XMPP en het real-time web

Robin Rutten (0712620)
<RobinRutten@student.ru.nl>

Begeleider: dr. Theo Schouten

21 juni 2010

Inhoudsopgave

1 XMPP en het real-time web **3**
- 1.1 Inleiding . . . 3
- 1.2 Probleemstelling . . . 4
- 1.3 Overzicht . . . 4

2 Introductie XMPP **5**
- 2.1 Waarom XMPP? . . . 5
- 2.2 Geschiedenis . . . 6
- 2.3 Basisprincipes . . . 7

3 Kernprotocol **9**
- 3.1 Overzicht . . . 9
- 3.2 Architectuur . . . 9
  - 3.2.1 Server . . . 9
  - 3.2.2 Client . . . 10
- 3.3 XMPP Adressen . . . 11
  - 3.3.1 Domain Identifier . . . 11
  - 3.3.2 Node Identifier . . . 11
  - 3.3.3 Resouce Identifier . . . 11
  - 3.3.4 bare en full JID . . . 11
- 3.4 XML Streams: uitwisseling van gegevens . . . 11
- 3.5 XML Stanzas . . . 12
  - 3.5.1 XML Namespaces . . . 12
  - 3.5.2 Algemene attributen . . . 13
  - 3.5.3 Message stanza . . . 13
  - 3.5.4 Presence stanza . . . 14
  - 3.5.5 Info/Query stanza . . . 15

4 Publish-Subscribe uitbreiding **17**
- 4.1 Inleiding . . . 17
- 4.2 Node types . . . 17
- 4.3 Node adres . . . 18
  - 4.3.1 JID . . . 18
  - 4.3.2 JID en NodeID . . . 18

4.4 Node Access Models 18
4.5 Event types 19
4.6 Verkennen 19
4.7 Publiceren 21
4.8 Overige functionaliteiten 23

5 XMPP via het HTTP protocol 24
5.1 Inleiding 24
5.2 De techniek achter BOSH 25
5.3 Het opzetten van BOSH Sessie 26

6 Operational Transformation 28
6.1 Inleiding 28
6.2 Requirements 28
6.2.1 Convergence requirement 29
6.2.2 Intention-preservation requirement 29
6.2.3 Causality-preservation requirement 29
6.3 Operational transformation (OT) 29
6.4 Situatie 29
6.4.1 De naïeve manier 29
6.4.2 De correcte manier 30
6.5 OT framework 31
6.5.1 Control algoritmes 31
6.5.2 Transformatiefuncties 32
6.6 dOPT algoritme 33
6.7 Jupiter algoritme 33
6.7.1 Algoritme 37
6.8 Google Wave OT algoritme 39
6.8.1 Operaties 39
6.8.2 Transformatie 41
6.8.3 Compositie 41
6.8.4 Werking van het algoritme 42

7 XMPP en Operational Transformation 47
7.1 Wave Federation Protocol 47
7.2 Uitbreiding 49
7.2.1 Architectuur 49
7.2.2 Client-Server communicatie 51
7.2.3 Communicatie remote component 54
7.2.4 Communicatie host component 56
7.2.5 Karakteristieke elementen 57

8 Conclusie en verder onderzoek 59
8.1 Conclusie 59
8.2 Verder onderzoek 60

Hoofdstuk 1

XMPP en het real-time web

1.1 Inleiding

Het internet, met name zoals we dat kennen binnen een webbrowser, was lange tijd zeer statisch in de zin dat informatie pas getoond werd als er specifiek om werd gevraagd. Nieuwe informatie verscheen pas als een website werd herladen. Dit statische karakter van het internet veranderde de laatste jaren al enigszins doordat steeds meer websites periodiek nieuwe informatie ophalen om de website te verversen. De nieuwste trend lijkt te zijn dat het internet verandert in een *real-time informatiebron*. Dit betekent dat informatie wordt ontvangen zodra deze wordt gepubliceerd, in plaats van dat er periodiek op updates gecontroleerd moet worden. Deze trend lijkt voornamelijk ingezet te zijn door het feit dat internet steeds meer wordt gebruikt als een communicatiemedium. Zo stijgt het gebruik van zogenaamde social media als Hyves, Facebook en Twitter snel. Men wil direct in contact staan met vrienden en kennissen en daarom direct op de hoogte worden gesteld van updates. Het real-time web staat ons onder andere toe om te chatten, spellen te spelen en samen te werken aan documenten. [1]

Het ontwikkelen van deze real-time toepassingen is echter niet zo gemakkelijk: protocollen die binnen de webbrowser gebruikt worden, zoals HTTP, zijn niet ontwikkeld voor real-time communicatie. Hier komt bij dat gelijktijdige acties door meerdere partijen aan hetzelfde object (bijvoorbeeld een tekstdocument) moeten worden gesynchroniseerd. Achter veel toepassingen die we op het web tegenkomen gaan dezelfde technieken en praktijken schuil. Het protocol dat in dit onderzoek besproken zal worden is het *eXtensible Messaging and Presence Protocol (XMPP)*. Enkele voorbeelden van toepassingen die hiervan gebruik maken zijn:

[1] Het is zo dat deze toepassingen binnen webbrowsers al langer mogelijk zijn met externe plugins zoals Flash. Het protocol dat in dit onderzoek besproken wordt kan in combinatie met Flash gebruikt worden. Daarnaast kan het protocol van HTTP gebruik maken om gegevens te verzenden en te ontvangen. Het kan hierdoor in combinatie met de scriptingtaal Javascript binnen een webbrowser toegepast worden zonder extra plugins.

Chesspark Een online schaakspel waarbij gebruikers via de browser tegen elkaar kunnen schaken.

Drop.io Drop.io richt zich op het in real-time delen van online bestanden.

Google Wave Met deze applicatie kan men onder andere tegelijkertijd met meerdere mensen aan eenzelfde document werken. Het Google Wave Federation Protocol is een open uitbreiding op het XMPP protocol. Het achterliggende algoritme om gelijktijdige operaties van verschillende partijen te synchroniseren zal in dit onderzoek besproken worden.

1.2 Probleemstelling

In het onderzoek staat de volgende vraag centraal: *Hoe kan het eXtensible Messaging and Presence Protocol (XMPP), en (zelf voorgestelde) uitbreidingen hierop, ingezet worden voor real-time communicatie en collaboratie op het web?*. Hierbij worden de volgende definities gehanteerd:

Real-time communicatie directe of nagenoeg onvertraagde communicatie tussen twee of meer partijen.

Real-time collaboratie gelijktijdige samenwerking tussen twee of meer partijen aan hetzelfde object waarbij wijzigingen direct, met verwaarloosbare latentie, tussen verschillende partijen worden gedeeld en gesynchroniseerd.

1.3 Overzicht

Grotendeels is het onderzoek beschrijvend van aard. Hoofdstuk 3 gaat in op het kernprotocol achter XMPP. In de hoofdstukken 4 en 5 worden voor dit onderzoek relevante uitbreidingen op het kernprotocol besproken. Hoofdstuk 6 bespreekt Operational Transformation (OT), een framework voor concurrency control. Het belang hiervan voor real-time collaboratie wordt in hoofdstuk 7 behandeld. Daarnaast definieert hoofdstuk 7 een uitbreiding voor XMPP, deze uitbreiding moet het op grote schaal toepassen van Operational Transformation voor verschillende documenttypes vergemakkelijken. Tot slot geeft hoofdstuk 8 op grond van de bevindingen een antwoord op de onderzoeksvraag en bespreekt het aanknopingspunten voor verder onderzoek.

Eerst zal nu een introductie tot XMPP gegeven worden. Deze introductie is onder andere bedoeld om de keuze voor XMPP duidelijk te maken en schetst een beknopt overzicht van de geschiedenis van het protocol.

Hoofdstuk 2

Introductie XMPP

2.1 Waarom XMPP?

In de inleiding is opgemerkt dat het internet lange tijd zeer statisch van karakter was. Hier is de kanttekening bij geplaatst dat hiermee vooral het internet binnen een webbrowser bedoeld wordt. Om dit duidelijk te maken, wordt eerst kort de architectuur van het internet besproken. De verzameling van protocollen die gebruikt worden voor het internet en vergelijkbare netwerken staan wel bekend onder de naam *TCP over IP (TCP/IP)*. Deze naam verwijst naar de twee belangrijkste protocollen: het *Transmission Control Protocol (TCP)* en het *Internet Protocol (IP)*. De protocollen zijn onderverveeld in verschillende lagen, de twee belangrijkste lagen voor dit onderzoek zijn de *Transportlaag* en de *Toepassingslaag*. Op de Transportlaag bevinden zich protocollen die grofweg verantwoordelijk zijn voor het overbrengen van gegegevens van de toepassingslaag tussen verschillende partijen. Het meest gebruikte protocol van deze laag is het Transmission Control Protocol (TCP). Op de Toepassingslaag vindt men protocollen die 'diensten bieden aan de toepassing'. Deze protocollen maken gebruik van protocollen van de Transportlaag om onderliggende verbindingen tot stand te brengen.

Op de transportlaag zijn verschillende protocollen te vinden die real-time communicatie bieden. Maar weinig van deze protocollen zijn open standaarden, waaronder XMPP en *SIMPLE*. Veel van deze protocollen gebruiken TCP als transportlaag. Het *Hypertext Transfer Protocol (HTTP)* protocol, dat binnen een webbrowser wordt gebruikt voor communicatie tussen een client en webserver, bevindt zich ook op de toepassingslaag. Het is een request-response protocol waarbij een verzoek van de client wordt beantwoord door de server. Dit request-reponse mechanisme is niet bedoeld voor real-time communicatie. Om XMPP direct in een browser te gebruiken zal de browser XMPP ondersteuning daarom moeten inbouwen. Een andere mogelijkheid is om XMPP, indirect, in combinatie met HTTP te gebruiken, dit zal worden besproken in hoofdstuk 5.

De keuze voor XMPP als communicatieprotocol is gemaakt op basis van de

sterke eigenschappen van het protocol (open, uitbreidbaar) en het feit dat het reeds de basis vormt van verschillende real-time toepassingen op het web. Er zal in dit onderzoek bewust geen vergelijkende studie gedaan worden met andere protocollen. De focus ligt op een diepgaande bespreking van de technieken en methoden. De volgende paragraaf geeft een beknopt overzicht van de geschiedenis van XMPP. Dit overzicht laat onder andere het belang zien van een open protocol als XMPP.

2.2 Geschiedenis

XMPP ontstaat eind jaren negentig, in een tijd waarin diverse instant messaging systemen op de markt verkrijgbaar zijn. In 1996 introduceerde Mirabilis ICQ. Een jaar later kwam AOL met de AOL Instant Messanger. Yahoo lanceerde in 1998 haar Yahoo Pager (Yahoo Messenger). Microsoft volgde in 1999 met, het in Nederland veel gebruikte, MSN Messenger (later hernoemd naar Windows Live Messenger). Een groot nadeel van al deze systemen was dat ze niet goed gespecificeerd waren en daarnaast niet goed samenwerkten. Gebruikers van het ene Messaging systeem konden niet communiceren met gebruikers van het andere Messaging systeem. Voor Jeremie Miller was het een grote frustratie dat hij hierdoor meerdere Instant Messaging (IM) clients naast elkaar moest draaien.

In 1998 startte *Jeremie Miller* daarom met het Jabber project. In januari 1999 verscheen de eerste versie van de zogenaamde *jabberd* server. Er vormde zich al snel een enthousiaste community die begon met het ontwikkelen van eigen clients en servers. De details van de protocollen, die onder andere real-time XML streaming en messaging omvatten, werden verder uitgewerkt. In mei van het jaar 2000 waren de Jabber protocollen gestabiliseerd en verscheen de production release, versie 1.0, van de jabberd server.

Om het groeiende aantal open en commerciële Jabber projecten te coördineren werd in 2001 de *Jabber Software Foundation (JSF)* opgericht. Eind 2002 legde de JSF het core protocol voor aan het IETF proces. Daarnaast werd een IETF working group opgericht om de formalisatie van het protocol kracht bij te zetten. De protocollen werden verzameld onder de neutrale naam *eXtensible Messaging and Presence Protocol (XMPP).*

Uiteindelijk werd het protocol in 2004 goedgekeurd als standaard. In oktober 2004 werden de volgende documenten gepubliceerd:

RFC 3920 Extensible Messaging and Presence Protocol (XMPP): Core

RFC 3921 Extensible Messaging and Presence Protocol (XMPP): Instant Messaging and Presence

RFC 3922 Mapping the Extensible Messaging and Presence Protocol (XMPP) to Common Presence and Instant Messaging (CPIM)

RFC 3923 End-to-End Signing and Object Encryption for the Extensible Messaging and Presence Protocol (XMPP)

Begin 2007 veranderde de Jabber Software Foundation haar naam in de *XMPP Standards Foundation* (*XSF*), om de focus op het ontwikkelen van open uitbreidingen op het XMPP kernprotocol te benadrukken. Op dit moment kent XMPP vele uitbreidingen en wordt het ondere andere toegepast in producten van onder andere Apple, Google, Nokia en IBM.

2.3 Basisprincipes

Het *eXtensible Messaging and Presence Protocol (XMPP)* is een open, op XML gebaseerd, protocol voor real-time communicatie. In essentie is XMPP een verzameling van technologieën en methoden om (gestructureerde) gegevens te streamen over een netwerk. Het protocol is oorspronkelijk ontworpen als alternatief voor Instant Messaging systemen zoals ICQ, Yahoo Messenger en MSN. Dit is terug te zien in de specificatie van het kernprotocol (RFCs 3920 en 3921), die vooral gericht is op messaging en presence functionaliteiten. Onder leiding van de XMPP Standards Foundation zijn diverse uitbreidingen, zogenoemde XMPP Extension Protocols (XEPs), gedefinieerd die nieuwe toepassingen mogelijk maken. Dankzij deze uitbreidingen kan XMPP nu ook ingezet worden voor onder andere multi-party chat, voice/video calls, collaboration, content syndication en file transfer.

Net als andere communicatieprotocollen definieert XMPP een formaat voor de uitwisseling van gegevens tussen twee of meer partijen. Deze communicatiepartijen zijn doorgaans een client en een server. Maar ook communicatie tussen twee servers of clients onderling behoort tot de mogelijkheden. De data die uitgewisseld wordt is opgemaakt volgens de *Extensible Markup Language (XML)*. Met XML kunnen gestructureerde gegevens weergegeven worden in de vorm van platte tekst. Een discrete semantische eenheid van gestructureerde informatie wordt in XMPP een *XML stanza* (paragraaf 3.5) genoemd. Het core protocol kent drie verschillende types XML stanzas: de message, presence en iq stanza. Via zogenaamde *XML streams* (paragraaf 3.4) worden één of meerdere XML stanzas tussen twee partijen over het netwerk verstuurd.

In het *OSI-model* vinden we XMPP terug in de toepassingslaag. De toepassingslaag vormt een verbinding tussen het netwerk en de applicatie. XMPP bevindt zich hiermee op dezelfde laag als (bekende) protocollen zoals SMTP, IMAP, FTP en HTTP. XMPP is niet gebonden aan een specifieke netwerk architectuur. Gebruikelijk wordt er gekozen voor een client-server implementatie waarbij TCP als transportlaag wordt gebruikt. Dit houdt in dat een client over een TCP verbinding met de server communiceert, evenals servers onderling (via twee TCP verbindingen). Verbindingen kunnen ook op een andere manier tot stand gebracht worden, zoals bijvoorbeeld via het HTTP protocol. Hoofdstuk 5 gaat hier verder op in.

De belangrijkste eigenschappen van het protocol, waarmee bij het ontwerp rekening is gehouden, zijn:

Open Iedereen kan zijn eigen XMPP applicatie of server schrijven. De specificaties zijn publiek beschikbaar.

Gedecentraliseerd De architectuur is te vergelijken met die van e-mail. Er is geen centrale hoofdserver. Iedereen kan zijn eigen XMPP-server draaien. Al deze servers kunnen met elkaar communiceren en worden als het ware verbonden tot een groot netwerk.

Secure Er worden verschillende erkende beveiligingsprotocollen ondersteund. (SASL en TLS)

Uitbreidbaar Het gebruik van XML (Namespaces) geeft iedereen de mogelijkheid aangepaste functionaliteiten bovenop het kernprotocol te bouwen.

Hoofdstuk 3

Kernprotocol

3.1 Overzicht

Dit hoofdstuk geeft een overzicht van de kerntechnologieën van XMPP zoals die zijn gedefinieerd in RFCs 3920 [6] en 3921 [7]. Deze bieden onder andere:

- De streaming laag (3.4),
- Encryptie gebruikend makend van het [Transport Layer Security protocol] (TLS),
- Authenticatie volgens het [Simple Authentication and Security Layer protocol] (SASL),
- Messaging en Presence notification

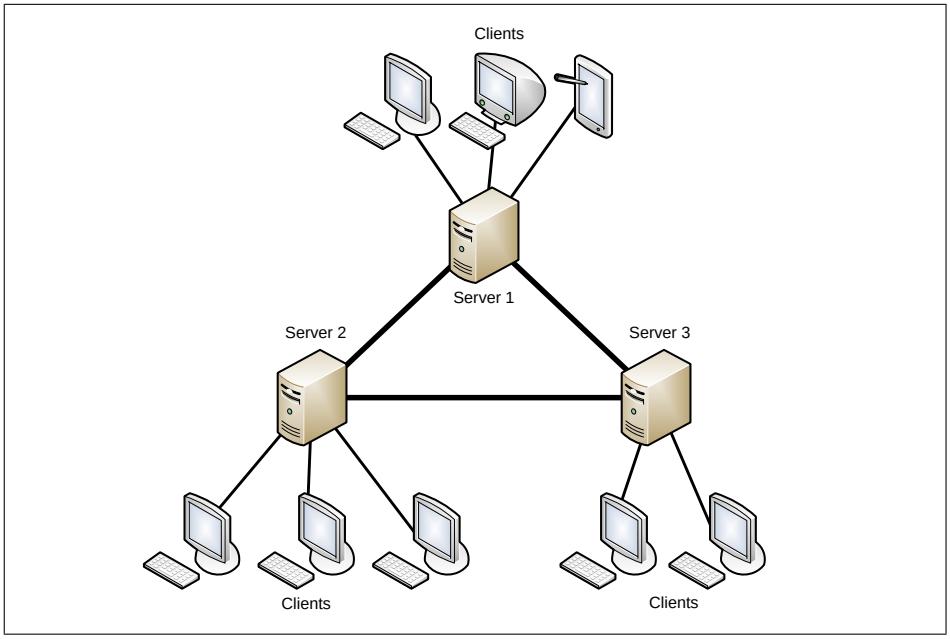
Encryptie en authenticatie worden niet verder besproken omdat deze niet van belang zijn in ofwel het vervolg van het onderzoek ofwel voor het begrip van XMPP. De volgende paragraaf begint met de bespreking van de algemene architectuur van XMPP.

3.2 Architectuur

Standaard kent XMPP twee soorten communicatiepartijen: servers en clients. Belangrijk in de architectuur van XMPP is dat het verkeer niet wordt geregeld door een centrale server. Door meerdere servers te verbinden ontstaat een groot netwerk. De clients zijn verbonden met een eigen server. De verschillende servers transporteren berichten tussen clients. Zie figuur 3.1.

3.2.1 Server

Als we kijken naar de taakverdeling tussen de server en client wordt het meeste werk gedaan door de server. De belangrijkste verantwoordelijkheden van de server zijn:



Figuur 3.1: XMPP architetuur

- het beheren van verbindingen met clients en andere servers.
- het afleveren berichten op het juiste adres.
- meestal ook de opslag van gegevens van clients, zoals bijvoorbeeld contactenlijsten.

3.2.2 Client

Gebruikelijk maakt een client een directe XMPP verbinding met een server, maar ook andere verbindingen zijn mogelijk (zie hoofdstuk 5). Clients worden meestal aangedreven door de mens. Maar ook machinegedreven clients zijn mogelijk, deze worden ook wel *robots* genoemd.

Meerdere instanties van een client kunnen tegelijkertijd met dezelfde server verbinden. Zo kan er bijvoorbeeld op meerdere geografische locaties of apparaten een verbinding worden opgezet. Dit verschil in locatie of apparaat kan uitgedrukt worden via de *resource identifier* (zie paragraaf 3.3.3).

In de volgende paragraaf (3.3) zullen we zien dat elke entiteit, communicatiepartij, een uniek adres heeft waarmee deze benaderd kan worden.

3.3 XMPP Adressen

Elke entiteit is uniek adresseerbaar door een zogenaamde *Jabber Identifier (JID)*. Deze adressen hebben erg veel weg van e-mailadressen. Een JID is opgebouwd uit drie gedeeltes: de *domain identifier*, *node identifier*, en *resource identifier*.

3.3.1 Domain Identifier

De domain identifier is het enige element dat verplicht is voor elke JID. Deze identifier specificeert doorgaans de server waarmee andere partijen kunnen verbinden om data over te brengen. In RFC 3920 is opgenomen aan welke eisen een domain identifier precies moet voldoen. Een voorbeeld van een (fictieve) domain identifier is *xmpp.cs.ru.nl*.

3.3.2 Node Identifier

De node identifier is optioneel en wordt geplaatst voor de domain identifier, gescheiden door een '@'. Gebruikelijk representeert deze identifier de gebruiker die van de server gebruik maakt. Het kan echter ook andere entiteiten representeren zoals bijvoorbeeld een chat room bij multi-user chat. Elke chat room heeft in dat geval een eigen node identifier.

3.3.3 Resouce Identifier

Eerder is opgemerkt dat de resource identifier gebruikt kan worden om verbindingen van verschillende locaties of apparaten te onderscheiden. De resource identifier is optioneel en wordt geplaatst achter de domain identifier, gescheiden door het '/'-karakter. Per client kunnen meerdere resources tegelijk verbonden zijn met dezelfde server.

3.3.4 bare en full JID

JIDs zijn te onderscheiden in twee categorieën: *bare JIDs* en *full JIDs*. Een bare JID is een adres zonder resource identifier maar met domain en node identifier, in de vorm *[node@domain]*. Bij een full JID is naast de domain en node identifier ook de resource identifier aangegeven. Een full JID heeft de vorm *[node@domain/resource]*. Voorbeelden van respectievelijk een fictieve bare en full jid zijn *rrutten@xmpp.cs.ru.nl* en *rrutten@xmpp.cs.ru.nl/library*.

3.4 XML Streams: uitwisseling van gegevens

XML streams en *XML stanzas* zijn de twee begrippen die verantwoordelijk zijn voor het uitwisselen van relatief kleine hoeveelheden gestructureerde informatie tussen communicatiepartijen. Via een XML stream worden XML elementen tussen twee entiteiten uitgewisseld over een netwerk. Volgens de syntax van XML wordt een stream gestart met de <*stream*> tag en beëindigd met de

</*stream*> tag. Over een stream kunnen één of meerdere XML elementen worden verstuurd. Deze elementen zijn XML Stanzas of elementen om de verbinding tot stand te brengen. Een XML Stanza is een eenheid van gestructureerde informatie die verstuurd wordt van de ene naar de andere entiteit over een XML stream. Alle XML Stanzas bevinden zich op de eerste laag onder het <stream> element. Met andere woorden: ze zijn directe kinderknopen van het <stream> element. XML Stanzas kunnen op zichzelf weer kinderknopen bevatten die extra informatie verschaffen. Als in het vervolg gesproken wordt over een element dat een ander element bevat, betekent dit dat het twee element een kinderknoop is van het eerste element. Het kernprotocol kent drie verschillende stanzas: <*presence*>, <*message*> en <*iq*>. Het volgende voorbeeld laat zien hoe meerdere XML stanzas over een enkele XML stream kunnen worden verzonden:

```
<stream>
  <message to='somebody@example.com' type='chat'>
    <body>Hello World</body>
  </message>
  <iq type='get'>
    <query xmlns='jabber:iq:roster' />
  </iq>
  <presence type='unavailable' />
</stream>
```

3.5 XML Stanzas

3.5.1 XML Namespaces

In deze paragraaf zullen we zien dat de XML stanzas verschillende mechanismen bieden om data tussen verschillende communicatiepartijen uit te wisselen. De stanzas zijn standaard uitgerust met een beperkt aantal functionaliteiten. Om deze basisfunctionaliteiten uit te breiden maakt XMPP gebruik van XML namespaces. Met XML namespaces kunnen uniek genaamde XML elementen worden gemaakt. Deze elementen moeten voorzien zijn van een *xmlns* namespace attribuut.Message en Presence stanzas kunnen, bovenop de standaard elementen, een of meerdere van deze elementen bevatten. Een IQ stanza mag hoogstens één zo'n element bevatten.

Onderstaand voorbeeld toont een message stanza, met onderliggend een <*body/*> en <*html/*> element. We zullen zien dat het <body/> element een van de drie standaardelementen voor message stanzas is, het <html/> element daarentegen wordt gespecificeerd in de *XHTML-IM* uitbreiding (xep-0071). Deze uitbreiding biedt de mogelijkheid voor XHTML opmaak in berichten. Merk op dat het element is gespecificeerd met de 'http://jabber.org/protocol/xhtml-im' namespace.

```
<message>
  <body>hi!</body>
```

```
<html xmlns='http://jabber.org/protocol/xhtml-im'>
  <body xmlns='http://www.w3.org/1999/xhtml'>
    <p style='font-weight:bold'>hi!</p>
  </body>
</html>
</message>
```

In de volgende paragrafen zullen we er bij het beschrijven van de stanzas vanuit gaan dat er gebruik wordt gemaakt van de standaard namespaces ('jabber:client' of 'jabber:server') en zullen alleen de basisfunctionaliteiten besproken worden. De elementen hoeven hierbij niet te worden voorzien van een xmlns attribuut.

3.5.2 Algemene attributen

Voordat ingegaan wordt op de specifieke stanzas, wordt eerst een overzicht gegeven van de algemene attributen die alle stanzas gemeenschappelijk hebben.

to De to attribuut geeft de JID van de bedoelde ontvanger van de stanza aan. Wanneer er geen to attribuut de stanza opgenomen is, gaat de server er van uit dat het bericht voor de server zelf bedoeld is.

from De from attribuut geeft de JID aan van de zender van de stanza.

type Het type attribuut geeft gedetailleerde informatie over het doel of de context van de stanza. Elk van de drie basis stanzas heeft verschillende mogelijke waardes voor het type attribuut. De enige gemeenschappelijke waarde die alle stanzas kunnen hebben is error. Dit geeft aan dat de stanza een foutmelding als antwoord is op de ontvangen stanza van dezelfde soort.

id Stanzas kunnen een id attrribuut bevatten. Reacties op een stanza met een id attribuut moeten een id attribuut met dezelfde waarde bevatten. De waarde van de ID moet uniek genoeg zijn zodat de zender het kan gebruiken om reacties te onderscheiden.

xml:lang Als de stanza XML data bevat die getoond wordt aan de gebruiker moet de stanza een xml:lang attribuut bevatten, deze geeft de standaardtaal aan van de data.

3.5.3 Message stanza

De *<message/>* stanza kan gebruikt worden om informatie te *"pushen"* naar een andere entiteit. Het is de taak van server om het bericht af te leveren bij de beoogde ontvanger.

Type berichten

Het is aanbevolen een type attribuut op te nemen bij elk message. Dit type attribuut mag één van de volgende waardes bevatten:

chat Het bericht is verzonden in een één-op-één chatconversatie.

groupchat Het bericht is verzonden in een chatconversatie van meer dan twee gebruikers.

normal Het bericht is verzonden buiten het domein van een chatconversatie. Het verwacht dat de ontvanger dit bericht beantwoordt.

headline Het bericht is gegenereerd door een dienst die, al dan niet automatisch, content (bijvoorbeeld nieuws of RSS feeds) verspreidt.

error Er is een fout opgetreden met betrekking tot het voorgaande bericht van de server.

Onderliggende elementen

Standaard kan een <message/> stanza de volgende elementen bevatten: *<subject/>*, *<body/>* en *<thread/>*. Een <subject/> element bevat informatie over het onderwerp van het bericht. Er kunnen verschillende subject elementen opgenomen worden voor verschillende talen. Het <body/> element specificeert de (door de mens leesbare) content van het bericht. Ook hier geldt dat voor verschillende talen een element opgenomen kan worden. Tot slot kan het <thread/> element gebruikt worden om een bepaalde conversatie aan te geven. Elke bericht binnen een conversatie moet hiervoor dezelfde waarde van het thread element hebben.

3.5.4 Presence stanza

De *<presence/>* stanza wordt gebruikt voor het verzenden en ontvangen van informatie over de beschikbaarheid van entiteiten. Daarnaast worden presence stanzas gebruikt voor het tot stand brengen en beheren van abonnementen op presence notificatie. Het werkt op basis van het *publish-subscribe* principe, alle geïnteresseerde partijen worden hierbij op de hoogte gesteld bij presence veranderingen.

Presence type

Optioneel heeft de presence stanza een *type* attribuut. Een presence stanza zonder type wordt gebruikt om aan te geven dat de verzendende partij online en beschikbaar is. Het type attribuut moet één van de volgende waardes hebben:

unavailable De partij is niet langer beschikbaar.

subscribe De verzendende partij wil zich abonneren op de presence veranderingen van de ontvanger.

subscribed De verzendende partij staat de ontvanger toe presence notificaties te ontvangen.

unsubscribe De verzendende partij zegt zijn abonnement op en wil niet langer presence notificaties ontvangen.

unsubscribed Het verzoek is geweigerd of een eerder abonnement wordt ingetrokken.

probe De server vraagt in naam van de gebruiker de huidige presence van de ontvanger op.

error Een fout is opgetreden bij verwerken of afleveren van de voorgaande presence stanza.

Onderliggende elementen

De standaard (optionele) onderliggende elementen voor de <presence/> stanza zijn *<show/>*, *<status/>* en *<priority/>*. Het <show/> element geeft de status van de entiteit aan. Als het show element niet is opgenomen, wordt aangenomen dat de entiteit online en beschikbaar is. Mogelijke waardes voor het element zijn:

away De entiteit is (tijdelijk) afwezig.

chat De entiteit is geïnteresseerd om te chatten.

dnd De entiteit is bezet (Do Not Disturb).

xa De entiteit is afwezig voor een langere tijd.

De <status/> element geeft in natuurlijke taal aan wat de status van de entiteit is. Meestal wordt dit attribuut gebruikt in combinatie met het <show/> element. Het <priority/> element geeft de prioriteit van de specifieke resource (een entiteit kan via meerdere resources verbonden zijn, paragraaf 3.3.3) aan. Voor <message/> stanzas geldt bijvoorbeeld dat deze afgeleverd wordt bij de resource met de hoogste prioriteit.

3.5.5 Info/Query stanza

IQ, of *Info/Query* is een request en response mechanisme. Hiermee kan een entiteit verzoeken doen aan andere entiteiten waarop deze entiteiten reageren met een antwoord.

Elke stanza moet een *id* attribuut bevatten, zodat bij het antwoord aangegeven kan worden op welk verzoek het reageert. Er zijn vier verschillende waardes voor het type attribuut:

get De stanza is een verzoek voor informatie.

set Het bericht levert benodigde data of verandert waardes.

result De IQ stanza is een antwoord op een succesvol verzoek.

error Tijdens het verwerken of bij levering van het vorige get of set verzoek is een fout opgetreden.

De data van een IQ stanza wordt aangegeven door het eerst onderliggende element van de <iq> stanza, uitgerust met een namespace (paragraaf 3.5.1). Een voorbeeld hiervan is te zien in het volgende hoofdstuk (paragraaf 4.6).

Hoofdstuk 4

Publish-Subscribe uitbreiding

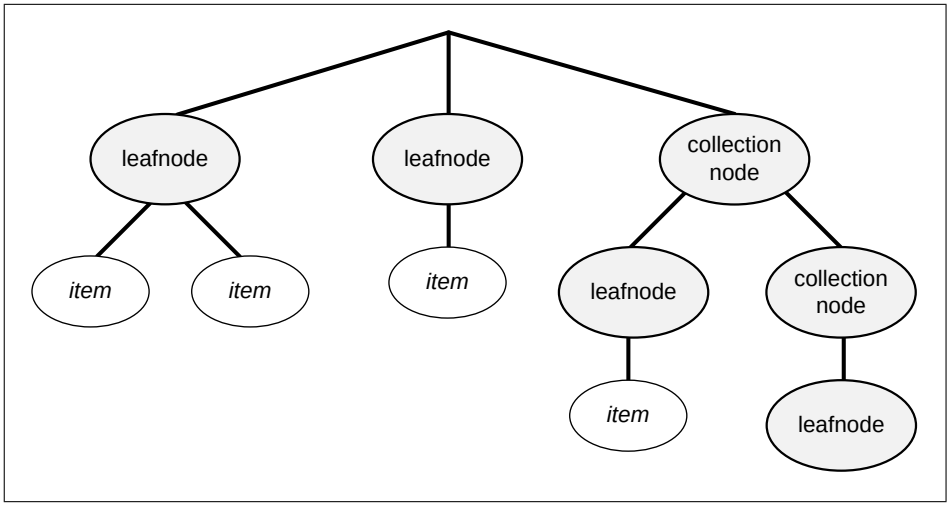
4.1 Inleiding

Een van de meest toegepaste uitbreidingen is de *Publish-Subscribe* extensie (xep-0060) [3], ook wel *pubsub* genoemd. Dit is een implementatie van het klassieke 'publish-subscribe' design pattern voor XMPP: een entiteit (een persoon of applicatie) publiceert informatie, vervolgens wordt een melding van deze gebeurtenis, een *event notification*, uitgezonden naar alle abonnees. Tussen de uitgevers en abonnees staat een dienst (de server), die onder andere publicaties ontvangt en afhandelt, en event notifications naar alle abonnees verzendt. Pubsub kan onder andere ingezet worden voor uitgebreide presence notification, multi-player games en netwerk beheersystemen.

Het doel van dit hoofdstuk is een overzicht te geven van de mogelijkheden van pubsub. Om dit hoofdstuk overzichtelijk te houden worden bepaalde details en mogelijkheden niet besproken.

4.2 Node types

Informatie (één of meer items) wordt gepubliceerd naar zogenaamde *nodes*. Entiteiten kunnen zich op deze nodes abonneren om meldingen van gebeurtenissen te ontvangen. Er zijn twee verschillende soorten nodes: *leaf nodes* en *collection nodes*. Een leaf node bevat alleen gepubliceerde items. Dit betekent dat deze node geen andere nodes mag bevatten. Een collection node bevat nodes, maar geen gepubliceerde items. Een voorbeeld van een mogelijke verzameling nodes en items is gegeven in figuur 4.1.



Figuur 4.1: Voorbeeld van een mogelijke structuur van een verzameling nodes en items

4.3 Node adres

Net als entiteiten (paragraaf 3.3) zijn ook nodes te benaderen via een adres. Nodes zijn adresseerbaar door ofwel een JID, ofwel een combinatie van een JID en de nodeID.

4.3.1 JID

Wanneer een node adresseerbaar is als een JID, moet de nodeID op de plaats van de resource identifier staan. Dit betekent dat de adressen er als volgt uit zien: *domain.tld/NodeID* of *user@domain.tld/NodeID*.

4.3.2 JID en NodeID

Wanneer een node adresseerbaar is door een combinatie van een JID en een nodeID, moet de waarde van zowel de Service Discovery node en pubsub *node attribuut* gelijk zijn aan het nodeID. Ter verduidelijking het volgende voorbeeld:

```
<iq to='user@domain.tld'>
  <query node='nodeID'/>
</iq>
```

4.4 Node Access Models

Voor de nodes bestaan er standaard zes *access models* om toegang tot nodes te reguleren.

Open Dit model staat iedereen toe zich te abonneren, zonder (expliciete) toestemming van de eigenaar van de node. Daarnaast kan elke entiteit items van een node opvragen zonder te zijn geabonneerd.

Whitelist Een entitieit kan zich alleen abonneren of items opvragen als deze op de whitelist staat. De eigenaar van de node beheert deze whitelist.

Authorize Alle abonnement aanvragen moeten worden goedgekeurd door de eigenaar van de node. Alleen abonnees mogen items opvragen.

Presence Elke entiteit die een abonnement heeft van het type 'from' of 'both' op de presence van de eigenaar (paragraaf 3.5.4) kan zich aanmelden voor de node en items opvragen.

Roster Entiteiten die zijn opgenomen in het aangegeven roster kunnen items opvragen en zich abonneren op de node.

4.5 Event types

Wanneer items worden gepubliceerd naar een node, zal de server event notificaties uitzenden naar de abonnees van de betrokken node. Er zijn verschillende soorten events. We kunnen de events onderscheiden in twee dimensies (merk op dat dit leidt tot vier mogelijkheden):

- persistent tegenover transient
- pure notification tegenover inclusion of payload.

Het verschil tussen deze events is of de data die gepubliceerd is wordt opgenomen bij de notificatie en de manier waarop identifiers voor items worden afgehandeld.

4.6 Verkennen

Om te weten te komen welke functies een pubsub service ondersteunt en welke content (nodes en items) deze heeft, zal een entiteit de service moeten verkennen. Deze verzoeken worden verstuurd door middel met een iq stanza (paragraaf 3.5.5).

De pubsub features van een service worden opgevraagd door een verzoek uitgerust met de 'http://jabber.org/protocol/disco#info' namespace. Een voorbeeld van hoe dit verzoek eruit ziet is hieronder weergeven. In dit voorbeeld en de voorbeelden later in dit hoofdstuk worden de *from* en *to* attribuutwaarden vervangen door een 'variabele' (x, y), uiteraard moet hier voor in de plaats een valide JID staan (paragraaf 3.3).

```
<iq type='get'
    from='x'
    to='y'
    id='z'>
```

```
  <query xmlns='http://jabber.org/protocol/disco#info'/>
</iq>
```

Zonder in te gaan op de betekenis van de specifieke onderdelen, is onderstaand voorbeeld een mogelijk antwoord van de server op dit verzoek.

```
<iq type='result'
    from='y'
    to='x'
    id='z'>
  <query xmlns='http://jabber.org/protocol/disco#info'>
    <identity category='pubsub' type='service'/>
    <feature var='http://jabber.org/protocol/pubsub'/>
  </query>
</iq>
```

Het is ook mogelijk om informatie op te vragen over een specifieke node, hiervoor dient naast de 'disco#info' namespace een node attribuut gespecificeerd te worden. De server moet het verzoek beantwoorden met een *<identity>* element inclusief de attributen *category* en *type* (leaf of collection node). Daarnaast kan er metadata teruggeven worden met meer informatie over de node (onder andere de titel, omschrijving en auteur). Zoals gebruikelijk volgt weer een voorbeeld van een mogelijke conversatie.

```
<iq type='get'
    from='x'
    to='y'
    id='z'>
  <query xmlns='http://jabber.org/protocol/disco#info'
         node='blogs'/>
</iq>

<iq type='result'
    from='y'
    to='x'
    id='z'>
  <query xmlns='http://jabber.org/protocol/disco#info'
         node='blogs'>
    <identity category='pubsub' type='collection'/>
  </query>
</iq>
```

Hoewel er nog meer gegevens opgevraagd kunnen worden, zoals informatie over abonnementen, zal hier alleen nog het opvragen van items besproken worden. Voor uitgebreidere informatie kan de officiële specificatie [3] geraadpleegd worden. Voor het opvragen van items is de 'http://jabber.org/protocol/disco#items' namespace beschikbaar. Met het volgende verzoek kunnen de items van een bepaalde node opgevraagd worden:

```
<iq type='get'
    from='x'
    to='y'
    id='z'>
  <query xmlns='http://jabber.org/protocol/disco#items'
         node='nodeID'/>
</iq>
```

Een mogelijk antwoord waarbij twee items als antwoord gegeven worden is:

```
<iq type='result'
    from='y'
    to='x'
    id='z'>
  <query xmlns='http://jabber.org/protocol/disco#items'
         node='nodeId'>
    <item jid='y' name='368866411b877c30064a5f62b917cffe'/>
    <item jid='y' name='3300659945416e274474e469a1f0154c'/>
  </query>
</iq>
```

4.7 Publiceren

Deze paragraaf bespreekt hoe informatie naar een node gepubliceerd kan worden. Hierbij wordt ook besproken hoe de abonnees op de hoogte worden gesteld. Ook voor publicatie worden <iq> stanzas gebruikt. Het onderliggende element is hier het <*publish*> element, met een *node* attribuut. Dit <publish> element heeft één of meerdere onderliggende <*item*> elementen. Ter verduidelijk weer een voorbeeld waarbij een eenvoudig bericht wordt gepubliceerd:

```
<iq type='set'
    from='x'
    to='y'
    id='z'>
  <pubsub xmlns='http://jabber.org/protocol/pubsub'>
    <publish node='myNode'>
      <item id='3de4rrr3ccd'>
        <entry xmlns='http://www.w3.org/2005/Atom'>
          <title>Hello World</title>
          <summary>This is a summary</summary>
          <published>2003-12-13T18:30:02Z</published>
          <updated>2003-12-13T18:30:02Z</updated>
        </entry>
      </item>
    </publish>
  </pubsub>
</iq>
```

Het <item> element heeft hier een attribuut *id* met de waarde '3de4rrr3ccd'. Wanneer dit attribuut niet is meegegeven zal de pubsub dienst zelf een unieke identifier moeten genereren binnen het domein van 'myNode'. Na publicatie worden alle abonnees van de betreffende node op de hoogte gesteld. Afhankelijk van het type event worden de gepubliceerde data wel of niet meegegeven. Event notificaties worden verzonden met de message stanza. Eerst een voorbeeld waarbij de data, ook wel *payload* genoemd niet wordt meegegeven. Merk op dat subscriber1 en subscriber2 geldige JIDs moeten zijn.

```
<message from='y' to='subscriber1' id='w'>
  <event xmlns='http://jabber.org/protocol/pubsub#event'>
    <items node='myNode'>
      <item id='3de4rrr3ccd'/>
    </items>
  </event>
</message>
<message from='y' to='subscriber2' id='w'>
  <event xmlns='http://jabber.org/protocol/pubsub#event'>
    <items node='myNode'>
      <item id='3de4rrr3ccd'/>
    </items>
  </event>
</message>
```

Nu volgt een voorbeeld waarbij de payload wel wordt meegegeven, het enige verschil is het <entry> element met de daarbij onderliggende elementen.

```
<message from='y' to='subscriber1' id='w'>
  <event xmlns='http://jabber.org/protocol/pubsub#event'>
    <items node='myNode'>
    <item id='3de4rrr3ccd'>
      <entry xmlns='http://www.w3.org/2005/Atom'>
        <title>Hello World</title>
        <summary>This is a summary</summary>
        <published>2003-12-13T18:30:02Z</published>
        <updated>2003-12-13T18:30:02Z</updated>
      </entry>
    </item>
    </items>
  </event>
</message>
<message from='y' to='subscriber2' id='w'>
  <event xmlns='http://jabber.org/protocol/pubsub#event'>
    <items node='myNode'>
    <item id='3de4rrr3ccd'>
      <entry xmlns='http://www.w3.org/2005/Atom'>
        <title>Hello World</title>
```

```
        <summary>This is a summary</summary>
        <published>2003-12-13T18:30:02Z</published>
        <updated>2003-12-13T18:30:02Z</updated>
      </entry>
    </item>
    </items>
  </event>
</message>
```

Optioneel kan in het <item> element een publisher en timestamp attribuut opgenomen worden om respectievelijk de uitgever van het item en het tijdstip waarop de notificatie uitgegeven is, aan te geven.

4.8 Overige functionaliteiten

De voorgaande paragrafen hebben laten zien hoe de pubsub service verkend kan worden, hoe items gepubliceerd worden en op welke manier abonnees worden geïnformeerd. Dit onderzoek gaat er van uit dat dit voldoende informatie is voor de lezer om een goed beeld te hebben van de mogelijkheden en de werking van de pubsub uitbreiding. Een aantal zaken zoals aanmaken/beheren van nodes en het beheren van abonnementen wordt daarom in dit hoofdstuk niet besproken. De pubsub uitbreiding komt terug bij de zelf gedefinieerde uitbreiding in paragraaf 7.2.

Hoofdstuk 5

XMPP via het HTTP protocol

5.1 Inleiding

Voor XMPP communicatie worden gebruikelijk TCP (Transmission Control Protocol) verbindingen gebruikt om data uit te wisselen tussen de verschillende partijen. Er zijn echter situaties waarbij een TCP verbinding ongewenst of zelfs onmogelijk is, enkele voorbeelden:

- Bij mobiele apparaten kost het constant in stand houden van een TCP verbinding veel batterijvermogen.
- Strenge firewall instellingen kunnen er voor zorgen dat het maken van een TCP verbinding met een XMPP-server onmogelijk is.
- Binnen een webbrowser is het (standaard) niet mogelijk een TCP verbinding op te zetten

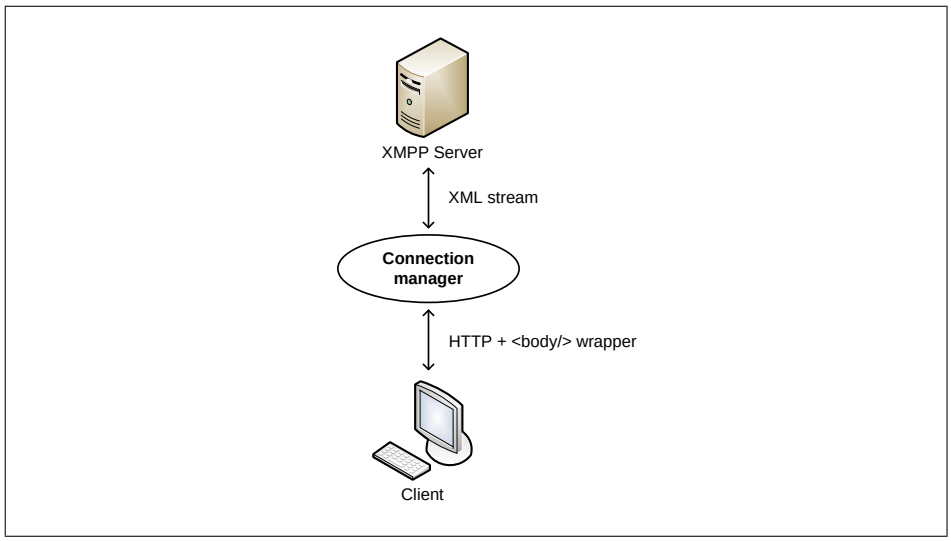
Bidirectional-streams Over Synchronous HTTP (BOSH) is een alternatieve methode die de werking van TCP simuleert. Hierbij wordt een bidirectionele stream nagebootst door gebruik te maken van meerdere, synchrone HTTP request paren. Het BOSH transport protocol wordt beschreven in de specificaties XEP-0124 en XEP-0206.

BOSH is oorspronkelijk ontwikkeld om de eveneens op HTTP gebasseerde techniek *Jabber HTTP Polling* (XEP-0025) te vervangen. Bij HTTP Polling worden de berichten, die opgeslagen zijn op de server, periodiek opgevraagd en verstuurd. Deze techniek kent verschillende nadelen. Zo worden berichten pas ontvangen wanneer er een periodische opvraging is gedaan. Om minimale latentie te bereiken zal er frequent met de server gecommuniceerd moeten worden. Daarnaast worden er opvragingen gedaan, ook wanneer er geen berichten beschikbaar zijn. Dit alles maakt HTTP Polling server- en data-intensief. De techniek die in BOSH wordt toegepast wordt *long polling* genoemd. Hierbij

wordt er gebruik gemaakt van 'lang levende' HTTP verbindingen. Dankzij dezen techniek worden berichten direct ontvangen wanneer ze beschikbaar zijn.

5.2 De techniek achter BOSH

De meeste implementaties gebruiken een *connection manager* om HTTP verbindingen af te handelen. Deze connection manager medieert tussen de client en de server. Het vertaalt de HTTP-verzoeken van de client naar berichten voor de server, en andersom berichten van de server in HTTP verzoeken. De serverberichten zijn doorgaans zoals gebruikelijk XML streams. Een grafische representatie is te zien in figuur 5.1.



Figuur 5.1: Connection manager

Het *Hypertext Transfer Protocol (HTTP)* is een synchroon request/response (verzoek/antwoord) protocol. Dit betekent dat de server voor elk verzoek van de client een antwoord teruggeeft. Bij HTTP polling wordt er periodiek een verzoek gedaan om te ontdekken of er berichten, bestemd voor de client, in de wachtrij staan. Bij deze techniek moet er een afweging gemaakt worden tussen latentie en bandbreedteverbruik. Lage latentie betekent een hoog bandbreedteverbruik, laag bandbreedteverbruik resulteert in hoge latentie.

De techniek achter BOSH bereikt zowel lage latentie als laag bandbreedteverbruik. Het basisidee is dat de server niet reageert op een verzoek van de client totdat het daadwerkelijk gegevens heeft die verzonden moeten worden naar de client. Zodra de client een antwoord heeft gekregen, start deze een nieuw verzoek (request) zodat er altijd een openstaande verbinding is waarover de server gegevens kan verzenden. Zo wordt een lage latentie gerealiseerd omdat de client altijd berichten ontvangt zodra deze beschikbaar zijn.

Wanneer de client (meer) data te verzenden heeft, maakt dez een nieuwe HTTP request. Dit verzoek wordt doorgaans over een nieuwe verbinding gestuurd. [1] Als er voor een langere periode geen verkeer in beide richtingen is geweest, zal de server het verzoek zonder data reageren. Hierdoor zullen zowel server als client verbindingsproblemen binnen een redelijke hoeveelheid tijd ontdekken. De client weet namelijk hoe lang de server over een reactie mag doen.

5.3 Het opzetten van BOSH Sessie

Elk HTTP-verzoek en antwoord bevat een <*body/*> element. Deze heeft de 'http://jabber.org/protocol/httpbind' namespace. De inhoud van dit element is de data die verstuurd moet worden. We laten nu zien hoe een BOSH sessie opgezet kan worden. De client zal de connection manager vragen een nieuwe sessie op te zetten. Voor dit eerste verzoek moet het <body/> element de volgende attributen bevatten:

to de doellocatie van de stream

xml:lang Dit attribuut geeft de standaard taal aan voor elk voor de mens leesbare karakter dat verstuurt wordt.

ver hoogste versie van het BOSH protocol dat de client ondersteunt.

wait Het *wait* attribuut geeft de langste tijdsperiode in secondes aan die de connection manager mag wachten voordat geantwoord moet worden op een verzoek. Dit attribuut zorgt er onder andere voor dat de client netwerkproblemen kan ontdekken.

hold Dit attribuut geeft het aan hoeveel verzoeken de client manager maximaal mag laten wachten op een antwoord. Wanneer de client geen HTTP Pipelining ondersteunt moet deze de waarde "1" hebben.

Daarnaast moet de client bij elk <body/> element een sequentiële identifier (bij elk verzoek opgehoogd) opnemen in de vorm van een *rid* attribuut. Enkele andere, optionele, attributen laten we buiten beschouwing. Een voorbeeld van een initieel verzoek is:

```
<body rid='1274010453'
      to='jabber.org'
      xml:lang='en'
      ver='1.6'
      wait='60'
      hold='1'
      xmlns='http://jabber.org/protocol/httpbind'/>
```

De connection manager antwoordt op dit verzoek met een <body/> element. De volgende attributen zijn hierbij verplicht:

[1] Wanneer de client de beschikking heeft over HTTP Pipelining kunnen meerdere verzoeken over dezelfde verbinding verstuurd worden

sid Dit is de sessie identifier, een unieke onvoorspelbare identifier voor de huidige sessie.

wait Het *wait* attribuut geeft de langste tijdsperiode in secondes aan die de connection manager zal wachten voordat een verzoek beantwoord wordt. De waarde hiervan is kleiner dan of gelijk aan de waarde gespecificeerd door de client.

ver De hoogste versie van het BOSH protocol dat de connection manager ondersteund.

polling Dit attribuut specificeert het kortst toelaatbare polling interval.

inactivity De langst toelaatbare periode van inactiviteit (in seconden). Als er geen openstaande verzoeken zijn, moet de client een nieuw verzoek maken voordat deze periode is verstreken.

requests Aantal gelijktijdige verzoeken die client kan maken. De aanbevolen waarde is één hoger dan de waarde van de hold attribuut.

hold Het maximaal aantal verzoeken dat de client manager op hetzelfde moment zal laten wachten. Deze waarde mag niet groter zijn dan de waarde gespecificeerd door de client in het sessie verzoek.

Onderstaand xml-bericht is een mogelijk antwoord van de server op het verzoek van de client.

```
<body sid='s43fr!xe3fm'
      wait='60'
      ver='1.6'
      polling='5'
      inactivity='30'
      requests='2'
      hold='1'
      xmlns='http://jabber.org/protocol/httpbind'/>
```

Wanneer de verbinding tot stand gebracht is, kan de client XML berichten verzenden door middel van HTTP verzoeken. Deze berichten worden opgenomen in het <body/> element van het verzoek. Het *sid* en *rid* attribuut moeten in het <body/> element aangegeven zijn. Het is de taak van de connection manager de berichten af te leveren bij de server, in volgorde van oplopend rid attribuut.

Hoofdstuk 6

Operational Transformation

6.1 Inleiding

De voorgaande hoofdstukken hebben laten zien hoe XMPP ingezet kan worden voor real-time communicatie. In de rest van dit onderzoek ligt de focus op *real-time collaboratie*. In het gebied van *CSCW (computer-supported cooperative work)* zijn systemen waarbij men in real-time coöperatief kan bewerken erg bruikbaar. Een *real-time coöperatief bewerksysteem* biedt meerdere gebruikers de mogelijkheid hetzelfde document te wijzigen op hetzelfde moment vanaf verschillende locaties. Hierbij gaat het niet alleen om documenten bestaande uit tekst, maar ook om grafische en multimedia documenten. Belangrijk hierbij is dat (gelijktijdige) wijzigingen door andere gebruikers vrijwel direct zichtbaar worden. Hiervoor is een *concurrency control* algoritme nodig om gelijktige operaties tussen verschillende partijen te synchroniseren.

In dit hoofdstuk wordt besproken waaraan de systemen moeten voldoen, wat de moeilijkheden zijn, en gaan we in op *Operational Transformation (OT)* technologie. Deze techniek wordt onder andere toegepast in CoWord, Gobby, SubEthaEdit en Google Wave. Het in 2009 gelanceerde Google Wave is een voorbeeld waarbij de OT gebruikt wordt in combinatie met XMPP.

6.2 Requirements

C.Sun, et al. hebben een constistentie model ontworpen met drie requirements waaraan een coöperatief bewerksysteem moet voldoen [10]. Een coöperatief bewerksysteem is constistent als het altijd aan de volgende eigenschappen voldoet:

- *Convergence*
- *Itention-preservation*
- *Causality-preservation*

6.2.1 Convergence requirement

Als vanaf verschillende locaties dezelfde verzameling operaties uitgevoerd wordt op een gedeeld object, moeten uiteindelijk alle kopieën van dit object identiek zijn.

6.2.2 Intention-preservation requirement

Het lokale effect van een operatie moet bewaard blijven op verschillende lokaties. Meer formeel: voor elke operatie O is het effect van het uitvoeren van O op alle lokaties hetzelde als de intentie (het bedoelde effect) van operatie O. Daarnaast verandert het (effect van) toepassen van operatie O het effect van onafhankelijke operaties niet.

6.2.3 Causality-preservation requirement

Operaties moeten uitgevoerd worden in volgorde van optreden. Voor elk paar operaties O_1 en O_2 geldt: als O_1 voorafgaat aan O_2 dan wordt O_1 op alle locaties eerder toegepast dan O_2.

6.3 Operational transformation (OT)

Operational Transformation bereikt zowel de convergence en intention preseveration requirement zonder beperkingen op te leggen aan activiteiten van gebruikers. De Causality-preservation requirement wordt doorgaans gegarandeerd door de transportlaag (bijvoorbeeld TCP). Andere modellen voor concurrency control zoals *locking* leggen wel beperkingen op aan de activiteiten van gebruikers.

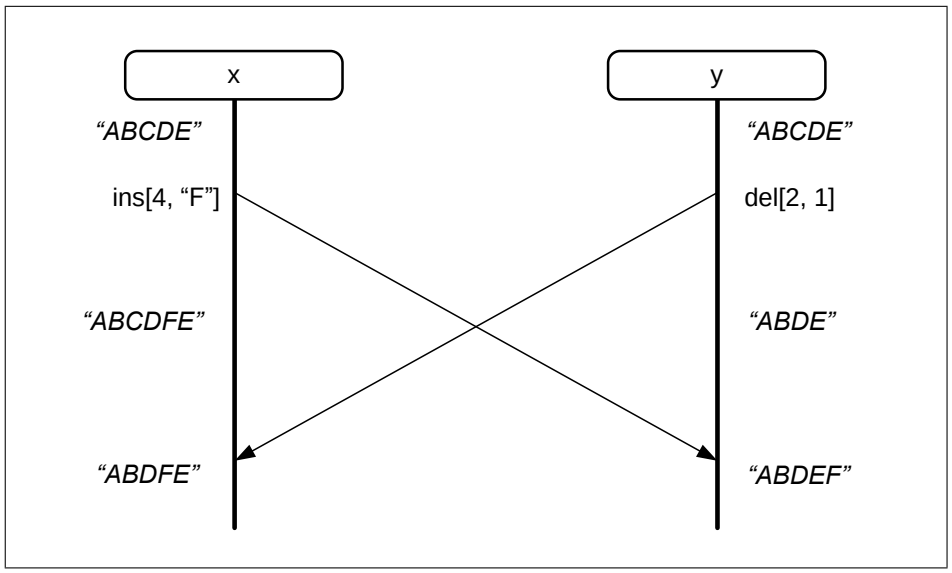
Oorspronkelijk is OT ontwikkeld voor *consistency maintenance* en *concurrency control* voor het collaboratief bewerken van documenten bestaande uit platte tekst. Het collaboratief bewerken van documenten betekent dat meerdere personen tegelijkertijd een gedeeld document kunnen bewerken. We spreken van live en concurrent (gelijktijdig) bewerken als de veranderingen die andere personen maken direct zichtbaar worden (bijvoorbeeld elke toetsaanslag). In de loop van de jaren zijn de mogelijkheden van OT toegenomen waardoor het nu onder andere mogelijk is gestructureerde documenten en afbeeldingen te bewerken.

6.4 Situatie

We beginnen met een voorbeeld van een concrete situatie waarop OT van toepassing is (figuur 6.1).

6.4.1 De naïeve manier

Stel er zijn twee gebruikers: X en Y. Ze hebben in beginsel hetzelfde document voor zich: “ABCDE”. Op (ongeveer) hetzelfde moment besluiten X en Y het



Figuur 6.1: De naïve manier waarbij de gebruikers elkaars operatie toepassen zonder deze eerst te transformeren aan de hand van het effect van de voorgaand uitgevoerde operaties.

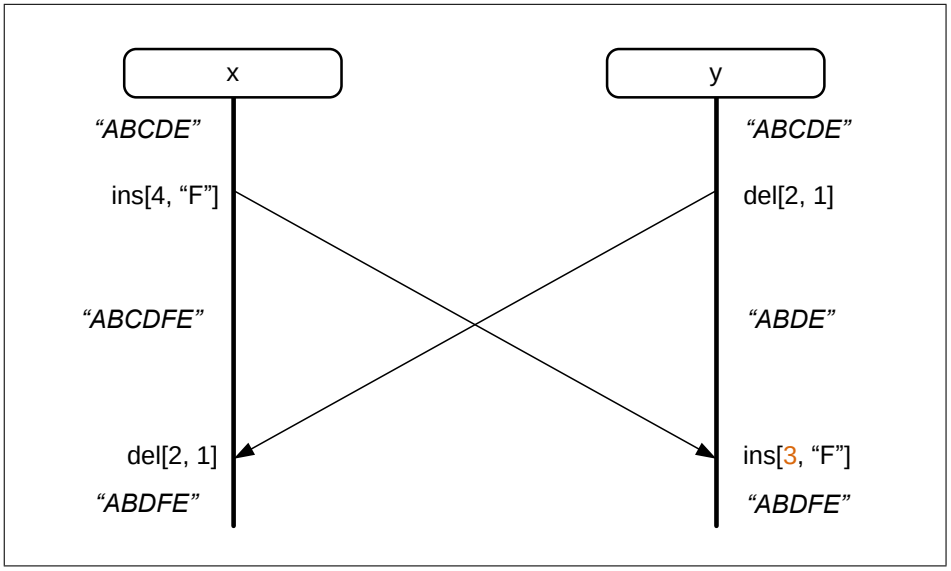
document te wijzigen. X voegt een karakter F toe op positie 4 (Ins[4, “F”]). Resulterend in document “ABCDFE”. Y verwijdert het derde karakter (Del[2,1]). Dit resulteert in document “ABDE”. Ze wisselen nu de uitgevoerde operatie uit. Het toepassen van elkaars operatie op het huidige document levert respectievelijk de volgende documenten “ABDFE” en “ABDEF” op. Doordat de twee operaties elkaar ‘beconcurreren’ heeft het uitvoeren van dezelfde verzameling operaties in verschillende volgorde er hier voor gezorgd dat de twee gebruikers uiteindelijk een verschillend document voor zich hebben.

Het idee achter OT is dat de parameters van een operatie worden *getransformeerd* aan de hand van het effect van de voorgaand uitgevoerde, concurrerende, operaties zodanig dat de getransfomeerde operatie het beoogte effect bereikt en consistentie bewaard blijft.

6.4.2 De correcte manier

We laten nu zien hoe OT toegepast kan worden op bovenstaand voorbeeld. Weer beginnen we met de gebruikers X en Y en document “ABCDE”. De gebruikers besluiten het document weer op hetzelfde moment te wijzigen met dezelfde operaties: (Ins[4, “F”]) en (Del[2,1]). Na uitvoeren van de operaties levert dit voor X en Y respectievelijk de documenten “ABCDFE” en “ABDE” op. De operaties worden nu uitgewisseld.

Voordat de operaties worden toegepast worden ze eerst getransformeerd aan de hand van het effect van de voorgaand uitgevoerde operaties. Omdat gebruiker



Figuur 6.2: De correcte manier waarbij operaties voor het toepassen eerst zijn getransformeerd.

Y het derde karakter heeft verwijderd wordt de operatie Ins[4, “F”] omgezet in Ins[3, “F”]. Vanaf positie 2 zijn alle karakters door de deletie namelijk een positie naar links opgeschoven. Gebruiker X heeft karakter F toegevoegd op positie 4. Deze operatie heeft geen concurrerend effect voor de operatie Del[2,1], de operatie Del[2,1] hoeft daarom niet getransformeerd te worden. Y voert nu de transformatie Ins[3, “F”] uit op document “ABDE” en krijgt “ABDFE”. X voert nu de operatie Del[2, 1] uit op het document “ABCDFE” en krijgt “ABDFE”. Dankzij OT is na het uitvoeren van alle operaties het document voor beide gebruikers gelijk.

6.5 OT framework

Het OT framework bestaat uit twee componenten: *control algoritmes* en *transformatiefuncties*.

6.5.1 Control algoritmes

Control algoritmes zijn in twee klassen te onderscheiden: *pessimistic* en *optimistic*.

Een *pessimistic* algoritme vereist communicatie met andere partijen (locaties of een centrale coördinator) voordat veranderingen worden doorgevoerd. De gebruiker moet hierbij wachten totdat informatie is uitgewisseld met andere partijen voordat de veranderingen worden toegepast. Bij *optimistic* algoritmes

worden veranderingen meteen lokaal toegepast. Daarna worden andere partijen geïnformeerd. De lokale responstijd wordt hierdoor ongevoelig voor netwerk latentie. Wijzigingen die de gebruiker aan een document maakt worden meteen zichtbaar. Daarnaast zijn de control algoritmes te onderscheiden in algoritmes voor volledig gedistributeerde systemen en gecentraliseerde systemen met een centrale coördinator.

6.5.2 Transformatiefuncties

Een transformatiefunctie bepaalt hoe een paar van operaties wordt omgezet. Deze transformatie is afhankelijk van het type operatie, de positie en andere parameters. Voor de transformaties geldt de volgende belangrijke eigenschap: Als T een transformatiefunctie is en c en s respectievelijk de client en de server operatie, dan geldt $T(c, s) = (c', s')$, waarbij $s \cdot c' = c \cdot s'$. Dit betekent dat wanneer de client c toepast gevolgd door s', en de server s toepast gevolgd door c', de client en de server in dezelfde eindtoestand komen.

De transformatiefuncties hebben niet altijd een paar als resultaat. Dit paar kan dan verkregen worden door de transformatie tweemaal uit te voeren door de invoerparameters om te wisselen. We geven nu een voorbeeld van een transitiefunctie met een enkelvoudig resultaat. Stel we hebben een Insert operatie met twee parameters, de positie en het karakter, en een Delete operatie met als parameter de positie van het karakter dat verwijdert moet worden. We definieren nu de transitiefuncties:

```
function Tid(Ins[p1, c1], Del[p2])
{
  if(p1 <= p2)
    return Ins[p1, c1]
  else
    return Ins[p1-1, c1]
}
function Tdi(Del[p1], Ins[p2,c2])
{
  if(p1 < p2)
    return Del[p1]
  else
    return Del[p1+1]
}
```

Stel we hebben het document “ABCDE” en twee operaties $Ins[5, F]$ en $Del[3]$. Om een paar als resultaat te krijgen berekenen we $(Tid(Ins[5, F], Del[3]), Tdi(Del[3], Ins[5, F]))$, met als resultaat $(Ins[4, F], Del[3])$. Er geldt nu inderdaad dat $Ins[5, F] \cdot Del[3] = Del[3] \cdot Ins[4, F]$.

6.6 dOPT algoritme

Het eerste OT algoritme, gebruikt in het GROVE (GRoup Outtie Viewing Edit) systeem, is ontwikkeld door Ellis en Gibbs bij MCC (Microelectronics and Computer Technology Corporation) en gepubliceerd in 1989. Het is een gedistribueerd en optimistic algoritme dat *dOPT* is genoemd [2].

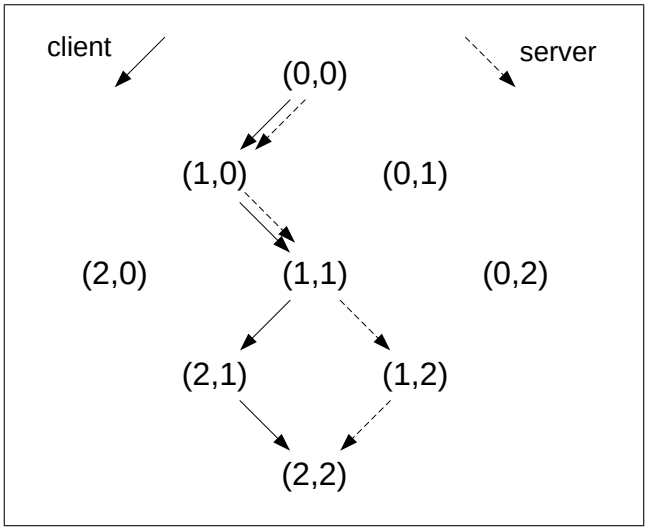
Er bleek echter een fout in dit algoritme te zitten (*The dOPT puzzle*), kort samengevat: wanneer een partij meerdere gelijktijdige operaties verzond, werkte het algoritme niet meer vanwege veranderde toestanden (veroorzaakt door eerdere operaties). In de jaren 90 zijn verschillende onderzoekers onafhankelijk met oplossingen voor de problemen gekomen. Het is het begin geweest van vele jaren onderzoek met als doel OT te verbeteren en uit te breiden.

6.7 Jupiter algoritme

Het OT algoritme van Google Wave is gebaseerd op het algoritme achter het *Jupiter Collaboration System* [5]. Dit control algoritme is afgeleid van het volledig gedistribueerde, optimistic dOPT algoritme dat gebruikt wordt in GROVE [2]. Voor het Jupiter algoritme is gekozen voor een gecentraliseerde benadering, waardoor het GROVE algoritme aanzienlijk vereenvoudigd kan worden. In een gedistribueerde omgeving kunnen alle partijen met elke andere partij operaties uitwisselen (n-way communicatie). Het algoritme voor gedistribueerde optimistische systemen moet daardoor op elk moment in staat zijn elke verandering van elke partij te verwerken. Dit blijkt erg ingewikkeld. Bij Jupiter communiceren de clients maar met één partij, de server. Alle clients synchroniseren onafhankelijk van elkaar met de server. De server houdt voor elke client een eigen toestandsruimte voor het gedeelde document bij. Alle clients bewaren een lokale kopie van het gedeelde document. Lokale veranderingen kunnen direct op deze lokale kopie toegepast worden. De lokale operatie delen de clients vervolgens met de centrale server. De server transformeert binnenkomende operaties, past deze toe op de gedeelde toestandsruimte, en verzendt de getransformeerde operaties naar andere partijen. Elke ontvangende partij moet vervolgens de inkomende operatie omzetten zodat de operatie consistent is met betrekking tot zijn huidige lokale toestand.

Wanneer een client of server een operatie toepast op het document, zeggen we dat deze in een nieuwe *toestand* terecht komt. Als een client of server meerdere operaties achter elkaar toepast, noemen we deze operaties gezamenlijk een *pad* door de toestandsruimte. Client en server kunnen operaties onafhankelijk van elkaar uitvoeren, ze hoeven niet noodzakelijk hetzelfde pad te volgen. Als ze de operaties in dezelfde volgorde verwerken, volgen ze hetzelfde pad. Worden de operaties niet in dezelfde volgorde verwerkt, dan zullen de paden *divergeren*. Via verschillende paden kunnen client en server (uiteindelijk) in dezelfde eindtoestand terechtkomen, ofwel *convergeren*. Wanneer client en server in dezelfde toestand zijn, betekent dit dat ze elkaars operaties hebben verwerkt en het gedeelde document, lokaal en op de server, gelijk is.

Deze toestandsruimte (en bijbehorende paden) kan grafisch gerepresenteerd worden. Zo'n grafische representatie van een mogelijke toestandsruimte is te zien in figuur 6.3. Het pad van de client wordt gerepresenteerd met een vetgedrukte pijl, het pad van de server met de gestippelde pijl. Elke toestand is gelabeld met het aantal operaties van respectievelijk de client en server die zijn verwerkt. Bij de toestand (2,3) zijn er twee operaties van de client en drie operaties van de server verwerkt. Een pijl naar rechts betekent dat een serveroperatie wordt verwerkt, bij een pijl naar links wordt een operatie van de client verwerkt. Ter verduidelijking: Voor een client betekent een pijl naar rechts dat deze een operatie van de server heeft ontvangen, deze heeft getransformeerd en vervolgens heeft toegepast op het document.

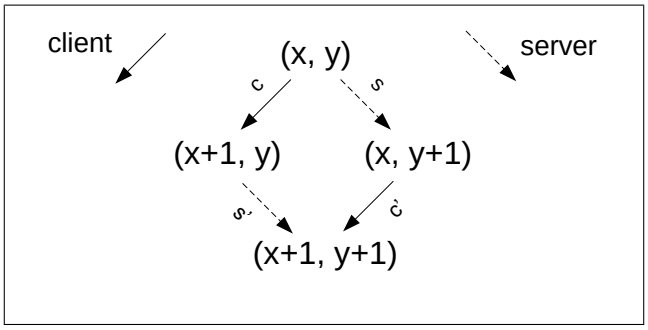


Figuur 6.3: Toestandsruimte

- Zowel de client als de server beginnen, met hetzelfde document, in toestand $(0, 0)$.
- Als eerste voert de client een lokale operatie uit vanaf toestand $(0, 0)$. Deze operatie wordt naar de server gestuurd, omdat de server in dezelfde toestand is, kan deze de operatie direct toepassen.
- Nu voert de server een operatie uit. De server komt hierdoor in toestand $(1, 1)$. De client wordt op de hoogte gesteld van deze operatie en voert de operatie uit. Ook de client bereikt hierdoor toestand $(1, 1)$.
- Tot dusver doorliepen de client en de server hetzelfde pad. Vanaf toestand $(1, 1)$ divergeert het pad echter. De client en server verwerken dan namelijk een verschillende operatie.

- De client past een lokale operatie toe. Gelijktijdig past de server een serveroperatie toe. De client bereikt nu in toestand $(2, 1)$, terwijl de server naar toestand $(1, 2)$ gaat. De client en server wisselen nu de operatie uit. De client transformeert de van de server ontvangen operatie met behulp van de transformatiefunctie. De server doet hetzelfde. Na toepassen van de getransformeerde operatie convergeren de client en server naar dezelfde toestand $(2, 2)$.

De transformatiefunctie in het Jupiter algoritme wordt *xform* genoemd. De invoer voor deze functie is een paar van client en server operaties, gegenereerd vanaf dezelfde starttoestand. Het resultaat is een paar getransfomeerde operaties waarmee de client en server dezelfde eindtoestand kunnen bereiken. Formeel wordt de transformatiefunctie als volgt gedefinieerd:



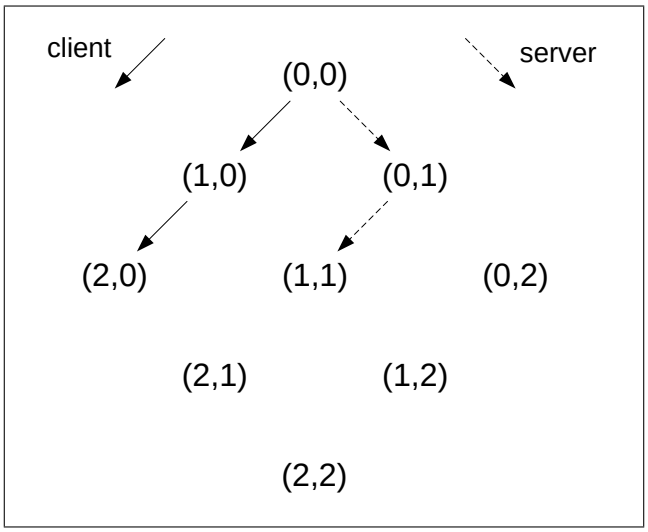
Figuur 6.4: Operaties berekend door de transformatiefunctie

$$xform(c, s) = \{c', s'\}$$

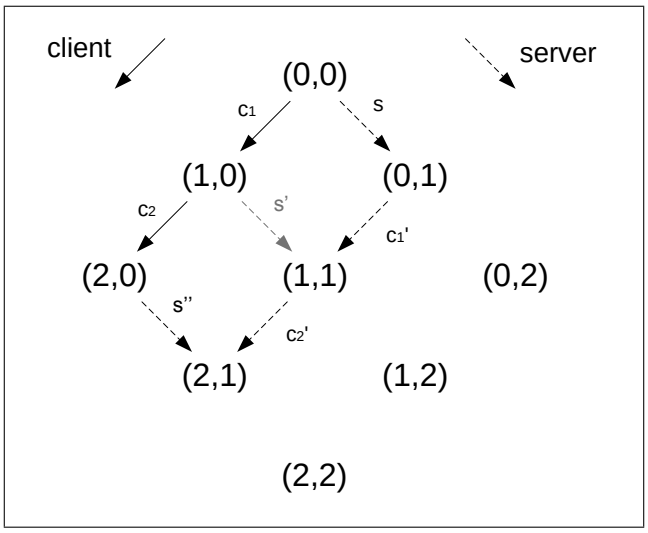
Voor c' en s' geldt de volgende eigenschap: als de client operatie c toepast gevolgd door s' en de server past operatie s toe gevolgd door c', dan eindigen de client en server in dezelfde state (figuur 6.4).

Zolang de client en server maar één operatie divergeren, zoals te zien in figuur 6.3 en 6.4, kan deze functie direct toegepast worden. Echter kunnen de paden meer dan een operatie verschillen. De ontvangende partij zal hier bij het verwerken van de operatie rekening mee moeten houden. Daarom moet de toestand voor het toepassen van de operatie altijd met de operatie meegestuurd worden.

Nu volgt een voorbeeld. Bekijk de situatie (toestandsruimte) in figuur 6.5. De client heeft twee operaties toegepast, waardoor deze terecht komt in toestand $(2, 0)$. Ondertussen heeft de server ook een eigen operatie toegepast. Na transformatie en uitvoeren van de eerste operatie van de client belandt de server in toestand $(1, 1)$. De server ontvangt vervolgens de tweede operatie van de client. Deze operatie is door de client toegepast vanuit toestand $(1, 0)$, de server heeft echter geen serveroperatie (vanuit toestand $(1, 0)$) die gebruikt kan worden voor de xform transformatiefunctie. (Herinner dat xform een paar server en client



Figuur 6.5: Voorbeeldsituatie



Figuur 6.6: Voorbeeldsituatie vervolg

operatie vanuit dezelfde toestand als invoer heeft). Figuur 6.6 representeert de oplossing. De oplossing is dat de server wanneer het c'_1 berekent, s' onthoudt. Dit is een een getransformeerde operatie van de server waarmee de client zich van toestand $(1, 0)$ naar toestand $(1, 1)$ zou kunnen verplaatsen. De server kan nu met $xform(c_2, s') = \{c'_2, s''\}$ operatie c'_2 berekenen, waarmee het in toestand $(2, 1)$ kan komen. De client kan op zijn beurt operatie s'' toepassen om toestand $(2, 1)$ te bereiken.

6.7.1 Algoritme

We bekijken nu het algoritme vanuit het oogpunt van de client. Het algoritme voor de server is gelijkwaardig.

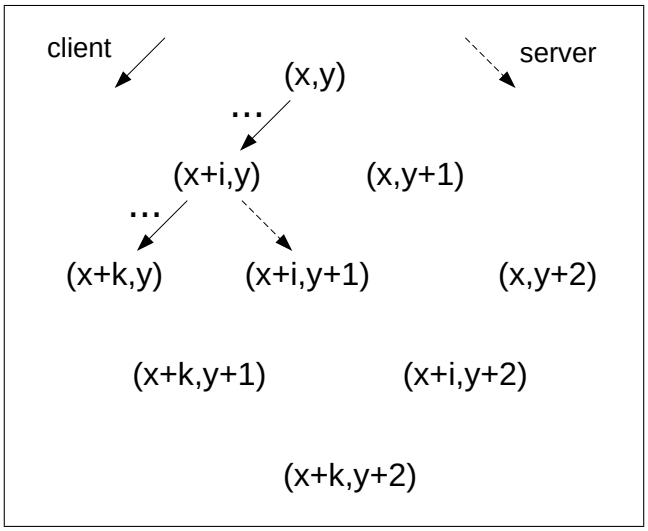
Verzenden

- Pas de operatie lokaal toe.
- Verzend de operatie naar de server, inclusief een identifier van de toestand waarin de client was voordat de operatie werd toegepast.
- Voeg de operatie toe aan de queue voor uitgaande operaties.

Ontvangen

- Op een zeker moment is de client in toestand (x, y). Hij voert k operaties uit en komt hierdoor in toestand $(x + k, y)$
- Zodra de client nu een operatie van de server ontvangt, heeft de server een of meerdere operaties van de client verwerkt en weet de client dat deze operatie ergens op het pad tussen (x, y) en $(x+k, y)$ inclusief is toegepast. We nemen aan dat er een bericht van de server komt vanuit toestand $(x+i, y)$. Waardoor de server in toestand $(x+i, y+1)$ terecht is gekomen.
- De client kan nu de opgeslagen operaties van toestand (x, y) tot $(x + i, y)$ verwijderen. Deze zijn door de server verwerkt en dus niet langer nodig. We weten immers dat de operatie van de server is toegepast in toestand $(x + i, y)$
- De resterende opgeslagen operaties $(x + i, y)$ tot $(x + k, y)$ worden nu gebruikt voor de transformatie van de ontvangen operatie. Aan de hand van de ontvangen operatie van de server, uitgevoerd vanuit toestand $(x + i, y)$ willen we een nieuwe operatie berekenen die hetzelfde effect heeft vanuit toestand $(x + k, y)$. We zagen eerder dat de transformatiefunctie een paar van client en server operaties, die vanaf dezelfde starttoestand zijn toegepast, als invoer heeft. De starttoestand van de serveroperatie is $(x + i, y)$, de eerste operatie die de client in de uitvoer queue heeft, is ook

vanuit deze toestand gegenereerd. Vervolgens worden de door transformatie opgeleverde serveroperatie en de eerstvolgende operatie in de uitvoerqueue gebruikt om transformatie te doen vanuit toestand $(x + i + 1, y)$. Dit proces herhaalt zich totdat toestand $(x + k, y)$ bereikt is, dit is het geval wanneer alle operaties in de queue gebruikt zijn voor transformatie. We hebben nu een operatie die ons brengt van $(x+k, y)$ naar $(x+k, y+1)$. Deze wordt lokaal toegepast.



Figuur 6.7:

De uitgaande queue blijft groeien totdat er een bericht van de server binnenkomt. Er is een situatie voor te stellen waarin de server lange tijd geen berichten (operaties) te verzenden heeft. Wanneer de client in de tussentijd veel bewerkingen doet, zal de queue een grote omvang aannemen. Dit is onwenselijk, de server zal er daarom zorg voor moeten dragen dat een periodieke *acknowledgement* naar de client gestuurd wordt.

Het beschreven algoritme laat zien hoe operaties tussen de client en server uitgewisseld worden. Echter moeten de operaties (via de server) ook tussen de clients onderling uitgewisseld worden. Hiervoor zal de server de operaties van de clients moeten omzetten tussen de verschillende toestandsruimtes. Dit maakt het algoritme van de server erg gecompliceerd. Een ander zwak punt van Jupiter algoritme is dat het erg geheugenintensief is omdat de server voor elke verbonden client een eigen toestandsruimte bij moet houden. In de praktijk zal het algoritme daarom niet genoeg functioneren wanneer er veel clients met de server verbonden zijn. Het OT algoritme achter Google Wave heeft enkele significante aanpassingen aan het besproken algoritme gedaan om de problemen op te lossen. Meer over dit algoritme is te vinden in de volgende paragraaf.

6.8 Google Wave OT algoritme

Helaas is er (nog) geen gedetailleerde literatuur beschikbaar over de werking van het Operational Transformation algoritme gebruikt in Google Wave. Op de website van het Wave Protocol wordt er in een artikel kort ingegaan op het OT algoritme achter Google Wave [11]. Daarnaast is er op internet videomateriaal beschikbaar. Voornamelijk is voor deze paragraaf het artikel van Daniel Spiewak op zijn blog codecommit.com gebruikt [9]. Dit OT algoritme wordt gebruikt in Novell Pulse en de schrijver heeft op basis van wat hij gehoord en gelezen heeft een sterk vermoeden dat het algoritme achter Wave op dezelfde manier werkt.

Bij het Jupiter algoritme kunnen clients en servers zoveel operaties na elkaar verzenden, zo snel als ze kunnen. Dit heeft als consequentie dat de client en de server, afhankelijk van wanneer ze operaties ontvangen van andere partijen, via verschillende paden de toestandsruimte kunnen doorlopen naar dezelfde convergerende toestand.

In Google Wave's uitbreiding moeten clients wachten op een bevestiging van de server voordat er nieuwe operaties verzonden kunnen worden. Deze zogenaamde *acknowledgement* van de server wordt pas verzonden als de server de operatie van de client heeft verwerkt, en de operatie naar alle verbonden clients is gestuurd. Operaties die op het lokale document worden toegepast worden opgeslagen en gezamenlijk verzonden als de acknowledgement is ontvangen. Door deze aanpassing is een client in staat het pad van de server af te leiden. De server hoeft hierdoor maar één toestandsruimte bij te houden, en niet langer een toestandsruimte voor elke verbonden client.

De documenten in Google Wave worden Waves genoemd. Elke Wave bestaat uit een verzameling wavelets, die elk een verzameling documenten bevatten. Een document is samengesteld uit XML en enkele annotaties. Deze annotaties worden hier buiten beschouwing gelaten. In figuur 6.8 is een voorbeeld van een XML document te zien. Elke XML tag (begin en eindtag) en elk ander

```
<blip>
  <p>Hello World</p>
</blip>
```

Figuur 6.8: Voorbeeld van een XML document (in Google Wave)

karakter wordt een *item* genoemd. Gaten tussen items worden posities genoemd. Document operaties kunnen verwijzen naar deze posities.

6.8.1 Operaties

Op een XML document kunnen operaties toegepast worden, in Wave de *document operaties* genoemd. Dit zijn handelingen die worden uitgevoerd op het document. Een operatie bestaat uit verschillende *componenten*. Deze voeren handelingen uit met betrekking tot de (huidige) *cursor* positie. Dit is een

denkbeeldige cursor die aan het begin van de operatie op positie 0 staat. Hieronder wordt een overzicht gegeven van de componenten.

skip (aantal) Verplaatst de cursor het gespecificeerde aantal posities in voorwaartse richting.

insert characters (karakters) Voeg de gespecificeerde karakters toe op de huidige positie, en plaatst de cursor aan het einde van de string.

delete characters (aantal) Verwijder het gespecificeerde aantal karakters op de huidige positie, laat de cursorpositie ongewijzigd.

insert element start (naam,attributen) Voeg een XML open-tag met de gespecifieerde naam toe op de huidige cursorpositie, verplaats de cursor één karakter voorwaarts. Er kan ook een verzameling attributen meegegeven worden.

insert element end Sluit de meest recent geopende XML tag af op de huidige positie, verplaats de cursor één karakter voorwaarts.

delete element start Verwijder de open-tag na de cursor, de cursorpositie blijft onveranderd.

delete element end Verwijder de sluit-tag na de cursor, de cursorpositie blijft onveranderd.

set attributes Verwijder de huidge attributen en voeg de gespecificeerde verzameling van attributen met bijbehorende waarde toe aan de xml tag na de cursor. De cursorpositie blijft ongewijzigd.

update attributes Update de attributen zonder ze te verwijderen, cursorpositie blijft ongewijzigd.

insert anti element start Voeg een sluit-tag voor het dichtbijzijnste linker element toe, verplaats de cursor één karakter voorwaarts.

insert anti element end Maak open-tag die door de anti element start gesloten werd. De cursor wordt één karakter voorwaarts geplaatst.

delete anti element start Verwijder de sluit-tag na de cursor, de cursorpositie blijft onveranderd.

delete anti element end Verwijder de start-tag na de cursor, de cursorpositie wordt niet gewijzigd.

Om het document van figuur 6.8 te wijzigen in `<blip><p>Hello</p> <p>World</p></blip>`, wordt er een anti-element-start na Hello en een anti-element-end voor World geplaatst. Een document kan gerepresenteerd worden als één operatie, voor het voorbeeld in figuur 6.8 is dit de volgende operatie:

```
insert element start "blip"
insert element start "p"
insert characters "Hello World"
insert element end
insert element end
```

6.8.2 Transformatie

Bij transformatie worden twee streaming operaties als input genomen, met twee streaming operaties als resultaat. Deze streaming methode moet ervoor zorgen dat de verwerking een paar van grote operaties efficiënt is. De werking van streaming wordt hier niet besproken.

6.8.3 Compositie

Een belangrijke toevoeging in vergelijking met het Jupiter algoritme is *compositie*. De operaties zijn zo ontworpen dat ze kunnen worden samengevoegd, de samenstelling van twee operaties is op zichzelf een enkele operatie. De compositie van twee operaties A en B wordt geschreven als $B \cdot A$. B(d) betekent: operatie B toegepast op document d. Voor compositie gelden de volgende requirements:

- Voor elk document d geldt

$$(B \cdot A)(d) = B(A(d))$$

- De transformatie van samengestelde operaties moet voldoen aan de eis dat als

$$transform(A, X) = (A', X') \wedge transform(B, X') = (B', X'')$$

dan geldt ook

$$transform(B \cdot A, X) = (B' \cdot A', X'')$$

en als

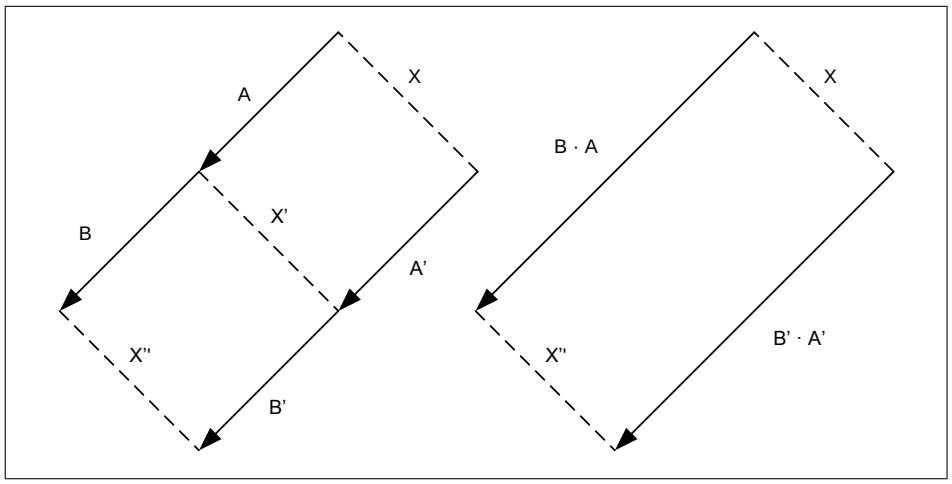
$$transform(X, A) = (X', A') \wedge transform(X', B) = (X'', B')$$

dan geldt

$$transform(X, B \cdot A) = (X'', B' \cdot A')$$

(figuur 6.9).

Voor een efficiënte werking is er in het Wave algoritme voor gekozen bij compositie de operaties als streams te verwerken.



Figuur 6.9: Tweede requirement voor compositie: De compositie van operaties heeft hetzelfde effect bij transformatie als het achtereenvolgens transformeren van de operaties

6.8.4 Werking van het algoritme

Ter herinnering: het grootste bezwaar tegen het Jupiter algoritme is overbelasting van de server doordat deze veel geheugenruimte nodig heeft voor het onthouden van de toestandsruimtes voor elke verbonden client, en het omzetten van operaties tussen toestandsruimtes de server veel rekenwerk kost. Het Wave algoritme lost deze problemen op door werk van de server naar de client te verplaatsen. Het resultaat is dat de server nog maar één toestandsruimte bij hoeft te houden.

Voordat de client een operatie naar de server verstuurt moet de client er voor zorgen dat de operatie ergens op pad van de server toepasbaar is. Het kan zijn dat de server operaties (van andere partijen) heeft toegepast waarvan de client nog geen weet heeft, maar dat maakt voor deze methode niet uit. Consequentie is dat we nooit meer dan één operatie op hetzelfde moment kan versturen. Stel namelijk dat we achtereenvolgens twee operaties C_1 en C_2 uitvoeren. Dit betekent dat C_2 is uitgevoerd vanaf een toestand waar operatie C_1 reeds is toegepast. Wanneer C_2 omgezet moet worden in een voor de server toepasbare operatie kan de client zich niet onttrekken aan feit dat C_1 toegepast is voorafgaand aan C_2. Ook in de toestandsruimte van de server zal C_2 toegepast moeten worden vanuit de toestand die bereikt wordt na het toepassen van (de getransformeerde) operatie C_1. Deze toestand is helaas niet te achterhalen totdat precies bekend is waar C_1 op het pad van de server is toegepast. De client zal daarom moeten wachten met het verzenden van de tweede operatie totdat een acknowledgment van de eerste operatie is ontvangen. Deze acknowledgement verstuurt de server zodra deze de operatie van de client heeft ontvangen en toegepast. Hoewel de client moet wachten met het verzenden van oper-

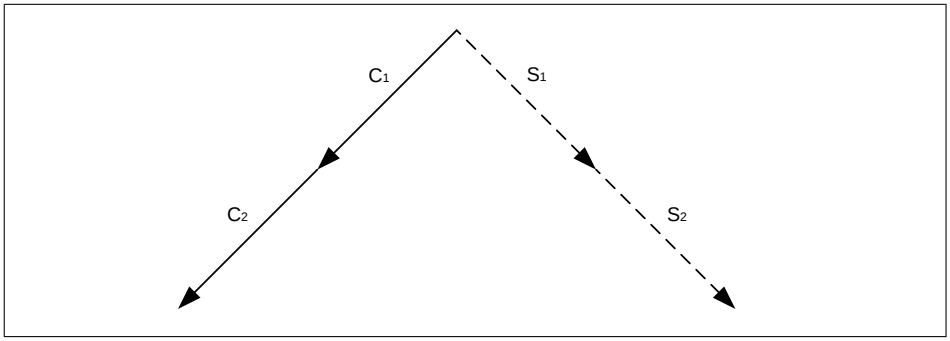
aties totdat een acknowledgement is ontvangen, kunnen de operaties wel direct lokaal worden toegepast. Wachtende operaties worden opgeslagen en uiteindelijk als één door compositie samengevoegde operatie verstuurd. De server zal elke toegepaste operatie naar elke andere partij versturen om deze op de hoogte te brengen van de operatie. Elke operatie wordt uigerust met een unieke identiteit. Wanneer de client een operatie ontvangt met dezelfde identiteit als een voorafgaand toegepaste operatie, kan het bericht worden opgevat als een acknowledgement. We nemen aan dat de communicatielaag ervoor zorgt dat berichten in volgorde van verzenden bij de client en server aankomen. Dit stelt de client in staat een kopie bij te houden van servergeschiedenis.

Hoewel de client niet meer vrij is in het verzenden van operaties, kan de toestand(sruimte) van de client en server nog steeds ver van elkaar verschillen. De client kan één of meerdere serveroperaties lokaal toepassen, en ook één of meerdere eigen operaties. Dit is waar het algoritme ingewikkeld wordt. Om inkomende operaties efficiënt te kunnen verwerken wordt er een *"brug"* tussen de laatst bekende toestand van de server en de huidige toestand van de client bijgehouden. Deze brug wordt gemaakt door alle operaties samen te voegen die lokaal zijn toegepast sinds het moment van divergatie met de server. Zodra de client een operatie van server ontvangt kan deze brug gebruiken om de operatie te transformeren. Zoals bekend is geeft de transformatie een paar van operaties als resultaat. De tweede operatie van dit resultaat vormt de nieuwe brug. Niet alleen moeten inkomende operaties efficiënt verwerkt worden, ook is het wenselijk om de uitgaande operatie op efficiënte wijze af te handelen. Hiervoor wordt een geschiedenis bijgehouden, de *buffer* genoemd, die alleen operaties bevat die nog niet zijn verzonden. De buffer is altijd één operatie omdat de verschillende operaties worden samengevoegd door compositie. Voor de buffer moet altijd het volgende gelden: Gegeven de operaties ontvangen van de server, als de server deze operatie toepast bereikt de server dezelfde toestand als de client. Net als de brug moet ook de buffer getransformeerd worden wanneer er een serveroperatie ontvangen wordt. Het blijkt dat de brug en buffer overlappen. Hoe dit alles precies in zijn werk gaat, wordt nu besproken aan de hand van een voorbeeld (figuur 6.10).

- De client past operatie C_1 toe en verstuurt deze operatie naar de server. Kort na deze operatie voert de client operatie C_2 uit. Zolang operatie C_1 onbevestigd (unacknowledged) is kan er geen nieuwe operatie gestuurd worden, operatie C_2 wordt gebufferd.

- Voordat de server operatie C_1 verwerkt, voert de server twee operaties uit: S_1 en S_2. Deze operaties worden naar alle verbonden clients verzonden. Even later zal de server operatie C_1 transformeren en de getransformeerde operatie naar alle verbonden partijen verzenden.

- De client ontvangt operatie S_1. Voor het toepassen van deze operatie moet de operatie getransformeerd worden. In de voorgaande paragraaf is te lezen hoe deze transformatie in zijn werk gaat. Echter, omdat meerdere operaties samengevoegd kunnen worden kan altijd volstaan worden met

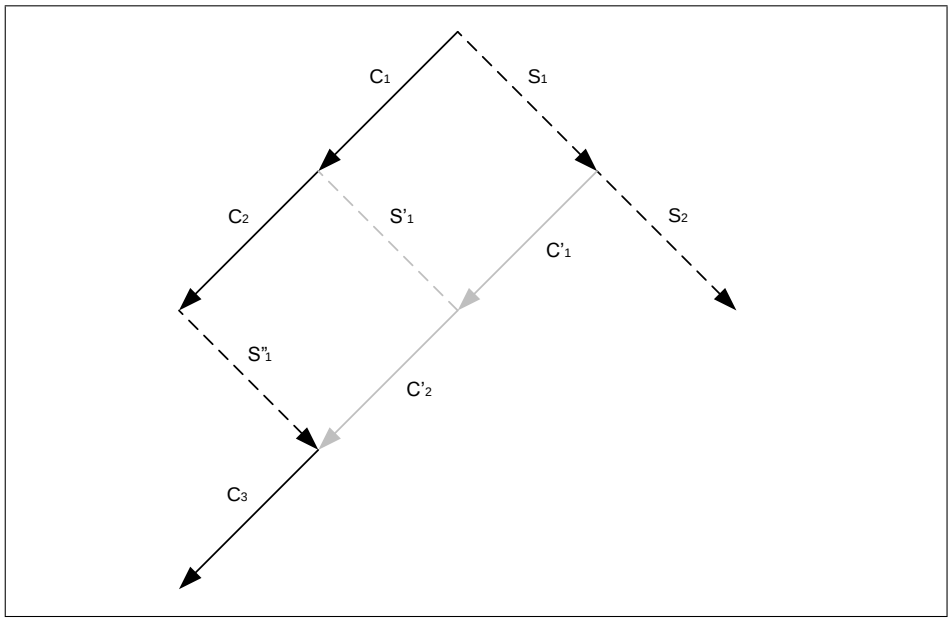


Figuur 6.10: Een toestandsruimte van een situatie waarin zowel de client als de server onafhankelijk twee operaties toepassen.

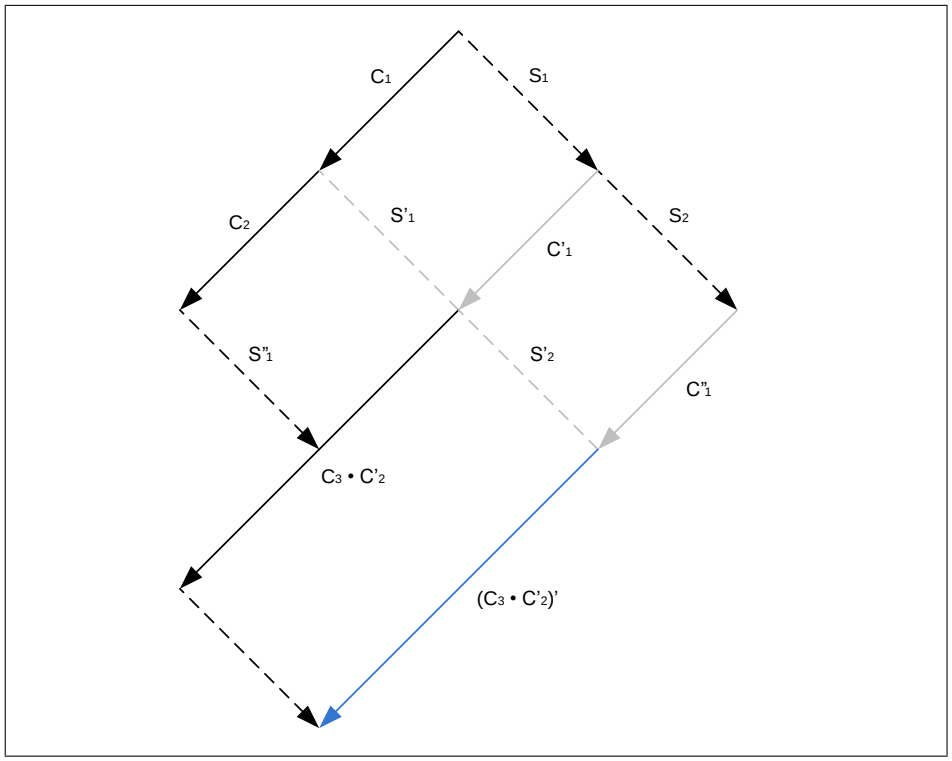
twee transformaties. Eerst wordt de reeds verzonden operatie (C_1) tegen de ontvangen serveroperatie (S_1) getransformeerd. Daarna wordt de buffer (in dit geval alleen operatie C_2) getransformeerd tegen S_1' (zie ook figuur 6.11). Met operatie S_1'' kan de client nu zijn gewenste toestand bereiken. Operatie C_2' vormt de nieuwe buffer. Merk op dat $C_1' \cdot C_2'$ gelijk is aan de brug.

- Stel dat de client nu een derde operatie toepast. Operatie C_1 is nog steeds niet bevestigd door de server. C_3 wordt daarom aan de buffer toegevoegd, dit betekent dat operatie C_1 en de buffer worden samengevoegd ($C_2' \cdot C_3$).
- Nu ontvangt de client operatie S_2. We gebruiken nu niet C_1 maar de eerder berekende operatie C_1' voor transformatie van de serveroperatie S_2. Vervolgens kan S_2' tegen de buffer getransformeerd worden.
- Dan eindelijk ontvangt de client de bevestiging (acknowledgment) van operatie C_1. Als alles goed is gegaan, is deze operatie gelijk aan operatie C_1''. Er is bij de berekening van de buffer rekening mee gehouden dat de buffer na deze operatie kan worden toegepast. De buffer operatie kan nu dus direct naar de server gestuurd worden. Als de server en client nu geen nieuwe operaties uitvoeren, zullen ze convergeren in dezelfde toestand (figuur 6.12).

Merk op dat de client telkens twee operaties bijhoudt: de buffer en de verzonden operatie. Wanneer de client een operatie van de server ontvangt transformeert deze de verzonden operatie tegen de ontvangen serveroperatie. Op deze manier is deze operatie altijd gelijk aan een operatie die de server mogelijk zal toepassen. Uiteindelijk zal de door de server getransformeerde operatie van de verzonden clientoperatie gelijk moeten zijn aan de laatst berekende verzonden operatie. In bovenstaande voorbeeld is dit operatie C_1''.



Figuur 6.11: De client berekent de nieuwe buffer door deze te transformeren tegen de ontvangen serveroperatie.



Figuur 6.12: De server bereikt dezelfde toestand als de client door de bufferoperatie uit te voeren.

Hoofdstuk 7

XMPP en Operational Transformation

De (kritische) lezer zal de vraag stellen waarom een groot gedeelte van deze scriptie over Operational Transformation gaat. Deze vraag zal in dit hoofdstuk beantwoord worden. Niet voor niets heeft deze scriptie de titel, XMPP en het real-time web. Met het real-time web in deze context wordt een internetervaring bedoeld waarbij informatie wordt ontvangen zodra het is gepubliceerd. Veel van de technieken die hier voor nodig zijn worden al geboden door de basis van XMPP en de reeds gedefinieerde uitbreidingen. Met de pubsub uitbreiding bijvoorbeeld, besproken in hoofdstuk 4, is het mogelijk dat abonnees op hoogte worden gesteld zodra informatie wordt gepubliceerd. De toepassing wordt geavanceerder wanneer gelijktijdige samenwerking mogelijk is tussen meerdere partijen. Dit betekent dat de wijzigingen die meerdere partijen tegelijkertijd aanbrengen aan eenzelfde object moeten worden gesynchroniseerd tussen de verschillende partijen (real-time collaboratie). Deze toepassing is precies waar OT voor gebruikt kan worden. Belangrijk voor deze keuze voor OT als concurrency control algoritme is dat het geen beperkingen oplegt aan de activiteiten van gebruikers. Operaties kunnen meteen lokaal op het document worden toegepast, de activiteiten van clients zijn onafhankelijk van latentie van het netwerk.

Dit hoofdstuk bespreekt kort het *Wave Federation Protocol* en zijn tekortkomingen en definieert daarnaast een uitbreiding voor het toepassen van OT in combinatie met XMPP.

7.1 Wave Federation Protocol

Google Wave is gebouwd met XMPP als onderliggende laag. De *Google Wave Federation Protocol Over XMPP* [1] is een open uitbreiding op de kernfunctionaliteiten van XMPP. Daarnaast wordt gebruik gemaakt van de, in hoofdstuk 4 besproken, Publish-Subscribe extensie. Google kiest er met Wave voor OT in te zetten voor het gezamenlijk bewerken van (gestructureerde) tekstdocu-

menten. In de introductie van OT werd al opgemerkt dat de techniek niet alleen ingezet kan worden voor documenten bestaande uit tekst, maar ook om grafische en multimedia documenten. De implementatie hiervan hoeft echter niet veel te verschillen: in Wave worden de documenten gerepresenteerd door XML documenten, de operaties zijn mutaties van deze documenten. Neem nu grafisch documenten, veel grafische documenten kunnen moeiteloos gerepresenteerd worden in het XML formaat. Een goed voorbeeld hiervan is het *Scalable Vector Graphics (SVG)* formaat, dit is een op XML gebaseerd formaat voor het representeren van tweedimensionale vectorafbeeldingen. In figuur 7.1 is te zien hoe met SVG een rechthoek, met een hoogte van 100 en een breedte 300 van pixels, weergegeven kan worden. Ook het OpenDocument formaat voor het

```
<?xml version='1.0' standalone='no'?>
<!DOCTYPE svg PUBLIC '-//W3C//DTD SVG 1.1//EN'
'http://www.w3.org/Graphics/SVG/1.1/DTD/svg11.dtd'>

<svg width='100%' height='100%' version='1.1'
xmlns='http://www.w3.org/2000/svg'>
<rect width='300' height='100'
style='fill:rgb(0,0,255);stroke-width:1;
stroke:rgb(0,0,0)'/>
</svg>
```

Figuur 7.1: Voorbeeld van een Scalable Vector Graphic

representeren van onder andere tekstdocumenten, presentaties en spreadsheets is op XML gebaseerd. Het blijkt dat een rijke variatie aan gegevens in het XML formaat weergegeven kan worden. Hierdoor zou het protocol bijvoorbeeld ook gebruikt kunnen worden voor een in real-time gedeelde agenda of zelfs een database systeem.

Helaas is het protocol achter Wave [1] redelijk specifiek. Het protocol is gericht op het Wave document model. Hoewel het waarschijnlijk niet onmogelijk is, maakt dit het wel lastig om het protocol te gebruiken voor andere documenttypes. Het doel nu is een meer generieke uitbreiding te definiëren voor Operational Transformation ondersteuning in XMPP. Hierbij is er een vaste server implementatie, zodat alle servers dezelfde taal spreken. De implementatie van de client hangt af van de toepassing: een tekstverwerker verschilt sterk van een programma voor het bewerken van afbeeldingen. Uiteraard kunnen er standaard bibliotheken worden ontwikkeld die functionaliteiten als transformatie en compositie afhandelen. Het formaat van de documenten is voor alle documenten gelijk (XML formaat), en ook de hierop toepasbare operaties zijn gelijk.

7.2 Uitbreiding

Deze paragraaf specificeert een uitbreiding op het XMPP kernprotocol [6] die het toepassen van Operational Transformation in combinatie met XMPP moet vergemakkelijken. Deze uitbreiding is geïnspireerd door het Wave (Federation) protocol [1]. Groot verschil is dat deze uitbreiding voor verschillende toepassingen, met verschillende documenttypes, inzetbaar moet zijn. De in dit hoofdstuk te bespreken uitbreiding is geen bestaande, officiële uitbreiding, maar een zelf voorgestelde uitbreiding. Hierbij moet opgemerkt worden dat dit een eerste voorstel is en er waarschijnlijk ruimte voor verbetering is. Daarnaast zou een werkende implementatie van deze uitbreiding erg bruikbaar zijn om de specificatie verder aan te scherpen. Hier ligt een uitdaging voor verder onderzoek.

7.2.1 Architectuur

De te bewerken objecten worden in het vervolg *documenten* genoemd. Wijzigingen aan documenten worden aangebracht door middel van *operaties*. Een operatie is opgebouwd uit verschillende componenten. Het achterliggende Operational Transformation, concurrency control, algoritme is deze zoals besproken in paragraaf 6.8.

Zoals gebruikelijk in XMPP zijn er twee communicatiepartijen: clients en servers. Clients communiceren alleen met de server waartoe ze behoren. Servers communiceren met lokale ('eigen') clients en andere servers. De server zal naast de uitbreiding in dit hoofdstuk ook de Publish-Subscribe uitbreiding moeten ondersteunen. Omdat clients van verschillende servers tegelijk aan hetzelfde document kunnen werken, moet elke server met (minstens) één deelnemende client een lokale kopie van het document bijhouden. Er is altijd één server die de oorspronkelijk kopie van het document bijhoudt, dit is de *host* van het document. We spreken van een *lokale* client of lokaal document als het document door de server gehost wordt. Als de server slechts een kopie bijhoudt spreken we van een *remote* client of document.

Documenten

De documenten zijn opgebouwd volgens het XML standaard. Een document bestaat uit een opeenvolging van verschillende items:

- start tags (<example>)
- eind tags (</example>)
- karakters

Een start tag heeft een naam en een verzameling van attributen. Deze attributen vormen sleutel-waarde paren, waarbij de sleutel en de waarde strings zijn. Elke start tag wordt afgesloten met een eind tag van dezelfde naam. Een eind tag is altijd gelijk aan de eerste onafgesloten start tag aan de rechter kant. Alle overige tekens zijn karakters.

Operaties

Een operatie definieert een verzameling van acties die specificeren hoe een document aangepast moet worden. Hierbij wordt er van links naar rechts door het document gelopen, en is elk item (start tag, eind tag of karakter) één element (dat bekent één stap in het document). Een operatie bestaat uit één of meerdere componenten. Deze componenten zijn deze zoals besproken in paragraaf 6.8.1:

- skip(aantal posities)
- insert characters(karakters)
- delete characters(aantal karakters)
- insert element start(naam, attributen)
- insert element end
- delete element start
- delete element end
- set attributes(attributen)
- update attributes(attributen)
- insert anti element start
- insert anti element end
- delete anti element start
- delete anti element end

Tijdens de verwerking van een operatie, wordt de huidige positie van de cursor bijgehouden. Deze cursorpositie wordt gewijzigd bij acties als *skip* en *insert characters*.

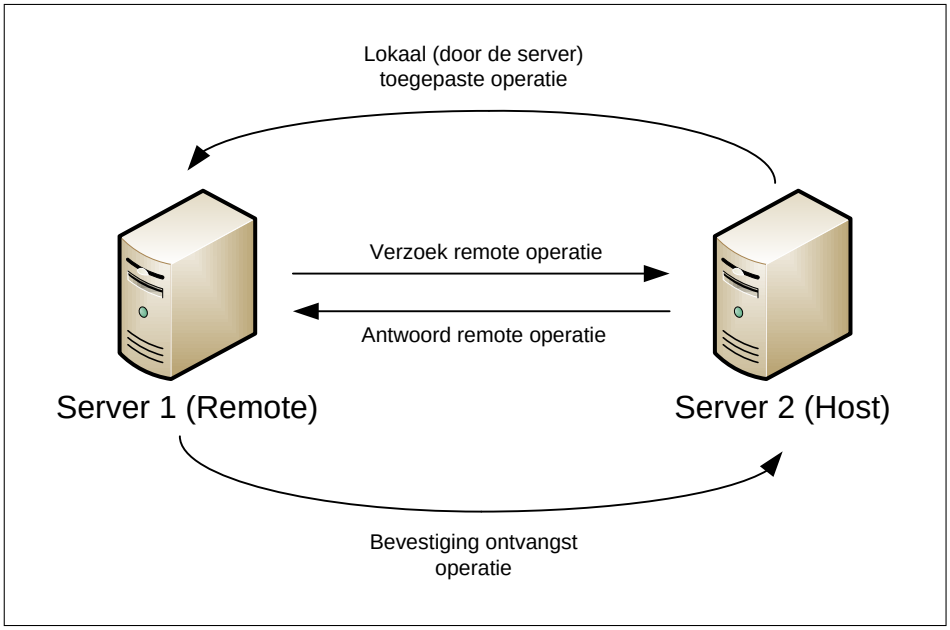
Naast deze document mutatie operaties zijn er operaties om partijen toe te voegen die het document mogen bekijken of bewerken:

- add observer (jid)
- add editor (jid)

Deze operaties kunnen alleen uitgevoerd worden door clients die toestemming hebben het document te bewerken of door de server die het document host.

Host en Remote component

De server kan zowel lokale als remote documenten bijhouden. Deze gebruikt daarom twee componenten voor communicatie tussen verschillende servers: een *host* en *remote* component. De *host component* verwerkt ontvangen operaties. Daarnaast moet deze component operaties die op het lokale document zijn toegepast doorsturen naar de servers van remote clients. Deze operaties die host component verstuurt worden ontvangen en verwerkt door de *remote component*. Operaties die remote clients toepassen op het document worden door de remote component doorgestuurd naar de server die het document host. Wanneer de remote server een operatie van een host ontvangt, zal deze de ontvangst moeten bevestigen door een *acknowledgment* te sturen. De host zal de uitgaande berichten opslaan totdat de ontvangst bevestigd is. Als de verbinding verbroken wordt, zullen de opgeslagen berichten nogmaals gestuurd worden zodra de verbinding weer herstelt is. In figuur 7.2 is de architectuur tussen servers gerepresenteerd. Om een server aan te geven die de host component gebruikt wordt



Figuur 7.2: Architectuur host en remote server

de term host of host server gebruikt. Een remote server of remote is een server die de remote component gebruikt.

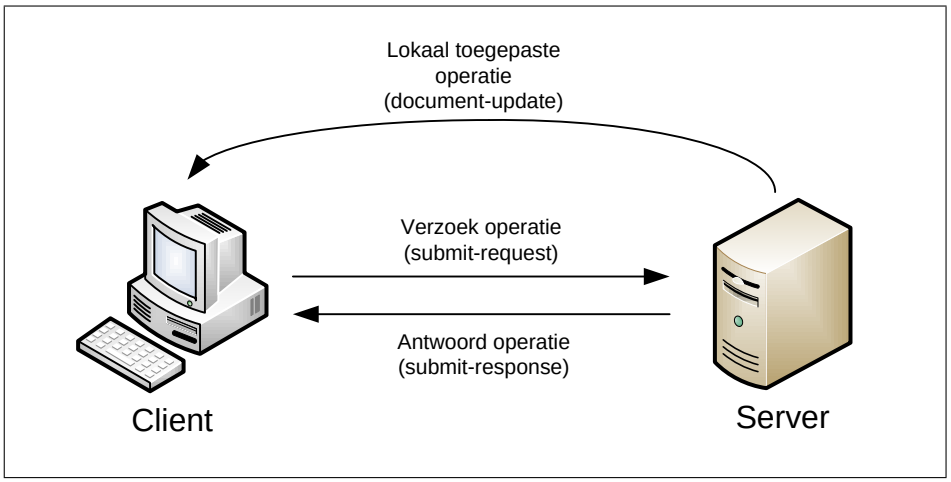
7.2.2 Client-Server communicatie

Eerst wordt nu de communicatie tussen client en server besproken. Vanuit het perspectief van de client maak het niet uit of de server een host of een remote

is: de berichten van en naar de server zijn voor beide componenten hetzelfde. De client moet er bij het versturen van een operatie wel rekening mee houden dat de verstuurde operatie ergens op het pad van de server ligt, in paragraaf 6.8 is besproken hoe dit gerealiseerd kan worden. Daarnaast kan de client pas nieuwe operaties verzenden als de server de vorige operatie heeft toegepast en hiervan een bevestiging is ontvangen.

Naast de operatie wisselen client en server informatie uit over het document waarop de transformatie van toepassing is. Dit omvat onder andere een versienummer dat een bepaalde versie van het document op de server aangeeft. Bij elke operatie die de server toepast zal dit versienummer met één opgehoogd worden. De client kan binnenkomende operaties ordenen met dit versienummer voordat ze getransformeerd en toegepast worden op het lokale document. Daarnaast kan een hash van het betreffende document meegegeven worden. Hiermee kunnen client en server nagaan of ze dezelfde toestand van het document voor zich hebben.

In figuur 7.3 is de communicatie tussen client en server schematisch weergeven. De nu volgende paragrafen bespreken voor de verschillende pijlen in de figuur



Figuur 7.3: Client en server communicatie

welke informatie er wordt overgebracht.

Versturen van een operatie

In deze uitbreiding wordt voor de communicatie gebruik gemaakt van de Publish-Subscribe uitbreiding (hoofdstuk 4). De client stuurt een operatie naar de server door een item te publiseren naar de 'http://example.org/protocol/ot' node. Deze node waarde geeft aan dat het bericht is gepubliceerd in de context van de uitbreiding gespecificeerd in deze paragraaf en verwijst direct naar de namespace van de payload. De operatie wordt opgenomen in het <*operation*>

element. Informatie over het document waarop de operatie toegepast moet worden, wordt gegeven door het <*document*> element. De unieke identifier van het document, dat bestaat uit een combinatie van het adres van de hostserver en een unieke string, wordt opgenomen door middel van het *id* attribuut. Met het *versie* attribuut is het laatst bekende versienummer van de server aangegeven. De client zorgt er altijd voor dat de verzonden operatie altijd vanuit het document (de toestand) behorend bij deze versie toegepast kan worden. De server hoeft de operatie hierdoor alleen te transformeren tegen de operaties die de server sindsdien heeft toegepast. Wanneer de client een nieuw document aan wil maken, verstuurt deze het <document> element zonder attributen. Een voorbeeld van een bericht waarmee een client een operatie naar de server kan versturen is hieronder gegeven. De termen client en server moeten (ook in latere voorbeelden) vervangen worden door het adres van de betreffende communicatiepartij.

```
<iq type='set' id='x' from='client' to='server'>
  <pubsub xmlns='http://jabber.org/protocol/pubsub'>
    <publish node='http://example.org/protocol/ot'>
      <item>
        <submit-request
          xmlns='http://example.org/protocol/ot'>
          <document id='host-server.com/4!e3mPc5'
            version='123'/>
          <operation>
            skip 5
            insert characters 'aba'
            skip 6
          </operation>
        </submit-request>
      </item>
    </publish>
  </pubsub>
</iq>
```

Zodra de server de operatie toegepast heeft zal deze het verzoek beantwoorden met een <*submit-response*> element. In dit bericht specificeert de server de versie en de hash van het document na het toepassen van de operatie. In het volgende voorbeeld heeft de server één andere operatie (van een andere client) toegepast voordat de verzonden operatie is toegepast (het versienummer is namelijk met twee gestegen).

```
<iq type='result' id='x' from='host' to='remote'>
  <pubsub xmlns='http://jabber.org/protocol/pubsub'>
    <publish node='http://example.org/protocol/ot'>
      <item>
        <submit-response
          xmlns='http://example.org/protocol/ot'>
          <document id='host-server.com/4!e3mPc5'
```

```
            version='125'
            hash='4d4380c965e6f137f2620d93dd7ecddf'/>
        </submit-response>
      </item>
    </publish>
  </pubsub>
</iq>
```

Ontvangen van een operatie

De server deelt operaties die op het lokale document zijn toegepast met participerende clients. Zoals gebruikelijk in de pubsub uitbreiding wordt dit bericht verstuurt met een message stanza waarin een <*event*> element is opgenomen. Voor het voorbeeld in deze paragraaf zal de server bijvoorbeeld het volgende bericht naar deelnemende clients versturen. Merk op dat de operatie is veranderd omdat deze getransformeerd is door de server.

```
<message type='normal'
  id='y'
  from='server'
  to='client'>
  <event xmlns='http://jabber.org/protocol/pubsub#event'>
    <items node='http://example.org/protocol/ot'>
      <item>
        <document-update xmlns='http://example.org/protocol/ot'>
          <document id='host-server.com/4!e3mPc5'
            version='125'
            hash='4d4380c965e6f137f2620d93dd7ecddf'/>
          <operation>
            skip 7
            insert characters 'aba'
            skip 6
          </operation>
        </document-update>
      </item>
    </items>
  </event>
</message>
```

7.2.3 Communicatie remote component

Deze paragraaf bespreekt de communicatie tussen servers, te beginnen met de communicatie van een remote naar een host server. In figuur 7.2 is schematisch de communicatie tussen servers weergegeven. De berichten in deze paragraaf zijn vrijwel gelijk aan de berichten gezien in paragraaf 7.2.2, ze worden hier daarom minder uitvoerig besproken.

Versturen van een operatie

Een remote server ontvangt operaties van zijn clients. Deze server stuurt de ontvangen operaties door naar de host server. Operaties worden verstuurd naar de 'http://example.org/protocol/ot' node. Onderstaande voorbeeld laat zien hoe de remote server een operatie verzendt door middel van een iq verzoek. De termen remote en host (ook in latere voorbeelden) moeten vervangen worden door het adres van respectievelijk de remote en host server.

```
<iq type='set' id='x' from='remote' to='host'>
  <pubsub xmlns='http://jabber.org/protocol/pubsub'>
    <publish node='http://example.org/protocol/ot'>
      <item>
        <submit-request
          xmlns='http://example.org/protocol/ot'>
          <document id='host-server.com/4!e3mPc5'
            version='123'/>
          <operation>
            skip 5
            insert characters 'aba'
            skip 6
          </operation>
        </submit-request>
      </item>
    </publish>
  </pubsub>
</iq>
```

Bevestiging operatie

De host reageert op dit verzoek, met een <*submit response*> element. Deze bevat een <*document*> element met het versienummer en de hash van het nieuwe document.

```
<iq type='result' id='x' from='host' to='remote'>
  <pubsub xmlns='http://jabber.org/protocol/pubsub'>
    <publish>
      <item>
        <submit-response
          xmlns='http://example.org/protocol/ot'>
          <document id='host-server.com/4!e3mPc5'
            version='125'
            hash='4d4380c965e6f137f2620d93dd7ecddf'/>
        </submit-response>
      </item>
    </publish>
  </pubsub>
```

```
</iq>
```

Operaties die een remote server ontvangt van zijn lokale client worden doorgestuurd naar de host server. De host zal vervolgens de getransformeerde, toegepaste operatie doorsturen naar alle remote servers (zie ook de volgende paragraaf). De remote server past deze operatie vervolgens toe op zijn lokale document.

7.2.4 Communicatie host component

Versturen van een toegepaste operatie

Wanneer de host een operatie heeft toegepast, verstuurt deze de toegepaste operatie naar alle remote servers. Hiervoor wordt de message stanza gebruikt. Om aan te geven dat het een update van het document betreft is in deze message stanza een <*document-update*> element opgenomen. Het <*document*> element geeft informatie over het document na toepassen van de operatie. In het <*operation*> element wordt de toegepaste operatie opgenomen. Ontvangstbevestiging wordt gedaan volgens de *Message Receipts* (XEP-0184) uitbreiding.

```
<message type='normal'
  from='host'
  id='x'
  to='remote'>
  <request xmlns='urn:xmpp:receipts'/>
  <event xmlns='http://jabber.org/protocol/pubsub#event'>
    <items node='http://example.org/protocol/ot'>
      <item>
        <document-update xmlns='http://example.org/protocol/ot'>
          <document id='host-server.com/4!e3mPc5'
            version='125'
            hash='4d4380c965e6f137f2620d93dd7ecddf'/>
          <operation>
            skip 7
            insert characters 'aba'
            skip 6
          </operation>
        </document-update>
      </item>
    </items>
  </event>
</message>
```

Bevestiging ontvangst operatie

De remote stuurt een *acknowledgement* om te bevestigen dat de operatie is ontvangen. Dit bericht ziet er als volgt uit:

```
<message id='y'
  from='remote'
  to='host'>
  <received
    xmlns='urn:xmpp:receipts' id='x' />
</message>
```

Het *id* attribuut in het *<received>* element moet gelijk zijn aan het id van het bericht dat bevestigd wordt.

De remote server past de van de server ontvangen operatie op zijn lokale document toe en verstuurt de operatie vervolgens naar zijn clients.

7.2.5 Karakteristieke elementen

Deze paragraaf geeft een overzicht van de voor deze uitbreiding karakteristieke elementen.

<submit-request> element

Het <submit-request> element wordt gebruikt om een nieuwe operatie naar een server te verzenden, het bevat de volgende elementen:

- <document> element
- <operation> element

<submit-response> element

Een <submit-response> element is een antwoord van de server op een submit-request. Deze bevat de volgende elementen:

- <document> element
- <operation> element

<document-update> element

Een <document-update> wordt gebruikt door de server om andere partijen op de hoogte te stellen van wijzigingen die aangebracht zijn aan het document op de server. De elementen die dit attribuut bevat zijn de volgende:

- <document> element
- <operation> element

<document> element

Een <document> geeft informatie over een document waarop een bepaalde operatie van toepassing is. Dit element bevat de volgende attributen:

id Identifier van het document (verplicht)

version Het nummer van de versie (verplicht)

hash De hash van het versie document

<operation> element

Dit element bevat een operatie, bestaande uit verschillende componenten, als platte tekst. Elk component moet op een nieuwe regel zijn geplaatst.

Hoofdstuk 8

Conclusie en verder onderzoek

8.1 Conclusie

Dit onderzoek begon met de vraag: *Hoe kan het eXtensible Messaging and Presence Protocol (XMPP), en (zelf voorgestelde) uitbreidingen hierop, ingezet worden voor real-time communicatie en collaboratie op het web?*.

Het onderzoek heeft laten zien dat het kernprotocol van XMPP een uitstekende basis biedt voor real-time communicatie. De kern definieert verschillende mechanismen om berichten tussen verschillende partijen te versturen en te ontvangen. Het open karakter van XMPP geeft daarnaast iedereen de mogelijkheid zijn eigen XMPP client of server applicatie te schrijven. Netwerken worden gevormd door meerdere gedecentraliseerde servers met elkaar te verbinden. Dankzij het uitbreidbare ontwerp kan XMPP gemakkelijk verrijkt worden met nieuwe functionaliteiten. Een goed voorbeeld hiervan is de Publish-Subscribe uitbreiding, besproken in hoofdstuk 4. Deze uitbreiding geeft partijen de mogelijk direct notificaties van nieuwe publicaties te ontvangen. In hoofdstuk 5 is de BOSH uitbreiding besproken. BOSH maakt het mogelijk XMPP in combinatie met het HTTP protocol te gebruiken, waardoor het onder andere ook binnen een webbrowser toegepast kan worden.

Om XMPP voor real-time collaboratie in te zetten blijkt een concurrency control algoritme nodig voor het synchroniseren van gelijktijdige operaties van verschillende partijen. In hoofdstuk 6 is daarom Operational Transformation (OT) besproken. Groot pluspunt van OT is dat het geen beperkingen oplegt aan de activiteiten van gebruikers. In hoofdstuk 7 is tenslotte een uitbreiding gedefinieerd die het toepassen Operational Transformation in combinatie met XMPP moet vergemakkelijken. Dit hoofdstuk besprak onder andere de architectuur en berichten die tussen verschillende partijen uitgewisseld worden om operaties op documenten tussen partijen over te brengen. Bij het ontwerp was van belang dat de uitbreiding voor verschillende soorten documenten te

gebruiken is.

8.2 Verder onderzoek

In paragraaf 7.2 is al opgemerkt dat een implementatie van de besproken uitbreiding kan helpen bij het aanscherpen van de uitbreiding. Een werkende implementatie kan bijvoorbeeld leiden tot optimalisaties, nieuwe toevoegingen aan de uitbreiding of helpen bij het vinden van eventuele fouten. Hierin ligt nog een grote uitdaging voor verder onderzoek.

In het onderzoek is bewust gekozen voor een diepgaande bespreking van bepaalde technieken, in plaats van een vergelijking tussen concurrerende technieken. Dit is ook iets waar verder onderzoek zich op kan richten. Zo zou men kunnen denken aan een comparatieve studie waarbij het OT algoritme vergeleken wordt met andere control algoritmes. Of een vergelijking van XMPP met andere real-time communicatie protocollen (bijvoorbeeld SIMPLE).

Bibliografie

[1] Anthony Baxter, Jochen Bekmann, Daniel Berlin, Soren Lassen, and Sam Thorogood. Google wave federation protocol over xmpp. `http://www.waveprotocol.org/draft-protocol-specs/draft-protocol-spec`, 2009.

[2] Clarence A. Ellis and Simon J. Gibbs. Concurrency control in groupware systems. In *SIGMOD Conference*, pages 399–407, 1989.

[3] Peter Millard, Peter Saint-Andre, and Ralph Meijer. Xep-0060: Publish-subscribe, 2007.

[4] Jack Moffitt. *Professional XMPP Programming with JavaScript and jQuery*. Wrox, January 2010.

[5] David A. Nichols, Pavel Curtis, Michael Dixon, and John Lamping. High-latency, low-bandwidth windowing in the jupiter collaboration system. In *ACM Symposium on User Interface Software and Technology*, pages 111–120, 1995.

[6] Peter Saint-Andre. *RFC 3920: Extensible Messaging and Presence Protocol (XMPP): Core*. Internet Engineering Task Force, October 2004.

[7] Peter Saint-Andre. *RFC 3921: Extensible Messaging and Presence Protocol (XMPP): Instant Messaging and Presence*. Internet Engineering Task Force, October 2004.

[8] Peter Saint-Andre, Kevin Smith, and Remko Tronçon. *XMPP: The Definitive Guide: Building Real-Time Applications with Jabber Technologies*. O'Reilly Media, Inc., May 2009.

[9] Daniel Spiewak. Understanding and applying operational transformation. `http://www.codecommit.com/blog/java/understanding-and-applying-operational-transformation`, 2010.

[10] Chengzheng Sun, Xiaohua Jia, Yanchun Zhang, Yun Yang, and David Chen. Achieving convergence, causality preservation, and intention preservation in real-time cooperative editing systems. *ACM Trans. Comput.-Hum. Interact.*, 5(1):63–108, 1998.

[11] David Wang and Alex Mah. Google wave operational transformation. http://www.waveprotocol.org/whitepapers/operational-transform.