die ondersteund worden door de binding. SOAP specificeert twee MEPs:
   (a) Het *SOAP Request-Response MEP*, waarbij tussen de twee SOAP nodes één SOAP message heen en één SOAP message teruggestuurd wordt.
   (b) Het *SOAP Response MEP*, waarbij als request een niet SOAP message verstuurd wordt en als response een SOAP message wordt teruggestuurd.
5. De adressering: nergens in de SOAP messages wordt aangegeven wie de ontvanger van de message zal zijn. Deze taak is weggelegd voor het transport protocol. Bij SMTP kan dit bijvoorbeeld d.m.v. het to-address.

De enige standaard binding voor SOAP is de http binding, andere transport protocollen kunnen gebruikt worden maar zijn op dit moment nog niet gestandaardiseerd. SMTP is een voorbeeld van zo'n ander protocol maar ook eigen protocollen kunnen voor het transport gebruikt worden.

Om met elkaar te kunnen communiceren is het noodzakelijk dat twee opeenvolgende SOAP nodes in het SOAP message path een binding overeenkomen. Het is echter niet noodzakelijk dat alle nodes in het message path dezelfde binding gebruiken. Met andere woorden, de features voorzien in de binding tussen node1 en node2 moeten niet overeenkomen met de features voorzien in de binding tussen node2 en node3.

3.7.1 SOAP HTTP binding

De SOAP HTTP binding [35] is een voorbeeld van een protocol binding voor SOAP. Dit is tevens de standaard binding voor SOAP 1.2 messages. Het HTTP protocol beschikt over een ingebouwd mechanisme waarmee het mogelijk is om verbanden tussen messages terug te vinden. Applicaties die gebruik maken van de HTTP binding moeten dus op dit vlak geen eigen inspanningen (m.b.v. SOAP headers) leveren.

Uitwisseling van SOAP messages kan op twee manieren:

1. **HTTP POST** methode: voor het overbrengen van SOAP messages in de request en response messages. Dit Message Exchange Pattern wordt ook wel: *SOAP request-response MEP* genoemd.

2. **HTTP GET** methode: waarbij SOAP messages enkel in de response message voorkomen. Een andere naam voor dit Message Exchange Pattern is: *SOAP response MEP*.

Beide MEPs kunnen gebruikt worden als instantiatie van de MEPs van een SOAP binding. Applicaties die gebruik maken van features die enkel ondersteund kunnen worden d.m.v. SOAP headers, en dus niet via een binding, zijn genoodzaakt om gebruik te maken van de HTTP POST methode. Als je enkel de toestand van een resource wil opvragen dan kan dit door gebruik te maken van HTTP GET. Dergelijke interacties worden safe of indempotent genoemd aangezien ze geen wijzigingen aanbrengen. Voorbeeld 3.10 en 3.11 zijn voorbeelden van een safe interactie.

Voorbeeld 3.10.

```
GET /travelcompany.example.org/reservations?code=FT35ZBQ
```

```
HTTP/1.1
Host: travelcompany.example.org
Accept: text/html;q=0.5, application/soap+xml
```

Voorbeeld 3.11.

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

<?xml version='1.0' ?>
<env:Envelope xmlns:env="http://www.w3.org/2003/05/soap-envelope">
 <env:Header>
 . . .
 </env:Header>
 <env:Body>
 . . .
 </env:Body>
</env:Envelope>
```

In voorbeeld 3.10 wordt de toestand van een bepaalde reservatie opgevraagd. De waarde van het Accept veld geeft aan van welk media type of types de response message moet zijn.

Voorbeeld 3.11 bevat naast de HTTP header ook een volledige SOAP Evelope. De HTTP header geeft aan dat alles goed verlopen is (200 OK) en bepaalt het type van de message. In dit geval dus SOAP.

Voorbeeld 3.12 is een voorbeeld van een RPC ingebed in een SOAP message en voorbeeld 3.13 bevat de response SOAP message. RPC responses kunnen meerdere return waardes hebben zoals te zien in voorbeeld 3.13 waar naast de som ook het verschil wordt teruggegeven, dit laatste louter ter illustratie.

Voorbeeld 3.12.

```
POST /Computations HTTP/1.1
Host: calculation.org
Content-Type: application/soap+xml; charset="utf-8"
Content-Length: nnnn

<?xml version='1.0' ?>
<env:Envelope xmlns:env="http://www.w3.org/2003/05/soap-envelope">
 <env:Body>
  <calc:add
         env:encodingStyle="http://www.w3.org/2003/05/soap-encoding"
         xmlns:calc="http://calculation.org/">
   <m:term1 xmlns:m="http://math.com">
    <m:number>24</m:number>
```

```
   </m:term1>
   <m:term2 xmlns:m="http://math.com">
    <m:number>12</m:number>
   </m:term2>
  </calc:add>
 </env:Body>
</env:Envelope>
```

Voorbeeld 3.13.

```
HTTP/1.1 200 OK
Content-Type: application/soap+xml; charset=utf-8
Content-Length: nnnn

<?xml version='1.0' ?>
<env:Envelope xmlns:env="http://www.w3.org/2003/05/soap-envelope">
 <env:Body>
     <calc:addResponse
         env:encodingStyle="http://www.w3.org/2003/05/soap-encoding"
         xmlns:calc="http://calculation.org/">
       <m:sum>36</m:sum>
       <m:difference>12</m:difference>
     </calc:addResponse>
 </env:Body>
</env:Envelope>
```

Ook in voorbeeld 3.13 is de HTTP statuscode gelijk aan 200 wat aangeeft dat alles in orde is. Er zijn verschillende klassen van statuscode waaronder: 2xx: succesvol, 4xx: client error, 5xx: server error,... Voorbeeld 3.14 is een voorbeeld van zo'n foutmelding.

Voorbeeld 3.14.

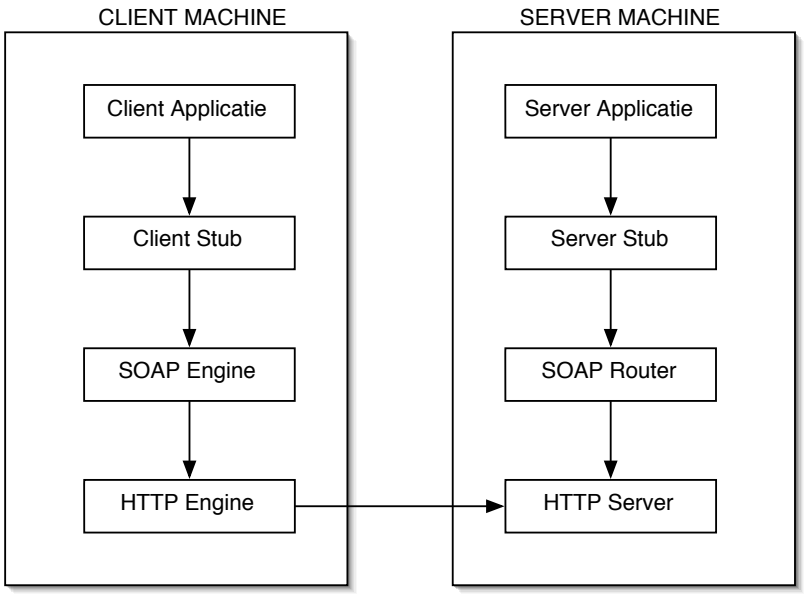
```
HTTP/1.1 500 Internal Server Error
Content-Type: application/soap+xml; charset="utf-8"
Content-Length: nnnn

<?xml version='1.0' ?>
<env:Envelope xmlns:env="http://www.w3.org/2003/05/soap-envelope"
              xmlns:calc="http://calculation.org">
  <env:Body>
   <env:Fault>
     <env:Code>
       <env:Value>env:Sender</env:Value>
       <env:Subcode>
        <env:Value>calc:BadArguments</env:Value>
       </env:Subcode>
     </env:Code>
     <env:Reason>
```

```
      <env:Text xml:lang="en-US">Not A Number</env:Text>
      <env:Text xml:lang="nl">Geen nummer</env:Text>
     </env:Reason>
     <env:Detail>
      <e:myFaultDetails
        xmlns:e="http://calculation.org/faults">
        <e:message>Term1 is not a number</e:message>
        <e:errorcode>004</e:errorcode>
      </e:myFaultDetails>
     </env:Detail>
   </env:Fault>
 </env:Body>
</env:Envelope>
```

3.8 SOAP implementatie

Figuur 3.3 is een voorbeeld van hoe SOAP kan geïmplementeerd worden:



Figuur 3.3: Mogelijke SOAP Implementatie

De verschillende stappen in een SOAP interactie aan de client zijde zijn:

1. De client applicatie wenst een remote applicatie op te roepen en doet hiervoor een beroep op de client stub

2. De client stub zorgt ervoor dat de local oproep van de client applicatie vertaald wordt naar een remote oproep.

3. Alle informatie wordt in een SOAP message gezet die dan weer verwerkt wordt in een HTTP message. Dit is de taak van de SOAP engine.

4. De HTTP engine verstuurt tot slot de message.

Voor de server is de procedure in feite hetzelfde maar dan omgekeerd:

1. De HTTP server ontvangt de HTTP message van de client en geeft deze door aan het SOAP systeem.

2. In de SOAP router wordt de HTTP informatie verwijderd, de inhoud wordt uit de SOAP message gehaald en de informatie wordt doorgegeven aan de juiste stub (= routing naar de juiste stub)

3. De Server stub zorgt ervoor dat de juiste server applicatie met de correcte data wordt opgeroepen.

4. De server applicatie voert de opdracht uit en kan op zijn beurt een reply message terugsturen.

3.9 Aanvaarding en toekomst van SOAP

SOAP is één van de meest gebruikte standaarden binnen het domein van web services. Toch zijn er nog verbeteringen mogelijk zoals standaarden voor het verzenden van binaire data, standaarden voor het aangeven van de te volgen route en de eindbestemming van SOAP messages, standaarden voor beveiliging van SOAP messages, ...

3.9.1 SOAP en binaire data

Het XML formaat is zeer handig voor het linken van totaal verschillende applicaties, of voor het linken van applicaties die data uitwisselen waarvoor er nog geen standaard data formaat bestaat.

Enige kritiek ten op zichte van het XML formaat is echter niet misplaatst. Het gebruik van XML is niet altijd een voordeel, soms introduceert het extra complexiteit en inefficiëntie. Bijvoorbeeld bij het versturen van binaire data zoals figuren is dit het geval. Er zijn hiervoor verschillende oplossingen zoals het gebruik van URLs of het Direct Internet Message Encapsulation protocol (DIME). [9]

3.9.2 Nog meer standaarden

Bovenop de bestaande standaarden zijn er momenteel nog enkele voorstellen tot standaard:

WS-Addressing [29] We hebben gezien dat SOAP geen mechanisme voorziet voor het adresseren maar dat daarvoor een beroep gedaan wordt op het gebruikte transport protocol. WS-Addressing is een SOAP extensie waarmee binnen in de SOAP message het adres van de bestemming kan gespecificeerd worden. Het specificeren van de bestemming gebeurt dan aan de hand van gestandaardiseerde header blocks.

WS-Routing [34] Daarnaast hebben we ook gezien dat de route die de SOAP messages afleggen bepaald wordt door het transport protocol. Users hebben dus in feite zeer weinig controle over de gevolgde route. WS-Routing, eveneens een SOAP extensie, biedt hiervoor een oplossing. Het voorstel is om een SOAP message path op te nemen in een gestandaardiseerde header block. Zo'n message path bestaat dan uit een opeenvolging van referenties naar SOAP nodes.

WS-Security [29] SOAP biedt geen enkele ondersteuning voor security. Dit wordt opgevangen door WS-Security, opnieuw een SOAP uitbreiding. WS-Security gebruikt binaire tokens voor de authentificatie, digitale handtekeningen voor de integriteit van de data en encryptie van de content om vertrouwelijke gegevens af te schermen. Ook hier wordt gebruik gemaakt van SOAP header blocks.

WS-Policy [29, 17] WS-Policy is een soort van onderhandelings framework tussen client een server. Beiden specificeren hun eisen aan de hand van zogenaamde "assertions". Een voorbeeld van een assertion is de eis tot authentificatie alvorens gebruik kan gemaakt worden van de services, WS-Policy biedt de mogelijk om deze assertions te groeperen. Er bestaan verschillende soorten groepen. De groep "all" geeft aan dat alle assertions geldig moeten zijn terwijl de groep "one" aangeeft dat één van de assertions voldaan moet zijn. Een Policy is een groep van assertions. WS-Policy voorziet dus een mechanisme voor het specificeren van policies. Het beschrijven van een assertion wordt overgelaten aan andere standaarden. WS-Policy werd ontworpen om samen te werken met WSDL in UDDI.

Hoofdstuk 4

WSDL: Web Service Description Language.

4.1 Wat is WSDL?

WSDL versie 1.1 [24] is het resultaat van een samenwerking tussen IBM, Microsoft en Ariba en is een W3C Note die net als SOAP gebaseerd is op XML. WSDL is een taal voor het beschrijven van web services. Ze definieert in zekere zin de structuur van de SOAP messages door de namen van de operaties, het aantal parameters, de parameter volgorde en dergelijken reeds van te voren vast te leggen. Zo'n beschrijving beantwoordt onder andere de volgende vragen:

1. Wat doet de web service?
2. Welke parameters moeten worden meegegeven?
3. Wat levert dit als resultaat op, en krijgen we hoegenaamd wel een resultaat?
4. Waar kunnen we de web service vinden?
5. Hoe moet de service opgeroepen worden?

Service descriptions spelen een zeer belangrijke rol in de reeds eerder besproken SOA architectuur, één van de basis concepten van web services.

4.2 Functie binnen webservices?

Het hoofddoel van WSDL is om te komen tot een automatisatie van de communicatie tussen applicaties. WSDL speelt min of meer dezelfde rol als de

interface definition language (IDL) in conventionele middleware platformen. (zie sectie 1.2.1) We plaatsen de twee tegenover elkaar:

IDL heeft als functie het beschrijven van interfaces. Deze beschrijvingen kunnen dan op hun beurt gebruikt worden voor de creatie van stubs. Deze stubs schermen dan de complexiteit van remote oproepen voor de gebruiker af.

WSDL wordt ook gebruikt voor het beschrijven van interfaces en deze beschrijvingen kunnen eveneens gebruikt worden voor het genereren van stubs. WSDL beschrijvingen zijn echter complexer dan IDL beschrijvingen wegens de volgende redenen:

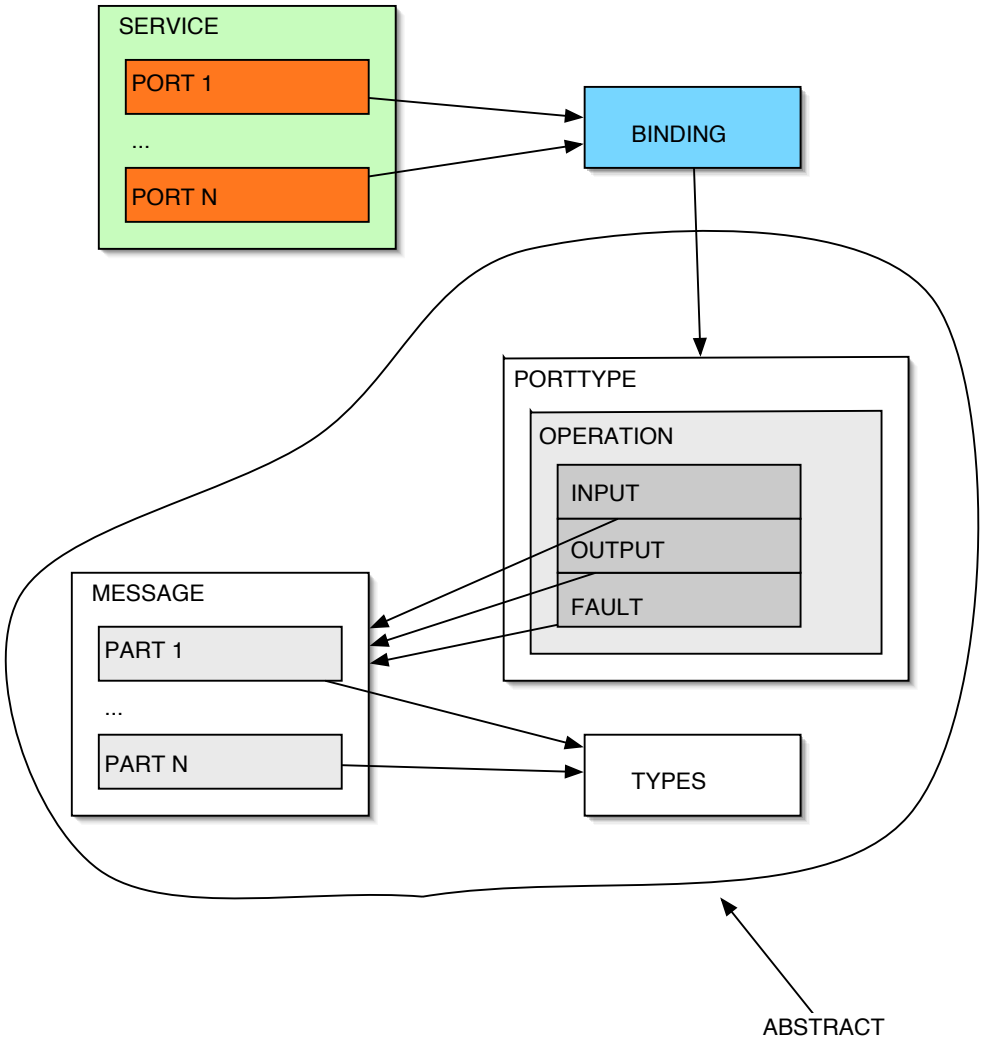
- Er moeten bindings of onderliggende transportprotocollen gespecificeerd worden wat bij traditionele middleware niet noodzakelijk is aangezien daar steeds hetzelfde, vooraf bekende, protocol gebruikt wordt.
- Bij traditionele middleware platformen zorgt de middleware ervoor dat de messages bij de juiste bestemming terecht komen, de locatie van de server is op die manier transparant voor de gebruiker. Web services beschikken niet over zo'n centraal orgaan, vandaar dat ook de locatie van de server gedefinieerd moet worden.

4.3 Opbouw van WSDL messages

De grammatica in tabel 4.1 geeft een overzicht van de structuur van een WSDL message. Hierbij staat "?" voor optionele elementen, "*" voor elementen die 0 of meerdere keren voorkomen en "+" voor elementen die 1 of meerdere keren voorkomen.

Het documentation element kan in elk WSDL element voorkomen en bevat informatie in een menselijk leesbaar formaat. De inhoud van een documentation element kan zowel tekst - als sub - elementen bevatten.

We kunnen een WSDL document in twee delen splitsen: een abstract deel en een concreet deel. Dit kan het best getoond worden aan de hand van figuur 4.1 waarbij de delen buiten de verzameling het concrete gedeelte voorstellen en de delen binnen de verzameling het abstracte en herbruikbare gedeelte voorstellen. De pijlen geven aan dat er referenties gebruikt worden.



Figuur 4.1: WSDL structuur

<definitions>	::=	<import>* <documentation>? <types>? <message>* <portType>* <binding>* <service>*
<types>	::=	<documentation>? <schema>*
<message>	::=	<documentation>? <part>*
<portType>	::=	<documentation>? <operation>*
<operation>	::=	<documentation>? <input>? <output>? <fault>*
<input>	::=	<documentation>?
<output>	::=	<documentation>?
<fault>	::=	<documentation>?
<binding>	::=	<documentation>? <operation>*
<service>	::=	<documentation>? <port>*
<port>	::=	<documentation>?

Tabel 4.1: structuur van een WSDL message

Alle elementen binnen de verzameling zijn abstract en kunnen bijgevolg door verschillende applicaties gedeeld worden. Elke applicatie kan een eigen concretisering van deze abstracte verzameling maken, a.d.h.v. de gekleurde elementen: service, port en binding.

De stappen die gezet moeten worden bij het ontwikkelen van een WSDL interface of anders gezegd het **abstracte deel** van de WSDL description zijn:

1. Het definiëren van de **data types**. Types worden meestal gedefinieerd aan de hand van XML Schema maar andere schema talen zijn ook toegelaten. Het definiëren van de types gebeurt in het types element. Hierin kunnen XML Schema documenten geplakt worden of er kan naar verwezen worden met import.

2. Het definiëren van de **messages** die gebruik maken van de types. Dit gebeurt in het message element. Elk message element is opgebouwd uit een aantal part elementen die oveenkomen met een bepaald data type.

3. Het definiëren van **operaties** in het operation element. Operaties hebben typisch een aantal parameters (input messages), een result waarde (output messages) en een aantal mogelijke excepties (fault messages). Er bestaan 4 verschillende basis operaties:

   (a) **One way**: deze interactie is beperkt tot het versturen van één message van de client naar de server.

(b) **Request-response**: hier worden twee messages uitgewisseld. De eerste message gaat van de client naar de server en de tweede van de server naar de client.

(c) **Solicit-response**: hier worden twee message uitgewisseld. De eerste message gaat van de server naar de client en de tweede van de client naar de server.

(d) **Notification**: deze interactie is beperkt tot het versturen van één message van de server naar de client.

De eerste en de laatste operaties worden gebruikt voor assynchrone interactie terwijl de middelste twee synchrone interactie voorzien.

4. Het definiëren van een interface door het groeperen van een aantal operations in een **portType** element.

Tot nu toe ging het over het abstracte deel van WSDL. Het concrete gedeelte van WSDL definieert onder andere: de gebruikte binding, de service definitie bestaande uit verschillende operaties, het adres waar de service te vinden is.

De ontwikkeling van het **concrete gedeelte** bestaat uit 3 stappen:

1. Het definiëren van de protocol binding en de encoding style voor alle messages in een gegeven portType. Dit gebeurt in het **binding** element. De gegevens die hier voorzien moeten worden zijn:

   (a) Message encoding style: literal encoding of SOAP encoding.

   (b) Interaction style: document style of RPC style.

   (c) Transport protocol: HTTP, SMTP, ...

   (d) Communication protocol: meestal SOAP. WSDL beperkt zich echter niet tot het SOAP communication protocol, ook andere protocollen kunnen gebruikt worden maar dit wordt zelden gedaan.

2. Het definiëren van de locatie van de web service gebeurt in het **port** element.

3. Het definiëren van services gebeurt in het **service** element. Typisch bestaat een service uit een aantal operaties. Elk service element bevat daarom een verzameling van port elementen.

Het opsplitsen van een WSDL bestand gebeurt in de praktijk zeer vaak. De verzameling van abstracte elementen noemt men ook wel de "Service Interface Definition" de concrete elementen krijgen de naam "Service Implementation Definition" toegekend. De verschillende documenten kunnen nadien

samengebracht worden via imports.

Met het import element is het mogelijk verschillende niveau's van abstractie op te splitsen om zo veel mogelijk hergebruik toe te staan. Op deze manier kunnen WSDL documenten modulair opgebouwd worden. Voorbeeld 4.1, 4.2 en 4.3 tonen een opsplitsing in drie files.

Voorbeeld 4.1. http://example.com/stockquote/stockquote.xsd

```
<?xml version="1.0"?>
<schema targetNamespace="http://example.com/stockquote/schemas"
       xmlns="http://www.w3.org/2000/10/XMLSchema">

    <element name="TradePriceRequest">
        <complexType>
            <all>
                <element name="tickerSymbol" type="string"/>
            </all>
        </complexType>
    </element>
    <element name="TradePrice">
        <complexType>
            <all>
                <element name="price" type="float"/>
            </all>
        </complexType>
    </element>
</schema>
```

Voorbeeld 4.2. http://example.com/stockquote/stockquote.wsdl

```
<?xml version="1.0"?>
<definitions name="StockQuote"

targetNamespace="http://example.com/stockquote/definitions"
          xmlns:tns="http://example.com/stockquote/definitions"
          xmlns:xsd1="http://example.com/stockquote/schemas"
          xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
          xmlns="http://schemas.xmlsoap.org/wsdl/">

   <import namespace="http://example.com/stockquote/schemas"
           location="http://example.com/stockquote/stockquote.xsd"/>

    <message name="GetLastTradePriceInput">
        <part name="body" element="xsd1:TradePriceRequest"/>
    </message>
```

```
    <message name="GetLastTradePriceOutput">
        <part name="body" element="xsd1:TradePrice"/>
    </message>

    <portType name="StockQuotePortType">
        <operation name="GetLastTradePrice">
           <input message="tns:GetLastTradePriceInput"/>
           <output message="tns:GetLastTradePriceOutput"/>
        </operation>
    </portType>
</definitions>
```

Voorbeeld 4.3. http://example.com/stockquote/stockquoteservice.wsdl

```
<?xml version="1.0"?>
<definitions name="StockQuote"

targetNamespace="http://example.com/stockquote/service"
          xmlns:tns="http://example.com/stockquote/service"
          xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
          xmlns:defs="http://example.com/stockquote/definitions"
          xmlns="http://schemas.xmlsoap.org/wsdl/">

   <import namespace="http://example.com/stockquote/definitions"
           location="http://example.com/stockquote/stockquote.wsdl"/>

    <binding name="StockQuoteSoapBinding" type="defs:StockQuotePortType">
        <soap:binding style="document" transport="http://schemas.xmlsoap.org/soap/http"/>
        <operation name="GetLastTradePrice">
           <soap:operation soapAction="http://example.com/GetLastTradePrice"/>
           <input>
               <soap:body use="literal"/>
           </input>
           <output>
               <soap:body use="literal"/>
           </output>
        </operation>
    </binding>

    <service name="StockQuoteService">
        <documentation>My first service</documentation>
        <port name="StockQuotePort" binding="tns:StockQuoteBinding">
           <soap:address location="http://example.com/stockquote"/>
        </port>
    </service>
</definitions>
```

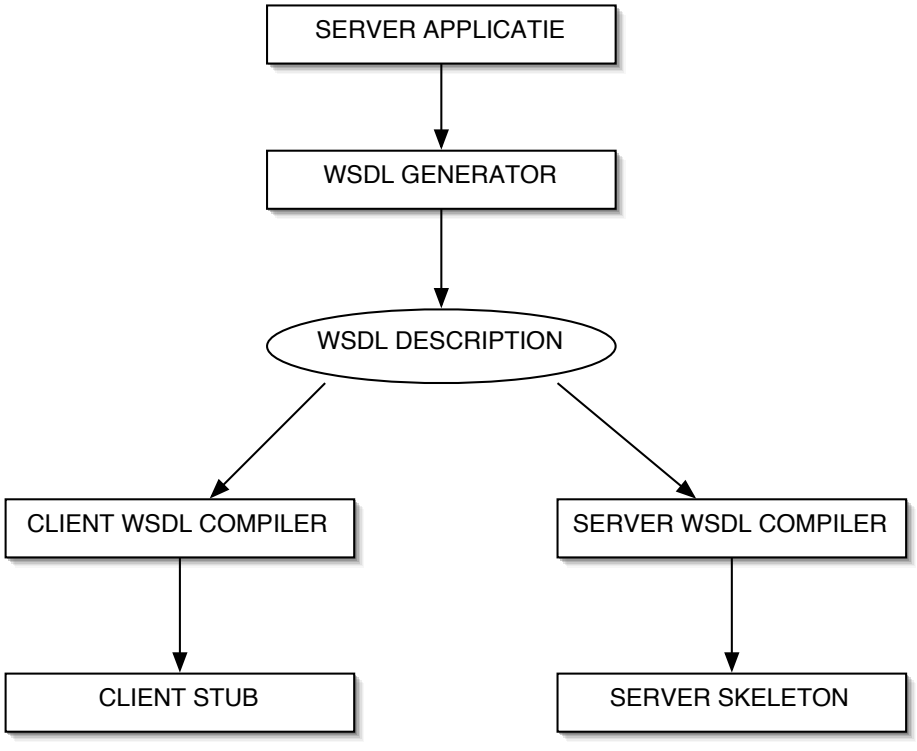
4.4 Het gebruik van WSDL

De W3C Web Services Description Working Group beschrijft 3 mogelijke gebruiksmanieren voor WSDL waarvan er momenteel slechts 2 gebruikt worden.

1. Als beschrijving voor web services.
2. Als basis voor stub compilers.
3. Voor het beschrijven van de semantiek.

WSDL documenten worden momenteel enkel gebruikt voor de eerste twee punten, het derde punt betreffende de beschrijving van semantiek wordt momenteel overgelaten aan andere specificaties.

Figuur 4.2 geeft aan hoe WSDL in de praktijk meestal gebruikt wordt:



Figuur 4.2: Het gebruik van WSDL

Verloop van bovenstaand proces:

1. De server wil een bepaalde applicatie aanbieden.
2. De WSDL generator genereert de WSDL description behorend bij de service.
3. Deze WSDL description wordt dan als input geleverd aan respectievelijk de Client WSDL Compiler en de Server WSDL Compiler.
4. Die er op hun beurt een Client stub en een Server skeleton mee genereren.
5. De Client stub kan dan gebruikt worden door de client applicatie en het skeleton door de server applicatie.

4.5 Aanvaarding en toekomst van WSDL

WSDL is vrij goed aanvaard alhoewel er heel wat alternatieven voor bestaan. Deze zijn meestal gericht op een specifiek applicatie domein zoals bijvoorbeeld het bankwezen. Het voordeel van dergelijke description languages is dat ze meer belang kunnen hechten aan de specifieke semantiek van dat domein. De prijs die WSDL betaalt voor zijn algemeenheid is dat het minder geschikt is voor het in detail beschrijven van services en voor het weergeven van semantiek.

Wie als uiteindelijke overwinnaar de strijd zal verlaten is nog onduidelijk. De toekomst zal uitwijzen of het nu WSDL, een van de alternatieve talen of een combinatie zal worden.

Hoofdstuk 5

UDDI: Universal Description, Discovery and Integration.

5.1 Wat is UDDI

UDDI versie 3 [22] werd ontwikkeld in samenwerking tussen Ariba, IBM en Microsoft en is een formele OASIS standaard terug te vinden op: [4]. UDDI is op XML gebaseerd en specificeert een framework voor het beschrijven en ontdekken van web services.

5.2 Functie binnen web services

Tot nu toe hebben we gezien welke standaarden gebruikt worden voor de communicatie tussen web services en voor het beschrijven van web services. Vooralleer gebruik kan gemaakt worden van een bepaalde service moet je eerst de locatie en de manier van oproepen van de services kennen. Dit alles maakt deel uit van "Service Discovery". Zoals reeds eerder vermeld werd is Service discovery een zeer belangrijk onderdeel van de SOA Architectuur.

De taak van UDDI bestaat uit 3 delen:

1. Het beschrijven van web services, (zie sectie 5.3)
2. Dynamic binding mogelijk maken, (zie sectie 5.4)
3. Ervoor zorgen dat mogelijke clients de services kunnen vinden. (zie sectie 5.5)

5.3 Het beschrijven van web services

5.3.1 Welke informatie wordt er beschreven?

De informatie die in een UDDI registry wordt bijgehouden is op te splitsen in 3 groepen:

1. **White Pages**: hierin worden contactgegevens zoals: namen, mailadressen, telefoonnummers, ... in verband met de provider van de service bijgehouden. Deze informatie kan gebruikt worden om de web services van een gegeven business terug te vinden.

2. **Yellow Pages**: bevat classificatie informatie omtrent de types en de locaties van de services die door de business aangeboden worden. Clients kunnen op zoek gaan naar services binnen een gegeven categorie- en classificatie-systeem.

3. **Green Pages**: bevat de details over hoe de aangeboden services opgeroepen moeten worden. Deze gegevens worden verstrekt door middel van referenties naar documenten die typisch buiten het registry bijgehouden worden.

5.3.2 Hoe wordt deze informatie bijgehouden?

Een UDDI registry maakt gebruik van 5 datatypes:

1. **businessEntity**: bevat informatie over het bedrijf en kan referenties bevatten naar één of meerdere businessService elementen.

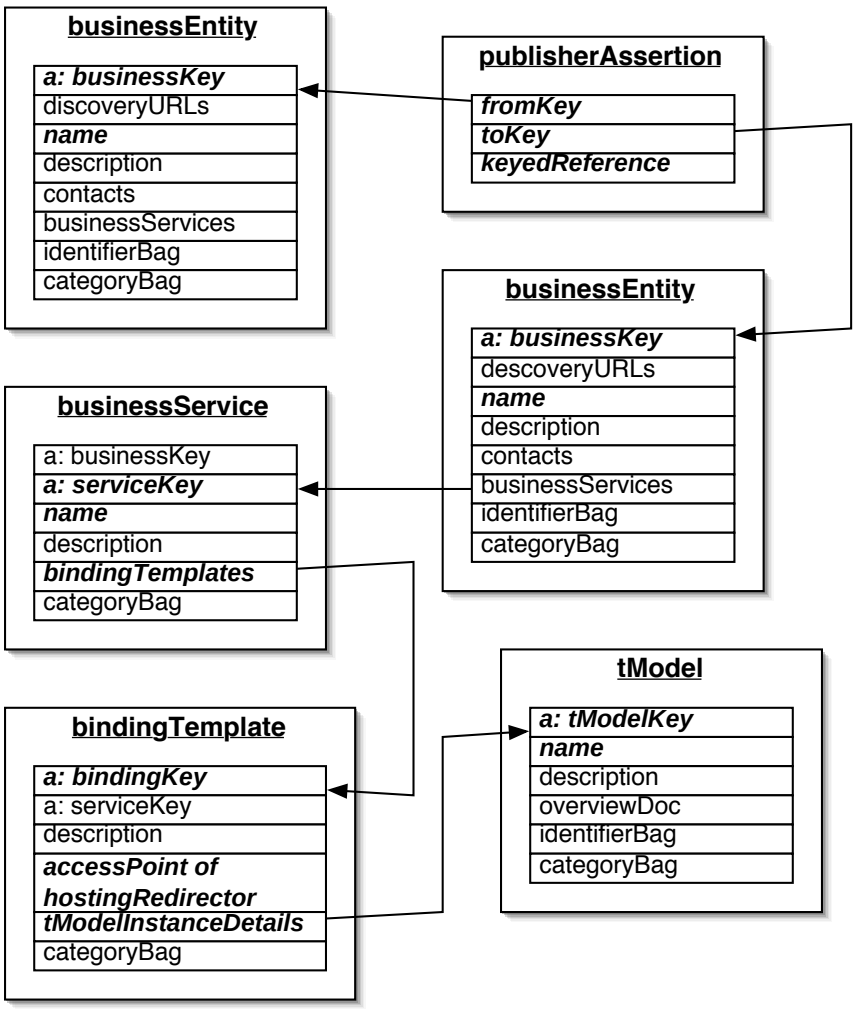
2. **businessService**: beschrijft een groep van verwante services aangeboden door een businessEntity. Dit element bevat ook één of meerdere bindingTemplate elementen.

3. **bindingTemplate**: bevat de technische informatie die nodig is voor het oproepen van een service. Deze informatie bestaat uit: de locatie van de service, de te gebruiken protocollen, de encoding style, ... Sommige van deze gegevens kunnen voorzien worden aan de hand van één of meerdere tModel elementen.

4. **tModel**: staat voor "technical model" en definieert de technische specificaties van een service. Daarnaast kunnen tModels ook gebruikt worden voor classificatie en identificatie.

5. **publisherAssertion**: voor het definiëren van relaties tussen businesEntity elementen.

De structuur van UDDI data kan grafisch voorgesteld worden zoals in figuur 5.1:



Figuur 5.1: UDDI structuur

Figuur 5.1 is afgeleid uit het XML Schema bestand dat terug te vinden is op: [1]. De in vet en cursief gedrukte woorden stellen verplichte elementen of attributen (regels beginnend met a:) voor. Zo moet een tModel element bijvoorbeeld steeds beschikken over een tModelKey attribuut en een name

element. De pijlen stellen referenties voor. Zo verwijst de fromKey van het publisherAssertion element naar de businessKey van het businessEntity element.

Hoe de UDDI informatie intern bewaard wordt door de UDDI Registries is niet gedefinieerd. Wat wel zeker is is dat de tModels als aparte entiteiten bijgehouden worden. Men verwacht namelijk dat naarmate er steeds meer services worden aangeboden dat er ook steeds meer services zullen zijn die gebruik maken van dezelfde tModels. Ook standaard interfaces kunnen op deze manier gedefinieerd worden. Het werken met tModels en referenties heeft twee voordelen:

1. tModels komen hoogstens 1 maal voor waardoor de opslagcapaciteit beter benut kan worden.

2. Er kan naar services gezocht worden die gebruik maken van een bepaald tModel.

In de volgende secties geven we een stap voor stap beschrijving van de UDDI datatypes te beginnen met het tModel.

tModel

tModels worden gebruikt als classificatie systeem voor businesses en services. Daarnaast worden tModels ook gebruikt om technische gegevens zoals het message formaat, het transport protocol of de gebruikte interface te specificeren.

Categorisatie en Identificatie Eén van de doelstellingen van UDDI registries is dat je services kan ontdekken. Als je een bepaalde zoekopdracht start krijg je een hele hoop resultaten. UDDI voorziet verschillende mechanismen om dit aantal te beperken:

- **Categorisatie**: het proces waarbij categorieën gecreëerd worden.
- **Classificatie**: het proces waarbij objecten toegekend worden aan voorgedefinieerde categorieën.
- **Identificatie**: het proces waarbij businesses en tModels zich op een unieke manier kunnen identificeren.

UDDI beschikt over een verzameling ingebouwde classificatie schema's waaronder o.a.:

- Het **North American Industry Classification System** (NAICS): voor het classificeren van bedrijven per industrietak. Meer informatie hierover is terug te vinden op: [3].
- **Universal Standard Product and Services Classification** (UNSPSC): is een classificatie systeem voor producten en services. Meer informatie is terug te vinden op: [6].
- De **ISO 3166 Standaard**: voor classificatie in functie van geografische locatie. Meer informatie is terug te vinden op: [2].

Elk van deze taxonomieën beschikt over een voorgedefinieerde tModel key:

NAICS:	uddi:ubr.uddi.org:categorization:naics:1997
UNSPSC:	uddi:ubr.uddi.org:categorization:unspsc
ISO 3166:	uddi:ubr.uddi.org:categorization:geo3166-2

Tabel 5.1: tModelKeys

Een tModel kan dus gebruikt worden om UDDI datatypes in categorieën in te delen. Het tModel komt dan overeen met een bepaald categorisatie systeem. Dit kan één van de 3 bovenstaande systemen zijn maar classificatie kan ook t.o.v. een eigen categorisatie systeem gebeuren. Alle entiteiten die beschikken over een categoryBag element kunnen geclassificeerd worden. Zie figuur 5.1.

Een onderneming die badminton artikelen verkoopt in Colorado kan dan gebruik maken van het categoryBag element in voorbeeld 5.1.

Voorbeeld 5.1.

```
<categoryBag>
  <keyedReference keyName="Sporting and Athletic Goods Manufacturing"
      keyValue="339920"
      tModelKey="uddi:ubr.uddi.org:categorization:naics:1997"/>
  <keyedReference keyName="Colorado"
      keyValue="US-CO"
      tModelKey="uddi:ubr.uddi.org:categorization:geo3166-2"/>
</categoryBag>
```

De relevante classificatie informatie wordt voorzien in het **categoryBag** element. Dit element bevat een verzameling van **keyedReference** elementen die op hun beurt drie attributen bevatten: **tModelKey**, **keyName** en **keyValue**. Het tModelKey attribuut specificeert het categorisatie systeem en

doet tegelijkertijd dienst als namespace voor de andere twee attributen. Het keyValue element bevat de categorie naam en het optionele keyName element tracht een beschrijvende naam van de categorie te geven.

Voorbeeld 5.1 maakt zowel gebruik van het NAICS als het ISO 3166 classificatie systeem. Het eerste keyedReference element geeft de industrietak van het bedrijf weer aan de hand van het keyValue "339920". Het keyName element verduidelijkt deze waarde door middel van de korte tekst: "Sporting and Athletic Goods Manufacturing". Het tweede keyedReference element maakt gebruik van ISO 3166 en geeft aan dat de business zich in Colorado bevindt door het keyValue "US-CO".

Andere categorisatie schema's kunnen worden toegevoegd aan een UDDI registry door middel van tModels. We kunnen daartoe het tModel in voorbeeld 5.2 publiceren.

Voorbeeld 5.2.

```
<tModel tModelKey="uuid:3D4EC875-9CBF-E45F-84DD-346D48BCAD9C">
  <name>Yahoo Business Taxonomy</name>
  <description xml:lang="nl">Dit is de Yahoo Business Taxonomy</description>
  <categoryBag>
    <keyedReference
      keyValue="categorization"
      tModelKey="uddi:uddi.org:categorization:types"/>
  </categoryBag>
</tModel>
```

Merk op dat ook tModels zelf geclassificeerd kunnen worden t.o.v. andere tModels. Het tModel met tModelKey uuid:3D4EC875-9CBF-E45F-84DD-346D48BCAD9C wordt hier geclassificeerd t.o.v. het tModel met tModelKey uddi:uddi.org:categorization:types. De waarde van het keyValue attribuut is hier gelijk aan "categorization" en geeft daarmee aan dat het tModel met tModelKey uuid:3D4EC875-9CBF-E45F-84DD-346D48BCAD9C gebruikt kan worden om andere datatypes te classificeren. Bovenstaande sport handel kan zijn categoryBag element dan uitbreiden op de manier zoals weergegeven in voorbeeld 5.3

Voorbeeld 5.3.

```
<categoryBag>
  <keyedReference keyName="Sporting and Athletic Goods Manufacturing"
      keyValue="339920"
      tModelKey="uddi:ubr.uddi.org:categorization:naics:1997"/>
```

```
  <keyedReference keyName="Colorado"
      keyValue="US-CO"
      tModelKey="uddi:ubr.uddi.org:categorization:geo3166-2"/>
  <keyedReference
      keyValue="business_and_economy/shopping_and_services/
                sports/badminton/gear_and_equipment/"
      tModelKey="uuid:3D4EC875-9CBF-E45F-84DD-346D48BCAD9C"/>
</categoryBag>
```

Het categoryBag element heeft er een nieuw keyedReference element bijgekregen. Het nieuwe element kent onze onderneming toe aan de categorie business_and_economy/shopping_and_services/sports/badminton/gear_and_equipment/ volgens het classificatie systeem van het tModel in voorbeeld 5.2.

Naast een categorisatie mechanisme beschikt UDDI ook over een identificatie systeem waarmee ondernemingen en tModels zich kunnen identificeren. Dit gebeurt in het identifierBag element. Ondernemingen en tModels kunnen geassocieerd worden met een gedeclareerd identificatie schema. Ook in een identifierBag element komen één of meerdere keyedReference elementen voor. Een voorbeeld van zo'n identificatie systeem in België is het BTW nummer of een handelsregister nummer. Elke onderneming heeft een uniek nummer en kan hiermee geïdentificeerd worden.

Naast enkele standaard identificatie schema's zoals het Amerikaanse DUNS nummer kunnen ook hier eigen schema's gedefinieerd en gepubliceerd worden. Voorbeeld 5.4 illustreert dit.

Voorbeeld 5.4.

```
<tModel tModelKey="uuid:44444444-9CBF-E45F-84DD-346D48BCAD9C">
  <name>BTW nummer</name>
  <description xml:lang="nl">
        Dit is het BTW nummer van de onderneming
  </description>
  <categoryBag>
    <keyedReference keyName="BTW-Nummer"
      keyValue="identifier"
      tModelKey="uddi:uddi.org:categorization:types"/>
  </categoryBag>
</tModel>
```

Net zoals hierboven wordt dit tModel opnieuw geclassificeerd ten opzicht van een ander tModel. Deze keer is de klasse niet categorization maar identifier. Daarmee wordt aangegeven dat het om een identificatie systeem gaat.

Voorbeeld 5.5 toont een identificatie aan de hand van het BTW nummer:

Voorbeeld 5.5.

```
<identifierBag>
  <keyedReference keyName="Mijn BTW Nummer"
    keyValue="123.456.789"
    tModelKey="uuid:44444444-9CBF-E45F-84DD-346D48BCAD9C"/>
</identifierBag>
```

Figuur 5.1 geeft aan in welke UDDI elementen het identifierBag en het categoryBag element kunnen voorkomen.

Specificatie van technische gegevens Bij de bespreking van WSDL hebben we gesproken over het feit dat WSDL documenten vaak opgesplitst worden in een abstract en herbruikbaar deel enerzijds en een concreet deel anderzijds. Het abstracte deel komt overeen met de interface en wordt ook wel "service interface description" genoemd terwijl het concrete deel de naam "service implementation definition" krijgt.

Deze opsplitsing heeft ook nut in de context van UDDI. Meer bepaald is het dankzij deze opsplitsing mogelijk om via referenties naar service interface descriptions op een dynamische manier op zoek te gaan naar services die een bepaalde interface implementeren.

Om aan te geven dat een service conform is met een bepaalde interface wordt in het bindingTemplate element een tModel voorzien dat refereert naar een WSDL document, meerbepaald een "service interface description".

UDDI ondersteunt naast WSDL verschillende types van Service description. Een web service die geregistreerd is in een UDDI Registry als een BusinessService kan beschreven worden in:

1. WSDL
2. ASCII tekst
3. RosettaNet pip
4. RDF
5. . . .

Zoals reeds eerder vermeld kunnen tModels gebruikt worden voor het uitdrukken van verschillende technische gegevens. In voorbeeld 5.6 wordt een tModel gebruikt voor het specificeren van een interface.

Voorbeeld 5.6.

```
<tModel tModelKey="uuid:12345678-ABCD-ABCD-ABCD-123456789123">
  <name>Bestel service</name>
  <description xml:lang="nl">
      Service interface definitie voor de bestel service
  </description>
  <overviewDoc>
    <description xml:lang="nl">
        Referentie naar een WSDL document dat de service definition bevat
    </description>
    <overviewURL useType="wsdlInterface">
      http://www.mijnOnderneming.be/bestelServiceInterface.wsdl
    </overviewURL>
  </overviewDoc>
  <overviewDoc>
    <overviewURL useType="text">
      http://www.mijnOnderneming.be/bestelServiceInterface.html
    </overviewURL>
  </overviewDoc>

  <categoryBag>
    <keyedReference keyName="uddi-org:types:wsdl"
        keyValue="wsdlSpec"
        tModelKey="uddi:uddi.org:categorization:types"/>
  </categoryBag>
</tModel>
```

UDDI legt geen regels op voor het beschrijven van interfaces, properties of semantiek. Zoals hierboven te zien is kan dit zowel met WSDL, html of om het even welk ander formaat gebeuren. Op deze manier is UDDI meer flexibel en tegelijk minder gevoelig voor evoluties. Het categoryBag element geeft aan dat dit tModel van het type wsdlSpec is.

Publisher Assertion

Met behulp van Publisher Assertions kunnen twee businessEntity elementen met elkaar in verband gebracht worden. Het **fromKey** element bevat de eerste businessEntity, het **toKey** element bevat de tweede businessEntity. Het soort relatie wordt aangegeven d.m.v. het **keyedReference** element. Dit element bevat 3 attributen: tModelKey, keyName en keyValue. TModelKey refereert naar het relationship type system om aan te geven dat hier een

relatie beschreven wordt. De keyName en keyValue attributen specificeren het type relatie. Enkele voorgedefinieerde keyValues zijn:

- **Parent-child**: De fromKey businessEntity is de parent van de businessEntity in de toKey.
- **Peer-peer**: De businessEntity in de fromKey is een gelijke (peer) van de businessEntity in de toKey.
- **Identity**: De businessEntity in de fromKey is dezelfde organisatie als die van de toKey.

Stel we hebben twee businessEntity elementen waarvan de ene onderneming een soort dochteronderneming is van de andere (Parent-child). Beide ondernemingen beschikken over een eigen businesskey:

- Parent onderneming: 1111111-AAAA-AAAA-AAAA-111111111111
- Dochter onderneming: 22222222-BBBB-BBBB-BBBB-222222222222

Om de parent-child relatie kenbaar te maken wordt de PublisherAssertion uit voorbeeld 5.7 gepubliceerd.

Voorbeeld 5.7.

```
<publisherAssertion>
  <fromKey>
    1111111-AAAA-AAAA-AAAA-111111111111
  </fromKey>
  <toKey>
    22222222-BBBB-BBBB-BBBB-222222222222
  </toKey>
  <keyedReference keyname="subsidiary"
      keyValue="parent-child"
      tModelKey="uddi:uddi.org:relationships"/>
</publisherAssertion>
```

BusinessEntity

Een BusinessEntity element bevat informatie over de business en over de services die deze business aanbiedt. Voorbeeld 5.8 bevat een voorbeeld van een BusinessEntity element:

Voorbeeld 5.8.

```
<businessEntity businessKey="12312312-123E-12D9-C874-010235CD2A12">
  <discoveryURLs>
    <discoveryURL useType=homepage>http://www.bedrijfX.be</discoveryURL>
```

```
    ...
  <discoveryURLs>
  <name>Bedrijf X</name>
  <description>UDDI businessEntity van Bedrijf X</description>
  <contacts>
    <contact>
      <personName>Christof Delsupehe</personName>
      <email>christof.delsupehe@student.luc.ac.be</email>
      <address>
        <addressLine>Stenenbrug 18</addressLine>
        <addressLine>3550 Zolder</addressLine>
        <addressLine>Belgium</addressLine>
      </address>
    <contact>
    ...
  </contacts>
  <businessServices>
    <businessService ... />
    ...
  </businessServices>
  <identifierBag> ... </identifierBag>
  <categoryBag> ... </categoryBag>
</businessEntity>
```

De identifierBag en categoryBag elementen hebben we reeds besproken. Het BusinessService element wordt in de nu volgende tekst nader toegelicht.

BusinessService

Een BusinessService element beschrijft een verzameling van verwante web services aangeboden door een organisatie beschreven in een BusinessEntity element. Voorbeeld 5.9 bevat een voorbeeld van een BusinessService element:

Voorbeeld 5.9.

```
<businessService serviceKey="12AB34CD-1234-4321-ABCD-1B3D5A789A1"
    businessKey="12312312-123E-12D9-C874-010235CD2A12">
  <name>Bestel Service</name>
  <description>Bestel Service van bedrijfX</description>
  <bindingTemplates>
    <bindingTemplate ... />
    ...
  </bindingTemplates>
  <categoryBag> ... </categoryBag>
</businessService>
```

Het businessKey attribuut van dit BusinessService element komt overeen met de businessKey uit voorbeeld 5.8. Het bindingTemplate element wordt in de nu volgende tekst nader toegelicht.

BindingTemplate

Een BindingTemplate element beschrijft de technische informatie die nodig is voor het gebruik van de web service. Voorbeeld 5.10 bevat een voorbeeld van een BindingTemplate element:

Voorbeeld 5.10.

```
<bindingTemplate bindingKey="4323BA34-5634-4657-A435-567234BAC23"
    serviceKey="12AB34CD-1234-4321-ABCD-1B3D5A789A1">
  <description>SOAP gebaseerde bestel service</description>
  <accessPoint>http://www.bedrijfX.be/bestelService</accessPoint>
  <tModelInstanceDetails>
    <tModelInstanceInfo
        tModelKey="uuid:12345678-ABCD-ABCD-ABCD-123456789123">
      <description>
          Referentie naar een tModel web service interface definitie
      </description>
    </tModelInstanceInfo>
  </tModelInstanceDetails>
  <categoryBag> ... </categoryBag>
</bindingTemplate>
```

Het serviceKey attribuut van dit bindingTemplate element komt overeen met de businessKey uit voorbeeld 5.9. Het tModelInstanceInfo element bevat in dit voorbeeld een referentie, via het tModelKey attribuut, naar het tmodel uit voorbeeld 5.6.

5.4 Dynamic Binding

TModels kunnen zoals gezegd gebruikt worden voor het classificeren en identificeren van bedrijven en services maar ook voor het definiëren van technische informatie. Dankzij tModels wordt dynamic binding mogelijk. Elk tModel beschikt over een unieke id waar naar gerefereerd kan worden. Stel dat een tModel met id gelijk aan 123 een bepaalde standaard interface definieert. Dan kan een client op een dynamische manier binden met een service provider door “at runtime” op zoek te gaan naar services in het UDDI registry die in het bindingTemplate gebruik maken van het tModel met id gelijk aan 123. Dynamische binding kan ook gebruik maken van geografische locatie, industrietak of eender welke andere classificatie van bedrijven of services.

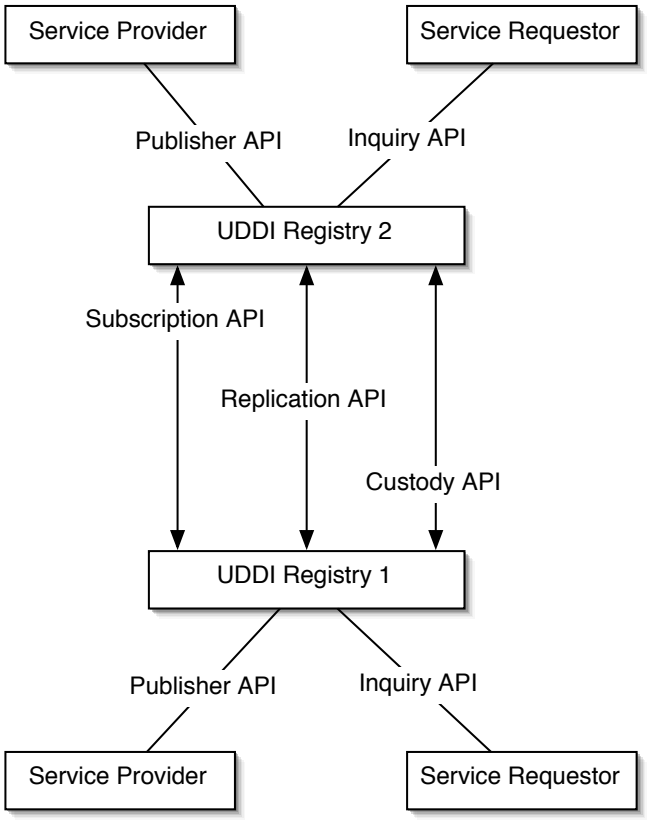
5.5 UDDI Registries en Service Discovery

UDDI definieert data structuren en API’s voor het publiceren van service descriptions in het registry en voor het ondervragen van dit registry op zoek

naar bepaalde gepubliceerde descriptions. De UDDI API's zijn zelf ook web services, elk met een eigen WSDL description.

5.5.1 UDDI API's

Er zijn drie soorten users die interageren met een UDDI Registry. Figuur 5.2 geeft aan wie deze users zijn en welke API's er voor hen beschikbaar zijn:



Figuur 5.2: UDDI API's

De drie partijen waarmee gecommuniceerd wordt zijn: de service requestors, de service providers en andere UDDI registries. Communicatie tussen deze partijen gebeurt via API's.

Er bestaan zes soorten API's: [22]

1. **Inquiry API**: is gedefinieerd tussen de service requester en de UDDI registry en wordt gebruikt voor het opvragen van informatie.

2. **Publisher API**: is gedefinieerd tussen de service provider en de UDDI registry en wordt gebruikt voor het toevoegen, verwijderen en aanpassen van entiteiten in het registry.

3. **Security API**: zorgt voor de uitwisseling van de "authentication tokens" nodig voor communicatie met het UDDI registry. Authentication tokens zijn, afhankelijk van de gebruikte authorization policy, noodzakelijk voor het gebruik van de inquiry API, de publisher API, de custody en ownership API en de subscription API.

4. **Custody en ownership API**: Elke entiteit staat onder de "voogdij" van een bepaalde UDDI registry en is eigendom van een bepaalde service provider. De Custody en ownership API maakt het mogelijk om zowel de eigendom als de voogdij over bepaalde entiteiten aan te passen.

5. **Subscription API**: clients kunnen zich subscriben voor wijzigingen van entiteiten binnen een registry zodat ze verwittigd worden indien er veranderingen plaatsvinden.

6. **Replication API**: zorgt voor de synchronisatie tussen twee of meerdere UDDI registries.

Een groep van web services die minstens één van bovenstaande API's implementeert noemt men een UDDI node. Een UDDI registry bestaat uit een groepering van één of meerdere UDDI nodes en elke UDDI node behoort tot precies één UDDI registry. Het zijn de UDDI nodes die via de API's instaan voor de interactie met de UDDI data.

Afhankelijk van de gebruikte API wordt een ander access point van het UDDI Registry gebruikt. Eén van de redenen hiervoor is dat de verschillende API's ook door verschillende UDDI nodes, met elk hun eigen security, geïmplementeerd kunnen worden. Zo kan het bijvoorbeeld zijn dat de inquiry API geen authentication vereist terwijl dat voor de publisher API wel noodzakelijk is.

5.5.2 Toegang tot de API's

De authorization policy van een registry definieert hoe toegangscontrole, indien aanwezig, door het registry geïmplementeerd wordt. Aangezien toegangscontrole een belangrijk gegeven is binnen UDDI voorziet UDDI in de publication, inquiry en subscription API's het optionele authInfo element.

AuthInfo is van het type String en staat in voor de eventuele authentication.

AuthInfo elementen kunnen bekomen worden aan de hand van de volgende operaties die deel uit maken van de **Security API**:

- **Get_authToken(userID, cred)**: voor het opvragen van een authentication token in de vorm van een authInfo element. Get_authToken vereist twee parameters: een userID en een wachtwoord om de user te kunnen identificeren.
- **Discard_authToken(authInfo)**: om aan te geven dat het als parameter meegegeven token niet langer geldig is.

5.5.3 Hoe kan informatie toegevoegd worden?

Het verkregen authentication token kan daarna eventueel als parameter meegegeven worden aan de **Publisher API** die de volgende services aanbiedt:

- add_publisherAssertions
- delete_binding
- delete_business
- delete_publisherAssertions
- delete_service
- delete_tModel
- get_assertionStatusReport
- get_publisherAssertions
- get_registeredInfo
- save_binding
- save_business
- save_service
- save_tModel
- set_publisherAssertions

In bijlage A bevindt zich een beknopte beschrijving van de operaties aangeboden door de Publisher API.

Bij het publiceren van entiteiten in een UDDI registry gelden de volgende regels in verband met eigendom en voogdij:

- De service provider die de betreffende entiteit voor het eerst gepubliceerd heeft is tevens de eigenaar van die entiteit. Dit houdt in dat eventuele updates of deletes enkel door hem, de oorspronkelijke service provider, kunnen uitgevoerd worden.
- Bijkomend geldt ook dat de service registry waar de entiteit voor het eerst gepubliceerd werd de voogdij krijgt over deze entiteit. Dit wil zeggen dat eventuele updates of deletes enkel kunnen gebeuren bij de registry waar de entiteit voor het eerst gepubliceerd werd.

Deze gegevens betreffende eigendom en voogdij kunnen echter wel aangepast worden. Hiervoor wordt dan gebruik gemaakt van de **Custody en ownership API** die de volgende functies aanbiedt:

- get_transferToken
- transfer_entities
- transfer_custody

In bijlage B bevindt zich een beknopte beschrijving van de operaties aangeboden door de Custody en ownership API.

5.5.4 Hoe kan informatie opgevraagd worden?

Voor het opvragen van informatie wordt gebruik gemaakt van de **Inquiry API**. Het aantal operaties aanwezig in de Inquiry API is eerder beperkt. Het staat natuurlijk elke Registry vrij om bijkomende services aan te bieden, al dan niet tegen betaling. Een voorbeeld van dergelijke services is een query die het mogelijk maakt om in het registry op zoek te gaan naar de service met de kleinste vertraging. Deze informatie wordt standaard niet door UDDI aangeboden maar kan dus als extra geïmplementeerd worden.

De volgende operaties moeten standaard in elk UDDI conform registry aanwezig zijn:

- Find_binding

- Find_business
- Find_relatedBusinesses
- Find_service
- Find_tModel
- Get_bindingDetail
- Get_businessDetail
- Get_operationalInfo
- Get_serviceDetail
- Get_tModelDetail

In bijlage C bevindt zich een beknopte beschrijving van de operaties aangeboden door de Inquiry API.

5.5.5 Subscription API

De **Subscription API** biedt clients in dit geval ook wel "subscribers" genoemd de mogelijkheid om zich te subscriben voor bepaalde veranderingen in het UDDI Registry zoals het updaten, deleten of toevoegen van nieuwe entiteiten. De subscribers definiëren aan de hand van de find en get operaties uit de vorige API's in welk type wijzigingen ze geïnteresseerd zijn.

Er zijn twee soorten subscription patronen:

Assynchronous notification: Bij Assynchronous notification krijgen subscribers een melding van nieuwe, aangepaste of verwijderde entities relatief ten opzichte van het moment waarop de subscription plaatsvond. Deze melding kan ofwel per mail gebeuren ofwel implementeren de subscribers zelf een eigen webservice waaraan dergelijke gebeurtenissen kunnen gemeld worden. Meldingen gebeuren periodisch in plaats van op het moment dat de effectieve wijziging gebeurt. Subscribers kunnen de frequentie van notification aanpassen door zelf het begin en einde van een periode vast te leggen.

Synchronous change tracking: verloopt als volgt:

1. De subscriber geeft aan in welke entiteiten hij geïnteresseerd is.

2. Het UDDI Registry geeft die entiteiten terug die in de gespecificeerde tijdsperiode aangepast werden.

Dus bij Synchronous change tracking krijgt de subscriber onmiddellijk de veranderingen van zijn interesse die plaatsgrepen binnen het gedefinieerde tijdsinterval. Bij assynchrone notification vermeldt de subscriber waarin hij geïnteresseerd is en wordt na x aantal tijd gemeld welke entiteiten van zijn interesse er eventueel aangepast werden.

5.5.6 Replication API

De **Replication API** tot slot staat in voor de consistentie tussen verschillende registries en gebruikt hiervoor de volgende operaties:

- Delete_subscription
- Get_subscriptionResults
- Get_subscriptions
- Save_subscriptions

In bijlage D bevindt zich een beknopte beschrijving van de operaties aangeboden door de Replication API.

Replication van data is vooral nuttig in situaties waarbij een registry niet toegankelijk is. Het "clonen" van registries gebeurt meestal op een vast tijdstip eens per dag. Merk wel op dat zoekopdrachten zoals find_service niet tussen de verschillende registries geforward worden.

5.6 Aanvaarding en toekomst van UDDI.

Er bestaan twee types UDDI registries:

1. Public Registry of UDDI Business Registry (UBR)
2. Private Registry

UBRs worden gehost door een klein aantal bedrijven waaronder o.a. IBM en Microsoft. Dit werd echter geen echt succes wegens een gebrek aan vertrouwen. Het gaat dan zowel over vertrouwen in IBM, Microsoft of andere aanbieders van UBRs als het vertrouwen in de bedrijven die gebruik zullen maken van de services. (Je kan je services wel publiekelijk aanbieden maar wie zegt dat een gebruiker van de service wel zo'n goede bedoelingen heeft) Enkele voorbeelden van UBR zijn terug te vinden op:

- http://uddi.microsoft.com/default.aspx
- https://uddi.ibm.com/ubr/registry.html

De private registries daarentegen werden wel een succes en zij kunnen gehost zijn op een intranet of het Internet.

5.6.1 Public Registry

Een grote vraag wat betreft public registries is of de voorziene data wel correct is. Er is namelijk weinig controle op de data die aan het registry toegevoegd wordt. Wie zegt dat er wel degelijk een service terug te vinden is op een bepaalde URL, klopt die geografische informatie wel?, ... UDDI zelf voorziet enkel zeer eenvoudige validatie systemen voor het controleren van categorie informatie. Verdere controle op correctheid van data kan geïmplementeerd worden door anderen. Maar voorlopig is er nog geen mechanisme op de markt dat de inkomende gegevens in voldoende mate controleert.

Ook de rechtsgeldigheid van over internet gesloten deals, anders dan via email, is nog niet vastgelegd. Is dit even rechtsgeldig als een ondertekend contract of als een verzonden email?

Dit zijn heel wat openstaande vragen die we in de toekomst misschien beantwoord zullen zien.

5.6.2 Private Registry

Bij interne registries doen deze vragen zich niet voor aangezien op voorhand geweten is wie de services aanbiedt en wie ze zal gebruiken. De mate van gebruik en de kans op succes liggen hier dan ook veel hoger.

5.6.3 Dynamic Binding

Dynamic binding is een veel belovende en geavanceerde techniek. In de praktijk wordt dit echter maar weinig gebruikt aangezien het gaat over interacties tussen verschillende bedrijven. Meestal moeten eerst contracten afgesloten worden om via authentificatie toegang te krijgen tot bepaalde services. Het idee dat bedrijven hun services kosteloos zouden openstellen voor het grote publiek is weinig realistisch.

De mogelijkheden wat betreft load balancing en het aanbieden van alternatieve servers bij fouten worden door UDDI niet benut. UDDI voorziet op dit

vlak geen enkele functionaliteit alhoewel ze hiervoor wel in de juiste positie verkeren. Dit is toch wel een gemiste kans. Elk UDDI registry kan natuurlijk wel op eigen houtje dergelijke services aanbieden.

Hoofdstuk 6

Service Coördinatie

Zoals reeds eerder gezegd in het hoofdstuk 2 bestaan web service uit 3 basisconcepten:

1. SOA Architectuur
2. Standaardisatie: SOAP, WSDL, UDDI
3. Herontwerp van allerhande middleware protocollen

De eerste 2 punten werden reeds behandeld in de hoofdstukken 2, 3, 4 en 5. Punt 3 komt hier aan bod.

Tot nu toe ging het enkel over eenvoudige interacties waarbij slechts 1 operatie werd opgeroepen. Wanneer er gewerkt wordt met meer complexe conversaties is er nood aan:

1. Een mechanisme waarmee service providers kunnen aangeven welke conversaties ze ondersteunen.
2. Een mechanisme om te checken of de interactie tussen service provider en service requestor aan de regels voor toegestane conversaties voldoet.

In dit hoofdstuk behandelen we vooral het tweede punt. Voor het beschrijven van de ondersteunde conversaties (punt 1) kan gebruik gemaakt worden van de **Web Services Conversation Language (WSCL)** [20] of van **BPEL4WS** (zie hoofdstuk 7).

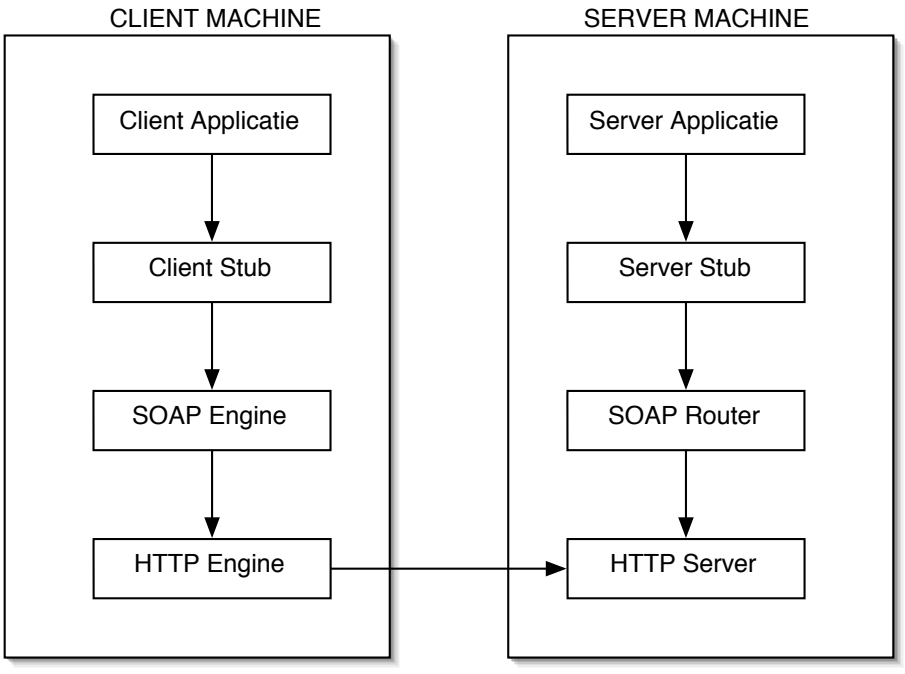
WSCL is een uitbreiding van WSDL voor het beschrijven van geldige interacties die door de service ondersteund worden. BPEL4WS was oorspronkelijk bedoeld voor service compositie maar dezelfde formalismen kunnen hier gebruikt worden voor het formuleren van een conversatie.

6.1 Classificatie van web service protocollen

Web service protocollen kunnen in twee groepen opgesplitst worden: de horizontale protocollen en de verticale protocollen. De verticale protocollen zijn applicatie gericht en behoren tot een specifiek domein. Horizontale protocollen daarentegen zijn meer algemeen en bijgevolg bruikbaar voor alle web services. Zij worden uitgevoerd door de web service middleware en dienen als bouwstenen voor verticale protocollen en andere horizontale protocollen. Voorbeelden van horizontale protocollen zijn WS-Coordination en WS-Transaction waarover meer in de secties 6.3 en 6.4. Een voorbeeld van een verticaal protocol is RosettaNet. RosettaNet is een consortium dat zich bezighoudt met het ontwikkelen van standaard e-commerce interfaces voor het op elkaar afstellen van de processen tussen supply chain partners.

6.2 Situatie schets

Zoals reeds eerder aangegeven in sectie 3.8 kan SOAP geïmplementeerd worden zoals in figuur 6.1:



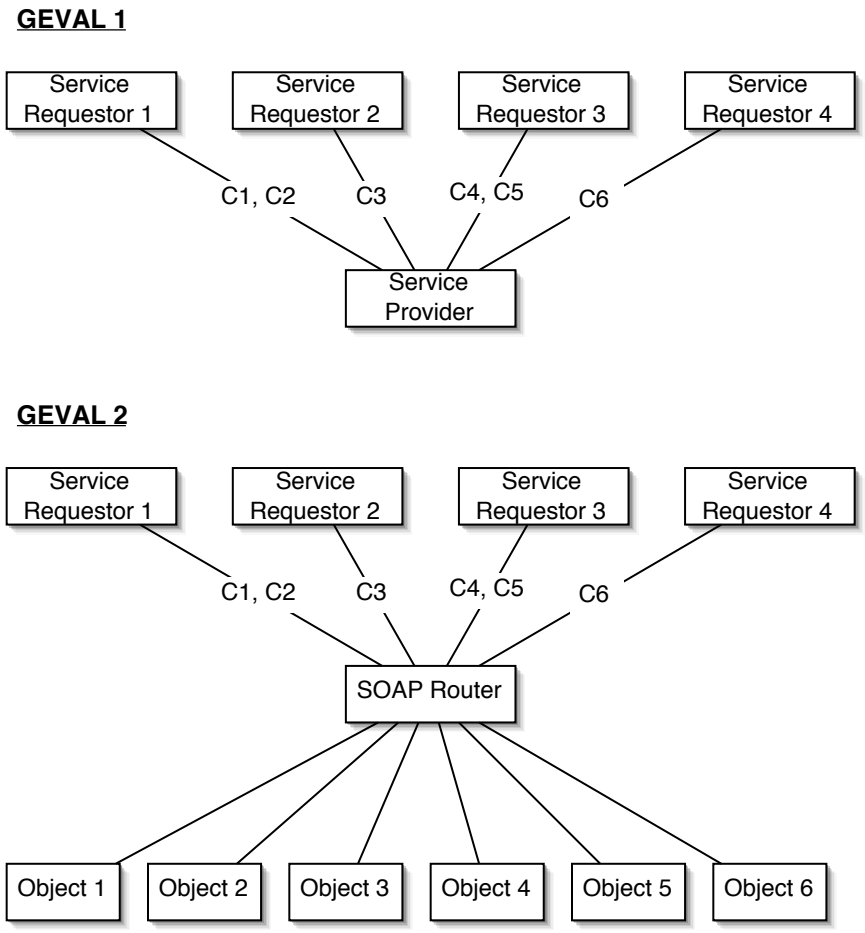
Figuur 6.1: Voorbeeld SOAP implementatie

De algemene werking staat beschreven in sectie 3.8. In deze context ligt de focus op de SOAP router. De taak van de SOAP router bestaat erin de SOAP messages naar het juiste server object te sturen. Wat betreft het server object zijn er 2 verschillende mogelijkheden:

1. Eén server object dat alle clients bedient.

2. Elke client krijgt zijn eigen server object toegewezen. Dus voor elke client wordt een factory functie opgeroepen die een nieuw object aanmaakt.

Schematisch kunnen we deze twee gevallen weergeven zoals in figuur 6.2, waarbij C1, C2, ... staan voor conversatie 1, conversatie 2, ...



Figuur 6.2: SOAP router: 1 object vs. meerdere objecten

In beide gevallen is het belangrijk om te weten welke message bij welke conversatie hoort. Dit gebeurt in de praktijk aan de hand van een identifier. In geval 1 is het de web service zelf die instaat voor:

1. Het bijhouden van de toestand van elke conversatie.
2. Het analyseren van de binnenkomende messages en het toewijzen van messages aan conversaties.
3. Het checken van de conversaties.

Dit is een vrij complexe taak die voor elke web service implementatie herhaald dient te worden. In geval 2 kan deze taak volledig door de web service middleware geïmplementeerd worden. Een deel van het werk wordt opgeknapt door het WS-Coordination framework. Het doel van WS-Coordination en WS-Transaction is de aanmaak van een framework dat kan gebruikt worden als basis voor het checken van conversaties.

6.3 WS-Coordination

WS-Coordination [16] staat voor Web Service Coordination en is een protocol dat instaat voor de coördinatie van acties van gedistribueerde applicaties. WS-Coordination is een uitbreidbaar framework dat als basis voor onder andere het ondersteunen van transacties en workflow management kan gebruikt worden.

Uit bovenstaande beschrijving blijkt het belang van conversation identifiers. WS-Coordination gebruikt hiervoor een eigen "coordination context". Naast een gewone identifier wordt ook andere informatie in de coordination context bijgehouden waarover meer in de nu volgende sectie.

6.3.1 De werking van WS-Coordination

WS-Coordination definieert een coördinator die verantwoordelijk is voor het besturen van de verschillende web services, participanten genoemd, die deel uitmaken van één conversatie. De uiteindelijke bedoeling is ervoor te zorgen dat de coördinator weet welke services er allemaal deelnemen (participeren) aan de conversatie. Ook belangrijk is dat de coördinator weet waar de participanten zich bevinden en omgekeerd zodat communicatie tussen de coördinator en de participanten mogelijk wordt.

Zowel de coördinator als de participanten implementeren minstens de volgende interfaces:

1. **Activation interface**: Participanten die een conversatie willen starten melden zich aan via deze interface. Belangrijk is ook dat de participant hierbij vermeldt welke rol de coördinator in de conversatie dient aan te nemen. Op deze manier wordt dan ook gespecificeerd over welk type conversatie het gaat. Bijvoorbeeld een transactie of een protocol dat instaat voor gegarandeerde aflevering,...

2. **Registration interface**: Elke participant die deel uitmaakt van de conversatie moet zich bij de coördinator registreren. Als onderdeel van deze registratie geeft de participant een poort nummer door waarover coördinatie messages verzonden kunnen worden.

3. **Protocol interface**: voor het versturen van protocol gerelateerde messages.

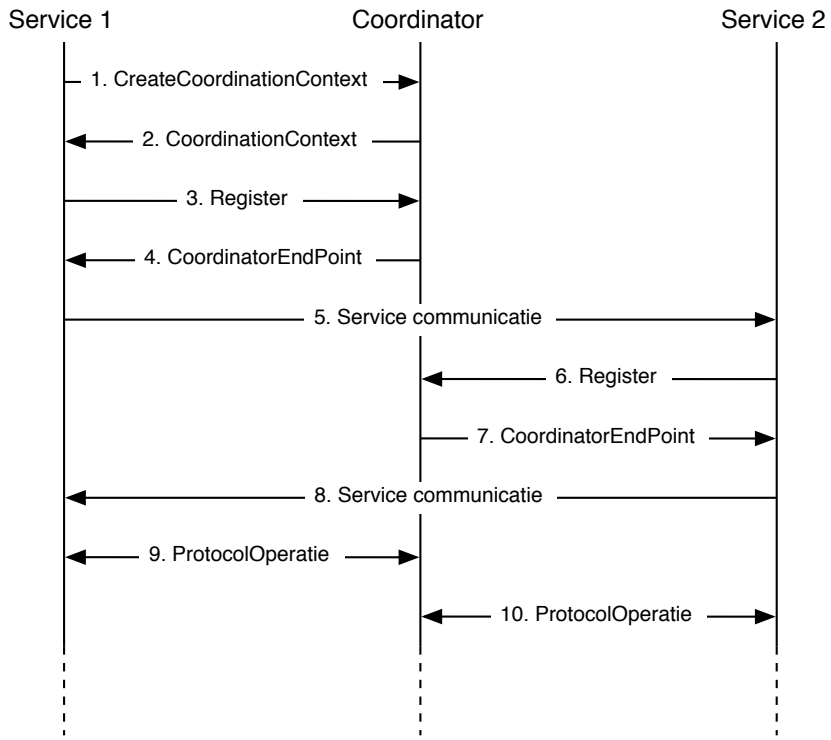
Zowel de participanten als de coördinator voorzien voor elk van deze interfaces een aparte poort. Voor de participanten komt er nog een extra poort, de web service poort, bij voor de interactie met andere participanten. Samengevat beschikt de coördinator dan over een activation, registration en een protocol poort terwijl de participanten over een activation, registration, protocol en web service port beschikken.

De werking van WS-Coordination kan het best uitgelegd worden aan de hand van figuur 6.3:

Stap 1: CreateCoordinationContext: Telkens een bepaalde applicatie een nieuwe conversatie wil beginnen vraagt deze aan de coördinator om een nieuwe context te creëren. Daartoe wordt de SOAP message uit voorbeeld 6.1, vertrekkende van de activation port van de participant naar de activation port van de coördinator gestuurd.

Voorbeeld 6.1.

```
<soapenv:Envelope
       xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
       xmlns:wscoor="http://schemas.xmlsoap.org/ws/2002/09/wscoor">
   <soapenv:Body>
      <wscoor:CreateCoordinationContext>
         <wscoor:CoordinationType>
            http://schemas.xmlsoap.org/ws/2002/09/wsat
         </wscoor:CoordinationType>
```



Figuur 6.3: Werking van WS-Coordination

```
      </wscoor:CreateCoordinationContext>
   </soapenv:Body>
</soapenv:Envelope>
```

Een *coordination type* bestaat uit een verzameling van verwante coordination protocollen. Een *coordination protocol* is een set van regels waaraan de conversatie tussen coördinator en participanten moet voldoen. Een voorbeeld van een coordination type is een atomic transaction. Dit coordination type bevat als coordination protocollen onder andere een 2PC protocol en een completion protocol waarmee de participant aangeeft dat hij op het einde van de transactie verwittigd wil worden van het resultaat. (zie sectie 6.4.1). Elke participant kan dan in de registration fase aangeven welke rol hij precies wil spelen binnen een bepaald coordination type. In bovenstaand voorbeeld gaat het om een atomic transaction. Dit wordt aangegeven met de link: "http://schemas.xmlsoap.org/ws/2002/09/wsat".

Stap 2: CoordinationContext: De coordination context die door de coördinator via de activation ports van de coördinator en de participant teruggestuurd wordt kan er dan uitzien zoals in voorbeeld 6.2:

Voorbeeld 6.2.

```
<soapenv:Envelope
       xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
       xmlns:wscoor="http://schemas.xmlsoap.org/ws/2002/09/wscoor"
       xmlns:wsu="http://schemas.xmlsoap.org/ws/2002/07/utility"
       xmlns:wsa="http://schemas.xmlsoap.org/ws/2003/03/addressing">
   <soapenv:Body>
      <wscoor:CreateCoordinationContextResponse>
         <wscoor:CoordinationContext>
            <wsu:Identifier>http://voorbeeld.be/trans3</wsu:Identifier>
            <wsu:Expires>2005-08-10T14:00:00Z</wsu:Expires>
            <wscoor:CoordinationType>
               http://schemas.xmlsoap.org/ws/2002/09/wsat
            </wscoor:CoordinationType>
            <wscoor:RegistrationService>
               <wsa:Address>http://coordinator.be/register</wsa:Address>
            </wscoor:RegistrationService>
         </wscoor:CoordinationContext>
      </wscoor:CreateCoordinationContextResponse>
   </soapenv:Body>
</soapenv:Envelope>
```

CreateCoordinationContextResponse geeft aan dat dit het antwoord is op de CreateCoordinationContext messages van hierboven. Naast een identifier (http://voorbeeld.be/trans3) wordt ook een Expires, een CoordinationType

en een RegistrationService element in de context opgenomen. Het Expires element geeft aan wanneer deze coordination context ten vroegste kan verlopen zijn. Het CoordinationType element hebben we reeds besproken en het RegistrationService element tenslotte geeft d.m.v. de registration poort aan waar we onze service kunnen registreren bij de coördinator.

Stap 3: Register: Nu we de locatie voor de registratie kennen kan er een Register operatie uitgevoerd worden waarmee we als service aan de coördinator willen laten weten dat wij deelnemen aan de conversatie. De SOAP message uit voorbeeld 6.3 wordt verstuurd van de registration port van de participant naar de registration port van de coördinator.

Voorbeeld 6.3.

```
<soapenv:Envelope
       xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
       xmlns:wscoor="http://schemas.xmlsoap.org/ws/2002/09/wscoor"
       xmlns:wsu="http://schemas.xmlsoap.org/ws/2002/07/utility">
   <soapEnv:Header>
      <wscoor:CoordinationContext mustUnderstand="true">
         <wsu:Identifier>http://voorbeeld.be/trans3</wsu:Identifier>
         <wsu:Expires>2005-08-10T14:00:00Z</wsu:Expires>
         <wscoor:CoordinationType>
            http://schemas.xmlsoap.org/ws/2002/09/wsat
         </wscoor:CoordinationType>
         <wscoor:RegistrationService>
            <wsa:Address>http://coordinator.be/register</wsa:Address>
         </wscoor:RegistrationService>
      </wscoor:CoordinationContext>
   </soapEnv:Header>
   <soapEnv:Body>
      <wscoor:Register>
         <wscoor:ProtocolIdentifier>
            http://schemas.xmlsoap.org/ws/2003/09/wsat\#Durable2PC
         </wscoor:ProtocolIdentifier>
         <wscoor:ParticipantProtocolService>
            <wsa:Address>http://participant/service1</wsa:Address>
         </wscoor:ParticipantProtocolService>
      </wscoor:Register>
   </soapEnv:Body>
</soapenv:Envelope>
```

Nadat de coordination context in stap 2 ontvangen werd, wordt deze steeds als SOAP header in de messages opgenomen. Het body element bevat hier de oproep van de register operatie. Het element ProtocolIdentifer definieert het te gebruiken coördinatie protocol, in dit geval een Durable2PC. (zie sectie

6.4.1) Het ParticipantProtocolService element bevat het adres (de protocol port) dat de coördinator kan gebruiken voor het oproepen van coördinatie operaties.

Stap 4: CoördinatorEndPoint: Het antwoord van de coördinator, verstuurd via de registration port van de coördinator naar de registration port van de participant ziet er dan uit zoals in voorbeeld 6.4:

Voorbeeld 6.4.

```
<soapenv:Envelope
       xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
       xmlns:wscoor="http://schemas.xmlsoap.org/ws/2002/09/wscoor"
       xmlns:wsu="http://schemas.xmlsoap.org/ws/2002/07/utility">
   <soapEnv:Header>
      <wscoor:CoordinationContext mustUderstand="true">
         <wsu:Identifier>http://voorbeeld.be/trans3</wsu:Identifier>
         <wsu:Expires>2005-08-10T14:00:00Z</wsu:Expires>
         <wscoor:CoordinationType>
            http://schemas.xmlsoap.org/ws/2002/09/wsat
         </wscoor:CoordinationType>
         <wscoor:RegistrationService>
            <wsa:Address>http://coordinator.be/register</wsa:Address>
         </wscoor:RegistrationService>
      </wscoor:CoordinationContext>
   </soapEnv:Header>
   <soapEnv:Body>
      <wscoor:RegisterResponse>
         <wscoor:CoordinatorProtocolService>
            <wsa:Address>http://coordinator.be/coordinator</wsa:Address>
         </wscoor:CoordinatorProtocolService>
      </wscoor:RegisterResponse>
   </soapEnv:Body>
</soapenv:Envelope>
```

Het CoordinatorProtocolService element bevat het adres (de protocol port) dat de participant kan gebruiken voor het oproepen van coördinatie operaties.

De volgende stappen zijn analoog. In het kort gebeurt het volgende:

- In stap 5 roept service 1 een bepaalde operatie van service 2 op. Dit gebeurt via een SOAP message met daarin de coordination context.
- Service 2 merkt de coordination context en gaat zich bij de coördinator registreren. (stap 6)
- De coördinator stuurt zijn protocol port terug als bevestiging. (stap 7)

- Service 2 kan nu gewoon communiceren met service 1. (stap 8)
- Tot slot kunnen ook coordination messages verstuurd worden tussen de coördinator en de participanten. (stappen 9 en 10)

Het uiteindelijke doel van WS-Coordination, nl. dat alle participanten en de coördinator elkaar kennen en met elkaar kunnen communiceren is dus bereikt. WS-Coordination wordt als basis voor verschillende andere protocollen zoals WS-Transaction gebruikt.

Bovenstaand voorbeeld is in feite een heel eenvoudig voorbeeld waarbij slechts 2 services betrokken zijn. In de praktijk is dit aantal zeker groter, maar de principes blijven hetzelfde.

Wat de coördinator betreft moeten we toch even opmerken dat er niet altijd 1 centrale coördinator gebruikt wordt, maar dat men soms ook 1 coördinator per service inzet. De coördinatoren werken in dat geval als een soort proxy voor de services. Het spreekt voor zich dat er dan heel wat meer synchronisatie nodig is om ervoor te zorgen dat alle coördinatoren en participanten elkaar kennen.

6.4 WS-Transaction

WS-Transaction [18] bouwt verder op het framework voorzien door WS-Coordination. Redenen waarom de gewone transactie protocollen van traditionele middleware systemen niet kunnen gebruikt worden zijn:

1. Het ontbreken van een centraal besturingsorgaan.
2. Het feit dat de verschillende deel processen langere tijd in beslag kunnen nemen waardoor database locks in de praktijk onhaalbaar worden.
3. Het feit dat locks, commits, resources en dergelijken niet eenduidig gedefinieerd zijn binnen web services in tegenstelling tot database systemen. (Wat is bijvoorbeeld de rollback van het versturen van een mail?)

Het is bijgevolg niet altijd mogelijk om aan de ACID eigenschappen [28] te voldoen. Transacties die aan de ACID eigenschappen voldoen, voldoen aan:

1. **Atomicity**: ze worden als 1 geheel uitgevoerd. (ofwel volledig ofwel niets)

2. **Consistency**: de consistentie moet bewaard blijven.

3. **Isolation**: of meerdere transacties nu tesamen of na elkaar uitgevoerd worden, het resultaat moet steeds hetzelfde zijn.

4. **Durability**: als de transactie afgelopen is mogen de wijzigingen niet verloren gaan.

WS-Transaction voorziet daarom twee soorten protocollen waarbij de tweede versie slechts gedeeltelijk aan de ACID eigenschappen voldoet:

1. **Atomic transactions**: dit zijn de transacties die een korte tijd in beslag nemen en die volledig voldoen aan de ACID eigenschappen.

2. **Business activities**: dit zijn transacties die meer tijd in beslag nemen en waarvoor er gebruik gemaakt wordt van de afgezwakte ACID eigenschappen.

WS-Transaction maakt dus voor de business activities gebruik van een **afgezwakte versie** van de ACID eigenschappen. Die afzwakking houdt in dat web services na elke operatie de wijzigingen kunnen bewaren. De eigenschappen waaraan dan niet meer voldaan wordt zijn atomicity en isolation. Hiervoor zijn compensatie mechanismen uitgewerkt.

Het compensatie mechanisme bestaat erin dat voor elke operatie een soort van compensatie operatie moet ontwikkeld worden die de operatie ongedaan maakt. De compensatie operatie moet voorzien worden door dezelfde instantie als degene die ook de oorspronkelijke operatie aanbood en maakt bijgevolg geen deel uit van het WS-Transaction protocol.

6.4.1 Atomic transaction

Zoals reeds eerder gezegd is een transactie atomair wanneer ofwel alle operaties van de transactie worden uitgevoerd ofwel geen enkele. Als een applicatie een atomic transactie [14] wil starten krijgt de coordinationType parameter de waarde: “http://schemas.xmlsoap.org/ws/2003/09/wsat”.

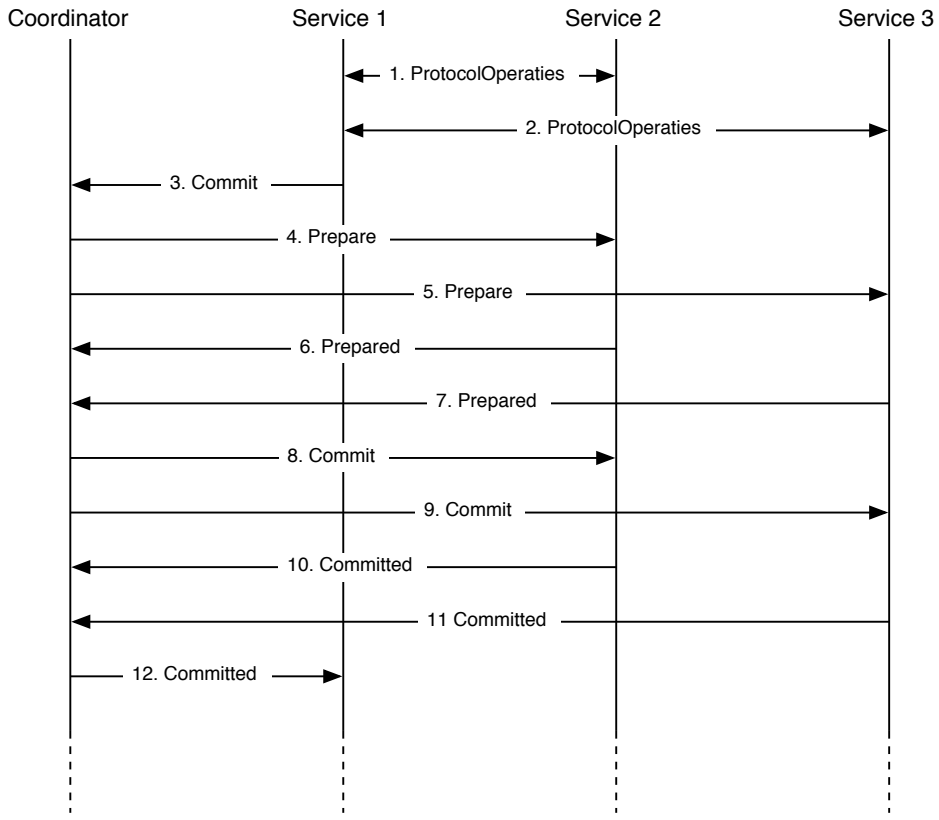
Bij de registratie geeft elke participant aan de hand van het coordination protocol aan welke rol hij binnen het Coordination type, in dit geval een atomic transaction, speelt. Voor atomic transactions zijn er 3 mogelijkheden:

1. **Completion**: een participant die deze rol speelt geeft aan dat hij op het einde van de transactie verwittigd wil worden in verband met het resultaat van de transactie.

2. **Durable2PC**: meest gebruikte protocol op basis van een gewone 2PC.

3. **Volatile2PC**: speciale versie van 2PC waarbij in fase 1 een prepare message verstuurd wordt naar een volatile memory zoals het cache geheugen. Het cache geheugen krijgt de message binnen en maakt de wijzigingen definitief door ze naar non volatile geheugen te schrijven. Fase 2 communiceert dan verder met dit non volatile geheugen om de 2PC af te werken.

Figuur 6.4 geeft een algemeen overzicht over de gevolgde procedure tijdens een Atomic Transaction.



Figuur 6.4: Werking van een Atomic Transaction

We nemen aan dat Service 1, 2 en 3 zich geregistreerd hebben als deel van een atomic transaction waarbij ze allen de Durable2PC rol spelen. Service 1 is de service die de transactie gestart heeft en in stappen 1 en 2 worden

gewone protocol operaties van Service 2 en 3 opgeroepen.

Op een gegeven moment wenst Service 1 de transactie te beïndigen. Er zijn hier 2 mogelijkheden: ofwel wordt een rollback messages naar de coördinator gestuurd ofwel een commit message. De structuur van deze messages is zeer eenvoudig.

```
<soapenv:Body>
   <wsat:Rollback>
</soapenv:Body>

<soapenv:Body>
   <wsat:Commit>
</soapenv:Body>
```

Beide voorbeelden worden dan in een SOAP envelope element verpakt met als SOAP header de coordination context afkomstig van WS-Coordination.

In ons voorbeeld hebben we gekozen om te committen. In stap 3 verstuurt Service 1 daarom een commit message naar de coördinator waarop deze een 2PC start door het versturen van prepare messages naar Service 2 en Service 3. (stappen 4 en 5)

Een Service die een prepare message binnenkrijgt kan hierop 3 mogelijke antwoorden geven:

1. **Prepared**: om aan te geven dat de Service klaar is om te committen.
2. **Aborted**: om aan te geven dat er iets mis is en dat de volledige transactie afgebroken moet worden.
3. **ReadOnly**: om aan te geven dat deze Service geen data wijzigingen aanbrengt en dat er dus geen commit of abort moet gebeuren. Dit heeft tot gevolg dat deze Service niet meer deelneemt aan de tweede fase van het 2PC protocol.

Ook de structuur van deze messages is zeer eenvoudig:

```
<soapenv:Body>
   <wsat:Prepared>
</soapenv:Body>

<soapenv:Body>
```

```
   <wsat:Aborted>
</soapenv:Body>

<soapenv:Body>
   <wsat:ReadOnly>
</soapenv:Body>
```

In de stappen 6 en 7 sturen zowel Service 2 als Service 3 een prepared message naar de coördinator. De coördinator die alle antwoorden binnenkrijgt kan dan ofwel committen ofwel aborten. In dit geval wordt er gekozen om te committen daar alle Services prepared waren. Hiertoe worden commit messages verstuurd naar de beide Services. (stappen 8 en 9)

De coördinator krijgt daarna de antwoorden van de Services toe waaruit blijkt dat ze beiden gecommit hebben. (stappen 10 en 11) Tot slot zendt de coördinator een committed message naar Service 1, de Service die het hele 2PC protocol startte.

6.4.2 Business Activities

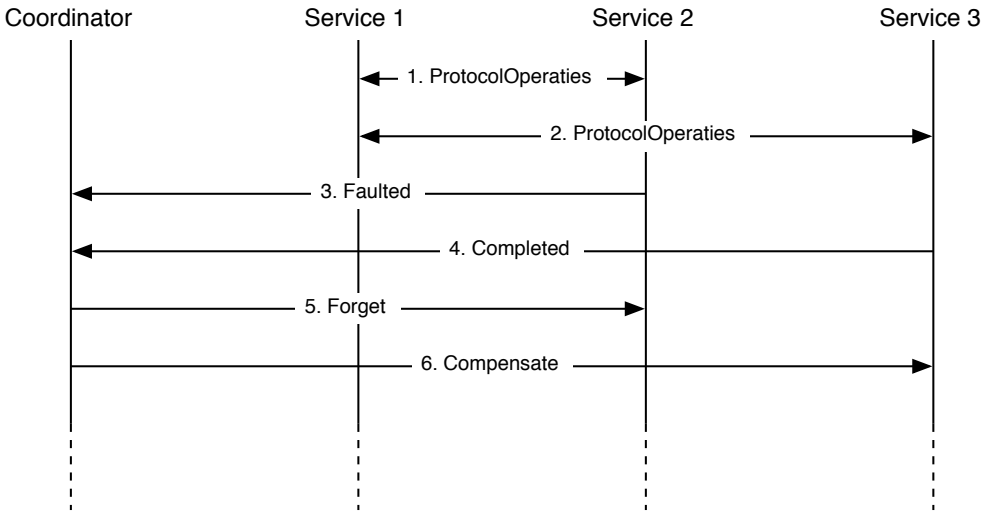
Business Activities [15] zijn transacties waarbij de verschillende operaties typisch een langere tijd in beslag nemen dan bij atomic transaction. Gewone locks zijn dus niet meer haalbaar waardoor het principe van de 2PC zijn nut verliest. Bij Business Activities wordt elke individuele operatie op zich onmiddellijk gecommit. Rollbacks zijn mogelijk via compensatie operaties.

Als een applicatie een business activity wil starten krijgt de coordinationType parameter de waarde: “http://schemas.xmlsoap.org/ws/2002/08/wsba”. Voor een Business activity zijn er 2 mogelijke coördinatie protocollen:

1. De **businessAgreement rol** wordt gebruikt als een participant de status van zijn execution aan de coördinator wil meedelen. De mogelijke waardes voor het status veld zijn: Exited, Completed of Faulted.

2. De **businessAgreementWithComplete rol** is een uitbreiding van de vorige rol waarbij de coördinator aan de participant moet laten weten wanneer de participant geen opdrachten meer moet verwachten. Op deze manier kan de betreffende participant zich voorbereiden op een eventuele complete of compensate message.

Figuur 6.5 geeft een algemeen overzicht van de gevolgde procedure tijdens een Business Activity.



Figuur 6.5: Werking van een Business Activity

We nemen aan dat Service 1, 2 en 3 zich geregistreerd hebben als deel van een Business Activity en dat ze allen de businessAgreement rol spelen. Service 1 is de Service die de transactie gestart heeft en in stappen 1 en 2 worden gewone protocol operaties van Service 2 en 3 opgeroepen.

In stappen 3 en 4 melden respectievelijk Service 2 en 3 hun status. Uit de "faulted" status blijkt duidelijk dat er in Service 2 iets is misgelopen.

Als alle resultaten binnen zijn kan de coördinator antwoorden met een Close, Complete, Compensate of Forget message. In stap 5 wordt naar Service 2 een forget message gestuurd zodat deze alle gegevens van de transactie kan vergeten. Service 3 die zijn operatie wel heeft uitgevoerd moet dit ongedaan maken door het uitvoeren van een compensatie operatie. Dit wordt hem gemeld door de Compensate message uit stap 6.

6.5 Andere protocollen

Zoals reeds eerder besproken moet de web service middleware tal van protocollen opnieuw ontwikkelen. We hebben er nu 2 besproken maar er zijn er natuurlijk nog veel meer zoals onder andere WS-ReliableMessaging, WS-Security (zie 3.9.2), ... WS-ReliableMessaging is in feite zeer belangrijk in de context van transacties. Als er bijvoorbeeld bepaalde messages verloren

gaan in het WS-Transaction protocol zou dit ertoe kunnen leiden dat volledige transacties afgebroken worden alhoewel er in feite toch niets mis was. De horizontale protocollen bouwen steeds verder op elkaar en worden ook als basis voor de verticale protocollen gebruikt.

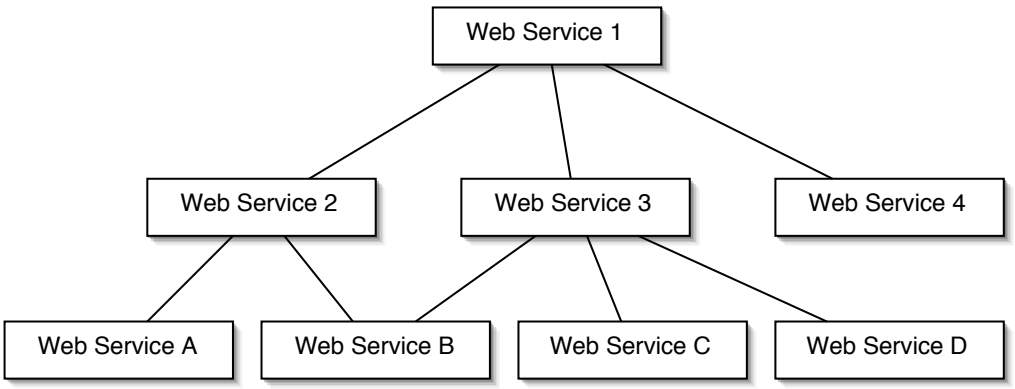
Hoofdstuk 7

Service Compositie

7.1 Wat is Service Compositie?

Zoals we reeds vermeld hebben zijn composite web services, web service die zelf gebruik maken van andere web services.

Service compositie kan gebruikt worden om complexiteit af te schermen. Daarnaast biedt het ook de mogelijkheid om met verschillende niveau's van abstractie te werken zoals te zien is op figuur 7.1.



Figuur 7.1: Voorbeeld van Service Composition

Momenteel worden composite web services veelal ontwikkeld met behulp van gewone programmeertalen zoals C, C++,... Gewone programmeertalen zijn hier echter niet voor bedoeld en het programmeren van compositie web services op deze manier is dan ook nogal omslachtig.

Daarom zijn er verschillende voorstellen voor web service compositie talen geweest. Al deze voorstellen samen hebben uiteindelijk geleid tot de algemene taal: "BPEL4WS". [12]

BPEL4WS staat voor Business Process Execution Language for Web Services en is de meest gebruikte taal op het vlak van web service compositie en web service coördinatie. Deze taal werd in 2002 ontwikkeld door BEA, Microsoft en IBM en is momenteel bij OASIS ingediend voor standaardisatie.

7.1.1 Service coördinatie vs. service compositie.

Bij service coördinatie moeten de operaties binnen een bepaalde web service voldoen aan een voorafgedefinieerde volgorde. Deze volgorde is publiek gekend en wordt daarom ook in service registries opgenomen.

Service compositie daarentegen is bedoeld voor intern gebruik, binnen een bedrijf of instelling. Hier wordt een service opgebouwd uit een set van andere services. Informatie omtrent welke services in welke volgorde gebruikt worden blijft voor de gebruiker afgeschermd. Voor de gebruiker is er geen onderscheid te maken tussen een composite web service en een gewone web service.

7.2 Hoe werkt Service Compositie?

7.2.1 Web Service Compositie Middleware

De web service compositie middleware voorziet een infrastructuur die het definiëren en uitvoeren van dergelijke composite web services moet vergemakkelijken. Dit gebeurt d.m.v. een service compositie model dat instaat voor het definiëren van:

- de services die gebruikt worden,
- de manier waarop de services met elkaar verbonden worden,
- en de volgorde waarin de services opgeroepen worden.

7.2.2 Service Compositie modellen

Een service compositie model moet de volgende vragen op een eenduidige manier kunnen beantwoorden:

1. Welke aannames kunnen omtrent de deelnemende web services (componenten) gedaan worden?
2. In welke volgorde moeten deze componenten opgeroepen worden?
3. Hoe communiceren de componenten onderling? Anders gezegd hoe ziet de data eruit die onderling verstuurd wordt en hoe komt de data bij de juiste component terecht?
4. Hoe worden services geselecteerd?
5. Hoe worden transacties ondersteund?
6. En wat gebeurt er in geval van fouten?

In de nu volgende bespreking geven we een overzicht van de mogelijke antwoorden op deze vragen.

Aannames omtrent de deelnemende web services?

We formuleren het antwoord op deze vraag in functie van BPEL4WS, de leider op het vlak van service compositie. De aannames die BPEL4WS over de deelnemende componenten doet zijn:

1. Elke deelnemende service beschikt over een WSDL description.
2. Verschillende standaarden of standaarden in wording moeten door de web service ondersteund worden. Voorbeelden van dergelijke standaarden zijn: XPath en WS-Addressing.

Volgorde van de deelnemende web services?

De volgorde van de componenten kan aangegeven worden aan de hand van zogenaamde orchestration models. Aangezien de uitvoer van eerdere componenten en de resultaten daarvan het verdere verloop van een composite web service kunnen beïnvloeden maken deze modellen gebruik van condities. Er bestaan verschillende soorten orchestration models:

1. **Activity Diagrams**: deze worden het meest gebruik.
2. **Statecharts**: bij dit model ligt de focus op de toestanden en niet op de activiteiten waardoor deze manier van weergave meer geschikt is voor het controleren van het verloop van composite web services. Als de toestanden duidelijk namen hebben is het makkelijker te zien hoe ver het staat met een bepaalde service. Dit is bij activity diagrammen niet het geval.

3. **Petri Nets**: is een soort combinatie tussen activity diagrams en statecharts.

4. **Π-Calculus**: kan gezien worden als een zeer eenvoudige programmeertaal waarmee de volgorde van de componenten kan aangegeven worden door middel van de primitieven ".", "+" en "|". [var=value] wordt gebruikt als notatie voor condities.

5. **Activity Hierarchies**: splitsen elke activiteit op in een boom van "sub" activiteiten.

6. **Rule-based Orchestration**: maakt gebruik van rules in de vorm van <event, action> of <event, condition, action>.

De activity diagrams worden veruit het meest gebruikt omdat deze het makkelijkst te begrijpen zijn. BPEL4WS maakt gebruik van een combinatie van activity diagrams en activity hierarchies.

Hoe communiceren de componenten onderling?

Om verschillende web services met elkaar te verbinden moet er natuurlijk data uitgewisseld kunnen worden. Data kan opgesplitst worden in enerzijds applicatie specifieke data en anderzijds control flow data.

De applicatie specifieke data wordt typisch verzonden in SOAP messages.

De waarde van control flow data wordt via een user-defined mapping verkregen uit XML gebaseerde messages afkomstig van web services binnen de compositie. Typisch is control flow data beperkt tot een aantal basis types zoals: strings, booleans, integers, ... en wordt het gebruikt voor het evalueren van condities waarmee het verdere verloop van de uitvoer bepaald wordt.

Nu we een data formaat gedefinieerd hebben moet nog bepaald worden hoe data van de ene service naar de andere doorgegeven wordt. Er zijn twee transfer methodes: de blackboard methode en de explicit data flow methode.

De werking van **blackboard** is als volgt:

1. Je beschikt over een verzameling variabelen met hun waardes, blackboard genaamd.

2. Op het moment dat een message moet verstuurd worden, worden de variabelen opgezocht en wordt de waarde in een message verwerkt.

3. Bij aankomst van een message worden de variabelen uit de message gehaald en wordt het blackboard atomair geüpdatet.

De **explicit data flow methode** werkt volgens een principe waarbij data niet centraal wordt bijgehouden, maar van de ene web service naar de andere wordt doorgegeven. Deze data kan dan door de opeenvolgende web services aangepast worden.

Hoe worden services geselecteerd?

Services kunnen aan de hand van URI's zowel statisch als dynamisch geselecteerd worden.

Hoe worden transacties ondersteund?

Transacties worden ondersteund door de mogelijkheid om atomic regions binnen een orchestration schema te definiëren. Ofwel worden dan alle activities binnen de atomic region uitgevoerd ofwel geen enkele. De implementatie kan gebruik maken van het 2PC protocol. Maar net zoals bij service coordination dient een afzwakking van de ACID properties zich in bepaalde situaties aan. Ook hier wordt in dat geval gebruik gemaakt van compensatie mechanismen. Verschillende composities maken daarom gebruik van het saga model. Het saga model verdeelt langdurende transacties in een verzameling sub transacties met overeenkomstige compensatie transacties. Elk van de sub-transacties voldoet hierbij wel aan de ACID properties.

Wat gebeurt er in geval van fouten?

Er bestaan op dit vlak verschillende technieken:

- **Flow-based approaches**: aan het einde van elke operatie oproep wordt het resultaat gecontroleerd op fouten.
- **Try-catch-throw approaches**: werkt volgens hetzelfde principe als de java try, catch en throw instructies. Deze techniek wordt gebruikt door BPEL4WS.
- **Rule-based approaches**: werkt volgens het principe van <event, condition, action>.

Hoofdstuk 8

Implementatie van web services met Apache Axis.

Dit hoofdstuk bestaat uit een korte beschrijving van wat Axis [7] is en waarvoor het kan gebruikt worden. Enkele van deze functionaliteiten worden daarna toegelicht aan de hand van voorbeelden. We maken hierbij gebruik van de Jacarta Tomcat Application Server [8].

8.1 Wat is Axis

Axis staat voor Apache eXtensible Interaction System. Axis is een package dat instaat voor de verwerking van SOAP messages, het is een java framework voor de ontwikkeling van SOAP processors zoals servers, clients en intermediaries. Een C++ versie van Axis is momenteel nog in ontwikkeling.

8.2 Welke functionaliteiten biedt Axis aan

Axis biedt tal van functionaliteiten die het makkelijker maken om services te ontwikkelen en te consumeren. Enkele van de mogelijkheden zijn:

- **Instant Deployment**: Een gewone java file kan mits wijziging van de extensie en kopiëren naar de juiste map onmiddellijk als web service aangeboden worden.
- **WSDL2Java**: maakt het mogelijk om vertrekkende van WSDL files Java frameworks te creëren die gebruikt kunnen worden als basis voor het consumeren en aanbieden van web services.

- **Java2WSDL**: Aan de hand van een Java interface kunnen java frameworks en WSDL bestanden aangemaakt worden.
- Een verzameling van APIs voor het bewerken van SOAP messages en diens componenten: envelopes, headers en bodies.
- Het onderling mappen van java types en SOAP/XML types.
- Het dynamisch oproepen van SOAP gebaseerde web services.
- Mechanismen voor het hosten van web services.
- De mogelijkheid tot het opbouwen van ketens van SOAP verwerkende componenten, handlers genaamd.
- Ondersteuning voor session georiënteerde services.

8.3 Case study: enkele voorbeelden

In de nu volgende sectie bespreken we 3 hoofdpunten van Axis zijnde:

1. Het deployen van web services: instant deployment vs. WSDD (zie sectie 8.3.1)
2. Het consumeren van web services: call en service objecten vs. WSDL2Java. (zie sectie 8.3.2)
3. Aanzet tot het opzetten van ketens van SOAP verwerkende componenten. (zie sectie 8.3.3)

8.3.1 Het Deployen van web services

Indien je als server een bepaalde service over het netwerk wil aanbieden als web service dan gaat dit op verschillend manieren:

Manier1

Voor eenvoudige java bestanden die geen gebruik maken van packages bestaat een zeer snelle en makkelijke oplossing. Het deployen van de web service kan al in drie eenvoudige stappen:

1. Maak een java file aan met daarin enkele eenvoudige functies zoals in voorbeeld 8.1.

2. Wijzig de extense van ".java" naar ".jws".

3. Kopieer tot slot het jws bestand naar de webapp folder van de applicatie server.

Voorbeeld 8.1. : Voorbeeld van een java file die in aanmerking komt voor een dergelijke procedure

```
public class Calculator {
  public int add(int i1, int i2)
  {
    return i1 + i2;
  }

  public int subtract(int i1, int i2)
  {
    return i1 - i2;
  }
}
```

Dit bestand wordt dan at runtime gecompileerd. Deze manier is zeer snel en makkelijk maar heeft ook enkele nadelen. Zo is het bijvoorbeeld niet mogelijk om fouten te ontdekken tenzij at runtime na deployment, je hebt ook steeds de broncode van de classes nodig. Een ander nadeel is natuurlijk de beperkingen die op de java file zelf gelegd worden. Onder andere packages kunnen niet gebruikt worden.

Manier2

De tweede manier is iets complexer maar geeft ook meer controle mogelijkheden. In tegenstelling tot de vorige manier kan hier wel met met packages en class bestanden gewerkt worden. Stel dat we de code in voorbeeld 8.2 als web services willen aanbieden:

Voorbeeld 8.2.

```
package voorbeelden.voorbeeld2;

public class Math{
    public int add(int x, int y) {
        return x + y;
    }
    public int subtract(int x, int y) {
        return x - y;
    }
    public int  multiply(int x, int y) {
        return x * y;
    }
```

```
    public int divide(int x, int y) {
        return x / y;
    }
}
```

De eerste stap die moet gezet worden is het registreren van de web service. Axis werkt met WSDD wat staat voor **Web Service Deployment Descriptor**. WSDD is een XML formaat dat door Axis gebruikt wordt voor het opslaan van configuratie en deployment informatie. Deze informatie wordt bijgehouden in de bestanden server-config.wsdd en client-config.wsdd. In deze stap is het de bedoeling om bovenstaande functies als web services op te nemen in het server-config.wsdd bestand. Dit kan ofwel door rechtstreeks in dit bestand te schrijven ofwel door de deployment descriptor uit voorbeeld 8.3 mee te gevan aan de AdminClient.

Voorbeeld 8.3. Deployment descriptor: deploy.wsdd

```
<deployment xmlns="http://xml.apache.org/axis/wsdd/"
    xmlns:java="http://xml.apache.org/axis/wsdd/providers/java">
  <service name="Math" provider="java:RPC">
    <parameter name="className" value="voorbeelden.voorbeeld2.Math"/>
    <parameter name="allowedMethods" value="*"/>
  </service>
</deployment>
```

Het buitenste element geeft aan dat het om een WSDD deployment gaat, het service element definieert de naam van de service en de manier van interactie: RPC. In de parameter elementen wordt aangegeven welke klasse met de service overeenkomt en welke de toegestane operaties zijn. Naast deze twee types van parameters zijn ook nog tal van andere parameters mogelijk waarover meer in sectie 8.3.3.

Na oproep van het volgende commando:

```
java org.apache.axis.client.AdminClient deploy.wsdd
```

wordt het server-config.wsdd bestand aangevuld met:

```
<service name="Math" provider="java:RPC">
  <parameter name="allowedMethods" value="*"/>
  <parameter name="className" value="voorbeelden.voorbeeld2.Math"/>
</service>
```

en is de math class beschikbaar als web service.

8.3.2 Het consumeren van web services

Wanneer je als client een bepaalde service wenst te gebruiken dan kan dit ook op verschillende manieren:

Manier1

Hier is het de bedoeling om de javacode volledig zelf op te bouwen gebruik makend van de Call en Service elementen. Een aanroep van de bovenstaande Math service ziet er dan uit zoals in voorbeeld 8.4

Voorbeeld 8.4. MathClient.java:

```
package voorbeelden.voorbeeld2;

import org.apache.axis.client.Call;
import org.apache.axis.client.Service;
import org.apache.axis.encoding.XMLType;
import org.apache.axis.utils.Options;

import javax.xml.rpc.ParameterMode;
import javax.xml.namespace.QName;

public class MathClient
{
    public static void main(String [] args)
    {
        try
        {
            Options options = new Options(args);
            String endpoint = options.getURL();
            String num1 = args[1];
            String num2 = args[2];

            args = options.getRemainingArgs();

            if(args.length != 3)
            {
                System.out.println("Usage: MathClient <add|subtract|multiply|divide>
                  arg1 arg2");
            }
            else
            {
                Service service = new Service();
                Call call = (Call) service.createCall();
                call.setTargetEndpointAddress( new java.net.URL(endpoint) );
                call.setOperationName( new QName("Math", args[0]) );
                call.addParameter("number1", XMLType.XSD_INT, ParameterMode.IN);
```

```
                call.addParameter("number2", XMLType.XSD_INT, ParameterMode.IN);
                call.setReturnType(XMLType.XSD_INT);
                int result = ((Integer)call.invoke(new Object[] {
                new Integer(Integer.valueOf( num1 ).intValue()), new Integer(
                Integer.valueOf( num2 ).intValue()) })).intValue();
                System.out.println("Het resultaat is: " + result);
            }

        }catch(Exception e){
            System.err.println(e.toString());
        }
    }
}
```

Met het commando "java voorbeelden.voorbeeld2.MathClient add 2 3" kan de service nu opgeroepen worden.

Manier2

De tweede manier vertrekt steeds van een WSDL description. Axis genereert automatisch voor elke deployed service een WSDL bestand. Dit document is toegankelijk via een gewone web browser door eenvoudigweg "?wsdl" achteraan de URL geassocieerd met de web service toe te voegen. Dit gaat zowel voor instant deployed services als voor services aangeboden via een deployment description.

De "Calculator" web service is terug te vinden op het adres:
http://localhost:8080/axis/Calculator.jws
De overeenkomstige WSDL description is bereikbaar via:
http://localhost:8080/axis/Calculator.jws?wsdl en ziet er uit zoals in voorbeeld 8.5:

Voorbeeld 8.5.

```
<?xml version="1.0" encoding="UTF-8" ?>
<wsdl:definitions targetNamespace="http://localhost:8080/axis/Calculator.jws"
        xmlns="http://schemas.xmlsoap.org/wsdl/"
        xmlns:apachesoap="http://xml.apache.org/xml-soap"
        xmlns:impl="http://localhost:8080/axis/Calculator.jws"
        xmlns:intf="http://localhost:8080/axis/Calculator.jws"
        xmlns:soapenc="http://schemas.xmlsoap.org/soap/encoding/"
        xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
        xmlns:wsdlsoap="http://schemas.xmlsoap.org/wsdl/soap/"
        xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <wsdl:message name="subtractRequest">
    <wsdl:part name="i1" type="xsd:int" />
    <wsdl:part name="i2" type="xsd:int" />
```

```
</wsdl:message>

<wsdl:message name="subtractResponse">
  <wsdl:part name="subtractReturn" type="xsd:int" />
</wsdl:message>

<wsdl:message name="addResponse">
  <wsdl:part name="addReturn" type="xsd:int" />
</wsdl:message>

<wsdl:message name="addRequest">
  <wsdl:part name="i1" type="xsd:int" />
  <wsdl:part name="i2" type="xsd:int" />
</wsdl:message>

<wsdl:portType name="Calculator">
  <wsdl:operation name="add" parameterOrder="i1 i2">
    <wsdl:input message="impl:addRequest" name="addRequest" />
    <wsdl:output message="impl:addResponse" name="addResponse" />
  </wsdl:operation>

  <wsdl:operation name="subtract" parameterOrder="i1 i2">
    <wsdl:input message="impl:subtractRequest" name="subtractRequest" />
    <wsdl:output message="impl:subtractResponse" name="subtractResponse" />
  </wsdl:operation>
</wsdl:portType>

<wsdl:binding name="CalculatorSoapBinding" type="impl:Calculator">
  <wsdlsoap:binding style="rpc"
      transport="http://schemas.xmlsoap.org/soap/http" />
  <wsdl:operation name="add">
    <wsdlsoap:operation soapAction="" />
    <wsdl:input name="addRequest">
      <wsdlsoap:body
      encodingStyle="http://schemas.xmlsoap.org/soap/encoding/
      namespace="http://DefaultNamespace" use="encoded" />
    </wsdl:input>

    <wsdl:output name="addResponse">
      <wsdlsoap:body
      encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
      namespace="http://localhost:8080/axis/Calculator.jws" use="encoded" />
    </wsdl:output>

  </wsdl:operation>

  <wsdl:operation name="subtract">
    <wsdlsoap:operation soapAction="" />
    <wsdl:input name="subtractRequest">
```

```
        <wsdlsoap:body
        encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
        namespace="http://DefaultNamespace" use="encoded" />
      </wsdl:input>

      <wsdl:output name="subtractResponse">
        <wsdlsoap:body
        encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
        namespace="http://localhost:8080/axis/Calculator.jws" use="encoded" />
      </wsdl:output>

    </wsdl:operation>

  </wsdl:binding>

  <wsdl:service name="CalculatorService">
    <wsdl:port binding="impl:CalculatorSoapBinding" name="Calculator">
      <wsdlsoap:address location="http://localhost:8080/axis/Calculator.jws" />
    </wsdl:port>
  </wsdl:service>

</wsdl:definitions>
```

Vertrekkende van deze WSDL descriptions kunnen met de WSDL2Java tool zogenaamde stubs (client zijde) en skeletons (server zijde) aangemaakt worden. Deze kunnen dan gebruikt worden als hulpmiddel voor onder andere het aanroepen van de betreffende web service. We tonen dit aan met een voorbeeld.

Op de web site: www.xmethods.net zijn tal van web services en hun WSDL bestanden terug te vinden. Enkele voorbeelden van WSDL document:

- http://www.xmethods.net/sd/2001/CurrencyExchangeService.wsdl
- http://www.xmethods.net/sd/2001/TemperatureService.wsdl.

Omdat we verderop gebruik zullen maken van de TemperatureService volgt nu de WSDL description: (zie voorbeeld 8.6)

Voorbeeld 8.6.

```
<?xml version="1.0" ?>
<definitions name="TemperatureService"
        targetNamespace="http://www.xmethods.net/sd/TemperatureService.wsdl"
        xmlns:tns="http://www.xmethods.net/sd/TemperatureService.wsdl"
        xmlns:xsd="http://www.w3.org/2001/XMLSchema"
        xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
        xmlns="http://schemas.xmlsoap.org/wsdl/">
```

```
  <message name="getTempRequest">
    <part name="zipcode" type="xsd:string" />
  </message>

  <message name="getTempResponse">
    <part name="return" type="xsd:float" />
  </message>

  <portType name="TemperaturePortType">
    <operation name="getTemp">
      <input message="tns:getTempRequest" />
      <output message="tns:getTempResponse" />
    </operation>
  </portType>

  <binding name="TemperatureBinding" type="tns:TemperaturePortType">
    <soap:binding style="rpc" transport="http://schemas.xmlsoap.org/soap/http" />
    <operation name="getTemp">
      <soap:operation soapAction="" />
      <input>
        <soap:body use="encoded" namespace="urn:xmethods-Temperature"
        encodingStyle="http://schemas.xmlsoap.org/soap/encoding/" />
      </input>

      <output>
        <soap:body use="encoded" namespace="urn:xmethods-Temperature"
        encodingStyle="http://schemas.xmlsoap.org/soap/encoding/" />
      </output>
    </operation>
  </binding>

  <service name="TemperatureService">
    <documentation>Returns current temperature in a given U.S.
        zipcode</documentation>
    <port name="TemperaturePort" binding="tns:TemperatureBinding">
      <soap:address
        location="http://services.xmethods.net:80/soap/servlet/rpcrouter" />
    </port>
  </service>

</definitions>
```

Om de TemperatureService te kunnen gebruiken creëren we eerst een stub dit gaat met het commando:

```
Java org.apache.axis.wsdl.WSDL2Java
http://www.xmethods.net/sd/2001/TemperatureService.wsdl
```

Hetgeen resulteert in een nog te compileren package dat kan gebruikt worden

om de betreffende web service op te roepen. Een voorbeeld van een java file die gebruik maakt van dit package is de Weather.java file in voorbeeld 8.7:

Voorbeeld 8.7.

```
import net.xmethods.www.sd.TemperatureService_wsdl.*;

class Weather
{
  public static void main(String args[]) throws Exception
  {
    String zip = "10001";
    TemperatureServiceLocator locator = new TemperatureServiceLocator();
    TemperaturePortType stub = locator.getTemperaturePort();

    float result = stub.getTemp(zip);
    System.out.println("In de stad met zipcode: " + zip + "
        is het " + result + " graden");
  }
}
```

8.3.3 Het opzetten van ketens van SOAP verwerkende componenten

WSDD documenten kunnen nog andere parameter elementen bevatten naast degene die we tot nu toe besproken hebben. Zo is het bijvoorbeeld mogelijk om de scope van een service of de te gebruiken handlers te definiëren.

Wat de scope betreft zijn er 3 mogelijkheden: Request, Application en Session. Afhankelijk van de gekozen waarde wordt er ofwel voor elk SOAP request een nieuw object aangemaakt (Request Scope), ofwel wordt er één object aangemaakt dat alle requests moet behandelen (Application Scope) ofwel wordt voor elke session enabled client een nieuw object gecreëerd (Session Scope). Situatie 1 en 2 komen respectievelijk overeen met geval 2 en geval 1 van figuur 6.2 in hoofdstuk 6.

Handlers kunnen gebruikt worden voor onder andere: het nemen van een log van een SOAP message, het registreren van het aantal keer dat een bepaalde service opgeroepen werd, het verwerken van SOAP headers, het encoderen van het body element van een SOAP message,...

Er bestaan twee soorten handlers. Allereerst zijn er de “preprocessor” handlers of **request handlers** die de message verwerken alvorens die de ultimate receiver bereikt. Daarnaast zijn er ook nog de “postprocessor” handlers, ook wel **response handlers** genoemd, die instaan voor de verwerking van de

message nadat die door de ultimate receiver verwerkt werd. Voorbeeld 8.8 bevat een voorbeeld van een WSDD file:

Voorbeeld 8.8.

```
<deployment xmlns="http://xml.apache.org/axis/wsdd/"
            xmlns:java="http://xml.apache.org/axis/wsdd/providers/java">

 <handler name="test" type="java:voorbeelden.voorbeeld3.TestHandler">
  <parameter name="filename" value="TestService.log"/>
 </handler>

 <service name="TestService" provider="java:RPC">
  <requestFlow>
   <handler type="test"/>
  </requestFlow>

  <parameter name="className" value="voorbeelden.voorbeeld3.MijnService"/>
  <parameter name="allowedMethods" value="*"/>
 </service>
</deployment>
```

Het requestFlow element komt overeen met de eerder vermeldde request handler. Daarnaast bestaat er ook nog een ResponseFlow element voor response handlers. Handlers kunnen gegroepeerd worden in chains. Vandaar dat binnen request- en response-Flow elementen zowel handlers als chains kunnen voorkomen.

In voorbeeld 8.8 wordt een service aangeboden onder de naam TestService. De service is toegankelijk via RPC interactie. Voordat de message verwerkt wordt passeert die eerst de TestHandler. Deze handler kan naar believen geïmplementeerd worden. Het parameter element dat behoort tot de handler zet de variabele filename gelijk aan de String “TestService.log”. Deze naam zal straks gebruikt worden voor het nemen van een log. Voorbeeld 8.9 bevat een implementatie van de klasse MijnService.

Voorbeeld 8.9.

```
package voorbeelden.voorbeeld3;

public class MijnService
{
    public String testMethod(String s)
    {
        return s;
    }
}
```

Voorbeeld 8.10 bevat een mogelijke implementatie van de klasse TestHandler.

Voorbeeld 8.10.

```
package voorbeelden.voorbeeld3;

import org.apache.axis.AxisFault;
import org.apache.axis.Handler;
import org.apache.axis.MessageContext;
import org.apache.axis.handlers.BasicHandler;

import org.apache.axis.Message;
import org.apache.axis.message.SOAPEnvelope;

import java.io.FileOutputStream;
import java.io.PrintWriter;
import java.util.Date;

public class TestHandler extends BasicHandler {
    public void invoke(MessageContext msgContext) throws AxisFault
    {
        try {
            Message requestMsg = msgContext.getRequestMessage();
            SOAPEnvelope env = requestMsg.getSOAPEnvelope();

            String filename = (String)getOption("filename");
            FileOutputStream fos = new FileOutputStream(filename, true);
            PrintWriter writer = new PrintWriter(fos);

            String result = env.toString();
            writer.println(result);
            writer.close();
        } catch (Exception e) {
            throw AxisFault.makeFault(e);
        }
    }
}
```

Deze handler krijgt een filename mee als parameter en gebruikt die file om een log te maken van de SOAP message die hij binnenkrijgt. Daarna wordt de SOAP message gewoon verder gestuurd. Naast het nemen van een log van voorbijkomende SOAP messages kunnen ook tal van andere operaties uitgevoerd worden zoals bijvoorbeeld het verwerken, toevoegen of verwijderen van headers, encryptie van het body-element, enz...

Als laatste hebben we dan natuurlijk nog een oproepende client nodig: (zie voorbeeld 8.11)

Voorbeeld 8.11.

```
package voorbeelden.voorbeeld3;
```

```
import org.apache.axis.client.Call;
import org.apache.axis.client.Service;
import org.apache.axis.encoding.XMLType;
import org.apache.axis.utils.Options;

import javax.xml.namespace.QName;
import javax.xml.rpc.ParameterMode;

public class Client {
    public static void main(String [] args) {
        try {
            Options options = new Options(args);
            String endpointURL = options.getURL();

            Service service = new Service();
            Call call = (Call) service.createCall();

            call.setTargetEndpointAddress( new java.net.URL(endpointURL) );
            call.setOperationName( new QName("TestService", "testMethod") );
            call.addParameter("arg1", XMLType.XSD_STRING, ParameterMode.IN);
            call.setReturnType(XMLType.XSD_STRING);

            String res = (String) call.invoke( new Object[] { "Hello" } );
            System.out.println( res );
        } catch (Exception e) {
            System.err.println(e.toString());
        }
    }
}
```

Na het uitvoeren van bovenstaande Client heeft de file TestService.log volgende inhoud:

Voorbeeld 8.12.

```
<soapenv:Envelope
        xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
        xmlns:xsd="http://www.w3.org/2001/XMLSchema"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <soapenv:Body>
    <ns1:testMethod
         soapenv:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
         xmlns:ns1="TestService">
       <arg1 xsi:type="xsd:string">Hello</arg1>
    </ns1:testMethod>
  </soapenv:Body>
</soapenv:Envelope>
```

Dit is slecht een greep uit de uitgebreide mogelijkheden van Apache AXIS. Apache Axis is vrij te downloaden op: http://ws.apache.org/axis/

Hoofdstuk 9

Conclusie

Ik vind de principes en de ideeën achter web services zeer goed. Er zijn standaarden voorzien voor:

- De communicatie met en tussen web services (SOAP)
- Het beschrijven van web services (WSDL), waardoor communicatie kan geautomatiseerd worden
- Het zoeken naar en publiceren van service descriptions a.d.h.v. een registry (UDDI). Dit registry houdt allerhande service descriptions bij en biedt ook de mogelijkheid om aan dynamic binding te doen.

Daarnaast zijn er tal van protocollen, afkomstig uit de traditionele middleware, opnieuw gedefinieerd. Web services hebben echter het grote voordeel dat ze in tegenstelling tot traditionele middleware, wel geschikt zijn voor gebruik over verschillende netwerken, inclusief het Internet.

Naarmate protocollen zoals WS-Adressing, WS-routing, WS-Transaction, WS-Coordination, WS-Composition, ... meer en meer gestandaardiseerd worden zullen de mogelijkheden van web services steeds meer overeenkomen met die van traditionele middleware systemen en zullen ook steeds meer bedrijven het potentieel van web services inzien.

Web services kunnen in principe op verschillende niveau's toegepast worden en kunnen uiteindelijk de taken van de traditionele middleware overnemen.

Er zijn momenteel verschillende grote software bedrijven die pakketten aanbieden voor het implementeren, deployen en consumeren van web services. Enkele voorbeelden hiervan zijn:

- **IBM**: http://www-306.ibm.com/software/websphere/
- **Oracle**: http://www.oracle.com/technology/products/jdev/index.html
- **Sun**: http://java.sun.com/webservices/index.jsp
- **Novell**: http://www.novell.com/products/extend/connectivity.html
- **Microsoft**: http://www.microsoft.com/net/

Het feit dat dergelijke grote bedrijven software schrijven voor het maken en gebruiken van web services wijst erop dat ook zij geloven in de toekomst van web services. Op de site http://www-306.ibm.com/software/success/ zijn verhalen te vinden van succesvolle toepassingen van web services.

We kunnen dus in het algemeen zeggen dat:

1. De mogelijkheden van web services zeer uitgebreid zijn.
2. Web services een oplossing bieden voor bestaande problemen.
3. Er open standaarden ontwikkeld werden.
4. Er ondersteuning is door grote software bedrijven.
5. Bestaande toepassingen hergebruikt kunnen worden door ze als web services te verpakken.

Alle voorwaarden voor succes zijn daarmee voldaan. Web services lijken een gouden tijd tegemoet te gaan. De toekomst zal moeten uitwijzen of web services deze hoge verwachtingen ook daadwerkelijk kunnen inlossen.

Bijlage A

Publisher API

De Publisher API definieert de volgende operaties waarbij * staat voor 0 of meer, + voor 1 of meer, ? voor 0 of 1 en | voor of:

- **add_publisherAssertions((authInfo?, publisherAssertion+)+)**: voor het toevoegen van een publisherAssertion. Een publisherAssertion bevat steeds een fromKey en een toKey die beide naar een businessEntity verwijzen. De publisher moet ook de eigenaar zijn van één van de betrokken businessEntities. Indien de publisher eigenaar is van beide businessEntities dan wordt de assertion meteen toegevoegd. In het andere geval wordt gewacht tot de andere businessEntity een overeenkomstige publisher assertion toevoegt.
- **delete_binding((authInfo?, bindingKey+)+)**: Voor het verwijderen van het bindingTemplate element dat overeenkomt met de gegeven bindingKey.
- **delete_business((authInfo?, businessKey+)+)**: Voor het verwijderen van de businessEntity die overeenkomt met de gegeven businessKey.
- **delete_publisherAssertions((authInfo?, publisherAssertion+)+)**: Voor het verwijderen van de publisherAssertion die overeenkomt met de gegeven publisherAssertion.
- **delete_service((authInfo?, serviceKey+)+)**: Voor het verwijderen van de businessService die overeenkomt met de gegeven serviceKey.
- **delete_tModel((authInfo?, tModelKey+)+)**: Voor het verwijderen van het tModel dat overeenkomt met de gegeven tModelKey. In feite wordt de tModel entiteit niet verwijderd maar gewoon verborgen zodat er later opnieuw naar kan gerefereerd worden.

- **get_assertionStatusReport((authInfo?, completionStatus?)+)**: Er zijn 3 soorten completion status:
  - **complete**: geeft alle completed publisher assertions terug.
  - **toKey_incomplete**: geeft alle publisher assertions terug waarvoor de businessEntity die overeenkomt met de toKey nog geen overeenkomstige assertion gemaakt heeft.
  - **fromKey_incomplete**: geeft alle publisher assertions terug waarvoor de businessEntity die overeenkomt met de fromKey nog geen overeenkomstige assertion gemaakt heeft.
- **get_publisherAssertions((authInfo?)+)**: Geeft alle assertions terug die geassocieerd zijn met de huidige publisher.
- **get_registeredInfo((authInfo?)+)**: Geeft een lijst van alle businessEntities en tModels waarvan de huidige publisher de eigenaar is.
- **save_binding((authInfo?, bindingTemplate+)+)**: Voor het bewaren of updaten van een bindingTemplate.
- **save_business((authInfo?, businessEntity+)+)**: Voor het bewaren of updaten van een businessEntity.
- **save_service((authInfo?, businessService+)+)**: Voor het bewaren of updaten van een businessService element.
- **save_tModel((authInfo?, tModel+)+)**: Voor het bewaren of updaten van een tModel.
- **set_publisherAssertions((authInfo?, publisherAssertion*)+)**: De tweede parameter is hier een lijst van assertions. De werking is als volgt: elke publisher assertion in de lijst die nog niet bestond wordt toegevoegd en elke bestaande publisher assertion die niet voorkomt in de lijst wordt verwijderd.

In bovenstaande voorbeelden is authInfo een optioneel argument dat afhankelijk van de implementatie wel of niet meegegeven moet worden. Ook zijn enkel de basis gevallen per operatie beschreven.

Bijlage B

Custody en ownership API

De Custody en ownership API definieert de volgende operaties waarbij * staat voor 0 of meer, + voor 1 of meer, ? voor 0 of 1 en | voor of:

- **get_transferToken((authInfo?, keyBag)+)**: het keyBag element bevat één of meerdere businessKeys of tModelKeys. Als de publisher geïdentificeerd door authInfo effectief de eigenaar is van de businessEntities en/of tModels dan wordt een transferToken teruggegeven. Dit token wordt daarna aan de toekomstige eigenaar doorgegeven die het dan kan gebruiken als parameter in de transfer_entities operatie.

- **transfer_entities((authInfo?, transferToken, keyBag)+)**: het keyBag element bevat ook hier één of meerdere businessKeys of tModelKeys. Als alles in orde is wat betreft de authentication en custody tokens worden de betreffende entiteiten naar de nieuwe eigenaar overgedragen.

- **transfer_custody((transferToken, keyBag, transferOperationalInfo)+)**: de transferOperationalInfo parameter bevat onder andere de NodeID van de UDDI node die na afloop de custody over de betreffende entiteiten zal krijgen. De tranfer_custody operatie wordt gebruik voor het overdragen van de custody van de ene UDDI node naar de andere. De UDDI node die deze oproep ontvangt moet indien het transferToken correct is de custody overgeven aan de UDDI node die vermeld is binnen het transferOperationalInfo element.

- **discard_transferToken((authInfo, (transferToken | keyBag))+)**: wordt gebruikt om een transferToken ongeldig te verklaren. Dit kan aan de hand van het transferToken zelf maar ook d.m.v. een set van één of meerdere businessKeys of tModelsKeys.

Bijlage C

Inquiry API

De Inquiry API definieert de volgende operaties:

- **Find_binding**: wordt gebruikt voor het localiseren van bindingTemplates binnen een gegeven businessService. De return waarde van find_binding is een bindingDetail structure.
- **Find_business**: wordt gebruikt om informatie te vinden in verband met één of meerdere businesses. De return waarde van find_business is een businessList structure.
- **Find_relatedBusinesses**: gaat op zoek naar businesses die verwant zijn met een gegeven business key. Voor deze verwantschappen wordt gezocht naar dochterondernemingen of andere departementen van een gegeven business. De return waarde van find_relatedBusinesses is een een lijst van businesses.
- **Find_service**: gaat op zoek naar specifieke services binnen een gegeven businessEntity. De return waarde van find_service is een serviceList structure.
- **Find_tModel**: gaat op zoek naar en returnt tModel structures.
- **Get_bindingDetail**: gaat op zoek naar en returnt bindingTemplate informatie. Geeft als return waarde een bindingDetail structure die voldoende informatie bevat om een service te kunnen oproepen.
- **Get_businessDetail**: geeft alle details van een gegeven, geregistreerde business. Returnt een businessDetail structure.
- **Get_operationalInfo**: wordt gebruikt voor het verkrijgen van operational informatie behorend tot één of meerdere entities in het registry. De returnwaarde is een operationInfos structure.

- **Get_serviceDetail**: geeft alle details van één of meerdere geregistreerde businessServices. Return waarde is een serviceDetail structure.
- **Get_tModelDetail**: geeft alle details van een gegeven verzameling van tModel data. Geeft een tModelDetail structure terug.

Bijlage D

Replication API

Elke aanpassing (change record) die in een bepaalde UDDI node gebeurt krijgt een volgnummer toegewezen. Dit volgnummer wordt het "Origination Update Sequence Number" **(USN)** genoemd.

Een **change record** wordt aangemaakt telkens er een verandering in een UDDI node gebeurt. Veranderingen kunnen ondermeer zijn: updates, toevoegen, verwijderen van businessEntities, tModels,... De gegevens die in een change record worden bijgehouden zijn:

1. **NodeId**: Id van de node waar het change record ontstaan is.
2. **Originating USN**: door de originating node aangemaakte identifier.
3. **Data**: beschrijving van de aanpassing.

In elke UDDI node wordt een lijst van <UDDI node, USN> bijgehouden, de highWaterMark vector. Zodat de node voor elke andere node kan bijhouden welke de laatste wijziging was.

De **highWaterMark vector** heeft voor elke UDDI node 1 entry in de vector. De gevens die in de entry worden bijgehouden zijn:

1. **OperatorNodeID**: identifier van de UDDI node
2. **Originating USN**: de originating USN van de meest recente aanpassing van de overeenkomstige UDDI node.

Een highWaterMark vector wordt dus gebruikt om per UDDI node aan te geven tot en met welke aanpassing de gegevens up to date zijn, en welke aanpassingen er dus nog moeten gebeuren.

De Replication API definieert de volgende operaties waarbij * staat voor 0 of meer, + voor 1 of meer en | voor of:

- **get_changeRecords((requestingNode, highWaterMark*, (responseLimitCount | responseLimitVector))+)**: wordt gebruikt om de changeRecords van een bepaalde node op te vragen. De requestingNode geeft aan naar waar het antwoord teruggestuurd moet worden. De optionele highWaterMark vector geeft aan welke aanpassingen al doorgevoerd zijn. Het responseLimitCount of responseLimitVector element geven aan hoeveel change records er maximaal teruggestuurd mogen worden.

- **notify_changeRecordsAvailable((notifyingNode, highWaterMark*)+)**: wordt gebruikt om andere nodes te laten weten dat er nieuwe change records beschikbaar zijn. De notifyingNode parameter geeft aan welke node deze change records beschikbaar heeft. De node die deze operatie oproept geeft ook een highWaterMark vector mee om zowel de locale als andere aanpassingen door te geven. Deze operatie gaat typisch vooraf aan de get_changeRecords() oproep.

- **do_ping()**: wordt gebruikt om het bestaan van een bepaalde node na te gaan.

- **get_highWaterMarks()**: wordt gebruikt om een lijst van highWaterMark elementen op te vragen.

Bibliografie

[1] http://uddi.org/schema/uddi_v3.xsd.

[2] Maintenance agency for iso 3166 country codes. http://www.iso.org/iso/en/prods-services/iso3166ma/index.html.

[3] North american industry classification system (naics). http://www.census.gov/epcd/www/naics.html.

[4] Oasis uddi specifications tc - committee specifications. http://www.oasis-open.org/committees/uddi-spec/doc/tcspecs.htm.

[5] *SOAP Tutorial*. http://www.w3schools.com/soap/default.asp.

[6] Unspsc. http://www.eccma.org/new/?page_id=32.

[7] Webservices - axis, 2000-2005. http://ws.apache.org/axis/.

[8] Welcome to the jakarta project, 2000-2005. http://jakarta.apache.org/.

[9] Direct internet message encapsulation (dime), 2002. http://msdn.microsoft.com/library/en-us/dnglobspec/html/draft-nielsen-dime-02.txt.

[10] *The Java Web Services Tutorial 1.0*, 2002. http://java.sun.com/webservices/docs/1.0/tutorial/.

[11] Web services architecture requirements, 2002. http://www.w3.org/TR/2002/WD-wsa-reqs-20020429.

[12] *Business Process Execution Language for Web Services Version 1.1*, 2003. ftp://www6.software.ibm.com/software/developer/library/ws-bpel.pdf.

[13] *Glossary of "SOAP Version 1.2 Part 1: Messaging Framework"*, 2003. http://www.w3.org/2003/glossary/subglossary/soap12-part1.rdf/.

[14] *Web Services Atomic Transaction (WSAtomicTransaction)*, 2003. ftp://www6.software.ibm.com/software/developer/library/ws-atomictransaction200309.pdf.

[15] *Web Services Business Activity Framework (WS-BusinessActivity)*, 2004. ftp://www6.software.ibm.com/software/developer/library/WS-BusinessActivity.pdf.

[16] *Web Services Coordination (WS-Coordination)*, 2004. ftp://www6.software.ibm.com/software/developer/library/WS-Coordination.pdf.

[17] *Web Services Policy Framework (WS-Policy)*, 2004. ftp://www6.software.ibm.com/software/developer/library/ws-policy.pdf.

[18] *Web Services Transactions specifications*, 2004. http://www-128.ibm.com/developerworks/library/specification/ws-tx/.

[19] Project: uddi4j: Summary, 2005. http://sourceforge.net/projects/uddi4j.

[20] *Web Services Conversation Language (WSCL) 1.0*, W3C. http://www.w3.org/TR/2002/NOTE-wscl10-20020314/.

[21] Gustavo Alonso, Fabio Casati, Harumi Kuno, and Vijay Machiraju. *Web Services*. Springer, 2004.

[22] Tom Bellwood, Luc Clement, David Ehnebuske, Andrew Hately, Maryann Hondo, Yin Leng Husband, Karsten Januszewski, Sam Lee, Barbara McKee, Joel Munter, and Claus von Riegen. *UDDI Version 3.0 UDDI Spec Technical Committee Specification, 19 July 2002*, 2002.

[23] Peter Brittenham, Francisco Cubera, Dave Ehnebuske, and Steve Graham. *Understanding WSDL in a UDDI registry*, 2002. http://www-128.ibm.com/developerworks/webservices/library/ws-wsdl/.

[24] Erik Christensen, Francisco Curbera, Greg Meredith, and Sanjavi Weerawarana. *Web Services Description Language (WSDL) 1.1*, 2001. http://www.w3.org/TR/wsdl.

[25] George Coulouris, Jean Dollimore, and Tim Kindberg. *Distributed Systems*, pages 520–523. Addison Wesley, 2001.

[26] Richard Dragan, Timothy Dyck, Sahil Gambhir, Michael Muchmore, and Matthew Sarrel. Webservices: Deel2. *PC Magazine*, pages 96–114, Januari 2004.

[27] Richard Dragan and Matthew Sarrel. Webservices: Deel1. *PC Magazine*, pages 106–120, December 2003.

[28] Hector Garcia-Molina, Jefrey D. Ullman, and Jennifer Widom. *Database Systems the complete Book*, pages 13–14, 397–408, 917–924, 1024–1028. Prentice Hall, 2002.

[29] Steve Graham, Doug Davis, Simeon Simeonov, Glen Daniels, Peter Brittenham, Yuichi Nakamura, Paul Fremantle, Dieter Konig, and Claudia Zentner. *Building Web Services With Java*. Sams Publishing, 2004.

[30] W3C XML Protocol Working Group. *SOAP Version 1.2 Part 0: Primer*, 2003. http://www.w3.org/TR/2003/REC-soap12-part0-20030624/.

[31] W3C XML Protocol Working Group. *SOAP Version 1.2 Part 1: Messaging Framework*, 2003. http://www.w3.org/TR/2003/REC-soap12-part1-20030624/.

[32] W3C XML Protocol Working Group. *SOAP Version 1.2 Part 2: Adjuncts*, 2003. http://www.w3.org/TR/2003/REC-soap12-part2-20030624/.

[33] Robert Lipschutz. Webservices: Deel3. *PC Magazine*, pages 100–107, Februari 2004.

[34] Eric Newcomer. *Understanding Web Services*. Addison-Wesley, 2002.

[35] David Orchard. *SOAP HTTP GET Binding Version 0.1*, 2002. http://www.w3.org/2001/tag/doc/ws-uri-05042002.html.

[36] Doug Tidwell, James Snell, and Pavel Kulchenko. *Programming Web Services with SOAP*. O'Reilly, 2001.

[37] Venu Vasudevan. *A Web Services Primer*, 2001. http://webservices.xml.com/pub/a/ws/2001/04/04/webservices/index.html.