Mijn Thesis Verdoeme

26 mei 2002

VRIJE UNIVERSITEIT BRUSSEL
Faculteit Wetenschappen - Departement Informatica
Academiejaar 2001 - 2002

Het Gebruik van een Declaratieve Taal ter Ondersteuning van het Concurrentie-beheer van Mobiele Multi Agent Systemen

Kristoff Van Buggenhout

Promotor: Prof. Dr. Theo D'Hondt
Begeleider: Werner Van Belle

Proefschrift ingediend met het oog
op het behalen van de graad van
Licentiaat in de Toegepaste Informatica.

In dit document presenteren we een meta-level synchronisatiemechanisme voor gedistribueerde multi agent systemen. Het mechanisme is gebaseerd op Aspect Oriented Programming, en zal de boodschappen doorgestuurd door agents unifiëren met in Prolog gedeclareerde guarding statements. Deze regels beslissen dan of de boodschap doorgegeven mag worden, indien niet wordt ze aan een queue toegevoegd. De geblokkeerde berichten in deze queue worden op regelmatige tijdstippen opnieuw geunifiëerd met de guarding regels.

We presenteren de primitieven die we nodig hebben om de programmeur toe te laten de guarding rules te declareren, en gebruiken ze om enkele voorbeelden te maken die demonstreren dat we een gespecialiseerd synchronisatiemechanisme kunnen ontwerpen voor elke specifieke situatie.

Vervolgens demonstreren we hoe we met de primitieven andere veel gebruikte synchronisatiemechanismen als semaforen, rendez-vous en gedistribueerde transactiesystemen kunnen bouwen met slechts enkele eenvoudige regels.

Tot slot tonen we de echte kracht en flexibiliteit van de primitieven door een adaptor te maken tussen binaire semaforen en resource agents uit een transactiesysteem, en maken we een high-level flow control mechanisme waar we concurrentie regelen vanuit een centrale agent die informatie bezig over welke messages conflicteren met welke andere, en dusdanig de synchronisatie kan regelen.

Dankwoord

Deze thesis is er gekomen dankzij de hulp en steun van verschillende mensen. Ik zou vooral mijn ouders willen bedanken voor hun morele steun, en voor me de mogelijkheid te schenken universitaire studies te volgen.

Verder ben ik Prof. Dr. Theo D'Hondt dankbaar om in mij te blijven geloven toen ik mezelf vorig jaar serieus in de knoei werkte.

Ook heb ik veel te danken aan mijn begeleider Werner Van Belle, niet enkel voor de wijze raad en hulp bij het maken van de implementatie en het schrijfwerk, maar ook voor de vele pep-talks waarin hij me telkens opnieuw probeerde te motiveren voort te werken als ik weer even het licht aan het eind van de tunnel uit het oog verloor.

Mijn broer en het voltallige bestuur van InfoGroep verdienen ook een vermelding, voor hun steun en motiverende gesprekken tijdens de finale fase van het werk. Merci mannen.

Voorts wil ik nog iedereen bedanken die op een of andere manier bijgedragen heeft tot de produktie van dit werk.

Inhoudsopgave

Lijst van figuren

1. Inleiding

Computers zijn reeds lange tijd in staat om met elkaar te communiceren via netwerken. Het duurde echter niet lang voor men ontdekte dat verschillende programmas, die met elkaar interageren, tegelijk uitvoeren voor problemen zorgt. Deze problemen noemen we concurrentieproblemen. Er zijn verschillende oplossingen gesuggereerd, elk met hun eigen voor- en nadelen, en in de meeste gevallen zijn de beschikbare mechanismen wel toereikend.

Iets recenter is de notie van agenten, onafhankelijke programmas, die in staat zijn niet enkel met elkaar te communiceren, maar ook netwerken kunnen gebruiken als transportmiddel, met andere woorden het zijn mobiele programmas. Agenten worden meestal gebruikt om grote, complexe problemen op te splitsen in kleinere, gemakkelijker behandelbare deelproblemen. Meestal vervullen verschillende agenten samen dus een grotere taak. Concurrentieproblemen in agent systemen zijn dus bijna niet te vermijden. De meeste concurrentiemechanismen zijn echter niet volledig geschikt voor gebruik binnen agent systemen, wegens gebrek aan flexibiliteit.

We zullen trachten de complexiteit van concurrentieproblemen in multi-agent systemen te vereenvoudigen door de concurrentie op een apart niveau aan te pakken, een niveau tussen het agent-niveau en het communicatie-niveau. Het basismechanisme wordt rule-based guarding op messages: alle berichten die de agent verstuurd en ontvangt worden eerst door het systeem gevalideerd. Berichten die problemen kunnen veroorzaken worden tijdelijk tegengehouden. Wanneer het weer veilig is de berichten te verzenden zal dit ook gebeuren. We gaan op zoek naar de geschikte primitieven om een zo krachtig en flexibel mogelijk synchronisatie-mechanisme te kunnen uitbouwen.

In het tweede hoofdstuk zullen we enkele belangrijke gevolgen van gedistribueerd programmeren aanhalen, en we presenteren enkele talen die ons toelaten gedistribueerd te programmeren.

In het volgende hoofdstuk leiden we concurrentieproblemen in: wat zijn ze, waardoor worden ze veroorzaakt en wat kunnen we eraan doen. We presenteren enkele populaire synchronisatiemechanismen, en bespreken hun voor- en nadelen.

Het vierde hoofdstuk presenteert de voordelen van het gebruik van een apart niveau voor agent-synchronisatie, en verduidelijkt waarom we een declaratieve programmeertaal verkiezen om het systeem te implementeren. We zullen de algemene design van het guarding systeem presenteren, uitgaande van een vergelijking met Aspect Oriented Programming.

In dit hoofdstuk presenteren we ook de benodigde synchronisatieprimitieven voor het meta-level guarding systeem, en we demonstreren de werking ervan via twee eenvoudige voorbeelden, en we doen een eerste validatie door de synchronzatieprimitieven te gebruiken om enkele andere synchronisatiemechanismen te implementeren (met name binaire semaforen, rendez-vous en een gedistribueerd transactiesysteem).

Verdere validatie gebeurt in hoofdstuk vijf, waarin we de ware kracht van het systeem demonstreren via twee voorbeelden. In het eerste voorbeeld tonen we aan dat het systeem bruikbaar is om op eenvoudige wijze verschillende synchronisatiemechanismen met elkaar te laten samenwerken, en het tweede voorbeeld demonstreert een high-level flow control mechanisme voor agenten zonder onderling concurrentiebeheer (zoals bijvoorbeeld agenten gemaakt door verschillende partijen).

Tenslotte trekken we de nodige besluiten, en we presenteren enkele mogelijke vormen van toekomstig werk dat kan verricht worden.

2. Gedistribueerd Programmeren

Wanneer we gedistribueerd programmeren, is het heel belangrijk in het achterhoofd te houden dat er grote conceptuele verschillen bestaan tussen gedistribueerd programmeren, en klassiek "lokaal" programmeren. In de volgende secties worden de grootste verschillen uitgelegd.

2.1. Componenten

Bij de meeste moderne programmeertalen merken we dat binnen een programma datastructuren en functies die samen een bepaalde taak vervullen gegroepeerd worden. Deze groepering geeft aanleiding tot het ontstaan van "objecten" (bijvoorbeeld een vector datastructuur en de functies PUSH, POP, PEEK en EMPTY? vormen een stack object). Deze objecten kunnen expliciet gedeclareerd worden, zoals in de meeste object-georienteerde programmeertalen, waar we klassen definiëren, meestal met een inheritance systeem; maar de vorming van objecten kan ook impliciet zijn, zoals in talen als C en Scheme. In deze laatste categorie zijn de objecten eerder conceptueel, de groepering in objecten is enkel zichtbaar voor de programmeur, niet voor de compiler of interpreter.

In gedistribueerde programmeertalen kunnen we ook deze opdeling in verschillende stukken maken, maar vermits onze objecten zich op verschillende machines kunnen bevinden, en dus eigenlijk kunnen beschouwd worden als op zichzelf staande programmas, elk met een eigen data space en execution stack. Het zijn dus geen eenvoudige objecten meer, maar onafhankelijke programmas die een deeltaak van een grotere taak vervullen.

2.2. Communicatie

Vermits gedistribueerd programmeren impliceert dat we werken met verschillende componenten, die zich meestal op verschillende machines bevinden, is het duidelijk dat de communicatie tussen deze verschillende componenten een belangrijk aspect van gedistribueerd programmeren is. We zullen verschillende aspecten van communicatie beschouwen, onder andere wat er doorgegeven wordt (het protocol), hoe we de informatie doorgeven (synchrone of asynchrone communicatie), en data ownership. Vervolgens zullen we enkele voorbeelden zien die demonstreren hoe enkele programmeertalen toelaten gedistribueerd te programmeren.

2.2.1. Protocol

De verschillende componenten van gedistribueerde programmas zullen met elkaar moeten communiceren. Om ervoor te zorgen dat zender en ontvanger elkaar verstaan zullen ze in hun communicatie vastgelegde "afspraken" over de opmaak en inhoud van de berichten moeten respecteren. Deze afspraken noemen we het protocol van de data transmissie. Om zinvol te kunnen werken moeten alle communicatiepartners dit protocol kennen, en respecteren.

Als voorbeeld zien we de communicatie tussen een email-client en server, gebruik makend van het SMTP protocol (Simple Mail Transfer Protocol) beschreven in [2]. Regels voorafgegaan door 'C' zijn boodschappen die de client verzendt naar de server, regels beginnend met 'S' zijn antwoorden van de server.

```
  kvbuggen@bug:~$ telnet mail-out.pandora.be 25
  Trying 195.130.132.45...
  Connected to mail-out.pandora.be.
  Escape character is '^]'.
S 220 horkos.telenet-ops.be ESMTP Postfix
C HELO nerom.cjb.net
S 250 horkos.telenet-ops.be
C MAIL FROM:kvbuggen@bug.nerom.cjb.net
S 250 Ok
C RCPT TO:kvbuggen@cthulhu.rave.org
S 250 Ok
C RCPT TO:kvbuggen@wendy.vub.ac.be
S 250 Ok
C RCPT TO:kvbuggen@vub.ac.be
S 250 Ok
C DATA
S 354 End data with <CR><LF>.<CR><LF>
C To: Kristoff Van Buggenhout <kvbuggen@cthulhu.rave.org>
C Cc: En nog eens <kvbuggen@wendy.vub.ac.be>
C Subject: Een email
C Met de hand gemaakt zelfs
C
C Groeten,
C --
C Bug
```

```
C .
S 250 Ok: queued as EBE3A84244
C QUIT
S 221 Bye
```

Na het verbinden zal de client zich identificeren door een `HELO` boodschap te sturen, daarna mag de client de te versturen boodschappen doorgeven. Het begin van een boodschap wordt aangeven door het `MAIL` commando, met als argument het return adres (het adres waar een eventuele foutboodschappen zullen toekomen). Vervolgens kunnen een of meer `RCPT` commando's volgen, deze bepalen de ontvangers van het bericht. Na het `DATA` commando kan de client de inhoud van de boodschap doorgeven, afgesloten met een punt op een voorts lege regel. Hierna mag een nieuwe boodschap doorgegeven worden, ofwel kan de verbinding worden gesloten met het `QUIT` commando.

Merk op dat de inhoud van de boodschap ook een bepaald protocol volgt, dit keer een protocol gebruikt door de verschillende beschikbare mail-programmas. De informatie gecodeerd in het protocol van de boodschap is echter niet relevant voor de SMTP server, deze zal altijd het bericht afleveren aan de opgegeven adressen, en een foutboodschap bij de afzender bezorgen indien de boodschap niet afgeleverd kan worden. Beschikbare email-readers zullen de informatie op de To:, Cc:, en Subject: regels in de boodschap gebruiken om een overzicht van de mailbox van een gebruiker te geven, en wanneer de gebruiker een reply stuurt op een bericht zal deze verstuurd worden naar de adressen in de To: en Cc: regels.

2.2.2. Synchroon of Asynchroon?

Agenten kunnen synchroon of asynchroon met elkaar communiceren. Method calls in object-georienteerde programmeertalen werken synchroon: het object dat een method van een ander object oproept zal wachten tot het andere object klaar is met het berekenen van het antwoord. De reden hiervoor is nogal duidelijk: vermits er maar één thread actief is kan het niet anders als synchroon werken.

Voor gedistribueerde systemen is dit echter niet optimaal: we onderbreken het werk van een agent liefst niet om te wachten op het antwoord van een andere agent, we zouden liever hebben dat de agent tijdens het wachten op antwoord andere dingen kan doen. Dit is asynchrone communicatie: bij het verzenden van een bericht (of oproepen van een method van een andere agent) zullen we niet onmiddelijk wachten op antwoord van de andere agent, zodat we intussen nog andere dingen kunnen doen. Een voorbeeld:

We evalueren in agent B (de -> operator staat voor asynchrone procedure call):

```
ping(x, ret) :: ret->pong(x * x)
:<function ping>
```

en in agent A:

```
pong(res) :: display("Result: ",res)
:<function pong>

{ B->ping(16, agentself), display(2 * 4, eoln)}
:8
:Result: 256
```

Wanneer we `B->ping(...)` oproepen zal deze functie onmiddelijk `<void>` teruggeven, en agent B zal functie `ping` beginnen uitvoeren. Intussen kan agent A eventueel andere dingen doen (zoals bijvoorbeeld 2 * 4 berekenen), en wanneer de berekening klaar is zal agent B het antwoord versturen naar agent A door functie `pong` op te roepen.

2.2.3. Ownership of data

De meest voor de hand liggende manier om gegevens over het netwerk te sturen is by value: de volledige datastructuur wordt geserialiseerd, over het netwerk gezonden, en gedeserialiseerd. Serialiseren is het omzetten van complexe datastructuren (zoals bijvoorbeeld bomen) naar een lineaire structuur, geschikt voor transport over een netwerk.

Voor kleine objecten, zoals getallen, strings, of kleine tabellen is dit goed, maar wanneer we grote structuren zoals bomen of objecten met een inheritance structuur moeten gaan kopieren over het netwerk kan hier serieus wat tijd verloren gaan. De oplossing op eerste zicht is duidelijk: we kunnen gewoon een referentie naar het object doorgeven, en elke keer we de gegevens nodig hebben, zal het systeem deze referentie volgen naar het systeem waar de data in het geheugen zit. Hier kunnen we echter serieus in de problemen komen wanneer de programmeertaal gebruik maakt van een garbage collector voor het geheugenmanagement: lokaal zijn er geen referenties naar het object, dus zal het verdwijnen. Zelfs in talen zonder garbage collector kan pass by reference voor problemen zorgen.

2.3. Partial Failure/Error Handling

Een ander typisch probleem bij gedistribueerde systemen is partial failure. In tegenstelling tot centrale systemen, waar een programma ofwel werkt, ofwel niet (total failure); kan een enkele component van het systeem uitvallen (partial failure). Ook het slecht functioneren van een component, eventueel door fouten in de omgeving waar de agent runt geeft aanleiding tot partial failure.

De enige oplossing tot dit probleem is ervoor zorgen dat agenten de mogelijkheid hebben de uitgevallen componenten opnieuw op te starten, en het systeem terug in een consistente staat te brengen (eventueel door rollbacks van reeds uitgevoerde instructies in andere agenten). Het opsporen van uitgevallen agenten kan gebeuren via timeouts: wanneer een agent een ander agent contacteerd, en gedurende een bepaalde periode geen antwoord krijgt mag de agent aannemen dat die component uitgevallen is. Deze periode is sterk afhankelijk van het type netwerk: bij lokale netwerken ligt de tijd tussen zenden en ontvangen van boodschappen veel hoger dan bijvoorbeeld bij satelietnetwerken, waar deze tijd kan oplopen tot zelfs enkele seconden. Wanneer het agentsysteem runt over een Wide Area Network (WAN) zoals bijvoorbeeld het internet, kunnen we echter op geen enkele manier op voorhand de grootte-orde van de delay bepalen, dus is het gok-en-meetwerk.

Het slecht functioneren van een agent opsporen is nog moeilijker: het enige dat we kunnen doen is de bewerking door verschillende agenten laten uitvoeren (op verschillende machines, om fouten in het systeem van de host te elimineren), en dan de resultaten vergelijken. Om een "*fail-once, still functional, fail-twice, still safe*"-systeem[1] te ontwikkelen hebben we tot 4 'klonen' van de originele agent nodig. Vermits elke bewerking moet uitgevoerd worden door de verchillende agenten, en de oplossingen achteraf moeten vergeleken worden, hebben we veel processing en network overhead. En zelfs dit systeem is niet volledig veilig: wat gebeurt er als het systeem verantwoordelijk voor het vergelijken van de resultaten zelf uitvalt?

Het partial failure fenomeen vormt een groot probleem binnen de wereld van gedistribueerde systemen, maar vermits het los staat van concurrentieproblemen, gaan we het verder niet beschouwen.

2.4. Gedistribueerde Programmeertalen

2.4.1. Sockets

Een socket is de meest eenvoudige manier om twee programmas met elkaar te laten communiceren. Een socket is een file descriptor die werkt als direct communicatiekanaal tussen twee processen: hetgeen het ene proces er in schrijft, kan het andere proces er uit lezen. Er bestaan twee soorten sockets: UDP sockets en TCP sockets. Het verschil zit in de Quality of Service (QoS), en de manier waarop beide soorten sockets een verbinding maken.

- UDP sockets zijn het best vergelijkbaar met een postbus: pakketten verzonden over UDP bevatten het adres waar ze naartoe moeten, en de route wordt bepaald per pakket. Er is ook geen enkele vorm van controle of pakketten hun doel bereikt hebben, en in welke vorm of volgorde.

  Het voordeel van deze sockets is dat de verbinding heel snel werkt: er is geen delay voor opzetten van connecties, en er moet geen QoS informatie mee doorgezonden

[1] Dit zijn systemen die zelfs na het uitvallen van een agent en een van zijn kloons, nog steeds kunnen detecteren of er fouten in de bewerkingen van de overblijvende kloons zitten.

worden. Deze sockets zijn dus geschikt voor toepassingen waarin veel informatie frequent over het netwerk verzonden wordt, zodat het niet veel uitmaakt dat er ergens een pakket niet toekomt. Een voorbeeld hiervan is Quake: de positie van een speler wordt verschillende keren per seconde verstuurd naar de server, dus het maakt niet uit als enkele van deze pakketten niet toekomen.

- TCP sockets zijn vergelijkbaar met een telefoonlijn: voor er kan gecommuniceerd worden moet een verbinding gemaakt worden, en elk pakket neemt dezelfde route naar de bestemming. Ook wordt gegarandeerd dat elk opgestuurd pakket zal toekomen, en in de originele volgorde.

  Dit protocol is geschikt voor toepassingen waarin we betrouwbare netwerkcommunicatie nodig hebben, zoals bijvoorbeeld een betaalautomaat. Omdat we connecties gebruiken zal bij TCP een extra overhead zijn om de connecties te maken en achteraf terug te sluiten, en om ervoor te zorgen dat alle pakketten in de juiste volgorde en foutloos aankomen moeten we in elk pakket buiten de nuttige data nog extra informatie steken (bijvoorbeeld pakketnummer en een checksum).

Zowel TCP als UDP pakketten worden over poorten verzonden. Poorten in een computer lijken op telefoons in een gebouw: zoals een gebouw vele telefoons kan hebben, en elke telefoon wordt opgenomen door een specifiek persoon, zo ook heeft een computer vele poorten, en elke poort hoort bij een specifieke service. Een FTP server bijvoorbeeld zal altijd gecontacteerd worden op poort 21. Ook de client heeft een poort nodig, het operating system zal hiervoor een random poort kiezen op het moment dat de verbinding gemaakt wordt. Het spreekt voor zich dat elke poort maar door één programma tegelijk kan gebruikt worden, dus bij het ontwerpen van netwerk software moet de maker ervoor zorgen dat het gekozen poortnummer niet reeds gebruikt wordt door een andere service.

Nog een probleem is het doorgeven van data, hier zien we weer een verschil tussen UDP en TCP:

- Bij UDP is de programmeur meestal zelf verantwoordelijk voor het vullen van de data-segmenten en het opsturen van de pakketten. Hiermee moet hij rekening houden met de maximum lengte van het data-segment van een UDP pakket. Vermits UDP geen garantie biedt dat verzonden pakketten ook aankomen, moet de programmeur ervoor zorgen dat alle informatie die hij wil doorzenden binnen één pakket valt. Hij zou de informatie eventueel kunnen opsplitsen over twee UDP pakketten, maar het kan zijn dat een van de twee pakketten niet toekomt, in dat geval is de originele data niet reconstrueerbaar.

- Omdat TCP communicatie connection management een zeker niveau van QoS aanbiedt, zorgen de meeste programmeertalen ervoor dat TCP sockets streams kunnen gebruiken. Elke socket heeft een inkomende en uitgaande datastream, deze kunnen gebruikt worden om strings te verzenden. Deze streams zorgen ervoor dat de programmeur geen rekening meer moet houden met de maximum lengte van pakketten, maar het beschikbare kanaal is nog altijd heel primitief: we kunnen enkel tekst doorsturen. In Java krijgen we hulp vanuit de Virtual Machine: we kunnen object streams gebruiken om objecten automatisch te laten serialiseren. Aan de kant van de ontvanger hebben we echter nog een probleem: we moeten weten van welk type een aangekomen bericht is voor we het kunnen typecasten. Wanneer we dus verschillende soorten objecten willen doorsturen moeten we eerst duidelijk maken aan de ontvanger welk type object we gaan sturen. De meest eenvoudige methode is eerst een identificatiecode doorsturen (een integer bijvoorbeeld), en dan het object zelf.

Als voorbeeld zien we hier de implementatie van een echo-server, wanneer we hiermee een verbinding maken zal de server alle data die hij ontvangt, onveranderd terug sturen naar de client. De server is geschreven in Java, gebruikt TCP sockets, en wacht op verbindingen op poort 7, de standaard poort voor een echo-service.

```
package sockets;

import java.net.*;
import java.io.*;

public class EchoServer extends Thread {

    private Socket incoming;

    public EchoServer(Socket incoming) {
        this.incoming = incoming;
    }

    public void run() {
        try {
            DataInputStream in =
               new DataInputStream(incoming.getInputStream());
            PrintStream out =
               new PrintStream(incoming.getOutputStream());
            boolean finished = false;
            while (!finished) {
                String text = in.readLine();
                if (text.equals("quit")) {
                    finished = true;
                    continue;
                }
                out.println(text);
            }
            incoming.close();
        } catch (Exception exc) {
            System.out.println(exc.toString());
        }
    }

    public static void main(String args[]) {
        try {
```

```
            ServerSocket serverSocket = new ServerSocket(7);
            while (true) {
                Socket incoming = serverSocket.accept();
                EchoServer e = new EchoServer(incoming);
                e.start();
            }
        } catch (Exception exc) {
            System.out.println(exc.toString());
        }
    }
}
```

Vervolgens een transcript van een test-run met telnet:

```
kvbuggen@bug:~$ java sockets.EchoServer &
kvbuggen@bug:~$ telnet localhost 7
Trying 127.0.0.1...
Connected to localhost.
Escape character is '^]'.
test
test
het werkt...
het werkt...
quit
Connection closed by foreign host.
kvbuggen@bug:~$
```

We kunnen besluiten dat sockets een te low-level communicatiemiddel zijn, de programmeur moet zelf teveel aandacht besteden aan implementatiedetails van het onderliggende communicatiesysteem, zoals lengte van pakketten bij UDP datatransmissie, beschikbare poortnummers, het formaat van de berichten die doorgezonden worden,...

Het is mogelijk om via sockets een groot en gecompliceerd gedistribueerd systeem te programmeren, net zoals het mogelijk is om gelijk welk programma in assembler te programmeren, maar het is een moeilijk en foutgevoelig proces. De volgende systemen bieden meer ondersteuning aan de programmeur door high-level communicatiemechanismen ter beschikking te stellen, die de socket-layer verbergen voor de programmeur. De meeste van deze systemen zullen zelf wel sockets en TCP of UDP communicatie gebruiken.

2.4.2. CORBA

De *Common Object Request Broker Architecture* (CORBA)[4] definieert een standaard voor remote method invocation, het oproepen van methods van objecten die zich op een andere machine bevinden. CORBA is bovendien language-onafhankelijk, wat wil zeggen dat het kan gebruikt worden om applicaties geschreven in verschillende programmeertalen transparant met elkaar te laten communiceren[2]. Een Java programma kan dus zonder problemen een method van een C++ klasse oproepen dankzij CORBA.

CORBA gebruikt een Interface Definition Language die toelaat de interface van een CORBA-object te beschrijven. De uiteindelijke implementatie kan dan ingevuld worden in een programmeertaal naar keuze, met een ORB naar keuze. Er zijn verschillende ORB's ter beschikking, maar de verschillende versies zijn niet rechtstreeks compatibel met elkaar. Servers en clients moesten dus gecodeerd worden met dezelfde ORB. Hiermee werd komaf gemaakt met de introductie van het Internet Inter-ORB Protocol (IIOP) als protocol voor alle communicatie tussen objecten en clients in elke ORB.

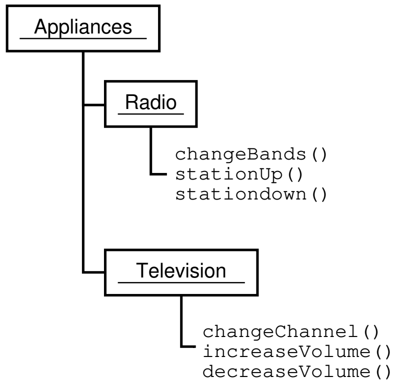
Voordelen van CORBA:

- Gedistribueerde Objecten: het is een high level communicatiesysteem, dat op niveau van objecten en methods werkt, zodat het mooi incorporeerbaar is in object oriented software design. Objecten op andere machines kunnen bijna volledig transparant opgeroepen worden, en de programmeur moet zelf niet meer de inhoud en opmaak van netwerk-pakketten bepalen.
- Platform- en Taalonafhankelijk: CORBA bestaat voor elke programmeertaal waarvoor een language mapping bestaat, en al deze systemen kunnen via CORBA met elkaar communiceren.
- Quality of Service: een van de handigheden aan high-level protocols is dat we automatisch een high-level QoS erbij krijgen. Inderdaad, wanneer we een remote method call doen, wordt ons gegarandeerd dat de oproep correct gebeurt, en we krijgen ofwel een return value terug, ofwel een exception.

CORBA stelt enkele speciale objecten ter beschikking van de programmeur, de *Common Object Services*, vergelijkbaar met class libraries in C++. Dit zijn objecten die handige services bieden voor gedistribueerde systemen, zoals bijvoorbeeld transactie objecten, event service objecten, relatie-objecten, en lifetime objecten.

Een van de meest bruikbare Common Object Services is de Naming Service. Dit is een service die ons toelaat namen te koppelen aan gedistribueerde objecten, vergelijkbaar aan de manier waarop DNS namen bindt aan IP adressen. De Naming Service bouwt een hierarchie van namen op waarin we onze objecten kunnen onderbrengen. In de hierarchie op figuur 2.1 kunnen we de `stationUp` method van het Radio object oproepen door `Applicances.Radio.stationUp()` uit te voeren (de exacte syntax om methods van CORBA objecten op te roepen hangt af van de gebruikte ORB). We kunnen on-the-fly eerder gelegde verwijzingen verwijderen of veranderen, en vermits de Common Object Services zelf ook CORBA objecten zijn, kunnen we een programma schrijven dat dit zelf ook kan.

[2] CORBA objecten kunnen geschreven worden in elke programmeertaal waarvoor een *language mapping* bestaat, hier komen we later nog op terug.



Figuur 2.1.: CORBA Naming Service

Type	Beschrijving	Java equivalent
long	geheel getal tussen -2^{31} en 2^{31}	int
short	geheel getal tussen -2^{15} en 2^{15}	short
float	IEEE single-precision floating point number	float
double	IEEE double-precision floating point number	double
char	char	char
boolean	booleaanse waarde	boolean
octet	byte	byte
string	sequentie van characters	java.lang.String
any	speciaal datatype, kan elk van bovenstaande zijn	Object

Tabel 2.1.: IDL Datatypes en hun Java equivalent

Wat betreft argumenten en return values hebben we de keuze tussen de eenvoudige datatypes uit tabel 2.1, en structured data types. Deze laatste zijn dezelfde structured types als we vinden in de programmeertaal C: structs, arrays, sequences, unions en enumerations.

Een struct is een manier om verschillende andere datatypes te groeperen, zo kunnen we bijvoorbeeld de volgende structuur maken:

```
struct AnsweringMachineMessage
{
  string dateStamp;
  string timeStamp;
  string message;
}
```

De IDL array heeft dezelfde eigenschappen als een C array: we moeten de grootte van de array vastleggen. Een sequence is een dynamische array, vergelijkbaar met een lijst uit Scheme. We kunnen een maximum grootte opgeven, maar dit is facultatief. Een union is een speciale vorm van struct, waar de types van de members kunnen verschillen naargelang het soort instance.

Enum is een datatype dat vergelijkbaar is met een constante array. Wanneer we `typedef enum _RadioBand { AM_BAND, FM_BAND } RadioBand` definiëren, zal elke variabele van het type `RadioBand` ofwel `AM_BAND`, ofwel `FM_BAND` als waarde hebben.

De interfaces die we definiëren in IDL zullen via een language mapping omgezet worden naar de programmeertaal waarin we deze interfaces zullen programmeren. Er zijn language mappings beschikbaar voor talen als Java, C++ en Smalltalk. In tabel 2.1 zien we de mapping tussen IDL datatypes en Java datatypes. Structs kunnen we mappen op klassen, en zowel arrays als sequences worden gemapt op Java arrays. Voor de overblijvende structured data types is het echter iets gecompliceerder: enumerated types en variabele klassen worden niet ondersteund door Java, dus om unions en enums te ondersteunen zijn speciale constructies toegevoegd.

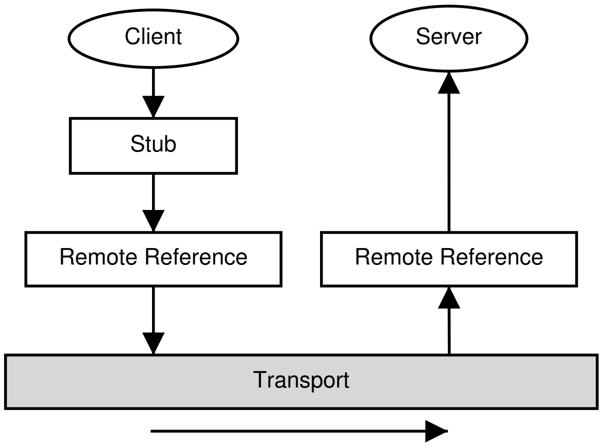
Een groot obstakel voor het gebruik van CORBA in Java Applets is de grootte van de CORBA klassen: zowel op de client als de server moet de volledige ORB beschikbaar zijn, dus als een andere ORB gebruikt werd dan de ORB die beschikbaar is in de browser van de client moet de volledige ORB gedownload worden, vermits een ORB ongeveer 400 klassen bevat is dit een aanzienlijke download.

2.4.3. Java RMI

Java Remote Method Invocation (RMI)[5, 6] is het kleine broertje van CORBA. Het gaat hier over een vereenvoudigde versie, die volledig in Java zelf geschreven is. Het voordeel hiervan is dat we geen aparte notatie nodig hebben om de remote interfaces te definiëren, het zijn gewone Java interfaces. Java RMI is veel eenvoudiger dan CORBA, onder andere de taalonafhankelijkheid is weg. RMI is dus enkel bruikbaar als zowel server als client in Java geschreven zijn. Ook is RMI minder krachtig, sommige features die CORBA wel aankan zijn in RMI nog niet mogelijk (zoals onder andere het automatisch laten starten van een remote object indien er nog geen actief zou zijn), maar er wordt nog actief ontwikkeld aan RMI, dus er zal in de toekomst nog functionaliteit bijkomen.

Een voordeel van de vereenvoudiging is dat RMI vriendelijker is in gebruik, en dat we voor het over het netwerk doorgeven van argumenten en return values niet moeten terugvallen op vertalingen van Java objecten en types naar een intermediair formaat, we kunnen gewoon gebruik maken van de object serialization die in Java zit en zo dus hele objecten als argumenten meegeven aan remote methods. Dit zorgt ervoor dat het onderscheid tussen lokale en remote objecten uit het standpunt van de programmeur volledig wegvalt.

RMI is opgebouwd uit 3 lagen, die elk een deel van de problematiek rond remote objects opvangen. Deze 3 lagen zijn (zie figuur 2.2):



Figuur 2.2.: Java RMI architectuur

- **Stub Layer**: op de client zal een stub object plaatsvervanger spelen voor het remote object, dat zich op de server bevindt. De stub is een implementatie van de remote interface, dus als we een remote method call willen uitvoeren zal eigenlijk een method van de stub opgeroepen worden. Deze gebruikt de onderliggende remote reference laag om de method call en argumenten naar de server te brengen. Daar aangekomen zal de method op het remote object uitgevoerd kunnen worden, en de return value wordt op dezelfde manier terug gezonden naar de client.

- **Remote Reference Layer**: deze laag zorgt voor het management van de referenties naar remote objecten. Deze laag bevat informatie over het type van remote object: unicast, multicast, self-activating servers,... Deze laag is ervoor verantwoordelijk dat de method calls terecht komen bij het juiste remote object, en zal de onderliggende transport layer gebruiken om te communiceren met de remote reference layer van de server.

- **Transport Layer**: de onderste laag in het model, deze laag is verantwoordelijk voor het opzetten van connecties tussen hosts, het doorgeven van de berichten, error control, het melden van eventuele netwerkproblemen (in de vorm van RemoteExceptions die naar de bovenliggende lagen geworpen worden)

De enige manier om een nieuwe reference naar een remote object te maken is door deze te krijgen van een ander remote object. We zien hier een mooi kip-en-het-ei-probleem, ware het niet dat er één remote object altijd beschikbaar is: `Naming`. Dit is een remote object met dezelfde taak als de Naming Service uit CORBA: het binden van namen aan remote objecten, zodat ze eenvoudig opvraagbaar zijn voor clients. De RMI versie is echter minder complex: we kunnen geen hierarchie opbouwen zoals in CORBA het geval was. Via `Naming.rebind(String name, Remote obj)` kunnen we een remote object registreren met naam name[3], `Naming.lookup(String name)` dient om een remote reference op te zoeken, en tenslotte kunnen we via `Naming.unbind(String name)` een remote object deregistreren.

We hebben reeds gezien er vanuit het standpunt van de programmeur geen verschil is tussen remote objecten en lokale objecten. Dit wil zeggen dat we de eigenschappen van lokale objecten ook terugvinden in de remote objecten. We zullen op de eigenschappen van lokale objecten relevant voor gedistribueerde systemen wat dieper ingaan.

Garbage Collection

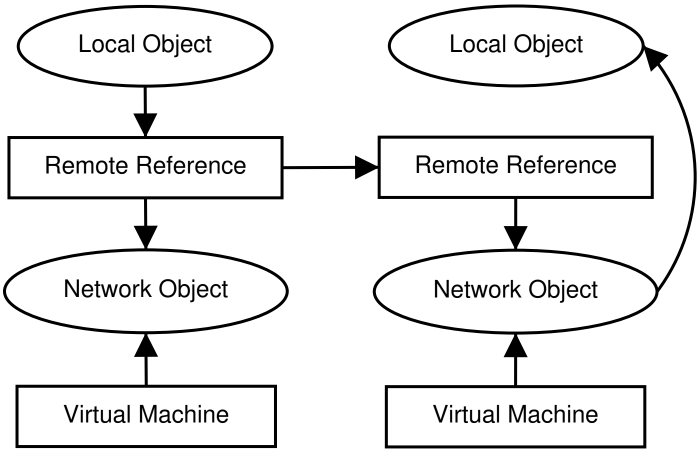
Een van de belangrijkste troeven van Java is garbage collection. Dit zorgt ervoor dat de programmeur geen aandacht moet besteden aan het alloceren en dealloceren van geheugen, wat zelfs zonder rekening te moeten houden met geheugen gealloceerd op remote machines al voor genoeg problemen zorgt.

Hoe zorgen we er nu voor dat een gedistribueerde pool geheugen op een min of meer betrouwbare manier kan worden gegarbage-collect? De mensen van Sun hebben hiervoor een garbage collection scheme opgesteld gebaseerd op het object reference counter systeem van netwerk objecten in Modula-3. Elk remote object krijgt een reference counter ingebouwd, en elke keer een nieuwe referentie met het object begint te communiceren wordt deze counter opgehoogd. Wanneer het remote object niet meer nodig is wordt de counter verlaagd. Zolang de counter groter is dan nul mag het object dus niet verdwijnen.

In de praktijk werkt het als volgt: aan de client-zijde zal de remote reference layer een netwerk-object maken voor elke remote method call van een lokaal object. Ook aan de server zijde wordt zo'n object gemaakt, dit netwerk object zal alle communicatie met het remote object regelen (en heeft dus ook een referentie ernaar). De gewone garbage collector van Java zal dus het remote object laten bestaan zolang het netwerk object er een referentie naar heeft.

Wanneer het lokale object het remote object niet meer nodig heeft, verwijdert de remote reference layer alle referenties naar het lokale netwerk object, dit zal in de volgende garbage collection cycle gedealloceerd worden, en in de `finalize` routine van het netwerk object wordt een boodschap verzonden naar het netwerk object aan de server kant, de reference counter wordt verlaagd, en als de counter op nul komt, zal het remote netwerk object haar referentie naar het remote object verwijderen. Het remote object zal dus verwijderd worden tijdens de volgende garbage collection cycle. Dit systeem geeft ons een robuuste bescherming tegen het te vroeg dealloceren van remote objecten. Indien er toch iets mis zou gaan, en we een method van een niet meer

[3] We kunnen `Naming.rebind(String name, Remote obj)` ook gebruiken wanneer we het object naar waar name verwijst willen updaten, bijvoorbeeld als we een nieuwe versie van het object gemaakt hebben. Vanaf de `Naming.rebind` oproep zal elke `Naming.lookup` een referentie geven naar het nieuwe remote object. Wanneer een client nog een remote reference had naar het oude remote object wordt dit echter niet automatisch upgedate.



Figuur 2.3.: Netwerk Objecten in RMI

bestaand remote object oproepen, zal er een `RemoteException` gethrowd worden in de lokale client. Een schematische voorstelling van de referenties tussen de verschillende soorten objecten vinden we in figuur 2.3.

We merken echter op dat vele remote objects nooit gedealloceerd zullen worden, om de eenvoudige reden dat er altijd een referentie zal bestaan naar het object, namelijk in het RMI registry. Zolang een remote object gebonden is in het registry, zal het niet gedealloceerd worden. We kunnen bindingen verwijderen, maar meestal gebeuren deze enkel als het programma beeindigd wordt, en dan wordt alle geheugen gebruikt door de virtual machine zowiezo gedealloceerd.

Platform Onafhankelijkheid

Java code is platform onafhankelijk. Eens gecompileerd, zal de code uitvoeren op elk systeem waarvoor een runtime environment voorhanden is. Vermits RMI ook maar gewoon Java code is, spreekt het voor zich dat RMI geen uitzondering is op deze regel.

Een van de voordelen van CORBA tegenover RMI is dat het niet enkel platform-onafhankelijk, maar ook taalonafhankelijk is. Hiertegenover staat dat RMI veel eenvoudiger in gebruik is, en we kunnen gebruik maken van de object serialization in Java om op een eenvoudige manier volledige objecten over het netwerk te transporteren. Dit laat ons toe niet enkel gegevens, maar ook gedrag te transporteren tussen client en server. Zoals we in een voorbeeld zullen zien (zie appendix A) is het niet moeilijk om een eenvoudig agent-systeem te programmeren via Java RMI.

Security

Wanneer we een applet starten, heeft deze applet toestemming om lokale objecten te maken. Echter, om begrijpelijke redenen, heeft de applet geen toestemming om nieuwe remote objecten te maken. RMI in applets is dus beperkt tot het oproepen van remote methods op remote objecten die al bestaan.

RMI servers moeten trouwens Java applications zijn. De servers kunnen niet embed worden in een webpagina. De belangrijkste reden waarom is dat het applet security mechanisme het verhinderd.

Werking

De manier waarop een object zichzelf als remote object profileert is door een van de verschillende remote object klassen te extenden. Op dit moment kunnen we kiezen tussen `UnicastRemoteObject` en `ActivatableObject`. Niet elke method in een remote object is remote oproepbaar, er kunnen ook methods voor lokaal gebruik enkel in de klasse zitten. De methods die remote oproepbaar zijn zetten we in een remote interface, het remote object implementeert deze interface. Clients krijgen enkel deze interface te zien. In een eenvoudig voorbeeld zullen we zien hoe we een remote twist kunnen geven aan het onsterfelijke Hello World! programma, voor een meer indrukwekkende demonstratie van de capaciteiten van RMI verwijzen we de lezer naar appendix A voor een gedistribueerde versie van de faculteit functie, die meteen ook demonstreert hoe een remote server tegelijk een remote client kan zijn, en dat elke remote call in een eigen thread uitvoert.

Eerst definiëren we de remote interface, we zullen één remote method hebben, `hello`, die een String als return value heeft. Deze method wordt opgeroepen door de client, en de teruggegeven String zal op het scherm geprint worden.

```
package rmi;

import java.rmi.*;

public interface HelloServer extends Remote {
    String hello() throws RemoteException;
}
```

Elke method uit de remote interface moet de RemoteException exception throwen, de client die de remote method oproept is verantwoordelijk voor het catchen en afhandelen van deze exception. Via deze exception worden connectieproblemen met de remote server aan de client duidelijk gemaakt.

De implementatie van de remote server gaat als volgt:

```
package rmi;

import java.rmi.*;
```

```java
import java.rmi.server.*;

public class HelloImpl extends UnicastRemoteObject
implements HelloServer {
    HelloImpl() throws RemoteException {
        super();
    }

    public String hello() {
        return ("Hello world!");
    }

    public static void main(String args[]) {
        try {
            HelloImpl blort = new HelloImpl();
            Naming.rebind("blort", blort);
            System.out.println("Server started.");
        } catch (Exception e) {System.out.println(e.toString());}
    }
}
```

In de constructor van het remote object roepen we de constructor van de superklasse op, deze kan eventueel een RemoteException throwen, dus de constructor van het remote object gaat deze gewoon doorgeven. Wanneer een nieuw remote object aangemaakt wordt moet deze exception opgevangen en correct afgehandeld worden.

De server is verantwoordelijk voor het invullen van de implementatie van de remote methods, de return value van de remote method zal automatisch terug naar de client getransporteerd worden. In dit voorbeeld zal de String "Hello World!" naar de client gestuurd worden.

In de main method wordt het remote object geinstancieerd, en via de `Naming.rebind(...)` oproep wordt de naam "blort" gekoppeld aan het remote object.

Vervolgens de remote client:

```java
package rmi;

import java.rmi.*;

public class HelloClient {
    public static void main(String args[]) {
        try {
            HelloServer srv = (HelloServer) Naming.lookup("blort"
                );
            System.out.println("Server contacted.");
            System.out.println(srv.hello());
        } catch (Exception e) {
            System.out.println("Problem: "+e.toString());
        }
    }
}
```

Via `Naming.lookup(...)` kunnen we het object gebonden aan de String “blort” opzoeken. Vermits we een object terugkrijgen van het type `Remote`, moet dit eerst getypecast worden naar de correcte remote interface (`HelloServer` in ons voorbeeld) voor we remote methods kunnen oproepen.

Vervolgens kunnen we op het bekomen remote object gewoon methods oproepen zoals we zouden doen bij een lokaal object. Het enige verschil: bij elke remote method call kan een `RemoteException` opgeworpen worden, dus moeten we deze exception opvangen.

Zoals eerder gezegd werkt RMI via een gedistribueerde garbage collector, dus we moeten zelf geen actie ondernemen om een remote object correct te laten verdwijnen, als het object niet meer nodig is zal de onderliggende virtual machine zelf de nodige acties ondernemen om het remote object te de-alloceren.

Dit eenvoudige voorbeeld was ruim voldoende om de basistechniek van RMI te demonstreren, voor meer indrukwekkende voorbeelden verwijzen we de lezer naar appendix A, waar we de implementatie zullen uitwerken van een gedistribueerd faculteit-algoritme, en een eenvoudige agent-server.

2.5. Conclusie

In dit hoofdstuk hebben we gezien welke zaken belangrijk zijn bij gedistribueerd programmeren, onder andere dat gedistribueerde programmeren een programma opdeelt in componenten die zelf volledig functioneel zijn, niet zoals objecten die enkel passief gebruikt kunnen worden. Verder moeten we aandacht besteden aan het opstellen van een gepast protocol voor elke applicatie, en gedistribueerde programmas voeren een nieuw soort fouten in: gedeeltelijk falen van software door uitvallen of verkeerd werken van agents.

We hebben verschillende voorbeelden van gedistribueerde programmas gepresenteerd, en de verschillende systemen waarin de voorbeelden gemaakt zijn hebben we verder geanalyseerd.

In het volgende hoofdstuk zullen we de zien hoe de interactie van concurrent lopende agenten voor problemen kunnen zorgen, en hoe we deze problemen kunnen oplossen.

3. Concurrentie

In dit hoofdstuk behandelen we een van de gevolgen van concurrent lopende agenten, namelijk concurrentieproblemen. We beginnen door te verklaren hoe ze exact veroorzaakt worden, en presenteren vervolgens een aantal manieren om agenten te synchroniseren. We zullen semaforen en monitoren zoals gedefiniëerd in [11], en transactiesystemen zoals gepresenteerd in [14].

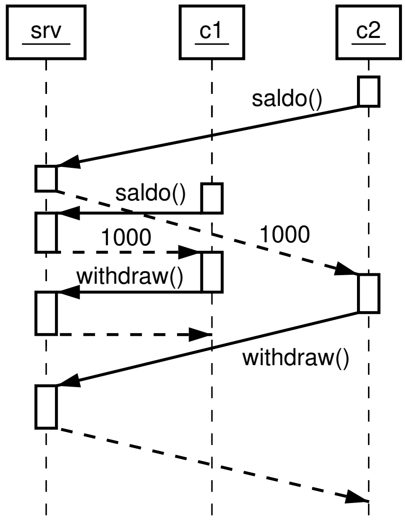
3.1. Race-Conditions

De verschillende agenten in een gedistribueerd systeem werken allemaal concurrent. Dit heeft serieuze implicaties wanneer deze agenten werken op shared data: we moeten dan rekening houden met concurrentieproblemen. Bij wijze van voorbeeld kunnen we een bank-agent beschouwen. De agent heeft twee operaties: saldo() en withdraw(), welke het saldo van een rekening kunnen opvragen en geld van de rekening opvragen. Er zijn twee client-agenten, die beide 500 euro willen opvragen indien het saldo van de rekening hoger is dan 1000 euro. De tweede client-agent (c2) heeft te kampen met zware netwerk-lag, terwijl de eerste client-agent (c1) een vlotte verbinding heeft. Beide agenten gebruiken de volgende code om hun opdracht uit te voeren:

```
if (srv.saldo() > 1000)
{
   src.withdraw(500);
}
```

Stel dat de oproepen van beide agenten verlopen zoals in het sequence diagram in figuur 3.1, we zien dat door de trage verbinding tussen de server en c2 het antwoord op de saldo() aanroep pas aankomt nadat c1 zijn opdracht al uitgevoerd heeft. Normaal gezien zou c2 dus geen geld meer mogen opvragen, vermits het saldo lager staat dan 1000 euro. De controle die c2 uitvoerde beweert echter dat alles in orde is, en ook c2 zal withdraw(500) uitvoeren.

Dit probleem noemen we een race condition: het resultaat van het programma hangt af van de volgorde waarin de agenten instructies uitvoeren. Meestal is het zo dat slechts een van de mogelijke resultaten een juist en consistent resultaat oplevert. We moeten



Figuur 3.1.: Concurrente agenten

dus een manier vinden om ervoor te zorgen dat de andere resultaten vermeden worden. De oorzaak ligt in de asynchrone aard van de processen, het ene proces heeft geen idee van de status (of het bestaan) van de andere processen. Om dit op te lossen moeten we ervoor zorgen dat de agenten gaan samenwerken op een bepaald niveau om hun acties op shared data te coordineren. De agenten moeten hun acties synchroniseren.

De meest gebruikte vorm van synchronisatie is locking: resources (variabelen, objecten, agenten,...) kunnen gereserveerd worden door een agent, andere agenten moeten wachten tot de lock vrijgegeven wordt. Er zijn twee soorten locks: read locks en write locks. Meerdere agenten kunnen tegelijk een read lock nemen op een resource, maar wanneer een agent een write lock aanvraagt, mag geen enkele andere agent reeds een read lock hebben op de resource. Indien dit wel het geval is moet de agent wachten tot de lock vrijgegeven wordt. Ook mag slechts één agent tegelijk een write lock op een resource nemen.

3.2. Synchronisatie

De eerste gebruikte methoden om concurrentieproblemen uit de weg te ruimen werden ontleend aan het onderzoeksgebied van de parallelle systemen: systemen waar gebruik gemaakt wordt van meerdere processoren, of gewoon meerdere processen op een processor. Deze systemen zijn erg gelijkaardig aan gedistribueerde systemen, het enige verschil is de latency: bij parallelle systemen kunnen de verschillende processen zeer snel gegevens uitwisselen, zo snel dat de delay verwaarloosbaar is. Bij gedistribueerde systemen (en zeker systemen die werken op een Wide Area Network zoals het Internet) is de latency helemaal niet waarloosbaar, deze kan soms oplopen tot meerdere seconden.

We zullen zien dat de eenvoudigere synchronisatiemechanismen minder geschikt zijn voor grote gedistribueerde systemen, zeker als we met object georienteerde gedistribueerde systemen beginnen te werken.

3.3. Concurrent Programmeren

We weten nu dat concurrente programmas die werken met shared resources inconsistentie van gegevens kunnen veroorzaken als ze niet correct gesynchroniseerd worden. We zullen in deze sectie enkele verschillende synchronisatiemechanismen bespreken, beginnend van eenvoudige mechanismen zoals semaforen tot complexe mechanismen als transactiesystemen.

3.3.1. Semaforen

Een *semafoor* zoals geïntroduceerd door Dijkstra[13] is een integer variabele die enkel positieve waarden kan aannemen. Er zijn twee operaties die de waarde van de semafoor kunnen veranderen:

- wait(s): als s > 0 dan s := s - 1, anders wordt het process dat wait(s) aanroepte gesuspend
- signal(s): als er een process P bestaat dat suspended is, wordt dit terug geactiveerd. Anders s := s + 1.

Semaforen worden gebruikt om kritische secties te beschermen. Een kritische sectie is een stuk code dat maar door een maximum aantal agenten tegelijk mag uitgevoerd worden. In de meeste gevallen zal een kritische sectie slechts door één agent tegelijk mogen uitgevoerd worden. De semafoor met beginwaarde 1 noemen we een *binaire semafoor*.

Het is duidelijk dat in we grote programmas snel de verschillende semaforen uit het oog kunnen verliezen. Wanneer we bijvoorbeeld via semaforen een gedistribueerde database moeten locken elke keer we een actie uitvoeren is de kans groot dat we ergens in de code vergeten een gelockte semafoor terug vrij te geven (door signal uit te voeren). Dit soort fouten is echter moeilijk op te sporen.

Hier zien we ook een extra probleem ontstaan: veronderstel dat er twee agenten zijn, a en b, met volgende code:

```
# agent a:
wait(n);
...
wait(m);
...
signal(m);
signal(n);

# agent b:
wait(m);
...
wait(n);
...
signal(n);
signal(m);
```

De kans bestaat dat agent a semafoor n locked, en dat agent b semafoor m locked. Indien dit gebeurt vallen beide agenten stil. Deze situatie noemt men een *deadlock*. Het wordt zelfs nog erger: beide agenten hebben al minstens 1 lock, die ze nooit meer zullen vrijgeven, dus die resources worden onbruikbaar, en wanneer andere agenten locks willen nemen op deze resources zullen ook die agenten stilvallen. Indien er genoeg tijd verloopt zal het hele systeem stilvallen.

Er zijn twee oplossingen voor dit probleem:

1. **Deadlock Detection**: ervoor zorgen dat agenten merken dat ze in een mogelijke deadlock-situatie zitten, en ze de mogelijkheid geven er iets aan te doen. Dit wordt meestal gedaan door het timeout-rollback-mechanisme: wanneer een agent langer dan een bepaalde periode moet wachten op een lock voor een resource, wordt aangenomen dat er een deadlock is, de agent zal al zijn reeds gekregen locks vrijgeven, en na een korte wachtperiode (om ervoor te zorgen dat andere agenten eventueel locks kunnen krijgen op resources die door hem gereserveerd waren) zal de agent opnieuw beginnen aan zijn opdracht. We zitten echter nog met een probleem: het kan zijn dat onze agent tussen het aanvragen van zijn eerste lock, en het detecteren van de deadlock reeds operaties uitgevoerd heeft, de effecten van deze operaties moeten ongedaan gemaakt worden voor de agent mag herbeginnen, dit is het rollback-deel. Het kan zelfs zijn dat de agent methods van andere agenten opgeroepen heeft, ook de effecten van die method calls moeten ongedaan gemaakt worden.

   Dit kan echter vermeden worden door agenten te verplichten alle locks die ze nodig hebben om hun opdracht te vervullen op hetzelfde moment te nemen, in een enkele niet-onderbreekbare opdracht, voor er operaties uitgevoerd zijn. Rollbacks zijn nu overbodig, vermits er geen operaties uitgevoerd worden voor de agent alle locks verkregen heeft, en dan kan er zich ook geen deadlock meer voordoen. Dit zorgt er wel voor dat we resources “te lang” reserveren: de lock wordt aangevraagd helemaal in het begin van de method, en wordt pas opnieuw released als al het werk gedaan is.

2. **Deadlock Avoidance**: in plaats van de deadlock eerst te laten gebeuren, en dan het systeem te herstellen, zal deze techniek het aanvragen van locks bestuderen, en ervoor zorgen dat een agent nooit een lock kan krijgen indien dit een deadlock kan veroorzaken. Dit kunnen we ofwel centraal regelen, ofwel door elke agent afzonderlijk laten doen. Elke methode heeft nadelen:

   - *Centrale locking* is niet schaalbaar: wanneer we de locking van een zeer groot systeem moeten laten doen door een enkel systeem zal er ten eerste veel netwerk overhead zijn, en ten tweede, als een deel van het netwerk afgezonderd wordt van de rest werkt het systeem niet meer.

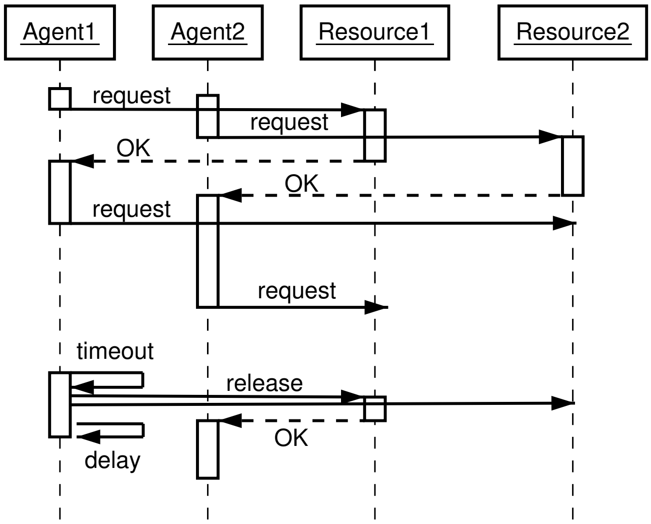
   - *Gedistribueerde locking* is zeer complex om te implementeren, vermits de agenten constant moeten synchroniseren bij het nemen van een lock.

Deadlock detection bij semaforen kan niet gebeuren door de clients zelf, vermits deze suspended zijn. Een deadlock avoidance mechanisme, gemaakt door de programmeur, is dus de enige oplossing.

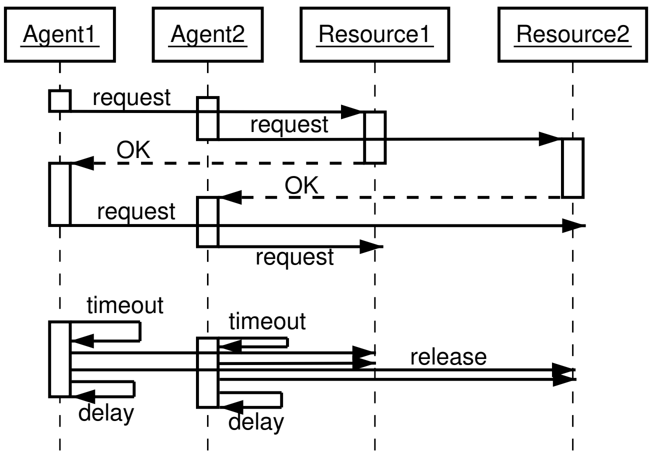
Alsof deadlocks nog niet voor genoeg problemen zorgen, kunnen we ook nog eens last krijgen van een neveneffect van deadlock resolution systemen: het kan zijn dat de agenten na het herstarten opnieuw in dezelfde deadlock terecht komen, en dit blijven herhalen. De agenten zijn technisch gezien niet deadlocked, maar ze krijgen ook geen instructies uitgevoerd, we noemen dit een **livelock**.

In de meeste gevallen zal een deadlock detection systeem werken zoals in figuur 3.2: wanneer we gedeadlocked zijn (een timeout deed zich voor in een agent), zal deze al zijn reeds verkregen en aangevraagde locks releasen, even wachten, en terug locks aanvragen. Intussen zal de agent die mede verantwoordelijk was voor het veroorzaken van de deadlock locks gekregen hebben op de net vrijgekomen resources, en deze agent kan zijn operaties uitvoeren.

In een bepaalde situatie geillustreerd door figuur 3.3 zien we hoe dit problemen kan veroorzaken: als beide agenten kort na elkaar hun locks aanvraagden, kan het zijn dat ze tegelijk (of ongeveer tegelijk) timeouten, locks vrijgeven, delayen, en opnieuw locks aanvragen. Dit zorgt ervoor dat dezelfde deadlock situatie zich zal voordoen, het deadlock detection systeem veroorzaakt hier een livelock.



Figuur 3.2.: Deadlock detection



Vermits beide agents op hetzelfde moment de delay
beginnen, zullen ze ook tegelijk terug voortwerken, en
zal dit patroon herhaald worden.
De agents zijn livelocked.

Figuur 3.3.: Livelock loop

De oplossing voor dit probleem is met random delays te werken: wanneer een agent een deadlock detecteert, zal hij net zoals bij gewone deadlock detection alle reeds verkregen locks vrijgeven, maar alvorens opnieuw te starten zal hij eerst gedurende een random gekozen periode wachten. Het gevolg is dat de kans zeer groot is dat een van beide agenten langer moet wachten dan de andere agent, en intussen een agent al zijn locks verkregen, en dus de deadlock-situatie vermeden is. De selectie van de bovengrens van de timeout moet goed overwogen zijn, vermits een te korte bovengrens ervoor zorgt dat de kans dat beide agenten even lang wachten groter wordt, en te lange delay zorgt ervoor dat processing tijd verloren gaat.

Een goed systeem om de goede bovengrens automatisch te laten afleiden is het volgende: wanneer we na een livelock opnieuw in een livelock situatie zitten, de bovengrens van de random delay verdubbelen.

Gedistribueerde systemen over WAN-netwerken zijn zeer gevoelig aan livelocking problemen, vermits het soms lange tijd kan duren voor we antwoord krijgen van onze communicatiepartner. Hierdoor zal het voor agenten lang duren alle locks te verkrijgen. Als we nu twee agenten A en B hebben die elk de resources r1, r2, r3 en r4 willen locken. Agent A begint van links naar rechts te locken, terwijl agent B van rechts naar links locked. Vermits we lange delaytijden hebben op de communicatie is de kans groot dat geen van beide agenten alle locks kan krijgen zonder dat de andere agent ook al een lock heeft. De agenten zullen dus alletwee hun locks terug vrijgeven en het boeltje opnieuw proberen. Dit probleem kan eenvoudig opgelost worden door een vaste locking-volgorde in de resources in te bouwen. Wanneer natuurlijk één enkele agent niet braaf meespeelt kan dit de werking van het hele systeem in gevaar brengen.

3.3.2. Monitors

Een monitor is een synchronisatiestructuur die we kunnen gebruiken wanneer de programmeertaal de notie van datastructuren heeft. Wat we gaan doen is de toegang tot deze structuren beperken. Bijvoorbeeld op een queue datastructuur (meestal geïmplementeerd als een vector), kunnen we slechts enkele operaties toelaten: `empty?`, `enqueue` en `dequeue`. Random access toelaten op de queue (ook al is de onderliggende structuur een gewone vector) zou tot onregelmatig gedrag kunnen leiden. We kunnen dus een monitor definieren, die voorgenoemde methods implementeerd. Alle processen hebben toegang tot deze monitor, maar de monitor zorgt ervoor dat slechts één van zijn methods tegelijk uitgevoerd wordt.

Java thread synchronisatie (via het `synchronized` keywoord) gebeurt via monitors. Elk Java object met minstens een `synchronized` method krijgt een monitor object toegekend, en wanneer een andere thread de gesynchroniseerde method probeert op te roepen zal het monitor object gelocked worden, zodat geen andere thread die gesynchroniseerde method kan oproepen in dat object.

3.3.3. Rendez-vous

Rendez-vous is een synchronisatiemechanisme voor message-based gedistribueerde systemen. Een object kan, via *"accept"* statements, een aantal "guards" instellen, wanneer er een boodschap binnenkomt die voldoet aan een guard wordt de bijhorende method uitgevoerd. De communicatie gebeurt synchroon, dus het object dat eerst de rendez-vous initieert wordt geblokkeerd tot het andere object de rendez-vous ook uitvoert. Tijdens de rendez-vous kunnen beide objecten gegevens uitwisselen aan elkaar.

Omdat blokkeren en wachten op een specifieke boodschap de mogelijkheden van het systeem te sterk zou beperken is er een constructie ingevoerd, *"select"* om ervoor te zorgen dat een agent kan wachten op een aantal boodschappen.

Dit systeem levert een veel krachtiger synchronisatiemechanisme dan semaforen en monitoren. Het enige probleem is dat alle communicatie tussen agenten sychroon gebeurt. Als dus een van de agenten uitvalt, zal uiteindelijk het hele systeem doodvallen. In kleinschalige gedistribueerde systemen is dit een oplosbaar probleem, maar indien we op een groot gedistribueerd netwerk werken is het allemaal niet meer zou eenvoudig. In dat geval is synchrone communicatie zelfs eerder een nadeel vanwege de lange delays tijdens netwerk-communicatie.

3.3.4. Transacties

Soms is het nodig om verschillende afzonderlijke acties te laten doorgaans als een geheel. Bijvoorbeeld het query-en en updaten van een database. We willen niet dat tussen het uitlezen en schrijven van gegevens opeens een andere agent ook in de database gegevens begint te wijzigen. Dit kunnen we verwezelijken via een transactiesysteem. In een transactiesysteem zal een artbitraire client een reeks operaties uitvoeren op een aantal abitraire resource agenten, zonder interferentie van andere agenten. De client agent zal dus tijdens de transactie locks hebben op de resource agenten. Dit locken kan gebeuren op twee manieren: *two phased locking*, waarin de client in het begin van de transactie alle benodigde resource agenten locked, en *incremental locking*, waar de resource agenten gelocked worden wanneer ze nodig zijn.

In de eerste situatie is het mogelijk deadlocks te vermijden door een vaste ordening in het aanvragen van de locks in te bouwen, maar doordat we alle locks nemen in het begin van de transactie houden we een aantal resource agent "te lang" gelocked. Het alternatief is echter nog complexer: wanneer we tijdens het uitvoeren van een transactie merken dat we een lock niet kunnen nemen, kunnen we het niet veroorloven van te wachten tot die lock vrijkomt, vermits dit tot deadlocks kan leiden (vergeet niet dat we reeds een aantal locks hebben). De transactie moet dus gerollbacked worden. Dit wil zeggen dat we alle reeds uitgevoerde operaties op resource agenten ongedaan moeten maken.

De meest eenvoudige methode om rollbacks te ondersteunen is tijdens het locken van de resource agent een copie van de state van de agent te nemen, en indien we moeten rollbacken deze oude state gewoon over de huidige state terugzetten. Dit is echter niet altijd mogelijk, bijvoorbeeld wanneer we operaties uitvoerden naar een database-backend. In dat geval moet ofwel de backend zelf rollbacks ondersteunen, ofwel moeten we van elke uitgevoerde operatie de inverse bijhouden om ze ongedaan te kunnen maken.

Wanneer de client de transactie beëindigd door ze te committen kunnen we de backup-state weggooien, de resouce agenten kunnen nu opnieuw gelocked worden door andere transacties.

Om transacties uniek te kunnen identificeren hebben we een transactie ID nodig. Deze ID moet uniek zijn over het hele transactiesysteem, dus wordt meestal toegekend vanaf een centrale agent, de transactieserver. Het bijhouden van de resource agenten die tot een transactie toegetreden zijn kunnen we op twee plaatsen bijhouden: op de centrale transactieserver, of op de resource agent zelf. De meest eenvoudige situatie is op de centrale server, maar wanneer de server uitvalt verliezen we alle informatie over de locks, dus moeten alle reeds gestarte transacties geabort worden. Indien locks zich op de resource agenten bevinden wordt het locking van de resources ingewikkelder, maar dit zorgt ervoor dat de transactieserver enkel nodig is om een uniek transactie ID te krijgen. Wanneer de transactieserver uitvalt kunnen alle reeds gestarte transacties nog altijd correct afgewerkt worden.

Het gebruik van een transactiesysteem is een heel generische manier om te verzekeren dat een aantal operaties uitgevoerd zullen worden als een atomisch geheel, maar hebben natuurlijk ook hun nadelen. Liefst zouden we een incrementele resouce locking strategie gebruiken, maar dit zorgt ervoor dat alle resource agenten moeten kunnen rollbacken, en incrementele locking is nog steeds kwetsbaar voor deadlocks. Anderzijds zijn incrementele systemen kwetsbaar voor livelocks: wanneer we lange transacties met veel resource agenten hebben is de kans dat twee transacties livelocken groot. Deze livelocks kunnen spijtig genoeg niet eenvoudig vermeden worden. De enige mogelijke strategie is random delays met groeiende bovengrens te gebruiken bij elke rollback. Dit zorgt er wel voor dat het uitvoeren van de transacties nog langer duurt.

3.4. Conclusie

In dit hoofdstuk hebben we het gehad over concurrentieproblemen. We zijn begonnen met in te leiden wat ze zijn, en hoe ze veroorzaakt worden door het gebruik van shared resources door concurrente agenten. We hebben een aantal manieren gezien om agenten te synchroniseren, de meeste gebruiken een vorm van resource locking. Hierbij hebben we gezien wat de gevaren zijn van deadlocks, en hoe het vermijden van deadlocks kan leiden tot livelocks.

Van de studie van enkele synchronisatiemechanismen kunnen we afleiden dat elk systeem zijn speciefiek specialisatieveld heeft, en wanneer we met het systeem iets willen bereiken dat ver buiten dit veld ligt zullen we veel moeilijkheden hebben. Het gedrag van een transactiesysteem implementeren via rendez-vous zal bijvoorbeeld enorm complex zijn.

4. Meta-Level Concurrentie

4.1. Inleiding

Dit hoofdstuk presenteerd het ontwerp van het guarding systeem dat we in het volgende hoofdstuk zullen implementeren. We zullen vertrekken van de notie van Aspect Oriented Programming, en aanpassingen hieraan voorstellen tot we een systeem krijgen dat aan onze eisen voldoet. De belangrijkste eis is dat het systeem met zo weinig mogelijk primitieven een zo flexibel mogelijk synchronisatiemechanisme aanbiedt.

4.2. Aspect Oriented Programming

Bij het maken van grote programmas tracht een programmeur het probleem op te splitsen in verschillende deelproblemen. Deze splitsing blijven we herhalen tot we een heel aantal basisproblemen, de *concerns*, gedefiniëerd hebben. In OO-talen worden deze concerns op objecten gemapt. Soms is het echter niet mogelijk een concern op een enkel object te mappen. Nemen we bijvoorbeeld thread synchronisatie in Java, dit kunnen we niet in een enkel object implementeren, het weeft zich als het ware tussen de andere objecten door.

Aspect Oriented Programming (AOP)[9] laat toe deze concerns op een meta-niveau te beschrijven. De hierdoor gevormde aspecten worden door een codeweaver door de base-code gewoven.

In gedistribueerde systemen is concurrentie een heel belangrijk en complex aspect. Het spreekt dus voor zich dat we om concurrentieproblemen aan te pakken naar AOP achtige systemen kijken. Puur AOP is echter niet geschikt voor toepassing in gedistribueerde agent-systemen, vermits de codeweaver er zich op baseert dat we alle agent-code voorhanden hebben. Dit is echter niet altijd het geval, zeker niet wanneer we met grote gedistribueerde systemen, zoals bijvoorbeeld WAN netwerken, werken. Dan zullen we zelfs meestal met agenten moeten samenwerken die geschreven zijn door andere partijen, zodat we geen toegang hebben tot de source-code van die agent.

Bovendien limiteert AOP zich tot object georiënteerde talen, niet alle multi-agent systemen zijn object georiënteerd.

4.3. Meta-Level Concurrentie

Wat we voorstellen is een guarding systeem te gebruiken om boodschappen verzonden tussen agenten te onderscheppen, en afhankelijk van regels gedefiniëerd door de programmeur boodschappen tijdelijk te blokkeren.

Vermits we boodschappen gaan matchen tegen regels zullen we een sterk unificatie-algoritme nodig hebben. We zullen daarom de guarding rules in Prolog implementeren. Logische programmeertalen hebben het extra voordeel dat ze uitermate geschikt zijn om over onderwerpen zoals synchronisatie tussen agenten te redeneren.

We implementeren dus het guarding systeem op meta-niveau boven de agenten, zoals geillustreerd in figuur x. In tegenstelling tot AOP zullen we echter geen code genereren om te weaven in de onderliggende agenten, we blijven op het meta-niveau. Dit heeft tot gevolg dat we ten allen tijde de guarding regels kunnen uitbreiden, of de volledige set vervangen door een andere set regels, zonder dat de onderliggende agent hier weet van hoeft te hebben. Wat hem betreft blijft alles bij het oude. Dit draagt toe aan de flexibiliteit van het systeem, hetgeen een van onze belangrijkste eisen was.

Dankzij dit guarding systeem is het niet nodig alle primitieve te voorzien voor elke mogelijke synchronisatiemethode, in plaats hiervan kunnen we gewoon de guarding regels van het benodigde synchronisatiemechanisme in het guarding systeem laden. Wanneer we de primitieven voorstellen die we zullen gebruiken om guarding regels te definieren, zullen we als eerste validatie de guarding regels opstellen die enkele bekende synchronisatiemechanismen uitdrukken.

Om veiligheidsredenen zal elke agent zijn eigen meta-guard krijgen. Indien we één meta-guard per distributed node (een distributed node is een “plaats” waar gedistribueerde agenten zich kunnen bevinden) zouden gebruiken, kunnen de agenten elkaars geblokkeerde boodschappen uitlezen en zelfs wijzigen.

We zullen geen volledige integratie van de meta-guards met een bestaand multi-agent systeem presenteren, vermits de nadruk van het werk niet ligt op deze integratie, maar op het ontwerp van geschikte primitieven om aan meta-guarding te kunnen doen. De implemenatie van de primitieven is dus nu ontworpen om de interactie met het guarding systeem via een Prolog console te doen. In een integratie met een multi-agent systeem moet de implementatie van de primitieven aangepast worden.

Elk guarding mechanisme heeft een queue van delayed messages. Dit zijn boodschappen die door het systeem tegengehouden worden. Bij elke nieuwe boodschap die in het systeem passeert worden alle delayed messages gechecked om te zien of de nieuwe status van de agent toelaat bepaalde boodschappen die eerder niet door mochten nu wel te laten passeren. Dit controleren gebeurt door een oproep aan het handlemessage-predikaat, dit wordt ingevuld door de programmeur. Deze kan dus bepalen aan welke criteria een boodschap moet voldoen om doorgelaten te worden. Op deze manier kan het synchronisatiemechanisme volledig afgesteld worden op de noden van de agenten. We zullen zien hoe we op een eenvoudige manier synchronisatiemechanismen zoals semaforen, rendez-vous, en zelfs hogere orde mechanismen zoals transacties kunnen implementeren.

4.4. Primitieven

We stellen boodschappen die agenten naar elkaar zenden voor als vectoren van drie elementen: de zender, de ontvanger, en de inhoud. De inhoud van een boodschap is op zijn beurt een vector, op de inhoud van deze vector zijn geen beperkingen opgelegd. Een boodschap van agent a naar agent b met inhoud “appelboom” zou er dus als volgt uitzien: `[a, b, [appelboom]]`.

Voor de implementatie van de queue gebruiken we een feature van Prolog: wanneer we nieuwe feiten asserten via de assertz oproep, komen deze feiten achter alle andere feiten met hetzelfde predikaat terecht. Wanneer we opzoekingen doen in het geheugen van Prolog zullen we dus steeds de eerst toegevoegde feiten kunnen opvragen. Delayed messages delen hetzelfde predikaat, namelijk `delayedmessage`.

Het prolog guarding systeem biedt de programmeur de volgende primitieven:

- **Sendmessage**: via deze primitieve komen boodschappen het systeem binnen. Zowel berichten van de agent naar andere agenten, als berichten ontvangen van andere agenten worden door het guarding systeem onderzocht. in deze primitieve wordt de te behandelen boodschap doorgegeven aan het `handlemessage` predikaat, hetgeen wordt ingevuld door de programmeur. Na het afhandelen van de boodschap door handlemessage zal het systeem alle delayed messages opnieuw checken (dit gebeurt in de checkwaitingmessages oproep). Dit is het beste moment om de delayed messages te controleren, vermits na het verzenden (of ontvangen) van een boodschap de concurrentiestatus van de agent zich in een significant andere staat kan bevinden. Voorbeeld: op een server agent is een resource gelocked door agent A. Agent B stuurt een lock request naar de server, deze boodschap wordt onderschept door het guarding systeem en wordt in de delayed queue gezet. Wanneer agent A aan de server mededeelt dat hij de lock niet meer nodig heeft, verandert de staat van de server waardoor agent B nu zijn lock wel kan krijgen. Onmiddelijk na het behandelen van de unlock message van agent A worden de delayed messages van de server opnieuw gecontroleerd, en nu mag de boodschap van agent B wel door.

  Het oproepen van de sendmessage rule uit het onderliggend agent systeem wordt gesimuleerd door manueel sendmessage oproepen in een prolog console in te geven.

- **Delaymessage**: primitieve die wordt opgeroepen wanneer het systeem ontdekt dat een boodschap nog niet verzonden mag worden. De boodschap in kwestie wordt aan de queue toegevoegd door `assertz(waitingmessage(Msg))` uit te voeren.

- **Dropmessage:** deze primitieve kan gebruikt worden wanneer we een boodschap willen “vergeten”. De boodschap in kwestie verdwijnt uit het systeem, en zal dus ook nooit door een agent ontvangen worden. Wanneer we een boodschap willen laten verdwijnen hebben we deze primitieve eigenlijk niet nodig, we bekomen hetzelfde effect door gewoon noch delaymessage, noch receivemessage op te roepen. We kunnen deze primitieve echter wel gebruiken wanneer we de zender van het bericht op de hoogte willen brengen dat zijn verzonden boodschap “vermoord” is.

- **Receivemessage**: de primitieve die zal opgeroepen worden wanneer het systeem beslist dat een boodschap verzonden mag worden. De primitieve is verantwoordelijk voor het doorgeven van de boodschap in kwestie aan de communicatielayer. In deze simulatie wordt het clearen van een boodschap voor verzending nagedaan door een mededeling op het scherm te zetten. Wat met de boodschap gebeurt na het verlaten van het prolog systeem hangt af van de bestemmeling van de boodschap. We onderscheiden twee gevallen:

    1. de bestemmeling is de agent zelf: het bericht wordt doorgegeven aan de onderliggende agent.

    2. de bestemmeling is een andere agent: het bericht zal aan de communicatielayer doorgegeven worden, deze zal het bericht versturen naar de bestemmeling. Daar aangekomen zal de boodschap eerst door het guardingsysteem daar aanwezig moeten verwerkt worden voor het bij de agent zelf uitkomt.

Voorts zijn er nog twee interne primitieven:

1. **Checkwaitingmessages**: deze rule zal voor elke delayed message controleren of ze nu wel verstuurd mag worden. Het proces wordt beëindigd wanneer we de speciale message tegenkomen die we voor het processen van de queue achteraan toegevoegd hebben.

2. **Checkthread**: de goal van een Prolog thread die om de 20 seconden de queue controleert. Om concurrentieprobemen tussen deze thread en de hoofd-thread (de thread waarin nieuwe boodschappen binnenkomen) te vermijden is het nodig het matchen van een boodschap met de guarding regels als kritische sectie te beschouwen, zodat maximum één thread tegelijk hiermee kan bezig zijn. Dit doen we door een Prolog mutex te locken in sendmessage, en in checkthread. Noot: access tot de queue wordt hierdoor ook gelimiteerd: slechts een van beide threads zal aan de queue kunnen op eender welk moment.

Deze primitieven implementeren we als volgt:

```
:- dynamic(waitingmessage/1).

:- mutex_create(checking).

sendmessage(Msg) :-
  mutex_lock(checking),
  handlemessage(Msg),
  assertz(waitingmessage(endofqueue)),
  checkwaitingmessages,
  mutex_unlock(checking).

delaymessage(Msg):-
  write('+ Queuing message '), write(Msg), nl,
  assertz(waitingmessage(Msg)).

dropmessage(Msg) :-
  write('x Dropping message '), write(Msg), nl.

receivemessage(Msg) :-
  write('- Passing message '), write(Msg), nl.

checkwaitingmessages :-
  retract(waitingmessage(Msg)),
  (Msg == endofqueue; (handlemessage(Msg), checkwaitingmessages
     )).

checkthread :-
  mutex_lock(checking),
  assertz(waitingmessage(endofqueue)),
  checkwaitingmessages,
  mutex_unlock(checking),
  sleep(20),
  checkthread.

:- thread_create(checkthread, _, [detached(true)]).
```

Met deze eenvoudige primitieven zullen we in volgende subsectie een eenvoudig synchronisatiemechanisme maken waarmee we op triviale manier onder andere het Dining Philosopher probleem kunnen oplossen.

4.5. Voorbeelden

4.5.1. Dining Philosophers

Het dining philosopher probleem bestaat uit het synchronisatieprobleem van vier hongerige filosofen. Ze zitten rond een tafel met elk een bord, en twee vorken. Deze vorken moeten ze delen met hun buren, en om te kunnen eten moet een filosoof beide vorken vastnemen. De filosofen zullen dus moeten samenwerken met hun buren om ervoor te zorgen dat ze allen kunnen eten. De filosofen kunnen we vergelijken met agenten, de vorken met resources. Een agent moet dus twee resources kunnen locken voor hij zijn opdracht kan uitvoeren.

We zullen volgende oplossing implementeren: een filosoof F mag een vork opnemen enkel en alleen als de vork die links van deze vork ligt ofwel nog op de tafel ligt, ofwel reeds in het bezit van de filosoof is. Dit zorgt ervoor dat wanneer de filosoof links van F zijn linkervork reeds genomen heeft, filosoof F zijn linkervork (en dus de rechtervork van de andere filosoof) niet mag opnemen. In het systeem komt dit overeen met agenten die resources van een tafel-agent locken voor gebruik. Een agent zal altijd eerst zijn linkervork, en dan de rechter proberen te locken, en na het eten onmiddelijk zijn vorken terug neerleggen (de resources unlocken).

Wanneer de tafel-agent merkt dat een agent een lock niet kan krijgen wordt deze request delayed. De queue zorgt ervoor dat agenten elkaar niet kunnen uitsluiten: als een agent ligt te wachten op een vork, en de agent die deze vork heeft legt hem neer, zal de agent deze vork onmiddelijk toegewezen krijgen door het systeem (als hij tenminste de eerstvolgende agent is die deze vork wou).

We zullen een heel eenvoudig mechanisme gebruiken: we definiëren guarding rules die gematched worden tegen verzonden boodschappen, als een guarding rule true evalueert mag de boodschap passeren, indien de rule false evalueert wordt de boodschap in de queue gestopt. De guarding rules definiëren we via `acceptmessage` rules, en `handlemessage` probeert de boodschap te unifieren met een van deze regels. De volgende guards definiëren we in het guarding systeem van de tafel-agent (de filosoof agenten hebben geen guarding regels nodig, alle boodschappen mogen altijd passeren):

```
:- dynamic(spoon/2).

prev(X,Y) :-
  Y is (X + 3) mod 4.

acceptmessage([From,_,[takespoon,Spoon]]) :-
  !, not(spoon(_,Spoon)),
  prev(Spoon,Pspoon),
  (not(spoon(_,Pspoon)); spoon(From,Pspoon)),
  asserta(spoon(From,Spoon)).

acceptmessage([From,_,[dropspoon,Spoon]]) :-
  !, spoon(From,Spoon),
  retract(spoon(From,Spoon)).

acceptmessage(_).

handlemessage(Msg) :-
  acceptmessage(Msg) -> receivemessage(Msg); delaymessage(Msg).
```

Een mogelijke run van het systeem:

```
-? sendmessage([1, table, [takespoon, 3]]).
 -> success
-? sendmessage([2, table, [takespoon, 0]]).
 -> fail
-? sendmessage([1, table, [takespoon, 0]]).
 -> success
-? sendmessage([1, table, [dropspoon, 3]]).
 -> success
-? sendmessage([1, table, [dropspoon, 0]]).
 -> success, 2 gets spoon 0
```

De primitieven laten ons dus toe op een eenvoudige manier synchronisatieproblemen zoals het dining philosopher probleem op te lossen. In het experimenten-hoofdstuk zien we hoe we met dit systeem de reeds bestaande synchronisatiemechanismen kunnen emuleren, en we zien hoe we met behulp van de vier eenvoudige primitieven een krachtig flow-control synchronisatiemechanisme kunnen bouwen.

4.5.2. Client - Server Synchronisatie

In een tweede voorbeeld hebben we een client / server situatie waarin clients "conferenties" kunnen starten op de server. Bij het opstarten van een conferentie zal de client opgeven hoeveel andere clients mogen inloggen op de conferentie. Vanaf dat moment kunnen clients inloggen op de conferentie, en boodschappen uitwisselen. Zolang nog niet alle client slots ingevuld zijn, zullen boodschappen verzonden vanuit de reeds ingelogde clients tegengehouden worden.

De code

```
:- dynamic(waiting/3), dynamic(joined/2).

acceptmessage([From, To, [startconf, ID, Num]]) :-
  !, \+waiting(ID, _, _),
  assert(joined(From, ID)),
  assert(waiting(ID, Num - 1, 1)).

acceptmessage([From, To, [connect, ID]]) :-
  !, waiting(ID, Open, Joined),
  Open > 0,
  \+joined(From, ID),
  retract(waiting(ID, Open, Joined)),
  asserta(joined(From, ID)),
  asserta(waiting(ID, Open - 1, Joined + 1)).

acceptmessage([From, To, [msg, ID, _]]) :-
```

```
  !, waiting(ID, 0, _),
  joined(From, ID).

acceptmessage([From, To, [leave, ID]]) :-
  !, retract(joined(From, ID)),
  retract(waiting(ID, Open, Joined)),
  ( Joined == 1; assert(waiting(ID, Open + 1, Joined - 1))).

acceptmessage(_) :-
  write('Boohoohoo, its all true!! :('), nl.

handlemessage(Msg) :-
  acceptmessage(Msg) ->
    (Msg = [From, From, _] -> dropmessage(Msg); receivemessage(
        Msg));
  delaymessage(Msg).
```

De client die een conferentie opstart wordt zelf automatisch toegevoegd aan de conferentie. Wanneer een client een conferentie wil toetreden, moet hij het conferentienummer van die conferentie opgeven, dit kan eventueel bekomen worden door aan de conferentieserver te vragen welke conferenties online zijn, vermits dit opvragen geen concurrentieproblemen oplevert hebben we er geen guarding rules voor nodig.

Zodra alle clients ingelogd zijn zullen alle reeds verzonden boodschappen toestemming krijgen verzonden te worden.

Clients kunnen natuurlijk ook conferenties verlaten, wanneer er geen clients meer ingelogd zijn in de conferentie, zal deze stopgezet worden (door het feit `waiting(ConferenceID, _, _)` te verwijderen).

Alle concurrentie handling gebeurt nu door het prolog guarding systeem, zodat de agent-code van de conferentieserver zich enkel moet bezighouden met het functionele aspect van message sending en connection management. Er zullen geen nieuwe connection requests binnenkomen als alle slots ingenomen zijn, of er zullen geen messages binnenkomen als nog niet alle client slots bezet zijn.

4.6. State Tracking

Voorgaande voorbeelden tonen hoe we in het guardingsysteem een idee kunnen opbouwen van de staat waarin de onderliggende agent zich bevindt. Dit wil zeggen dat we het synchronisatiemechanisme specifiek op de situatie kunnen afstellen indien een van de standaard mechanismen niet zou voldoen, of juist te uitgebreid is.

Wanneer het protocol van de communicatie tussen de agenten juist ontworpen is, kunnen we de relevante informatie voor het concurrentiemanagement afleiden uit de inhoud van de inkomende en uitgaande boodschappen van agenten. Zodoende moet de agent geen rekening houden met het bovenliggende prolog guarding systeem, vanuit het standpunt van de agenten is het guarding systeem volledig transparant.

Door het opbouwen van deze state kunnen we dus ook als dat nodig zou zijn een geschiedenis van de agent bijhouden, in welke staat hij zich bevind, in welke staten hij zich voordien bevond, enzovoort. Dit zouden we dan kunnen gebruiken om bijvoorbeeld rollbacks uit te voeren, zoals in transactiesystemen.

4.7. Synchronisatiemechanismen

Als eerste validatie zullen we de gedefiniëerde primitieven gebruiken om een aantal typische synchronisatiemechanismen te implementeren. We beginnen met de implementatie van binaire semaforen, om dan via rendez-vous tot een implementatie van een gedistribueerd transactiesysteem over te gaan.

4.7.1. Semaforen

Als eerste test zullen we een implementatie maken van de in sectie 3.3.1 gepresenteerde binaire semaforen. Het is enkel nodig om op de agent waar de te locken resources zich bevinden guarding rules te gebruiken. De agenten gebruiken het volgende protocol:

- `[a, b, [wait, n]]` -> agent a wil semafoor n op agent b locken
- `[b, a, [locked, n]]` -> agent a heeft lock op semafoor n verkregen
- `[a, b, [signal, n]]` -> agent a geeft lock op n vrij

De guarding regels definiëren we als volgt:

```
:- dynamic(locked/2).

myname(agt2).

handlemessage([From, To, [wait, Name]]) :-
  myname(To),
  \+locked(Name, _),
  assertz(locked(Name, From)),
  receivemessage([To, From, [locked, Name]]).

handlemessage([From, To, [wait, Name]]) :-
  myname(To),
  locked(Name, From),
  receivemessage([To, From, [locked, Name]]).

handlemessage([From, To, [signal, Name]]) :-
  myname(To),
  ignore(retract(locked(Name, From))).

handlemessage(Msg) :-
```

```
    Msg = [_, To, [wait, _]],
    myname(To),
    delaymessage(Msg);
    write('Default rule'), nl,
    receivemessage(Msg).
```

De implementatie van gewone semaforen is gelijkaardig, met het verschil dat per semafoor bijgehouden moet worden hoeveel agenten de kritieke sectie nog mogen binnentreden.

4.7.2. Rendez-Vous

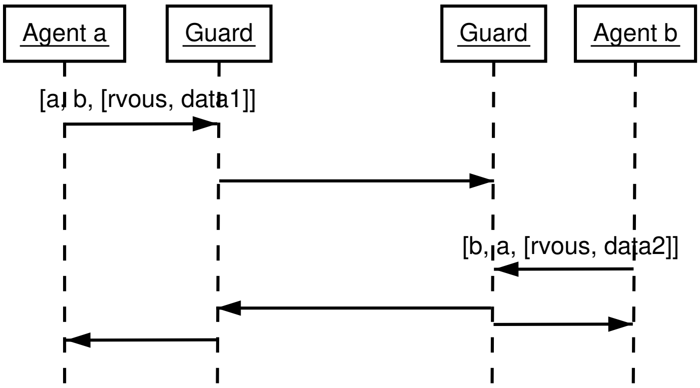
Het volgende systeem dat we implementeren is het rendez-vous systeem zoals gepresenteerd in sectie 3.3.3. Vermits de acties van de twee betrokken partijen dezelfde zijn, zullen we een set regels ontwerpen die op beide clients staan. Ziehier de regels voor een agent met naam 'agt2':

```
:- dynamic(sent/2).

myname(agt2).

handlemessage([From, To, [rvous, Data]]) :-
  myname(To),
  ( retract(sent(To, From)),
    receivemessage([From, To, [rvous, Data]]);
    delaymessage([From, To, [rvous, Data]])).

handlemessage([From, To, [rvous, Data]]) :-
  myname(From),
  assert(sent(From, To)),
  receivemessage([From, To, [rvous, Data]]).
```

Figuur 4.1 toont het sequence diagram van agenten a en b die een rendez-vous uitvoeren. We zien dat de eerste boodschap die verstuurd wordt (`[a, b, [rvous, data1]]`) in het guarding systeem van agent b delayed wordt, tot ook agent b zijn deel van de rendez-vous uitvoert. In het guarding systeem van agent a werd gemarkeerd dat agent a zijn rendez-vous al uitgevoerd heeft, dus wanneer de `[b, a, [rvous, data2]]` boodschap van agent b binnenkomt mag deze onmiddelijk doorgegeven worden aan de agent.

4.7.3. Transacties

Vermits transactiesystemen zeer populaire synchronisatiemechanisme zijn, gaan we ook dit implementeren in het prolog guarding systeem. We schetsen eerst de algemene werking van het transactiesysteem:



Figuur 4.1.: Rendez-vous

1. Een agent (vanaf nu client genoemd) wil een transactie starten. Hij contacteert hiervoor de transactie server agent, deze zorgt onder andere voor het toekennen van een uniek transactie ID.

2. De agent kan nu andere agenten (agenten waar zich resources bevinden die nodig zijn binnen de transactie) vragen tot de transactie toe te treden. Indien deze agent reeds een andere transactie gejoined is, zal de agent een abort message sturen naar de client. Deze zal alle reeds uitgevoerde acties rollbacken, en naar alle reeds gejoinde agenten een rollback message sturen.

3. Als de nodige resource-agenten gegetreden zijn, kan de client acties op deze resources ondernemen. Eventueel kunnen nog extra resource agenten gecontacteerd worden.

4. Als de client klaar is, stuurt hij een commit message naar alle toegetreden resource agenten. Deze zijn nu terug beschikbaar voor andere client agenten.

We hebben dus drie soorten agenten: client agenten die transacties starten, resource agenten die een resource bevatten die gebruikt wordt binnen transacties, en een transactie server. We zullen het systeem zodanig ontwerpen dat deze drie functies eventueel binnen een agent vervuld kunnen worden (dus bijvoorbeeld de transactie server die zelf een transactie wil starten).

We verwachten van de resource agenten dat deze in staat zijn om rollbacks uit te voeren als een rollback message binnenkomt, de meest eenvoudige manier om dit te ondersteunen is wanneer de agent een transactie joined een kopie van zijn state te maken, indien later de transactie gerollbacked moet worden kan gewoon deze kopie teruggezet worden. Als de transactie gecommit wordt kan de kopie weggegooid worden.

Dit zijn de regels voor de resource agenten, deze bevatten de regels om de locking van de agent te verzorgen:

```
:- dynamic(joined/1).

myname(agt3).

handlemessage([From, To, [join, ID]]) :-
  myname(To),
  joined(ID),
  receivemessage([To, From, [joined, ID]]).

handlemessage([From, To, [join, ID]]) :-
  myname(To),
  \+joined(_),
  assert(joined(ID)),
  receivemessage([From, To, [join, ID]]).
  receivemessage([To, From, [joined, ID]]).

handlemessage([From, To, [join, ID]]) :-
  myname(To),
  receivemessage([To, From, [abort, ID]]).

handlemessage([From, To, [commit, ID]]) :-
  myname(To),
  retract(joined(ID)),
  receivemessage([From, To, [commit, ID]]).

handlemessage([From, To, [rollback, ID]]) :-
  myname(To),
  retract(joined(ID)),
  receivemessage([From, To, [rollback, ID]]).

handlemessage([From, To, [doaction, ID, Args]]) :-
  myname(To),
  ( joined(ID),
    receivemessage([From, To, [doaction, ID, Args]]);
    delaymessage([From, To, [doaction, ID, Args]])).

handlemessage(_).
```

Vervolgens de regels van de transactie server:

```
:- dynamic(transid/1).

transid(0).

myname(srv).

handlemessage([From, To, [starttrans]]) :-
  retract(transid(Num)),
  New is Num + 1,
  assert(transid(New)),
```

```
    receivemessage([To, From, [transstarted, Num]]).
```

En tenslotte de regels van de client (de agent die de transactie start):

```
:- dynamic(locked/2).

myname(clt1).

commitlocks(Me, ID) :-
  retract(locked(Where, ID)),
  reveivemessage([Me, Where, [commit, ID]]),
  commitlocks(Me, ID).

abortlocks(Me, ID) :-
  retract(locked(Where, ID)),
  receivemessage([Me, Where, [rollback, ID]]),
  abortlocks(Me, ID).

handlemessage([From, To, [joined, ID]]) :-
  ignore(\+locked(To, ID), assert(locked(To, ID))),
  receivemessage([From, To, [joined, ID]]).

handlemessage([From, To, [abort, ID]]) :-
  abortlocks(To, ID),
  receivemessage([From, To, [abort, ID]]).

handlemessage([From, To, [commit, ID]]) :-
  myname(From),
  commitlocks(From, ID).

handlemessage(_).
```

4.8. Conclusie

We hebben in dit hoofdstuk de guarding primitieven gepresenteerd, en aan de hand van enkele eenvoudige voorbeeldjes hun werking aangetoond.

Verder hebben we aangetoond dat we de primitieven kunnen gebruiken om een aantal bestaande synchronisatiemechanismen te implementeren, gaande van eenvoudige binaire semaforen tot complexe gedistribueerde transactiesystemen.

In het volgende hoofdstuk zullen we het systeem verder valideren door aan te tonen dat we ook agenten met verschillende synchronisatiemechanismen, of zelfs agenten zonder onderling concurrentiebeheer (bijvoorbeeld agenten geschreven door verschillende partijen) met elkaar te laten synchroniseren.

5. Experimenten

We zullen nu de guarding primitieven verder valideren door twee krachtige toepassingen te maken, de eerste demonstreert dat we agenten die verschillende synchronisatiemechanismen gebruiken kunnen laten samenwerken; in de tweede toepassing zullen we de sychronizatie van agenten zonder ingebouwd synchronisatiemechanisme regelen.

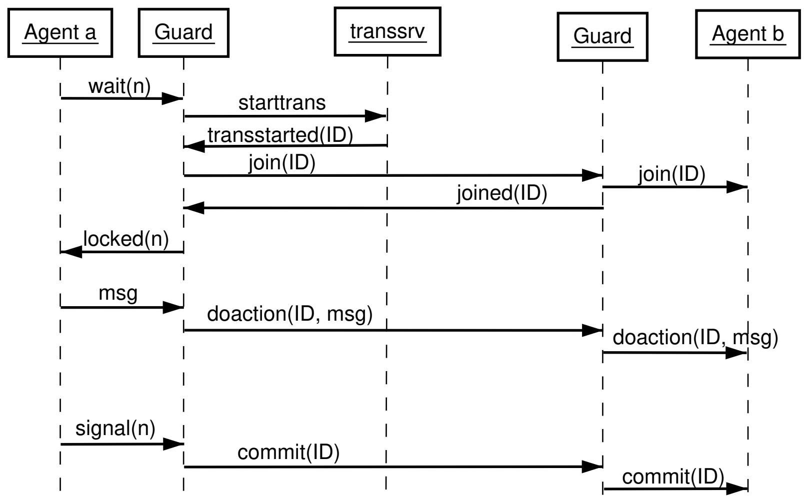
5.1. Adaptors

Het guarding systeem is in staat om adaptor te spelen tussen agenten die verschillende synchronisatiemechanismen gebruiken. Zo is het bijvoorbeeld mogelijk om een agent, die binaire semaforen gebruikt, te laten samenwerken met een resource agent uit een transactiesysteem. Dit doen we door het protocol van binaire semaforen te laten vertalen naar het protocol van transactiesystemen door het guarding mechanisme. Deze vertaling schetsen we via volgend voorbeeld, waarin agent a via binaire semaforen communiceert met agent b, welke een resource agent is uit een transactiesysteem met als transactieserver agent “transsrv” (zie ook figuur 5.1):

1. a stuurt `[a, b, [wait, n]]`. Om agent b te kunnen locken moeten we ervoor zorgen dat we een transactie starten waarin we agent b laten joinen. Als eerste stap moeten we dus een transactie starten. Dit doen we door een starttrans request te sturen naar de transactieserver.

2. Wanneer de transactieserver een bericht terugstuurd dat de transactie gestart is, wordt dit opgevangen door de guarding regels, deze slaan het feit `transaction(transactionID, resourceAgent, semaphoreName)` op. We weten nu dat resource `semaphoreName` gekoppeld is aan de transactie `transactionID`. Er wordt een join request naar de resource agent (b) gestuurd. Wanneer dit join request lukt, is de resource agent gelocked door de transactie, dus is de semafoor gelocked.

3. De resource agent stuurt `joined` of `abort` terug, in het eerste geval zal het guarding systeem dit doorvertalen naar een locked boodschap (dit is een boodschap die verstaan wordt door agent a), volgens agent a is de semafoor gelocked. Indien een abort message terugkomt (agent b is reeds gelocked door een andere transactie), zal de oorspronkelijke join message naar agent b delayed worden, zodat we later opnieuw kunnen proberen. Vermits er geen andere resource agenten in de transactie betrokken zijn is het niet nodig een rollback te starten.



Figuur 5.1.: Semafoor - transactie adaptor

4. Alle boodschappen die agent a nu verstuurd naar agent b (van de vorm `[a, b, Msg]`) worden eerst vertaald naar berichten van de vorm `[a, b, [doaction, transactionID, Msg]]`, en dan verstuurd naar agent b.

5. Als agent a de boodschap `[a, b, [signal, n]]` stuurt, wordt dit doorvertaald naar een commit message, deze wordt verstuurd naar agent b. De agent is nu opnieuw beschikbaar voor andere transacties.

Deze vertalingen gebeuren in het guarding systeem van de client, we kunnen volgende regels gebruiken om de vertalingen uit te voeren (deze regels vervangen de regels voor standaard binaire semaforen):

```
:- dynamic(transaction/3,requested/2).

handlemessage([From, To, [wait, X]]) :-
  \+transaction(_, To, X),
  assert(requested(X, To)),
  receivemessage([From, transsrv, [starttrans]]).

handlemessage([transsrv, To, [transstarted, ID]]) :-
  retract(requested(X, Rsc)),
  assert(transaction(ID, Rsc, X)),
  receivemessage([To, Rsc, [join, ID]]).

handlemessage([From, To, [joined, ID]]) :-
  transaction(ID, From, X),
  receivemessage([From, To, [locked, X]]).

handlemessage([From, To, [abort, ID]]) :-
```

```
  delaymessage([To, From, [join, ID]]).

handlemessage([From, To, [signal, X]]) :-
  retract(transaction(ID, To, X)),
  receivemessage([From, To, [commit, ID]]).

handlemessage([From, To, Msg]) :-
  transaction(ID, To, _),
  receivemessage([From, To, [doaction, ID, Msg]]).
```

In dit voorbeeld komt de kracht van het systeem te voorschijn: met een klein aantal eenvoudige regels zijn we in staat agenten met totaal verschillende synchronisatiemechanismen toch met elkaar te laten samenwerken, dit zonder iets te moeten veranderen aan de code van de agenten.

5.2. Flow Control

Tot nu toe hebben we enkel een bottom-up aanpak gehandhaafd: we baseren ons synchronisatiemechanisme op een low-level feature, zoals resources, een actie, of meerdere gekoppelde acties (semaforen, rendez-vous of transacties). We kunnen echter ook een heel ander aanpak nemen en de zaak van bovenuit bekijken: we kunnen “conflicten” definiëren, en aan de hand van deze conflict-definities afleiden of bepaalde acties mogen uitgevoerd worden of niet.

Veronderstel we hebben een agent, genaamd `dbinterface`, met een database-interface, die geen rekening houdt met concurrentie. De interface heeft twee methods: `get` en `store`. Dit levert geen problemen op zolang we niet met meerdere agenten tegelijk de database-interface gebruiken. We willen nu echter deze agent gebruiken vanuit verschillende client-agenten, dus we zullen zelf de synchronisatie moeten toevoegen.

De oproepen van de `get` en `store` methods van de interface moeten dus aan volgende beperkingen voldoen:

1. Een oproep van `get` mag gebeuren indien er momenteel geen `store` commando bezig is.

2. Een oproep van `store` mag gebeuren indien niets anders bezig is.

We kunnen deze twee regels als volgt uitdrukken in conflict-feiten:

1. `conflicts([_, dbinterface, [get, _]], [_, dbinterface, [store, _]]).` In mensentaal: een get-oproep aan dbinterface vanuit gelijk welke agent conflicteerd met een store-oproep aan dbinterface van gelijk welke agent.

2. `conflicts([_, dbinterface, [store, _]], [_, dbinterface, _]]`. In mensentaal: een store-oproep aan dbinterface vanuit gelijk welke agent conflicteerd met gelijk welke andere oproep (dus get of store) aan dbinterface vanuit gelijk welke agent.

Deze rules kunnen we gebruiken op een centrale agent (vanaf hier bossagt genoemd) om boodschappen verzonden naar de dbinterface agent te controleren. Dit kunnen we laten doen door volgende regels:

```
handlemessage([From, bossagt, Msg]) :-
  conflicts(Msg, Other),
  busy(Other),
  delaymessage([From, bossagt, Msg]);
  assert(busy(Msg)),
  receivemessage([bossagt, From, Msg]).

handlemessage([From, bossagt, Terminator]) :-
  busy(Msg),
  finished(Msg, Terminator),
  retract(busy(Msg)).
```

Wanneer een method van de dbinterface opgeroepen wordt zal het systeem eerst controleren of er conflicterende methods reeds aan het runnen zijn, indien niet mag de method gestart worden, anders wordt ze gedelayed. Het beëindigen van een method wordt opgevangen door de tweede regel, het finished predikaat wordt gedefinieerd als:

```
finished([From, To, [Method, _]], [To, From, [return, Method, _]]).
```

Anders gezegd: een method call naar een method `Method` binnen agent `To` vanuit agent `From` wordt beëindigd met een boodschap van agent `To` naar agent `From` met inhoud `[return, Method, ReturnValue]`.

Het enige waar we nu nog voor moeten zorgen is dat de potentieel conflicterende boodschappen gestuurd naar agent `dbinterface` eerst gevalideerd worden door agent `bossagt`. De volgende guarding regels, die we in het guarding systeem van agent `dbinterface` toevoegen, zorgen ervoor dat deze boodschappen worden doorgestuurd naar agent bossagt om gevalideerd te worden:

```
handlemessage(Msg) :-
  Msg = [_, dbinterface, [get, _]],
  receivemessage([dbinterface, bossagt, Msg]).

handlemessage(Msg) :-
  Msg = [dbinterface, _, [return, get, _]],
  receivemessage([dbinterface, bossman, Msg]),
```

```
  receivemessage(Msg).

handlemessage(Msg) :-
  Msg = [_, dbinterface, [store, _]],
  receivemessage([dbinterface, bossagt, Msg]).

handlemessage(Msg) :-
  Msg = [dbinterface, _, [return, store, _]],
  receivemessage([dbinterface, bossman, Msg]),
  receivemessage(Msg).
```

Deze regels kunnen we echter automatisch laten genereren vanop bossagt, aan de hand van de informatie in de conflict-regels. Het genereren van de regels voor een te valideren message (in dit geval bijvoorbeeld `[_, dbinterface, [store, _]]`) kunnen we doen via volgende regel:

```
generatereceivercode(Msg) :-
  Msg = [From, To, [Obj, Method, _]],
  writef('handlemessage(Msg) :-\n'),
  writef('  Msg = %q,\n', [Msg]),
  writef('  receivemessage([%q, bossagt, Msg]).\n\n', [To]),
  writef('handlemessage(Msg) :-\n'),
  writef('  Msg = [%q, %q, [return, %q, %q, _]],\n', [To, From, Obj, Method]),
  writef('  receivemessage([%q, bossman, Msg]),\n', [To]),
  writef('  receivemessage(Msg).\n\n\n').
```

Wanneer we al de forwarding code voor een bepaalde agent laten genereren (dus voor elke boodschap die de agent kan ontvangen die deel uitmaakt van een conflict-regel), en dan deze code naar de agent in kwestie transporteren om daar in te laden in het guarding systeem kunnen we ervoor zorgen dat alle mogelijk conflicterende boodschappen eerst even langs de `bossagt` passeren. Andere boodschappen, die geen conflicten kunnen veroorzaken, worden ongemoeid gelaten.

De enige guarding regel die we op een client manueel moeten installeren is de regel die toelaat berichten die gevalideerd zijn door de `bossagt` te ontvangen. De code voor de client is dus beperkt tot de volgende 4 regels:

```
myname(clt).

handlemessage([bossagt, To, Msg]) :-
  myname(To),
  receivemessage(Msg).
```

Tenslotte voor de volledigheid geven we de volledige code voor de `bossagt` agent:

```
conflicts([_, dbinterface, [get, _]], [_, dbinterface, [store, _
    ]]).
conflicts([_, dbinterface, [store, _]], [_, dbinterface, _]).

finished([From, To, [Method, _]], [To, From, [return, Method, _
    ]]).

generatereceivercode(Msg) :-
  Msg = [From, To, [Method, _]],
  writef('handlemessage(Msg) :-\n'),
  writef('  Msg = %q,\n', [Msg]),
  writef('  receivemessage([%q, bossagt, Msg]).\n\n', [To]),
  writef('handlemessage(Msg) :-\n'),
  writef('  Msg = [%q, %q, [return, %q, _]],\n', [To, From,
      Method]),
  writef('  receivemessage([%q, bossman, Msg]),\n', [To]),
  writef('  receivemessage(Msg).\n\n\n').

generatecode(Agent) :-
  conflicts([F1, Agent, Msg], Msg2),
  generatereceivercode([F1, Agent, Msg]),
  generatereceivercode(Msg2),
  fail; true.

handlemessage([From, bossagt, Msg]) :-
  conflicts(Msg, Other),
  busy(Other),
  delaymessage([From, bossagt, Msg]);
  assert(busy(Msg)),
  receivemessage([bossagt, From, Msg]).

handlemessage([From, bossagt, Terminator]) :-
  busy(Msg),
  finished(Msg, Terminator),
  retract(busy(Msg)).
```

Dit systeem laat ons toe met een minimum aan regels een hoog-level synchronisatiemechanisme te bouwen dat aan de hand van gedefiniëerde conflicten tussen method invocations van agenten kan beslissen of method invocations veilig zijn om uit te voeren of niet. Indien ze niet safe zijn, worden ze delayed tot ze geen conflicten meer veroorzaken met reeds uitvoerende methods.

5.3. Conclusie

We hebben de flexibiliteit en de kracht van de guarding primitieven aangetoond door met behulp van slechts enkele eenvoudige regels agenten met verschillende synchronisatiemechanismen, of zelfs agenten zonder onderling concurrentiebeheer met elkaar te laten synchroniseren.

We kunnen op eenvoudige manier andere synchronisatiemechanismen namaken, en het systeem is in staat een synchronisatiemechanisme uit te bouwen gebaseerd op high-level informatie. Het systeem biedt ons dus een zeer grote flexibiliteit, zonder dat hiervoor de implementatie zeer complex hoeft te worden, in tegendeel zelfs.

6. Conclusie

Nu zullen we het geleverde werk concluderen, en een aantal mogelijke toekomstige uitbreidingen presenteren.

6.1. Besluiten

In het eerste hoofdstuk hebben we de context van de thesis gesitueerd binnen gedistribueerde agent systemen, en we hebben een overzicht gegeven wat waar te vinden is in de tekst.

Het volgende hoofdstuk behandelde gedistribueerd programmeren, we hebben gezien wat gedistribueerd programmeren is, en welke grote verschillen gedistribueerd programmeren heeft tegenover klassiek "lokaal" programmeren. De belangrijkste verschillen zijn dat we in gedistribueerd programmeren meer met componenten werken, en dat in vele gevallen de verschillende componenten van een gedistribueerd programma asynchroon communiceren, dit om ervoor te zorgen dat het doodvallen van een agent niet leidt tot het stilvallen van een volledig systeem. Indien we timeouts gebruiken om te ontdekken of andere agents nog actief zijn moeten we ook rekening houden met de vertraging van het netwerk: hoe groter het netwerk, hoe trager de communicatie. De lengte van de timeout moet dus goed overwogen worden. Ook komen we een nieuw soort fouten tegen: gedeeltelijk falen van een programma door fout werken of uitvallen van agenten. Tenslotte hebben we enkele mechanismen gezien die ons toelaten gedistribueerd te programmeren.

In het volgende hoofdstuk zijn we wat dieper ingegaan op de problemen die concurrent lopende programmas kunnen veroorzaken inzake gedeeld geheugen, en we hebben enkele mechanismen gezien die ons toelaten de acties van concurrente agenten te synchroniseren. We hebben besloten dat de meeste van deze systemen voor grote, complexe programmas ontoereikend zijn, en dat de systemen weinig flexibel zijn. Ook hebben we besloten dat heterogene agenten (agenten geschreven op verschillende platformen en in verschillende talen) moeilijkheden hebben om met de complexere synchronisatiemechanismen te werken, dus moeten we voor die systemen terugvallen op eenvoudigere, generieke systemen. Deze zijn echter veel complexer om te beheren, wat de zaak er niet eenvoudiger op maakt.

In het vierde hoofdstuk hebben we geargumenteerd welke voordelen concurrentie aanpakken op een apart niveau oplevert, en wat de voordelen zijn van een guarding mechanisme te implementeren in een declaratieve programmeertaal zoals Prolog. De belangrijkste voordelen van het meta-level guarding mechanisme zijn:

- de flexibiliteit die het meta-niveau ons oplevert: de concurrentie guarding staat volledig los van de onderliggende agenten, of van het communicatiekanaal dat gebruikt wordt door de agenten.

- het krachtige unificatiealgoritme van Prolog, dit hebben we nodig voor het matchen van berichten tegen de guarding regels die gedeclareerd zijn in het systeem.

Vervolgens presenteren we de primitieven die we nodig hebben voor het guarding systeem, en we demonstreren hun werking door twee eenvoudige voorbeelden te implementeren. Verder doen we een eerste validatie door het concurrentie-gedrag van binaire semaforen, rendez-vous en een gedistribueerd transactiesysteem te implementeren via de gepresenteerde primitieven.

Verdere validatie gebeurt in het laatste hoofdstuk, waar we de kracht en de flexibiliteit van het systeem testen door aan te tonen dat we agenten, die gebruik maken van verschillende synchronisatiemechanismen, met elkaar kunnen laten samenwerken; en dat we zelfs agenten zonder ingebouwd concurrentiegedrag kunnen synchroniseren.

We kunnen besluiten dat we erin geslaagd zijn met enkele eenvoudige Prolog primitieven krachtige en flexibele sychronizatieregels op te stellen. We hebben aangetoond dat concurrentie aanpakken met een declaratieve taal, op een apart niveau, en losstaand van zowel het agent- als het communicatieniveau ervoor zorgt dat we zeer gespecialiseerde synchronisatiemechanismen kunnen ontwerpen, en dat we bovendien de mogelijkheid hebben het synchronisatiemechanisme te vervangen door een ander, zonder dat we hiervoor de agenten zelf moeten aanpassen. Hierdoor zijn we niet gebonden aan een specifiek synchronisatiemechanisme, zoals in de meeste andere systemen wel het geval is, en kunnen we de synchronisatie-regels van de agenten afstellen op de situatie.

Een ander handig gevolg van het op meta-niveau behandelen van concurrentie is dat we de communicatie tussen verschillende agent-platformen nog steeds kunnen synchroniseren via high-level mechanismen. Meestal zijn zulke high-level synchronisatiemechanismen (zoals bijvoorbeeld transactiesystemen) wel beschikbaar binnen één agent-platform (bijvoorbeeld Java), maar als we Java agenten willen laten communiceren met bijvoorbeeld Borg agenten kunnen we deze high-level mechanismen niet meer gebruiken, vermits we moeten terugvallen op eenvoudige, generieke communicatiemiddelen zoals sockets. Vermits de Prolog guarding regels los staan van het gebruikte agent-platform kunnen we nu echter wel deze high-level mechanismen gebruiken.

6.2. Toekomstige Uitbreidingen

Een eerste logische stap zou zijn om het guarding systeem volledig te implementeren, boven een werkend multi-agent platform. Het meest voor de hand liggend systeem zou Borg zijn, vermits we in Borg gemakkelijk de code voor verzenden en ontvangen van boodschappen kunnen aanpassen. We kunnen dus zonder veel werk ervoor zorgen dat elke binnenkomende en buitengaande boodschap eerst door het guarding systeem laten passeren.

Verder is het interessant te proberen het guarding systeem proberen te gebruiken om communicatie tussen Java remote objecten te synchroniseren. Hier komen we echter al meer problemen tegen: de boodschappen die door de Java Virtual Machines doorgestuurd worden naar elkaar zijn niet rechtstreeks leesbaar door Prolog. Wat we hiervoor dus moeten doen is de Java objecten vertalen naar structuren die wel leesbaar zijn door Prolog, en als het guarding systeem beslist dat een boodschap verstuurd mag worden, moet de Java versie verzonden worden.

Een andere zeer interessante uitbreiding is een uitbreiding van het high-level flow control experiment: het automatisch laten genereren van de conflict-regels. Hiervoor kunnen we een parser gebruiken die de source code van de agenten bestudeerd. Dit heeft een zeer handig gevolg: het concurrentie-management van de agenten gebeurt nu volledig geautomatiseerd.

Een mogelijk argument tegen de bruikbaarheid van het guarding systeem is dat de programmeurs Prolog moeten kennen. We zouden kunnen proberen deze restrictie weg te werken door het guarding mechanisme in het agent systeem op te nemen. Het beschikbaar zijn van een geschikt unificatiealgoritme in het agent systeem is hiervoor natuurlijk van levensbelang.

Dit is slechts een kleine greep uit de vele mogelijkheden van verdere uitbreidingen en verder werk die mogelijk zijn.

A. RMI Voorbeelden

A.1. Gedistribueerde Faculteit

In dit voorbeeld demonstreren we hoe een klasse zowel RMI client als server kan zijn, en uit het verloop van het programma kunnen we afleiden dat elke remote method call in een eigen thread loopt. Dit wil zeggen dat remote method calls concurrent runnen, dus bij de implementatie van de method moet opgelet worden voor concurrentieproblemen.

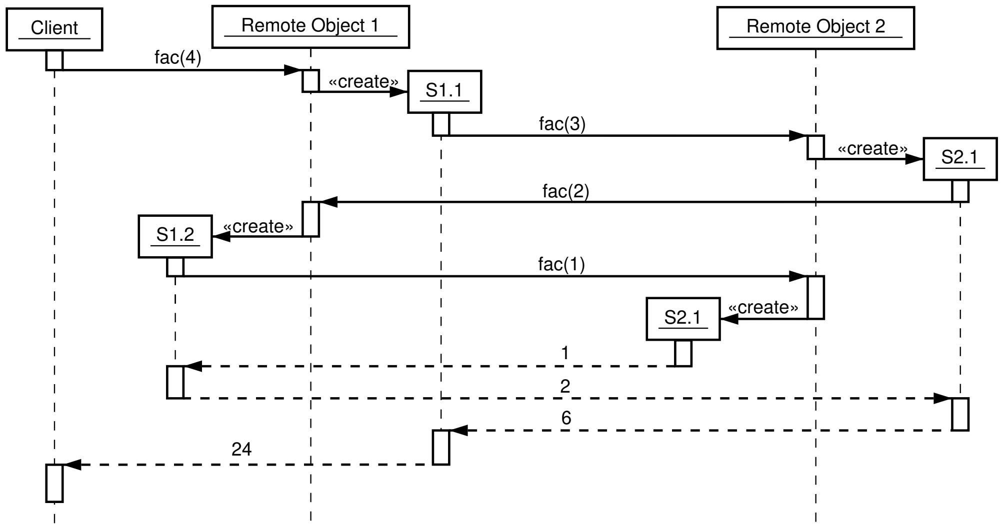
De remote interface bevat één method: `int fac(int n)`.

```
package distfac;

import java.rmi.*;

public interface FacSrv extends Remote {
    public int fac(int n) throws RemoteException;
}
```

De implementatie van het remote object is een beetje eigenaardig: we gebruiken een recursief faculteit-algoritme, voor elke recursiestap zal de `fac`-method van een ander remote object opgeroepen worden, het remote object is zelf dus ook een remote client. Als we hiervoor twee remote objecten gebruiken is het effect dat er remote method calls heen en weer gedaan worden tussen de twee remote objecten. Dit effect wordt verduidelijkt door figuur A.1.

We zullen twee remote objecten gebruiken in dit voorbeeld, elk remote object kan een referentie naar het andere opvragen om zo de recursiestap van het algoritme over te laten aan het andere remote object. In de figuur zien we tevens hoe een remote method call afgehandeld wordt door het remote object: er wordt een nieuwe thread gestart, waarin de method zal uitgevoerd worden. Op deze manier kan eenzelfde remote method vanuit verschillende clients tegelijk opgeroepen worden. Deze twee objecten worden geinstancieerd en geregistreerd (in het RMI registry) in de main-method van de FacImpl klasse.

```
package distfac;

import java.rmi.*;
import java.rmi.server.*;
```



Figuur A.1.: Gedistribueerde faculteit via RMI

```
public class FacImpl extends UnicastRemoteObject
implements FacSrv {
    private String other;

    public FacImpl(String other)
    throws RemoteException {
        super();
        this.other = other;
    }

    public int fac(int n)
    throws RemoteException {
        if (n > 1)
            try {
                FacSrv srv = (FacSrv) Naming.lookup(other);
                return (n * srv.fac(n - 1));
            } catch (Exception exc) {
                System.out.println(exc.toString());
            }
        return 1;
    }

    public static void main(String args[]) {
        try {
            FacImpl s1, s2;
            s2 = new FacImpl("eerste");
            Naming.rebind("tweede", s2);
            s1 = new FacImpl("tweede");
```

```
            Naming.rebind("eerste", s1);
        } catch (Exception exc) {
            System.out.println(exc.toString());
        }
    }
}
```

Het client-programma kan dan een van beide remote objecten opvragen aan het registry, en de fac-method oproepen. Dit zal het hele proces van recursieve oproepen tussen de twee remote objecten starten, en uiteindelijk zal de uitkomst naar de client gereturned worden.

```
package distfac;

import java.rmi.*;

public class FacClient {
    public static void main(String[] args) {
        try {
            FacSrv srv = (FacSrv) Naming.lookup("eerste");
            System.out.println(srv.fac(10));
        } catch (Exception e) {
            System.out.println(e.toString());
        }
    }
}
```

A.2. Agent Server

We zullen een eenvoudig mobiel agent systeem maken in Java. Agenten runnen op een AgentServer, en ze zijn in staat te verhuizen van een AgentServer naar een andere. De agenten kunnen aan de AgentServer een nieuwe bestemming vragen, hoe deze bestemming opgezocht wordt hangt af van de implementatie van de AgentServer. We zullen een agent maken die gedistribueerd faculteiten kan berekenen door zichzelf te klonen, en de kloon op te sturen naar een andere AgentServer. In het voorbeeld hebben we twee agentservers, die naar elkaar verwijzen. De FacAgent zal dus klonen heen en weer sturen tussen beide servers.

Allereerst zien we de implementatie van de AgentServer interface:

```
package facagent;

import java.rmi.*;

public interface AgentServer extends Remote {
    public AgentServer selectServer() throws RemoteException;
    public Object receiveAgent(Agent agent) throws
        RemoteException;
}
```

Een agent kan zichzelf transporteren naar een AgentServer door zichzelf mee te geven als argument aan de receiveAgent remote method van een AgentServer. Bij aankomst op de AgentServer zal de agent gestart worden door de server.

Vervolgens de implementatie van de AgentServer die we zullen gebruiken:

```
package facagent;

import java.rmi.*;
import java.rmi.server.*;

public class AgentServerImpl extends UnicastRemoteObject
implements AgentServer {
    private String otherServer;

    public AgentServerImpl(String otherServer)
    throws RemoteException {
        super();
        this.otherServer = otherServer;
    }

    public AgentServer selectServer()
    throws RemoteException {
        AgentServer other = (AgentServer) Naming.lookup(
            otherServer);
        return other;
    }

    public Object receiveAgent(Agent agent)
    throws RemoteException {
        System.out.println("Agent received.");
        return agent.execute(this);
    }

    public static void main(String args[]) {
        String serverName = args[0];
        String otherServer = args[1];
        try {
            AgentServerImpl server = new AgentServerImpl(
                otherServer);
            Naming.rebind(serverName, server);
        } catch (Exception exc) {
            System.out.println(exc.toString());
        }
        System.out.println("Server "+serverName+" started.");
    }
}
```

Alle agenten moeten volgende interface implementeren:

```
package facagent;

import java.io.*;
```

```
public interface Agent extends Serializable {
    public Object execute(AgentServer srv);
}
```

De faculteit agent werkt gedistribueerd recursief: zolang n niet 0 is zal de agent zichzelf klonen, en deze kloon n - 1 laten berekenen op een andere AgentServer. De agent blijft dan op antwoord wachten.

De implemenatie van de faculteit agent:

```
package facagent;

import java.rmi.*;

public class FacAgent implements Agent {
    int n;

    public FacAgent(int n) {
        this.n = n;
    }

    public Object execute(AgentServer srv) {
        Integer res;
        FacAgent agt;
        System.out.println("New fac agent started.");
        if (n > 1) {
            agt = new FacAgent(n-1);
            try {
                srv = srv.selectServer();
                res = (Integer)srv.receiveAgent(agt);
                return new Integer(res.intValue() * n);
            } catch (Exception e) {
                System.out.println(e.toString());
            }
        }
        return new Integer(1);
    }

    public static void main(String args[]) {
        FacAgent agt = new FacAgent(Integer.parseInt(args[1]));
        Integer res;
        String serverName = args[0];
        try {
            AgentServer srv = (AgentServer)Naming.lookup(
                serverName);
            res = (Integer)srv.receiveAgent(agt);
            System.out.println("Result: "+res.toString());
        } catch (Exception e) {
            System.out.println(e.toString());
        }
    }
}
```

Bibliografie

[1] Peter Flach, *"Simply Logical"*, John Wiley & Sons Ltd, 1994

[2] Jonathan B. Postel, "*RFC821: The Simple Mail Transfer Protocol*", 1982

[3] Prashant Sridharan, *"Advanced Java Networking"*, 1997

[4] De OMG CORBA homepagina,
`http://www.corba.org`

[5] Java Remote Method Invocation,
`http://java.sun.com/products/jdk/rmi`

[6] Kenneth Baclawski, *"Java RMI Tutorial"*,
`http://www.ccs.neu.edu/home/kenb/com3337/rmi_tut.html`

[7] Mark Skipper, *"Everything is an Aspect"*, 1999,
`http://www.doc.ic.ac.uk/~mcs98/research/aopsop/experiment2/everything.html`

[8] De BORG webpagina door Werner Van Belle,
`http://borg.rave.org`

[9] Website Aspect Oriented Software Development,
`http://aosd.net`

[10] Johan Fabry, *"Full distribution transparency, without losing control - AOP to the rescue"*, Master Thesis, 1999

[11] M. Ben-Ari, *"Principles of Concurrent Programming"*, Prentice-Hall, 1982

[12] C. A. Hoare, *"Communicating Sequential Processes"*, Prentice-Hall, 1985

[13] Edsger Wybe Dijkstra, *"Cooperating Sequential Processes"*, 1965

[14] Doug Lea, *"Concurrent Programming in Java"*, Addison-Wesley, 1997

[15] W. Richard Stevens, *"Unix Network Programming"*, Prentice-Hall, 1990

[16] Jannes Pockele, *"Concurrency Management as a Quality of Service Attribute in Distributed Asynchronous Collections"*, 2001

[17] Kris Gijbels, *"Aspect-Oriented Programming using a Logic Meta Programming Language to express cross-cutting through a dynamic joinpoint structure"*, 2001