Vrije Universiteit Brussel
Faculteit van de Wetenschappen
Departement Informatica

Ontwikkeling van een mobiele multi agent architectuur

Proefschrift ingediend met het oog op het behalen van de graad van licentiaat in de informatica.

Door: Werner Van Belle
Promotor: Prof. Theo D'Hondt
Academiejaar 1996 - 1997

Samenvatting

Heden ten dage worden een aantal mobiele agent systemen ontworpen en geschreven aan verscheidene research labs. Deze agent systemen worden vaak nogal ad hoc benaderd, zonder grondige kennis van de aspecten die agents aanbelangen. Het blijkt dat een belangrijk onderdeel van deze systemen bestaat uit location control, ofwel het bepalen waar een agent zich op een bepaald moment bevindt. In deze thesis zullen we een mogelijk efficiënt antwoord aanreiken.

Inhoudsopgave

1 Introductie

Het is opvallend hoe vandaag binnen computer netwerken de grens tussen client en server gaandeweg vervaagt om plaats te maken voor een nieuwe vorm van gedistribueerde programma's. Door de enorme toename in performantie van de modale PC is de zogenaamde client immers reeds in staat om zuivere netwerk programma's zoals FTP- en WWW-servers te ondersteunen. Anderzijds blijven er natuurlijk een hele reeks diensten over die niet lokaal kunnen aangeboden worden en die toch een hoge graad van interactiviteit langs de client-kant vereisen. Een voorbeeld hiervan zijn de talrijke search engines op het internet: door hun omvang en algemeenheid leveren ze minder relevante informatie op zodat een intelligente interactie met de eindgebruiker onontbeerlijk wordt. Denken we hierbij aan een adaptieve filter die tijdelijk lokaal wordt geïnstalleerd om relevante informatie te herkennen en die nauw samenwerkt met de eigenlijke search engine. Een ander voorbeeld zijn web crawlers: aangezien deze vanuit een lokale machine het internet doorlopen geeft dit aanleiding tot een spectaculaire verspilling van de beschikbare bandbreedte. Eigenlijk moet men dus stellen dat de enorme toename in stabiliteit en performantie van de hardware een drastische terugloop in de verhouding aantal clients/aantal servers met zich meebrengt.

Op het gebied van de software architectuur treffen we in dit kader de grootste afwijkingen van het traditionele client/server beeld; Java applets die delen van de programmatuur van een traditionele server exporteren naar de client zijn het best gekende voorbeeld. Nochtans zijn er reeds verschillende resultaten in de zoektocht naar een meer symmetrische structuur in gedistribueerde systemen: SUN en IBM ontwikkelen respectievelijk zogenaamde servlets [21] en aglets [18]. De visie die hierbij ontstaat is er een van ver doorgedreven uniformiteit: alle processen binnen een dergelijk netwerk zijn in hetzelfde medium uitgedrukt. Daarom nemen wij ons hier voor toekomstige gedistribueerde systemen te beschouwen als dynamische omgevingen waarbinnen mobiele software agents met elkaar en met menselijke gebruikers kunnen communiceren. [1]

De voornaamste reden om de notie van mobiele software in overweging te nemen is de reductie van nodeloze communicatie binnen een groot netwerk. In een systeem zoals het internet, waarbij een immens groot aantal machines op een niet reguliere manier met elkaar verbonden zijn kan het reactievermogen van agents, dat wil zeggen persistente en autonome software componenten, verhoogd worden door ze ook mobiel te maken. Zo kan bijvoorbeeld de mobiliteit van een menselijke gebruiker weerspiegeld worden in de mobiliteit van een persoonlijke agent die hem of haar op het internet vertegenwoordigt.

1.1 Technologie

Enerzijds is er dus het probleem van distributie, maar anderzijds is er ook de push die de huidige technologie geeft. Krachtige computers en netwerken laten

[1] Een totaal andere aanpak wat betreft distributie is een shared memory. Dit zijn de zogenaamde JavaSpaces [3].

toe het performantieverlies van virtuele machine (VM) technologie te accepteren om de voordelen ervan te kunnen uitbuiten.

In zijn thesis 'Language support for Mobile Agents' [10] beschrijft Frederick Colville wat een programmeertaal nodig heeft en welke voorzieningen aangaande taalconstructies en taalimplementatie aangenaam zijn als we zelf een agent framework willen programmeren. Laat ons hierover even uit wijden.

- Uit ervaring weten we dat geheugenbeheer op één machine met één processor de programmeur vaak onnodige kopzorgen baart en vaak tot onnodige fouten leidt. In deze context biedt garbage collection als technologie met zijn simulatie van een oneindig geheugen de uitkomst. In een gedistribueerde omgeving daarentegen waar men naar een object kan refereren van buiten uit de virtuele machine wordt dit een nog groter probleem. Stel dat 2 agents naar hetzelfde object refereren en de eerste agent verplaatst zich uit de machine. Op een gegeven moment houdt de tweede agent op te bestaan. Hoe kan de virtuele machine weten dat naar dit object nog wel degelijk gerefereerd wordt ? Omdat dit probleem te complex wordt hebben we nood aan automatisch geheugenbeheer.
- In gedistribueerde omgevingen en bijbehorende architecturen komt het maar al te vaak voor dat we een communicatielaag en transportlaag hebben. De transportlaag zorgt over het algemeen voor het verplaatsen van een byte stream van de ene machine naar de andere. De communicatielaag vormt de link tussen intern gerepresenteerde objecten naar netwerk gerepresenteerde objecten. Als we bijvoorbeeld communicatie beschouwen tussen een 386-architectuur en een SPARC workstation zullen we de convertie van big endian[2] naar little endian in acht moeten nemen. Deze convertie verwacht dat de programmeur semantiek op zijn byte streams plakt en zegt wat een entiteit vormt. 4 bytes zijn niet noodzakelijk een integer. Dit vraagt altijd de nodige interactie tussen het systeem en de communicatielaag. Meestal moet de programmeur aan elke klasse of structuur methoden `writeToStream` en `readFromStream` toevoegen. Het zou aangenaam zijn mocht de programmeertaal dit proces zo goed mogelijk automatiseren.
- Niet alleen op niveau van hardware wat betreft little endian en big endian treffen we problemen aan, maar ook wat betreft het operating system en aanwezige API's ontdekken we afwijkingen. We zouden bijvoorbeeld graag een agent verhuizen van een Unix workstation naar een NT-client. We hebben dus een taal nodig die in staat is deze heterogeniteit te virtualiseren.
- Eens dit allemaal opgelost is moeten we natuurlijk ook de code kunnen verplaatsen van machine x naar machine y. Dit kan gaan van de verplaatsing van agent beschrijvingen tot het verplaatsen van gecompileerde agents. Net zoals we opgemerkt hebben bij de transportlaag willen we dit zo doorzichtig mogelijk aanbieden.

[2] Dit is de byte volgorde waarin een integer in het hardware geheugen wordt voorgesteld.

- Het is duidelijk dat elke machine meerdere agents moet kunnen herbergen omdat we niet willen dat de aanwezigheid van een agent op een machine een andere agent belet te bestaan. Dit wil zeggen dat deze agents parallel moeten kunnen uitvoeren zonder in elkaars vaarwater te zitten. Ze zullen dus uitgevoerd moeten worden in hun eigen thread zonder dat coöperatie verwacht wordt van de agent.
- Stel dat we toch coöperatie verwachten van de agent, dan kan deze agent het hele systeem blokkeren en op oneerlijke wijze resources benutten die voor andere agents bestemd waren. Dit veiligheidsaspect van agents is er een dat opgenomen moet worden in de taal en zijn structuur, des te meer ook andere agents beschermd moeten worden tegen aanvallen van buiten uit. Interpreters vormen in deze context een onverwacht bruikbare oplossing. Door de virtualisering van de hardware en expliciete uitvoering van instructies houden we de executie van een agent totaal in de hand zoals opgemerkt in [9].

Het blijkt nu dat de Java bytecode interpreter aan veel van deze eisen voldoet en ons vandaag de mogelijkheid geeft snel een compacte implementatie te bouwen waarbinnen nagedacht en geëxperimenteerd kan worden over high-level communicatie tussen agents om zo tot beter onderhoudbare, abstractere modellen van interprocescommunicatie te komen. Het agent systeem wordt bijgevolg geprogrammeerd in Java, net zoals de agents zelf.

1.2 Doel

In deze thesis willen we testen hoe goed we in staat zijn dergelijk mobiele agents te implementeren met behulp van Java. Hiertoe zullen we dus de voor- en nadelen van bestaande agent systemen moeten onderzoeken. Aangezien dit nog een jonge technologie is die nog niet op punt gebracht is, is er ongetwijfeld plaats voor verbetering. We zullen trachten een aantal problemen zoals location transparency en communicatie te behandelen door het bouwen van een eigen agent systeem.

Het programmeren van agents in dit framework zou moeten overeenkomen met het expliciteren van de objecten die we ontdekken in een OO design. De gemakkelijkste wijze om hierover te redeneren is te denken aan diensten die *iemand* nodig heeft, aan diensten die *iemand* aanbied. Beschouw een agent als een representatie van de wensen van een gebruiker.

1.3 Reisgids

In het eerste deel onderzoeken we een aantal agent systemen die in Java geprogrammeerd zijn. Hierbij trachten we de voordelen en nadelen van elk systeem grondig af te wegen. Om dit te kunnen doen diepen we eerst allerhande aspecten van agents uit en creëren we het nodige referentiekader. Naderhand bespreken we in hoeverre de Java virtuele machine en zijn bijhorende taal voorzieningen bieden voor de ontwikkelingen van een dergelijk systeem.

Na de literatuurstudie zullen we in het tweede deel een eigen framework ontwikkelen dat de voordelen van de bekeken systemen tracht te integreren. Speciale aandacht zal hierbij worden geschonken aan location transparency en persistentie.

Om de leesbaarheid te verhogen zullen we, als we code in de tekst voegen, enkel het essentiële overhouden. Constructors die onmiddellijk hun super aanroepen, try-catch clauses en dergelijke worden niet vermeld. URL's en gelijkaardige namen worden sans serif gezet. Bijvoorbeeld http://ketchup.rave.org/~werner Klassenamen en methodenamen worden altijd in `typewriter` mode geprint. We nemen de conventie van Java en SmallTalk over en laten elke klassenaam met een hoofdletter beginnen; methodenamen laten we van start gaan met een kleine letter. Bijvoorbeeld: object `myDictionary` is een instantie van de klasse `MyDictionary`.

Deze thesis is niet in perfect algemeen Nederlands geschreven omdat sommige termen onvertaalbaar zijn zonder een onverstaanbare tekst over te houden. Waar we konden hebben we Nederlands op een consistente wijze gehanteerd. Vaktermen en code hebben we in hun standaardtaal, het Engels, gelaten. Zo nu en dan wordt zowel het Nederlands als het Engels voor een bepaald woord gebruikt, bijvoorbeeld 'message' of 'bericht'. Dit komt omdat we langzaam van een vakterm zoals 'message passing' overgaan naar 'het sturen van berichten'

1.4 Dankbetuiging

Eerst en vooral zou ik Wolfgang De Meuter (Wolf) willen bedanken voor de uren die hij samen met mij gespendeerd heeft om mij van 'char*' tot 'String' te promoveren. Professor Theo D'Hondt is zo gul geweest mij de vrijheid te geven allerhande onderzoeksdomeinen af te schuimen vooraleer ik dit onderwerp ter hand heb genomen. Patrick Steyaert heeft de tekst nagelezen en een aantal sublieme meta-meta-hints gegeven aangaande het nut van agents. Niels Boyen, de distributie expert van het 10e, heeft mij een hoop papers in verband met data distributie aan de hand gedaan. Geert Lathouwers heeft mijn werk geregeld geminimaliseerd door telkens weer opnieuw een paper op te diepen die net hetgene uitlegt dat ik bedacht had.

Technische ondersteuning heb ik gekregen van Joe Kiniry (de schrijver van de Infosphere) en Joachimm Baum (één van de ontwikkelaars van het Mole systeem). Carinne Lucas, Tom Lenaerts en Thomas Unger hebben de finale versie opgepoetst. Zeker wat betreft de LaTeXcode moet ik Thomas dankbaar zijn.

Deel I

Evaluatie van bestaande agent systemen

De inleiding van dit Deel bestaat uit het definiëren van een referentiekader voor agent frameworks. Nadien kijken we naar de taal Java en zijn virtuele machine en onderzoeken we in hoeverre deze taal de voorzieningen, beschreven in de introductie, ondersteunt. Daarna zijn we in staat bestaande agent systemen, die geïmplementeerd zijn in Java, onder de loep te nemen, waaronder Aglets (Hoofdstuk 4), Mole (Hoofdstuk 5) en de Infosphere (Hoofdstuk 6). Nadien zullen we in Deel II een eigen systeem ontwerpen en schrijven.

2 Agents

In dit hoofdstuk bespreken we allerhande aspecten van agents zoals persistentie, mobiliteit, intelligentie, autonomiteit en communicatie. Deze en andere eigenschappen dienen in acht genomen te worden als we een agent framework willen implementeren. We definiëren zo dus een referentiekader waarbinnen we agent systemen kunnen evalueren.

2.1 Hoog niveau aspecten van agents

In het wetenschappelijk domein van de artificiële intelligentie zijn agents geen vreemden. Het begrip agent is daar weliswaar een rekbaar iets, maar drie aspecten waar toch altijd opnieuw naar gerefereerd wordt en die beschreven worden in de IBM Open Blueprint [15] zijn:

- De *intelligentie* van een agent. Dit is een informeel begrip dat refereert naar de mate waarin een agent kan redeneren en leren uit bestaande situaties. Het begrip kan omschreven worden als de mogelijkheden van de agent om wensen van de gebruiker te aanvaarden en opgedragen taken uit te voeren. Men kan tenminste verwachten dat de gebruiker zijn voorkeur kan meegeven. Hogere niveaus van intelligentie omvatten het opstellen van een gebruikersmodel en het zich aanpassen aan het gedrag van de mens. Als men nog verder wil gaan, kan men denken hoe een agent leert en zich aanpast aan zijn omgeving, zowel op niveau van de beschikbare resources als het eigenlijke doel van de gebruiker. Op dit ogenblik worden intelligente autonome agents ontwikkeld door het bedrijf Autonomy Corporation [28], dit in samenwerking met Cambridge University. Een voorbeeld is een art agent die helpt zoeken naar kunst, zoals tekeningen of muziek, op het net.

- Als we het hebben over de *agency* van een agent spreken we over de mate waarin een agent autonoom kan handelen. Dit kan het best gemeten worden door de interactie tussen hem en zijn omgeving waar te nemen. Minimaal mag men verwachten dat een agent functioneert in naam van een gebruiker en met deze persoon communiceert. Een meer geavanceerde agent interageert met andere entiteiten van het systeem. Hierbij denken we aan data servers, resources en applications. Nog geavanceerder zijn agents die autonoom interageren onder mekaar en onderling onderhandelen om een bepaalde doelstelling waar te maken (dit noemt men **multi-agents** in tegenstelling tot **local agents**).
- *Mobiliteit* beschrijft hoe vrij een agent is van plaats te wijzigen. Om te beginnen zijn er natuurlijk de statische agents die op één machine verblijven en zich nooit verplaatsen. (Bijvoorbeeld user interfaces). Scripts kunnen van de ene machine naar de andere verplaatst worden en luiden ook een vorm van mobiliteit in. Het kenmerkende hier is dat de agent geen state data meesleept. Een hogere vorm van mobiliteit is er één waar zowel state als code verhuisd worden. In de limiet kan een agent zelfs in het midden van zijn uitvoering verplaatst worden zonder dat hij dat merkt. Hier spreekt men al eerder over mobiele objecten.

2.2 Definitie van een agent

Bovenstaande aspecten laten ons aanvoelen wat we met een agent bedoelen en wat het bereik van mogelijke software agents is. Doch, om het begrip mobiele agent beter te verstaan is nood aan verdere opdeling en beschrijving.

Uit hetgeen we reeds beschreven hebben blijkt dat een agent een software entiteit is die een actie moet uitvoeren voor zijn eigenaar. Het is dus te verwachten dat een agent een proces ten uitvoer brengt en bijgevolg dus ook zelf een proces bevat. Omdat we met meerdere processoren zitten en elke agent autonoom moet kunnen handelen zullen we aan een agent altijd moeten vragen om zijn status te wijzigen. Deze vraag tot statuswijziging gaat over machines heen en wordt het best verwezenlijkt met behulp van asynchrone communicatie. Synchrone communicatie met voorspelbare vertraging kunnen we niet verwachten omdat we enorme fluctuaties zullen zien in de wachttijden nodig om een agent te bereiken. Daarom definiëren we een **mobiele agent** als een code bevattende actieve autonome entiteit die tussen twee agent systemen getransporteerd kan worden en in staat is asynchroon berichten naar andere agents te sturen.

Wat betreft het gebruik van deze mobiliteit kunnen we een agent *zelf* laten beslissen wanneer en naar waar hij verhuist. In dit geval spreken we over **expliciet mobiele** software agents. Als er expliciet mobiele agents zijn, zijn er vanzelfsprekend ook **impliciet mobiele** agents die zelf geen initiatief zullen nemen om zich te verplaatsen. Tussen deze twee uitersten heerst een continuüm (een agent kan bijvoorbeeld 'hints' geven aan het agent systeem). Een voorbeeld van impliciete mobiliteit is een framework waarbinnen een agent verplaatst wordt omdat dit de algemene performantie ten goede komt.

2.3 Gebruik van mobiele agents

Afhankelijk van de context waarin we denken, ontdekken we andere toepassingen. Als we bijvoorbeeld aan de mobiliteit denken doemt onmiddellijk *mobile access* op. Dit is de meest voor de hand liggende toepassing aangezien gebruikers vandaag de dag zeer mobiel zijn. Stel dat een gebruiker zich met zijn laptop van Europa naar Amerika verplaatst, dan wil die gebruiker al de resources waar hij normaal aan kan, kunnen bereiken en gebruiken op veilige wijze. Zijn mobiliteit zou zo transparant mogelijk gemaakt moeten worden. Hoe de communicatie met remote resources verwezenlijkt wordt kan gaan van modem communicatie tot satelliet verbindingen. Zelfs docking stations zijn mogelijk. De autonomie en de intelligentie van dergelijke agents moet dermate hoog zijn dat ze zich aan nieuwe omgevingen gemakkelijk kunnen aanpassen. Het zou bijvoorbeeld nuttig zijn mocht een remote user zijn resources, bijvoorbeeld een demo van een project, naar hem kunnen roepen.

Net zoals de structuur van het net van plaats tot plaats zeer snel wijzigt, wijzigt ook de structuur van bedrijven en organisaties zeer snel. Men verhuist van statische fixed person/fixed tasks naar dynamische snel evoluerende structuren waarbij men vluchtige crisisgroepen heeft die tijdelijk resources aanspreken om een bepaalde taak uit te voeren. Voorbeelden hiervan zijn crisis management, teams voor proposal review en research collaboration. In netwerk gecentraliseerde omgevingen bieden intelligente agents een uitkomst om deze snelle herstructurering op te vangen. Meer specifieke voorbeelden zijn meeting makers en de collaboratieve design van formele specificaties. Hierbij worden de wensen en eisen van iemand zo goed mogelijk verdedigd. We kunnen ons dus verder concentreren op deze interactie met de omgeving en streven naar *Vloeiend aanpasbare virtuele organisaties en samenwerkingsverbanden* [12] met behulp van mobiele agents.

De hierboven beschreven snelle wijziging van inter-coöperatieve verbanden treffen we ook aan op niveau van systeem management. Systeembeheerders houden er niet van repetitief werk uit te moeten voeren. Soms moet men een machine configureren of diagnostiseren waar men fysisch niet aan kan. In deze gevallen willen we eigenlijk een actief netwerk dat systeem en netwerk management zoals *Remote diagnostics* heel wat vereenvoudigd. David L.Tennehouse beschrijft hetgeen hiervoor nodig is in 'Toward an active network architecture' [16]. De ordinaire pakketjes die over ethernet gestuurd worden, worden in dit voorstel vervangen door 'capsules', miniatuurprogrammaatjes die in elke netwerknode geïnjecteerd worden en daar actief kunnen deelnemen aan de routing van berichten.

Als we uit een heel ander vaatje tappen komen we aan de kant van intelligente agents te liggen, die door het opstellen van patronen van de gebruiker de nodige informatie proberen te filteren. FireFly is een voorbeeld van een intelligente agent die zich voorziet van dergelijke modellen en daarmee een muzieksmaak patroon van een gebruiker opstelt om zo na een tijdje zelf naar voor te komen met voorstellen. Dergelijke agents bevragen de enorme hoeveelheden informatie die voorhanden zijn steeds vanaf hun lokale machine. Hier kunnen mobiele agents zeker al zorgen voor snellere toegangstijden door op de servers hun filtering door te voeren in plaats van op de clients. Het actief vinden van informatie vraagt natuurlijk wel de nodige infrastructuur met voldoende meta data bevat om te beschrijven wat bepaalde resources inhouden. [19]

Als we bovenstaande scenario's bekijken komen we tot de constatatie dat we steeds de wachttijden voor de gebruiker proberen te beperken. In het eerste geval willen we de remote gebruiker snel bedienen door zijn software naar hem te verhuizen, in het tweede geval willen we de lokale interacties tussen clusters opdrijven door ze te herorganiseren. In het derde geval trachten we netwerk management te vereenvoudigen door de routing gemakkelijk aanpasbaar (lees: optimaliseerbaar) te maken en in het vierde scenario willen we de constante querying die optreedt bij opzoeken van informatie reduceren.

Een tweede aangename eigenschap van mobiliteit die gevonden kan worden in de paper 'Process Migration' [8] heet *fault resilience*. Als we in normale omstandigheden verplicht zijn een machine af te zetten, dan kunnen de mensen die hiervan gebruik maakten tijdelijk niet meer aan hun resources en/of tools. Door het automatisch verplaatsen van hun werkomgeving die ze gebruikten kan het uitvallen van die machine sterk verborgen worden.

2.4 Mobiliteit

Nu we het eens zijn over het nut dat verplaatsen van agents heeft, is het tijd om na te denken hoe we een agent en zijn bijbehorende proces(sen) van de ene machine naar de andere machine kunnen verplaatsen. Ruwweg genomen zijn er twee manieren om processen te verplaatsen. [17]

We kunnen een volledig nieuw proces creëren op de nieuwe lokatie: *remote execution*. Hierbij wordt de programmacode en zijn bijhorende data verplaatst naar de doellokatie. Daar wordt een nieuwe thread gestart die dan zelf moet zorgen dat al de nodige data opnieuw ingelezen wordt. Het uitvoeren van een nieuw proces kan gebeuren door de ontvangen code te compileren en te runnen of door ze onmiddellijk te interpreteren. Tegenwoordig is hier een mix tussen interpreteren en compileren aan te treffen, zoals JIT's (Just In Time Compilers) aantonen. De resultaten na een remote execution zijn een nieuw proces en een nieuwe status. Net zoals een fork onder Unix.

Migratie zorgt dat de huidige execution state volledig verplaatst wordt naar een andere virtuele machine. Het belangrijkste verschil met remote execution is dat het nieuwe proces zelf zijn data niet meer moet inlezen en dat dit op een transparante wijze plaats grijpt. De stappen waarin dit kan gebeuren zijn:

1. De *beslissing* tot verplaatsen wordt genomen door de agent zelf of door het netwerk. Belangrijk is hier dat de beslissing genomen wordt terwijl een agent aan het runnen is.

2. Nu moeten we het *runnende proces stilleggen.* Dit kan op twee manieren.

We onderbreken preëmtief het proces en beginnen direct met de verhuis. Dit noemt men telescripting. Ofwel brengen we de agent ervan op de hoogte dat hij verhuisd zal worden en wachten we tot hij er zin in heeft. (Coöperatieve verplaatsing, dit is wat men het looping model noemt)

3. Eens de agent klaar is om te verplaatsen en het proces gestopt is, moeten we de code en de data verplaatsen. In het telescripting model nemen we al de relevante data en code vast en verplaatsen die. Inclusief coroutine(s) natuurlijk ! In het looping model[3] vragen we de agent zichzelf te serialiseren en sturen we de ontvangen data op naar onze doelbestemming (dit is wat 'freezing of an agent' genoemd wordt). De data zelf kan verplaatst worden gebruik makend van het TCP/IP protocol gaande tot het sendmail protocol.

4. De ontvanger kan beginnen met de data en de code weer bij elkaar te sprokkelen. Dit wil zeggen dat we de *ontvangen code* eventueel compileren of dat we de ontvangen code dynamisch in het geheugen laden. Hierbij moet het systeem er ook voor zorgen dat de nodige voorzieningen (API's) aanwezig zijn zodat de agent naderhand gestart kan worden.

5. Het *laden van de data* is de volgende actie die ondernomen dient te worden. In het telescripting model creëren we de stack en data segmenten. In het looping model starten we een thread die de data zelf moet uitpakken (thawing of an agent).

6. En dan de finale actie: leven in het programma blazen. In het telescripting model: de agent verder laten lopen door *een proces toe te kennen* waarvan de instruction pointer direct goed staat; in het looping model: de main thread van de agent opnieuw starten.

	Telescripting	Looping model
Voordelen	1. Geen delay tot de agent beslist er de brui aan te geven. Een agent is niet in staat het systeem te blokkeren door tegen te werken. 2. Het schrijven van code voor dergelijke agents is hoogst aangenaam. We moeten geen rekening houden met het plotse onderbreken.	De samenspraak met de agent om te serialiseren zorgt dat we op zeer vlotte wijze compacte data kunnen samenstellen.
Nadelen	Serialiseren is totaal context-ongevoelig. Het is zeer goed mogelijk dat we veel te veel data opsturen.	Net zoals Windows oorspronkelijk co-operatieve multitasking was, zal blijken dat de eisen die we aan de programmeur stellen te hoog zijn. Het coderen van een dergelijke agent vraagt de nodige voorzichtigheid.

[3] Het looping model is in feite geen perfecte migratie en leunt ook een tikje aan tegen 'remote execution' omdat de agent gevraagd wordt zich te serialiseren. Langs de andere kant wordt het uiteen rafelen van de data mooi automatisch afgehandeld, zodat we dit toch nog als migratie kunnen beschouwen.

2.5 Security en persistentie

Als we in staat zijn onze processen en hun bijhorende data te verplaatsen komen we natuurlijk in de problemen wat betreft eigendom van resources en agents. In tegenstelling tot client/server, waar alleen maar de vraag rijst hoe we een server beschermen tegen aanvallen van buitenaf, snijdt hier het mes aan beide kanten. Op het ene moment willen we een agent systeem beschermen tegen aanvallen van binnenkomende agents die teveel resources vragen en op een ander ogenblik willen we een agent en zijn data beschermen tegen bepaalde eventueel vervalste agent systemen. Bijvoorbeeld het verwijderen en inpluggen van een nieuwe agent of het sturen van berichten onder naam van een niet aanwezige agent zorgt voor de nodige paranoia. Een typerend voorbeeld om systeem resources op te gebruiken is het recursief genereren van agents.

Samen met het gebruik van resources ontdekken we dat onze agents moeilijk persistent te maken zijn. Kan machine Y voldoende in vertrouwen genomen worden om agent A er zijn data op te laten stockeren ? Deze vraag is eigenlijk geen nieuwkomer en heeft juist aanleiding gegeven tot de situatie waarin we vandaag de dag verkeren. Omdat weinig mensen bestaande servers vertrouwen voor persistente data storage gebruiken ze hun eigen machine en zijn ze verantwoordelijk voor hun eigen data.

Omdat niet alle agents tegelijkertijd nodig zijn en gebruikt worden is het nuttig de mogelijkheid te bieden agents achter te laten op een spoolmedium en te (her)activeren als er een bericht voor hen binnenkomt. In bestaande agent systemen heeft men expliciete activatie- en deactivatie methoden zodat het agent systeem zelf niet moet raden wanneer een agent geactiveerd/gedeactiveerd mag worden. Dit komt architecturaal overeen met het runnen van inetd onder Unix systemen. Deze daemon zal andere services beschikbaar maken op het ogenblik er een vraag voor binnen komt. Httpd wordt bijvoorbeeld enkel gestart als er een http request binnenkomt.

2.6 Location control

We praten nu wel over *mijn* machine en *jouw* machine, maar deze naamresolutie is een tikje te ad hoc. Stel dat een machine ineens van eigenaar verandert en dat al de agents daar moeten verhuizen. Hoe lossen we dit dan op ? Hoe kunnen we zorgen dat deze agents, die dan tijdelijk moeten verhuizen, bereikbaar blijven zonder absurd veel manuele updates te moeten doen ? Het probleem zelf is tweedelig, enerzijds willen we een agent kunnen refereren door een naam te gebruiken, anderzijds willen we de agent kunnen bereiken door een naam te gebruiken.

Hoe we agents en agent systemen een eigen naam geven en hoe we naar een agent refereren op een unieke wijze is essentieel. Dit kan gaan van globaal unieke naming waarbij elke agent op het net een strikt unieke naam heeft tot unieke naming binnen bepaalde domeinen. Een ander aspect van een agentname is de duur waarin de naam zelf geldig blijft. Dit kan gaan van agentnames die onmiddellijk ongeldig worden na een verplaatsing tot namen die over meerdere virtuele machines en eventueel meerdere runs heen worden meegedragen. Afhankelijk van het agent systeem dat we onderzoeken zullen we andere benaderingen aantreffen.

Zoals reeds opgemerkt is naming niet het enige probleem dat optreedt. Eens we de naam van een agent kennen, willen we deze agent ook kunnen bereiken. Dit liefst op een wijze waarbij een verplaatsing van een agent geen of toch zo weinig mogelijk impact heeft op de naam die we gebruiken. We willen dus de positie van een agent transparant maken ten opzichte van de gebruikers van de agent, die enkel berichten kunnen sturen naar andere agents. Het is dus duidelijk dat **location transparency** enkel nodig is als we met andere agents communiceren. Dit kan verwezenlijkt worden door de routing van berichten voldoende slim te maken, of door de programmeurs van agents zelf de nodige voorzieningen te laten treffen.

2.7 Proxy agents

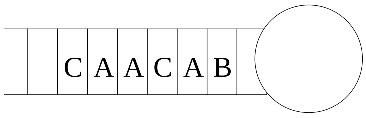
Eén manier om location transparency te voorzien is de idee van proxy agents. Hierbij leidt men de toegang tot een agent om door een zogenaamde proxy. Hierdoor kunnen we de agent in kwestie laten verhuizen zonder dat dit merkbaar wordt. De proxy zorgt er dan voor dat alle berichten geforward worden naar de juiste positie.

Een ander nut van deze proxies wordt gevonden in het zelf definiëren van een interface tot een agent. We kunnen bijvoorbeeld een proxy maken die aan ontvangen berichten een prioriteit toekent en zo dus een front end vormt voor de agent. Een proxy, gebruikt als interface tot een agent, kan bijvoorbeeld ook enkel de berichten die de agent wil ontvangen doorlaten. Als agent A enkel berichten wil ontvangen van 'hisFather' en niet van 'hisMother' kan de proxy hiervoor zorgen. Activatie- en deactivatie-methoden passen zeer goed in deze context. De proxy kan dan zelf kiezen wanneer de agent geactiveerd moet worden.

Andere mogelijkheden tot location transparency en naming worden besproken in Deel 2 waarbinnen we een eigen framework uit de grond stampen.

2.8 Communicatie

Berichten sturen is de enige manier voor agents om elkaars status te wijzigen. We kunnen niet zoals in de meeste objectgerichte systemen een object hardhandig binnendringen met een proces om er zijn status te wijzigen. Hoe we de message passing tussen agents implementeren is een ander vraagstuk. De communicatie tussen agents wordt vaak gebaseerd op asynchrone message passing. Zoals geweten is kan hier synchrone message passing over gelayerd worden. Daaruit volgt dat we niets missen door ons te beperken tot asynchrone message passing.



Algemeen komt asynchrone message passing neer op het aaneenschakelen van queues. Als we dit doen volgens het Actor message passing systeem [32] hebben we één queue waarbij men niet kan wachten op een bepaald type, waarbij men steeds het eerste bericht van de queue moet halen, in dit geval 'B'. Als we aan het Maisie [31] message passing system denken zouden we in staat zijn een bepaald message type uit de queue te halen en zo bijvoorbeeld berichttype B over te slaan en onmiddellijk het eerste 'A'-type te nemen.

2.9 Conclusie

In dit hoofdstuk hebben we getracht een beeld te schetsen van het vage begrip 'agent'. We hebben hieromtrent niets formeel kunnen vastleggen omdat het begrip zelf zo vaak in andere contexten gebruikt en misbruikt wordt. Wij zullen, telkens we over een agent spreken, het hebben over een mobiele actieve autonome software entiteit die bepaalde opdrachten moet volbrengen. Hiertoe kan hij op asynchrone basis communiceren met andere agents. Om een beeld te schetsen van wat een agent is hebben we een aantal toepassingen uitgelegd en hebben we geconstateerd dat het steeds de factor performantie is die komt bovendrijven.

Op het ogenblik dat we zelf begonnen na te denken over agents en agent systemen bleken er een aantal problemen op te doemen. Het eerste probleem op zich was het verplaatsen van de nodige processen. Een tweede probleem omvatte het geven van namen aan agents en het bereiken van agents door gebruik te maken van die naam. Onafhankelijk van deze voorzieningen moet overdacht worden hoe agents 'communiceren', met andere woorden, hoe berichten tussen agents behandeld worden. Het laatste probleem dat we geïdentificeerd hebben is het secure maken van agents en agent systemen.

3 Java

In dit onderdeel bespreken we in hoeverre de taal Java voorzien is om agent frameworks mee te implementeren. We gaan er onmiddellijk vanuit dat de agent description language dezelfde is als de taal waarin het framework geschreven is. Op het einde van deze sectie worden implicaties van het doorbreken van deze benadering overschouwd. Maar eerst en vooral moeten we onderzoeken wat de Java virtuele machine ons te bieden heeft.

3.1 De virtuele machine

3.1.1 Bytecode

Elke Java virtuele machine heeft een byte-code die zeer standaard geworden is. Elke www-browser, elke machine kan vandaag een Java bytecode interpreter runnen. We gaan er vanuit dat de lezer vertrouwd is met de Java VM maar vermelden toch nog even dat het een stack-machine is voor een object geöriënteerde taal waarbij de code zelf wordt bijgehouden in klasse-files. Eén enkele klasse-file komt overeen met één klasse voor de virtuele machine. De enige lijmstructuur zijn deze klassen en bijgevolg ligt het voor de hand te overwegen een soort agent voor te stellen als een klasse en de agent zelf als instance van deze klasse. Het eerste voordeel is dat de klasse-files veel compacter zijn dan de meeste gecompileerde native-code en veel kleiner dan de ongecompileerde source-files. Een ongedocumenteerde Java-file van 3K komt gemakkelijk overeen met een klasse-file van 900 bytes. Deze naar een nog lager niveau compileren mondt vaak uit in grotere data files. Dit biedt duidelijk voordelen voor snel transport over een netwerk.

Een ander voordeel van de bytecode is dat hiermee de heterogeniteit tussen verschillende machines opgeheven wordt, dit niet alleen op niveau van hardware, zoals big-endian en little-endian,[4] maar ook op gebied van aanwezige API's. Het virtualiseren van system calls naar het operating systeem om windows te creëren of het aanspreken van files en dergelijke zijn hiervan goede voorbeelden. In Java wordt dit met behulp van een standaard AWT opgelost.

Doordat deze bytecode geïnterpreteerd wordt kunnen we eveneens een sterke veiligheidscontrole uitvoeren wat betreft bescherming van de eigen hardware. De virtuele machine kan alles wat gebeurt, elke instructie die uitgevoerd wordt, controleren of laten controleren door een security manager. Dit houdt ook in dat we de runnende processen kunnen monitoren en beschermen tegen elkaar. De vertraging die ontstaat door het interpreteren wordt door betere technologie zoals 'Just In Time Compilers' teruggeschroefd naar zeer aanvaardbare snelheden.

[4] De volgorde van de bytes op een lager niveau.

3.1.2 Class loading & Threading

De class files moeten in een virtuele machine geladen worden. De bytecode heeft hier een paar natives voor die op hoger niveau gebracht worden door Classloaders die de data van een of andere medium laden (bijvoorbeeld netwerk, het lokale filesysteem, http, ...) en deze dan in geheugen dumpen. Deze class loading kan gebeuren op een willekeurig moment als de virtuele machine aan het runnen is. De nauwe interactie tussen de classloader en de virtuele machine zorgt ervoor dat enkel de klassen die nodig zijn geladen worden. Hierdoor komen we in een situatie waarbij enkel de nodige code geladen wordt. De idee van een classloader wordt verder in detail besproken in Hoofdstuk 8.1.

Tesamen met het in het geheugen laden van klassen willen we deze klasse, het stuk code dat uiteindelijk de representatie van het soort agent zal worden, kunnen uitvoeren. Dit kunnen we doen door het main proces binnen te laten dringen in nieuw gemaakte instances, met het grote nadeel dat we de control flow niet meer in handen hebben. We zijn in dit scenario dus verplicht tot coöperatief multitasken, en de ervaring wijst uit dat dit meestal oneerlijk uitdraait. Er zullen processen zijn die te veel resources vragen en de control flow amper teruggeven en er zullen moedwillige processen zijn die het hele systeem blokkeren. De virtuele machine biedt hier uitkomst door zijn multi-threading waardoor we een geladen klasse kunnen starten in een eigen thread. De multitasking tussen agents kan nu preëmtief uitgevoerd worden in één namespace. Dit wil effectief zeggen dat we elke agent autonoom kunnen starten door een agent een eigen thread toe te kennen.

3.1.3 JavaObject serialisation

Om objecten persistent te maken in doorlopende applicaties (over meerdere runs heen dus) is het nodig de status van objecten (of een graph van objecten) te representeren in hun doorlopende geserialiseerde vorm. JVM en Java bieden de nodige voorzieningen hiervoor door objecten te serialiseren en op een stream te schrijven [2]:

Bijvoorbeeld:

```
// Het schrijven van de huidige datum naar een file
FileOutputStream f = new FileOutputStream("myfile");
ObjectOutput s=new ObjectOutputStream(f);
s.writeObject("Today");
s.writeObject(new Date());
s.flush();

// Het lezen van string en datum
FileInputStream in = new FileInputStream("myfile");
ObjectInputStream s = new ObjectInputStream(in);
String today = (String)s.readObject();
```

```
Date date = (Date)s.readObject();
```

Het proces waarbij objecten op een stream worden geschreven noemt men marshalling. Het proces waarbij ze van een stream worden gelezen noemt unmarshalling. Als een object gedeserialiseerd wordt, wordt altijd een *nieuw* object gecreëerd. Java serialisation kan dus niet gebruikt worden om hetzelfde object te maken. (dus $u(s(a)) \neq a$, waar u het deserialiseren is, s het serialiseren en $\neq$ de vergelijking van de referenties is)

Een object is serialiseerbaar als hij de interface `Serializable` implementeert. Als we zelf willen bepalen hoe een object weggeschreven wordt, moeten we `Externalizable` implementeren.

3.1.4 Stackframes

Het zou prettig zijn mochten we de runtime stack als serialiseerbaar object kunnen beschouwen zodat we deze mee kunnen serialiseren. Mocht dit kunnen zouden we threads kunnen serialiseren en de coroutine verhuizen naar een andere VM. In dit geval zouden we processen samen met hun execution state kunnen verplaatsen, telescripting met andere woorden. Het verplaatsen van een agent thread zou bijvoorbeeld bestaan uit

```
FileOutputStream f = new FileOutputStream("tmp");
ObjectOutput s=new ObjectOutputStream(f);
s.writeObject(agent.getCurrentThread());
s.flush();
```

Doch, op het ogenblik is de stack van de Java virtuele machine een intern object waar we niet direct aankunnen.

3.2 De taal Java

Op dit ene minpunt na blijkt de Java virtuele machine voldoende technische ondersteuning te bieden voor het implementeren van agent frameworks. We moeten ons nu de vraag stellen in hoeverre de taal Java tekort schiet voor het schrijven van gedistribueerde applicaties. Eerst en vooral bekijken we een aantal van SUN's benaderingen om distributie te lijf te gaan.

3.2.1 RMI

Een feature van de nieuwe Java VM (JDK1.1) is Remote Method Invocation. Dit is een gedistribueerd object model voor de Java virtuele machine dat zoveel mogelijk de huidige semantiek van het Java object model behoudt.[5] Hierdoor kan men op eenvoudige wijze gedistribueerde objecten implementeren en gebruiken. Het systeem combineert aspecten van het Modula-3 Network Objects system en Spring's subcontract (aldus [1]).

Net zoals Remote Procedure Call (RPC) gebruikt wordt om synchroon te communiceren met een server, dus vragen stellen, kan men in Java RMI gebruiken om met objecten op afstand synchroon te communiceren. De doorsnede tussen server en client wordt gedefinieerd in een Java-interface die voor beiden bereikbaar moet zijn. Bijvoorbeeld

```
public interface RmiServer extends java.rmi.Remote
  {Object Print(Object toprint) throws java.rmi.RemoteException;}
```

Het opvangen van de conversie van interne message passing naar remote message passing wordt gedaan door het genereren van stubs en een skeleton. Een stub is de client-side proxy voor een remote object. Een skeleton van een remote object is een entiteit aan de serverkant, verantwoordelijk voor de dispatching van binnenkomende berichten naar method-calls. De server ziet er in dit geval als volgt uit:

```
public class RmiServerImpl
extends java.rmi.server.UnicastRemoteObject
implements RmiServer
  {private static String MyName=null;
  static RmiServerImpl RmiServerken=null;
  public Object Print(Object toprint)
    {System.out.println(toprint);
    return toprint;}
  public static void main(String argv[])
    {System.setSecurityManager(
      new java.rmi.RMISecurityManager());
    MyName = "rmi://IGWENT2/RmiServer";
    RmiServerken = new RmiServerImpl();
    java.rmi.Naming.rebind(MyName, RmiServerken);}}
```

In de bovenstaande voorbeeldcode zal de `main` functie een server instantiëren. De server implementeert de `Print` methode die door een client aangeroepen zal worden. Merk ook op dat na het creëren van een server deze server aan een naam gebonden moet worden; in dit geval rmi://igwent2/RmiServer.

De programmastructuur van de client kan hieronder aangetroffen worden. In de `main`-functie wordt eerst een representatie van de server gecreëerd. Dit wordt gedaan door een stub-klasse (aangemaakt door de bijgeleverde stub compiler: rmic) te instantiëren en te benaderen als interface. (`RmiServer` in dit geval)

[5] In die zin dat de referentie naar een remote object doorgegeven kan worden aan elke methode aanroep (zowel lokaal als niet lokaal). Remote message passing gebeurt daarentegen door call by value en niet meer door de standaard call by reference.

```
public class RmiClient
  {private static RmiServer RmiServerken=null;
  protected static void Initialize()
    {System.setSecurityManager(
      new java.rmi.RMISecurityManager());
    String ChannelName;
    RmiServerken =(RmiServer)java.rmi.Naming.lookup(
      "rmi://IGWENT2/RmiServer");}
  public static void main(String argv[])
    {Initialize();
    RmiServerken.Print("Test");
    RmiServerken.Print("Testbis");}}
```

Javaspaces [3] zijn een andere benadering van SUN om in een gedistribueerde omgeving voor communicatie te zorgen. Hierbij is er een ruimte, een markt, waarbinnen allerhande objecten uitgestald kunnen worden. De ene client kan nu op het gepaste ogenblik een object op de markt gooien en andere objecten kunnen dit eraf nemen. Dit is te vergelijken met een shared memory waar verschillende processen op concureren, zei het in een grotere internetomgeving. Om deze markten te implementeren maken ze gebruik van RMI. De idee achter dit coördination model is duidelijk te lezen in [30] waar Paolo Ciancarini aan de hand van het taal-framework 'Linda' allerhande toepassingen demonsteert.

3.2.2 Meta architectuur

Zoals we opgemerkt hebben bij RMI moeten we in gedistribueerde omgevingen vaak stubs en skeletons genereren. Dit is een vervelend tijdrovend werkje dat automatisch uitgevoerd kan worden door een zogenaamde stub-compiler. Bijvoorbeeld `rmic` wordt gebruikt om stubs en skeleton paren te genereren voor RMI.

Omdat we in een skeleton object geen jump table van methoden kunnen maken zijn we verplicht de mapping van incoming message naar method call hard te coderen, met als groot nadeel dat het hard coderen van een reeks condities onnodig veel code inneemt en dat deze harde codering hoogst onflexibel is, in die zin dat we niet snel een bericht kunnen toevoegen aan de interface van een object.

Als we met agents werken die een veel dynamischere interface hebben dan methode signatures zullen we berichten moeten afbeelden op eventueel andere, op voorhand onbekende, methoden. Zeker als we op meta niveau over mogelijke onderneembare acties willen redeneren. We hebben duidelijk een `perform` of `apply` nodig.

Een ander nadeel van dergelijke stub-generatie is dat deze stubs nog te statisch zijn. We kunnen niet gauw een methode hermappen op een ander nummer. Deze onflexibiliteit wordt nog verder benadrukt als we dynamisch, terwijl een systeem aan het runnen is, een stub willen genereren voor één of ander remote object.

Een ander gebrek van de taal Java op zich is dat we geen meta level controle kunnen doen wat betreft het versturen van berichten. Als we bijvoorbeeld elk bericht, dat van object A naar object B gestuurd wordt, op een eigen speciale wijze willen doorsturen zijn we verplicht gebruik te maken van deze stub/skeleton benadering. Een oplossing zoals Smalltalk waarbij de `messageNotUnderstood` wordt overschreven is nog niet ingeburgerd.

Door rekening te houden met dit eventueel gebruik van de VM, zijnde dat we een `perform` nodig hebben, en zijn huidige gebreken, zijnde dat we geen `perform` hebben, kunnen we de virtuele machine uitbreiden met de nodige primitieven. SUN heeft dit ook bedacht en vanaf JDK1.1 is de notitie van reflectie geïntroduceerd en kunnen we methoden als een object behandelen (first class methods). Helaas is er nog steeds geen `messageNotUnderstood` voorziening.

3.2.3 Agent Scripting Taal

Laat ons er nu even vanuit gaan dat we een andere taal dan Java zouden gebruiken om de agents te programmeren. Een mogelijke benadering is dat we scriptjes hebben die van de ene machine naar de andere verplaatsen. Een voorbeeld hiervan is AgentTcl [29], dewelke overigens ook een prachtige demonstratie is van telescripting. Hieronder vindt u een stuk voorbeeldcode van een dergelijke script agent. Het script in kwestie zal één agent maken die van machine naar machine springt om er de lijst van ingelogde users op te vragen.

```
proc who machines
  {global agent
  set list ""
  foreach m $machines
      {agent_jump $m
      set users [exec who]
      append list "$agent(local-server):\n$users\n\n"}
  return $list}

set machines "bald.cs.dartmouth.edu \
              tuolomne.cs.dartmouth.edu"

agent_begin
agent_submit $agent(local-ip) -vars machines
          -procs who -script {who $machines}
agent_receive code message -blocking
puts "\nWHO´S WHO on our computers\n\n$message"
agent_end
```

Een voordeel van dergelijke scripting agents is dat ze zeer gemakkelijk bereikbaar zijn voor de eindgebruiker. Dit omdat er geen code gecompileerd moet worden, hoewel dit slechts een psychologische drempel is. Nu de agents niet gecompileerd worden komen we wel tot de vaststelling dat deze agents veel plaats innemen wat betreft code, dit natuurlijk in vergelijking met de Java classfiles. In tegenstelling tot de code-loader (classloader) van Java moeten we ook altijd het volledige programma doorsturen, wat performantieverlies kan betekenen.

Stel nu dat we zulke agents willen runnen in een framework geschreven in Java. In hoeverre gelden de bovenvermelde features van de JavaVM en kunnen we er op steunen ? Wat zeker al vaststaat is dat we een eigen interpreter zullen moeten schrijven om deze scripting-code te vertolken. Dit wil zeggen dat we over geheugenbeheer zullen moeten nadenken.

We kunnen overwegen de garbage collector van de VM te gebruiken om automatisch geheugenbeheer voor de scriptingtaal te voorzien. Dit blijkt inderdaad mogelijk te zijn zolang we er voor zorgen dat elke referentie binnen de scriptingtaal exact overeen komt met één referentie binnen de interpreterende taal.

Omdat de agent scripting en het framework veel verder uiteen staan dan bij het scenario waar een agent een klasse is, komen we tot de constatatie dat we de interface tussen script-taal en framework nauwkeurig moeten definiëren. Dit niet alleen op het gebied hoe we een agent laten verplaatsen of hoe een agent zich aanmeldt, maar ook op gebied van hoe een agent zijn acties onderneemt. Hoe een agent bijvoorbeeld een dialogbox op het scherm krijgt. In feite moeten we hier één grote API voorzien die vragen van de scriptingtaal omzet naar method calls. Ook hier zouden higher order methods een handig geschenk zijn. Doch dit is gemakkelijk te omzeilen door een enorme skeleton te schrijven. Eens dit gedaan is hebben de agents onderling geen last meer van stub en skeleton problemen.

Nu kunnen we het looping model afwegen ten opzichte van het telescripting-model. Indien we het looping model nemen moeten we het serialiseren van de execution state zelf doen. We kunnen hierbij duidelijk gebruik maken van de Java serialisation, maar dan moeten we wel zorgen dat elke agent, gerepresenteerd in het geheugen, een eiland vormt dat in één blok opgestuurd kan worden naar de doelmachine. Eens we toch de serialisation van agents grondig hebben onderzocht en we een expliciete runtime stack voorzien hebben kunnen we ons nog moeilijk afwenden van de sublieme telescripting.

3.3 Conclusie

In dit onderdeel is gebleken dat interpreters een perfecte basis vormen voor agent systemen. Meer bepaald blijkt de Java virtuele machine nog een paar aangename eigenschappen zoals multithreading, serialisation en classloading te bevatten die het programmeren van agent frameworks heel wat vereenvoudigen. Als we de taal Java beschouwen, kunnen we RMI niet onopgemerkt voorbij laten gaan. Het nadeel aan RMI is dat dit zuiver synchrone communicatie is. Maar het heeft SUN wel verplicht aspecten zoals reflectie te introduceren.

Of we nu interpretatie van een scriptingtaal kiezen of de voorstelling van agents als objecten is een andere keuze. Beide methoden hebben hun voordelen en nadelen. In deze thesis hebben we bewust gekozen voor het schrijven van agents als objecten in Java omwille van de technische mogelijkheden die de Java VM biedt.

We zullen nu overgaan naar het onderzoeken van een aantal bestaande agent-frameworks, geschreven in Java.

4 Aglets

4.1 Overzicht

De Aglet Workbench ontwikkeld door het 'IBM Tokyo Research Laboratory' beschrijft en implementeert een agent framework waarbinnen expliciet mobiele agents (Aglets) kunnen verplaatsen van de ene agent context (machine) naar de andere, hierbij code en data transporterend. Hiervoor werd een Agent Transfer Protocol [18] (ATP) ontwikkeld, waarvan agents gebruik kunnen maken. Het protocol is er één waarbinnen naming van agent services en agent identifiers beschreven wordt. Agent verplaatsing en simpele agent bescherming werden in acht genomen. Op het ogenblik wordt de standaard nog uitgewerkt. In de toekomst zouden een query mechanisme, agent authentification, access permissions en agent registry (!) voorhanden moeten zijn.

4.2 Een Aglet

Omwille van al de beschreven voordelen die de Java VM biedt heeft men een agent gerepresenteerd als een instance van een klasse. De interface tot deze klasse bestaat zowel uit methoden die imperatief een actie uitvoeren op de agent (Creation/Disposing, Activating/Deactivating, Dispatching/Retracting), als uit methoden die aangeroepen worden als dergelijke actie plaats heeft gegrepen. Verder is er nog de `handleMessage()` methode die aangeroepen wordt door de proxy van de agent als er een bericht binnenkomt.

Onderstaande voorbeeld Aglet toont aan hoe een agent een nieuwe agent creëert, deze verplaatst en er berichten naar stuurt.

```
public class test3 extends Aglet
   {int i=0;
   boolean vader;
   AgletProxy child;  //vader heeft een referentie naar zijn kind
   public void onCreation(Object init)
      {if (vader=init==null)
         child=(AgletProxy)getAgletContext().
               createAglet(new URL("file://awb/aglets/public/")
               ,"MailSys.test3",this);}
   public void run()
     {if (vader)
       {for (int i=0;i<100;i++)
         child.sendAsyncMessage(new Message("hello"));
       child=child.dispatch(new URL("atp://134.184.49.51"));  (***)
       for (int i=0;i<100;i++)
         child.sendAsyncMessage(new Message("hello"));}}
   public boolean handleMessage(Message msg)
     {if ("hello".equals(msg.kind))
```

```
    System.out.println("Hello World");
  return true;}}
```

Aglets verhuizen in dit framework volgens het looping model. Men maakt natuurlijk gebruik van de Java serialisatie om objecten (zowel agent objecten als berichten tussen agents) over te sturen. Code wordt overgebracht door een zelf geschreven classloader. De Aglet Workbench bied zowel synchrone als asynchrone communicatie [6] tussen agents.

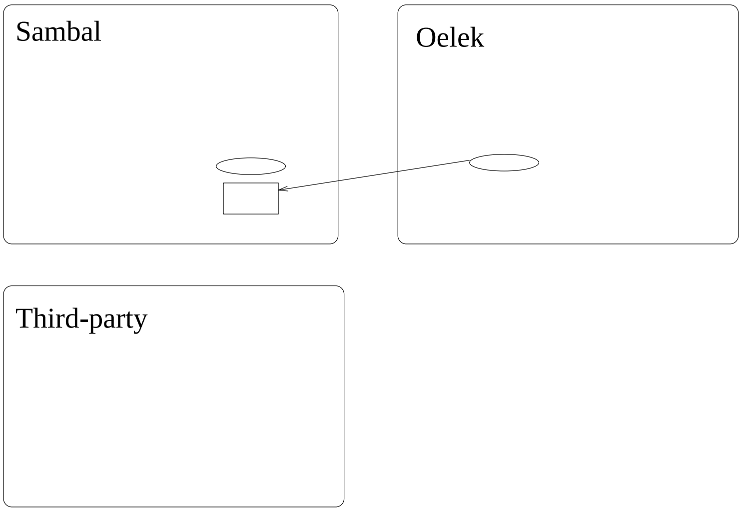
Persistentie wordt verwezenlijkt door alle Aglets die persistent moeten zijn lokaal te runnen. Dit wil zeggen dat deze agents zich niet kunnen verplaatsen. Signalisatie tot persistentie wordt gegeven door activation & deactivation methoden. Zolang een agent gedeactiveerd is kan hij op een of ander spool-medium bewaart blijven.

4.3 Proxy agents

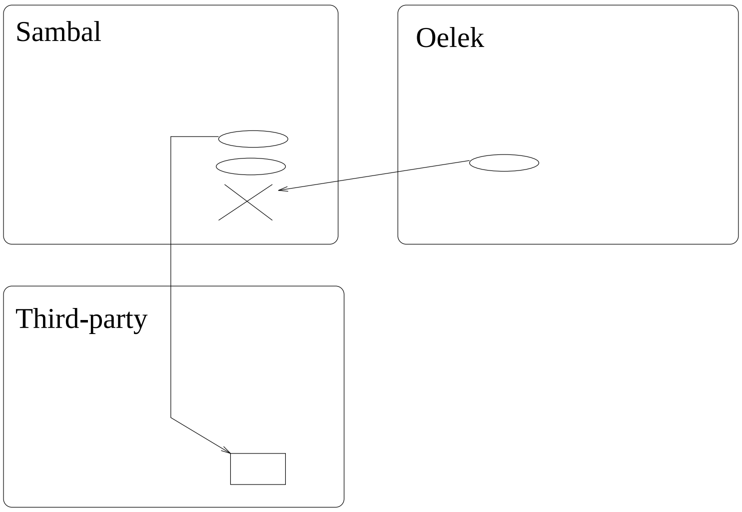
In dit framework treffen we proxy agents aan. Deze proxies laten toe te filteren welke berichten binnenkomen. Hier kunnen we bijvoorbeeld kiezen of we alle berichten van 'mummy-agent' negeren, of alle berichten van 'teacher-agent' omleiden naar 'mummy-agent'. Het systeem verplicht de programmeur al de vragen die hij normaal direct aan de Aglet zou stellen te richten aan de bijhorende agentProxy. Dit is vooral handig als we een gedeactiveerde agent weer willen activeren. Een agentProxy is altijd resident in een agentContext en kan deze niet verlaten.

Zoals we beschreven hebben kan een proxy gebruikt worden om location transparency te voorzien. We zullen dit nu trachten te implementeren. Eerst moet opgemerkt worden dat, als een Aglet van agentContext wijzigt, zijn bijhorende proxy van een `localproxy` naar een `remoteproxy` wijzigt. De grote onvolkomenheid hierin schuilt in het feit dat deze verandering van proxy absoluut niet transparant aanwezig is. Stel bijvoorbeeld dat we een agent hebben op machine 'Sambal' en op 'Oelek' hebben we een AgentProxy naar deze agent gedefinieerd. Op de figuren worden proxies voorgesteld als aureooltjes (ellipsen), de agents zijn de doosjes...

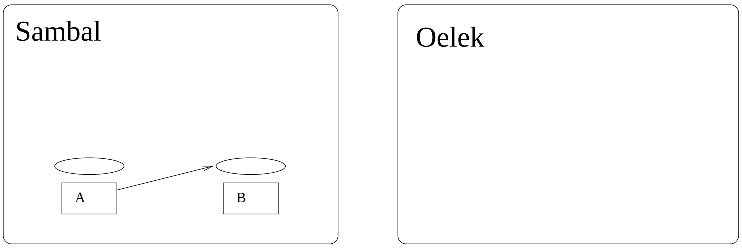
[6] Dit is overigens zeer mooi geïmplenteerd door als return value van een `sendAsyncMessage` een `FutureReply` weer te geven.



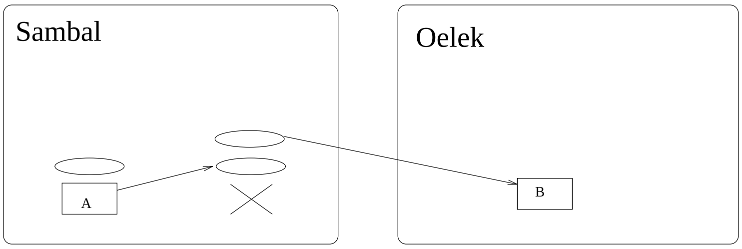
Als we de agent nu verhuizen naar een derde machine zullen we merken dat de proxy op 'Oelek' ongeldig geworden is omdat de proxy daar blijft wijzen naar een niet aanwezige agent op Sambal.



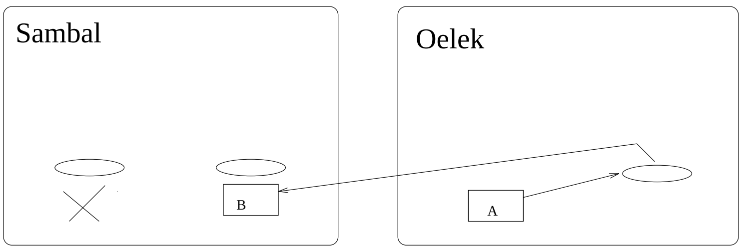
We hebben dus helaas geen location-transparency. Een nog groter probleem bestaat erin dat de referentiele transparantie die we met proxies proberen te halen zelfs binnen de virtuele machine niet geldig is. Als we een agent verhuizen door naar zijn proxy `dispatch` te sturen kunnen we nadien aan het oorspronkelijke proxy object niets meer vragen omdat de referentie binnen de virtuele machine naar de proxy ook gewijzigd is. (Dit zie je in de voorbeeldcode waar staat `child=child.dispatch(...)`). We krijgen dus een *nieuwe* proxy weer na een dispatch !



Na de verhuis van Aglet B van Sambal naar Oelek geeft dit binnen Sambal een referentie naar een proxy die niet meer geldig is omdat er een nieuwe gemaakt is.



Een ander probleem bestaat erin dat agentproxies niet te serialiseren zijn. Dit zou nochthans een zeer elegante manier zijn om de verplaatsing van een agent op te vangen. Als een agent bijvoorbeeld verhuist en al de proxies die hij nodig heeft meegenomen worden, zouden deze nieuwe proxies zich bij aankomst weer kunnen verbinden met het juiste remote object. Dit wil zeggen dat bepaalde `localproxies` na een verplaatsing `remoteproxies` worden. Bijvoorbeeld bij de verplaatsing van agent A naar 'Oelek', waar zowel A als B op Sambal stonden, zouden we volgend resultaat hebben:



We moeten dus duidelijk concluderen dat proxies binnen de Aglet Workbench enkel nut hebben als interface-omleiders. Location transparency zullen we ergens anders moeten zoeken. Laat ons daartoe eens kijken naar de naming

4.4 Naming

Naming in dit systeem wordt op het ogenblik nog op algoritmische basis gedaan. Elke agent krijgt een computer-gegenereerde naam die de string representatie is van een hexadecimaal getal. Uniciteit wordt verzekerd door voor elke identifier een hostname te plakken. (ter illustratie: atp://igwent1.vub.ac.be/0x52FC3DE is een typerende naam voor een agent) Wanneer een agent van plaats verandert, wijzigt zijn naam.

Zoals reeds werd opgemerkt is een groot probleem in dit framework hoe we een bepaalde agent kunnen bereiken. Dit probleem wordt nog vergroot omdat we geen enkele agent kunnen benoemen als we code aan het schrijven zijn omdat *elke* agent identifier gekozen wordt door het systeem. We hebben met andere woorden nood aan één of andere indirectie zoals een name server. Doch hier treedt het probleem op dat we eveneens niet in staat zijn de centrale agent die dienst doet als name server te benoemen en dus zijn we zelfs niet in staat onze name query naar iemand te richten. Een oplossing hiervoor is ervoor te zorgen dat er één factory is waarbinnen al de agents geïmplementeerd worden. Deze factory zou dan aan elke agent die hij creëert zijn eigen adres moeten meegeven zodat het kind de centrale name service kan bereiken.

Zoals men zelf vermeldt zou deze voorziening, een name service dus, aanwezig moeten zijn in een nieuwe uitgave van de Aglet Workbench.

4.5 Security

Hoewel deze naming een vreemde zaak is en veel nadelen heeft, zorgt ze ook voor een zeer originele benadering van security.[7] Door elke agent identifier algoritmisch te genereren zijn we niet in staat zomaar naar andere agents berichten te sturen omdat we gewoon de naam nog niet kunnen opvragen. Dit wil zeggen dat we steeds kunnen weten *wie* naar een agent refereert en dus wie potentieel kwaad kan berokkenen. Stel dat een kraker toch de naam van een agent heeft kunnen achterhalen, dan zal die naam de eerst volgende keer dat de agent verhuist weer ongeldig worden. Dit belet natuurlijk niet dat het onmogelijk is dit systeem te kraken, maar de moeilijkheid om een agent te benoemen en te blijven volgen vormt een onmiddellijk beletsel. Een eventuele indringer zal een niveau lager moeten gaan en protocols analyseren om zo fake agents te introduceren die leven op basis van bestaande lokaal runnende agents (die reeds voldoende permissies verkregen hebben).

4.6 Conclusie

We hebben nu het eerste framework onder de loep gelegd en gemerkt dat het zodanig veel restricties oplegt dat het onhandelbaar is in het gebruik. Dat we een agentname niet zelf mogen kiezen is het grootste gebrek, dat wel kan leiden tot een grotere security. Location transparency is onbestaande en bijna onmogelijk te implementeren, zelfs met het gebruik van de voorziene AgletProxies. Deze proxies kunnen enkel gebruikt worden om de interface tot een agent te reguleren.

[7] Ik ben er mij van bewust dat het misschien niet de bedoeling was dergelijke methode te gebruiken om security te implementeren, maar de mogelijkheid was het vermelden waard.

Er zijn geen voorzieningen die toelaten meta data en interface beschrijvingen te exporteren.

Het belangrijkste wat we uit dit framework kunnen onthouden is de wijze waarop security behandeld worden.

5 Mole

5.1 Overzicht

Mole [9] is een agent framework ontwikkeld aan 'the Institute for Parallel and Distributed Computer Systems' aan de universiteit van Stuttgart onder leiding van professor Rothermel.

Wat Mole betreft gaat men er vanuit dat mobiele agents, tesamen met hun mogelijkheid tot verplaatsing naar afgelegen hosts, moeten gerund worden in een veilige omgeving. Op deze afgelegen plaatsen/hosts vraagt men services die men lokaal niet onmiddellijk kan benaderen. Uiteindelijk zal men met behulp van agents aan client code distributie doen. Meer bepaald kan agenttechnologie gebruikt worden door klanten om ingewikkelde taken te delegeren naar autonome agents op andere hosts, die deze agents dan asynchroon kunnen runnen. Na het verzamelen en berekenen van de resultaten kan al de data teruggestuurd worden naar de klant.

Mole wordt op het ogenblik gebruikt in een aantal subprojecten zoals AIDA, ASAP en ATOMAS. In AIDA wil men bestuderen hoe groepen van agents zich gedragen en hoe deze in een veilig omgeving gerunt kunnen worden. ASAP loopt in samenwerking met het zwitserse Media project, waarmee men een commerciële databank en de documenten daarin bevat, beschikbaar wil maken. Het praktische onderdeel dat ASAP zou moeten aanbieden is een uitvoeringsplatform voor mobiele agents. In ATOMAS wil men een soort Linda-systeem (een systeem zoals de JavaSpaces, uitgelegd op pagina 17) implementeren door gebruik te maken van Mole.

5.2 Een Mole

Een Mole is een agent gemodelleerd als een cluster objecten zonder referenties naar andere objecten. Die groep als geheel kan verplaatst worden van de ene host naar de andere. Dit wil zeggen dat alle objecten waarvoor een agent direct of indirect kan refereren, de transitieve afsluiting van het main agent object, zijn eigendom zijn.

Dankzij dit eilandconcept kunnen agents door serialisatie gemakkelijk van de ene machine naar de andere verplaatst worden. Moles verhuizen in dit framework volgens het looping-model, zoals te verwachten is, gezien de voorzieningen die de Java VM biedt. De code wordt op het ogenblik net zoals in de Aglet Workbench overgebracht door een zelf geschreven classloader.

Deze mobiliteit laat toe dat agents vrij bewegen naar een zelf gekozen lokatie. Een lokatie komt overeen met de URL van een host. Agents kunnen en moeten zichzelf registreren bij een name service die enkel lokaal draait. Laat ons een voorbeeld geven van een typische Mole, gebaseerd op een voorbeeld geschreven door Fritz Hohl:

```
public class Flooder extends UserAgent
  {public Flooder() {}
  public Flooder(String einString, AgentName aName)
    {selfdescription = einString;
    myname = aName;}
  public void start()
    {Flooder son = new Flooder("son of " + selfdescription, myname);
    Engine.debug("Flooder: my name is " + myname.toString());
    try {Thread.sleep(5000);} catch (Exception e) {}
    actuallocation.createAgent(son);}}
```

Men ziet in dit voorbeeld dat een Mole een public constructor heeft die twee parameters neemt. Deze constructor wordt gebruikt om een Mole te maken met een bepaalde naam en beschrijving. De `start` methode wordt opgeroepen na een migratie of creatie en zal in dit voorbeeld onmiddellijk een nieuwe flooder maken. Na het maken van de agent moet hij nog geregistreerd worden binnen het omringende agent systeem. Dit wordt gedaan door `createAgent` aan te roepen.

5.3 Communicatie

In Mole zijn messages expliciet gemaakt. Als we van agent A naar agent B een bericht willen sturen creëren we een message object dat we meegeven aan de agent engine om op te sturen. Dit is op zich asynchroon omdat we niet wachten op onmiddellijk antwoord. Maar zoals algemeen geweten, kunnen we hier synchrone message passing over leggen. Een communicatie voorbeeld vindt u hieronder.

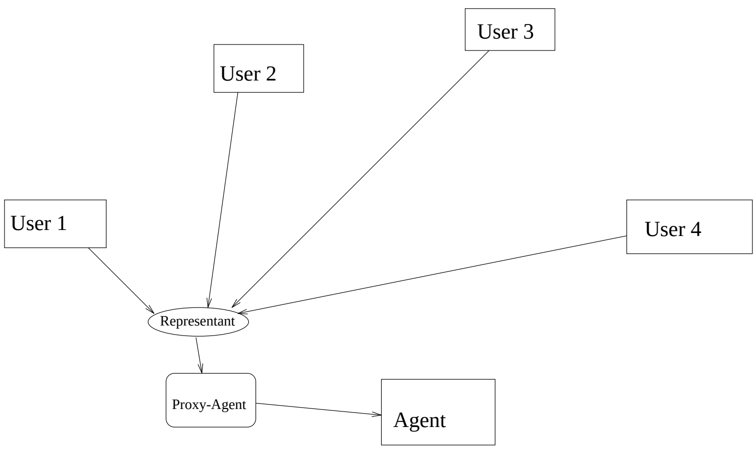
```
public class MessageTester extends UserAgent
  {static messagenumber=0;
  public MessageTester() {}
  public MessageTester(String einString, AgentName aName)
    {selfdescription = einString;
    myname = aName;}
  public void start()
    {Message m=
    new Message(myname, actuallocation.locationName(),
      new AgentName(1,2,3,4,5,6,7,10),
      new LocationName("location2.mole.informatik.uni-stuttgart.de"),
      1,"Test");
    m.messageid = messagenumber++;
    actuallocation.message(m);}
  public void receiveMessage(Message m)
    {Engine.debug("MT Test from "+ (m.sender).toString());}}
```

Een message wordt geconstrueerd door zowel van de zender als de ontvanger de naam en positie op te geven. Wat hierin onmiddellijk opvalt is dat we net zoals in de Aglet Workbench de lokatie moeten opgeven naar waar we een bericht sturen. In dit framework hebben we dus helaas ook geen onmiddellijke location transparency want als de agent beweegt moeten alle eventuele gebruikers weten naar waar hij verhuisd is.

5.4 Proxies

Zoals we weten kunnen proxies gebruikt worden om location transparency te voorzien. Dit framework heeft helaas geen proxies. Wat we wel kunnen is berichten automatisch laten forwarden naar een andere agentname als een bepaalde agent niet meer lokaal aanwezig is. Dus als agent x van plaats A naar plaats B verhuist is, kunnen we op A een representant achterlaten die alle berichten automatisch forward naar de nieuwe agentnaam op plaats B. Men is dus in staat lokaal een representant te maken voor een remote agent. De representant is een Mole die we zelf nog moeten implementeren. We zijn dus in staat zelf te experimenteren met het updaten van positie-gegevens.

Representanten bieden de mogelijkheid een reeks van algoritmes te testen. We kunnen bijvoorbeeld een agent maken die slechts één representant heeft staan op de masterlokatie en aan iedereen die een referentie naar de Mole nodig heeft, het adres van de master-representant weergeven. Deze master representant zal dan de berichten doorsturen naar de juiste agent in kwestie. Eigenlijk hebben we zo toch onze eigen proxy, onder de vorm van een agent, gecreëerd. In het hoofdstuk routing wordt een andere mogelijkheid besproken.



5.5 Meta level

Het Mole framework biedt de mogelijkheid Moles en de toegang tot Moles te monitoren, met behulp van MoleViews en MoleProts. Een MoleProt is een monitor die voor elk bericht dat binnenkomt voor de agent een kopie van dat bericht krijgt. Zo kunnen we dus het gedrag van een agent monitoren. Dit kan helaas alleen als we de agent die we willen monitoren laten afstammen van `MoleViewAgent`. Met behulp hiervan wil men een grafische representatie maken van de binnenkomende/buitengaande berichten en zo performantietesten visualiseren.

Om agents op te zoeken hebben we de mogelijkheid de interface tot een agent op te vragen met behulp van de functie `serviceProvidedBy`, en analoog hiermee `serviceProvidersOf`. Een extra stuk meta informatie kan gestockeerd worden in de `selfDescription` field van de agent. Dit is uiterst handig als we resources, die een agent gebruikt, moeten herbinden na een verplaatsing.

5.6 Conclusie

In dit hoofdstuk hebben we het Mole framework onderzocht. Dit framework bied geen location transparency aan, maar we zijn wel in de mogelijkheid er zelf voor te zorgen. Dit onder de vorm van zelf geschreven proxies. We kunnen agents opzoeken door lokaal de providers van een bepaalde service op te vragen. Deze vorm van meta data is een sterk pluspunt voor dit framework, zeker wanneer we aan resource binding willen doen. Over persistentie is niet nagedacht.

6 Infosphere

6.1 Overzicht

De Infosphere is een agent framework ontwikkelt door de Caltech Infosphere Group (in California). Onderstaande is vrij vertaald en geciteerd uit 'The Caltech Infosphere Project: an Overview' [11] en beschrijft wat men wil aanvangen met het framework:

"Iedere gebruiker op het netwerk heeft een Infosphere die gedefinieerd kan worden als zijn huidige status en verzameling middelen die hij heeft. Een hulpmiddel kan de status van medewerkers in een interactie wijzigen. Internet (en specifiek het web) ondersteund dergelijke interacties tussen Infospheres. De gemiddelde gebruiker zijn Infosphere omvat zijn files (persistent storage), zijn tools (browers, object brokers, compilers, editors) en processen die voor hem runnen (notifiers). Een wetenschapper zijn Infosphere kan bestaan uit apparaten zoals microscopen tot en met interfaces zoals secretaresen, collegas in de werkgroep, projectmanagers en zo verder.

Iemand zijn Infosphere verandert wanneer die persoon van plaats tot plaats rijst. De bandbreedte tot een bepaalde service kan sterk teruglopen of verhogen als we op een vliegtuig zitten, per modem inbellen of naar een andere universiteit trekken.

Net zoals iemands Infosphere wijzigt door te verplaatsen, wijzigt iemands Infosphere ook als die persoon zich in een andere rol moet inleven. Wanneer iemand uit zijn rol van coordinator in een multi-instituut research project stapt en in de rol van lesgever stapt verandert zijn Infosphere. De eerste Infosphere omvat instituten en interageert met andere instituten in het researchproject, de budgetten die toegekend zijn, PERT charts etc. De tweede Infosphere omvat interactie met studenten en de werken waarmee ze bezig zijn. Het is zo dat de beide Infospheres niet volledig onafhankelijk zijn omdat sommige delen van hun status gemeenschappelijk zijn voor beide rollen. (bijvoorbeeld de agenda die de persoon in kwestie gebruikt)"

De bedoeling van de Caltech groep is deze interacties tussen Infospheres te modelleren en te virtualiseren door een Infosphere Infrastructure te bouwen.

In tegenstelling tot voorgaande frameworks zijn agents hierin niet mobiel. Niettegenstaande heb ik het framework toch opgenomen in de tekst omdat men zich niet blind gestaard heeft op de mobiliteit maar wel aandacht schenkt aan geheel andere aspecten van agents zoals meta data en communicatie.

6.2 Een Djinn

Een Djinn [8] is de naam die men op Caltech toegekend heeft aan hun agents. Een Djinn bestaat uit een verzameling mailboxen voor asynschrone communicatie en een interne status, die persistent gemaakt kan worden. Laat ons eens naar een voorbeeld kijken. Hieronder staan twee Djinns. De eerste speelt server voor de tweede. Het enige wat aan de server gevraagd kan worden is de datum weer te geven van de host waar hij draait. In de `question` methode zien we hoe de datum opgevraagd wordt en hoe die teruggegeven wordt.

```
public class DateServer extends Djinn
  {boolean wasQuestioned = true;
  static DateServer djinn = new DateServer();
  static {
    djinnTrueName = new DjinnTrueName (
      "Infospheres Group, Caltech", // org
      "W. Tanaka",                  // author
      "infospheres@cs.caltech.edu", // email
      1, 0,              // release 1.0
      "beta1",           // release beta1
      "Fri, Nov 1 1996", // release date
      "DateServer Example Djinn", // name
      "http://www.infospheres.caltech.edu/"  // url
      );}
  public Answer question (Question q)
    {if (q instanceof TimeQuestion)
       {TimeQuestion tq = (TimeQuestion)q;
       Date d = new Date();
       return new TimeAnswer (d);}
    else return new TimeAnswer();}}
```

De client ziet er ver hetzelfde uit, met dit verschil dat de client niet moet wachten op een vraag maar onmiddellijk van start kan gaan door een vraag op te sturen.

```
public class DateClient
  {public static void main (String args[])
    {Lamp lamp = new Lamp();
    DjinnName djinnToSummon = new DjinnName (args[1], args[0],
      8080, "DateServer");
    DjinnName djinnName=lamp.summon(djinnToSummon, null, null);
    TimeQuestion q = new TimeQuestion();
    Answer a = lamp.question (djinnName, q);
    System.out.println ("The remote date is: " +
      ((TimeAnswer)a).getDate());}
```

[8] Distributed Java Infosphere Network Node

6.3 Mobiliteit

Hoewel het framework zelf geen mobiele Djinns ondersteunt is er toch een belangrijke vorm van mobiliteit aan te treffen in het verplaatsen van de berichten. Elk bericht is een afstammeling van een berichtklasse die serialiseerbaar moet zijn. Op het ogenblik is deze serialisering nog zeer expliciet door middel van `writeData` en `readData` methoden, maar het is te verwachten dat bij een nieuwe release van de JDK dit probleem opgelost zal worden door gebruik te maken van de Java serialisation. Als we nu dus een bericht van DateClient naar DateServer willen krijgen moeten we een gepaste instance van een messageklasse (in dit geval `TimeQuestion`) opsturen naar de DateServer-Djinn, die dan op zijn beurt een `TimeAnswer` terugsturen:

```
public class TimeQuestion extends Question
  {public void writeData (DataOutputStream dout) {}
  public void readData (DataInputStream din) {}}
```

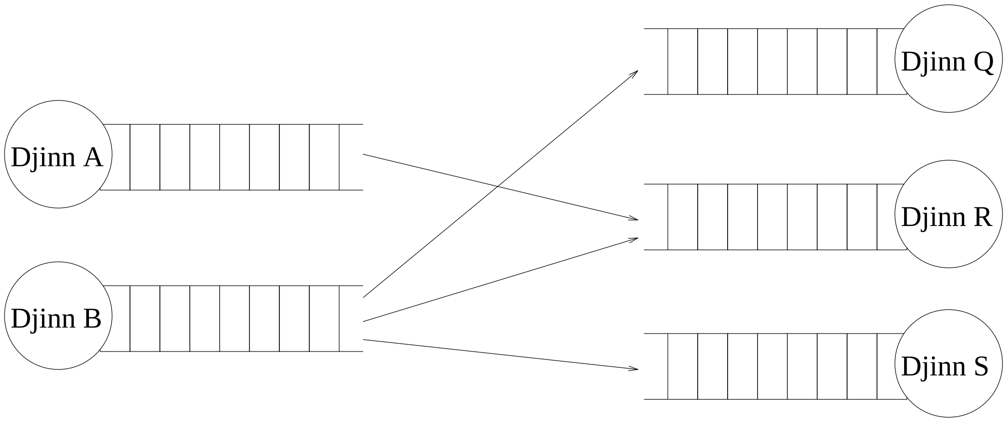
Men kan zien dat de `TimeQuestion` absoluut niets serialiseert en dus niets van potentiële interne status wordt opgestuurd. Het enige dat effectief wordt opgestuurd met dergelijke messageklasse is de type-informatie[9]. De server zal als hij dergelijk bericht ziet enkel moeten controleren op de klassenaam om te weten wat ermee aangevangen moet worden. Het antwoord daarentegen bevat toch wat meer eigen data dan enkel type-informatie:

```
public class TimeAnswer extends Answer
  {private Date date;
  public TimeAnswer () {}
  public TimeAnswer (Date d) {date = d;}
  public void setDate(Date d) {date = d;}
  public Date getDate () {return date;}
  public void writeData (DataOutputStream dout)
    {long time = date.getTime();
    dout.writeLong (time);}
  public void readData (DataInputStream din)
    {date = new Date (din.readLong ());}}
```

6.4 Communicatie

Een Djinn kan zelf mailboxen maken naar believen. De term mailbox refereert zowel naar in-queue's als out-queue's. Elke out-queue kan verbonden worden met meerdere in-queues zoals gevisualiseerd staat in onderstaande figuur.

[9] De klassesignatuur is het basistype van een object. Eventueel bovenliggende typeringen en interfaces zijn allen gebasseerd op de basistypering, zijnde de klassesignatuur.



Elke mailbox kan getypeerd worden. Zo kan een mailbox enkel objecten van type A bevatten en een andere mailbox enkel objecten van type B. Voor elke Djinn is er slechts één maildaemon die zorgt dat alle binnenkomende berichten in de juiste mailbox gestopt worden.

Het type van een mailbox komt overeen met het klassetype van de objecten die we er willen insteken. Als klasse A een afstammeling is van klasse B kunnen we zeggen dat mailbox A afstamt van mailbox B. De maildaemon zal dan automatisch de meest specifieke mailbox kiezen. Verder zorgt hij er ook nog voor dat elk bericht binnen een eindige tijd toekomt. Er zullen anders gezegd geen berichten eeuwig gequeued blijven.

6.5 Persistentie

Agents kunnen in dit framework geactiveerd en gedeactiveerd worden, met het natuurlijke voordeel dat een gedeactiveerde agent op een spoolmedium achtergelaten kan worden. De eenvoud in dit framework, met name dat agents niet mobiel zijn, laat toe de persistentie op doeltreffende wijze op te vangen met eenvoudige `freeze` en `thaw` methoden. Ook hier kunnen we in de toekomst rekenen op serialisering door de Java virtuele machine, hoewel dit natuurlijk niet altijd gewenst is, bijvoorbeeld het doorsturen van een sparse matrix kan veel efficiënter gebeuren als we de serialisatie zelf in de hand nemen. Als we zelf de serialisatie in de hand hebben kunnen we eventueel zelfs een voor de mens leesbare representatie gebruiken.

6.6 Proxies

De idee van proxy agents treffen we ook in dit framework aan, zij het onder de naam 'Lamp'. Met een 'Lamp' kan je een remote agent oproepen. We denken hier natuurlijk onmiddellijk aan Java-agents, doch dit is niet absoluut noodzakelijk. Het is zelfs mogelijk Java klassen te gebruiken die niet afstammen van een Djinn. Voorwaarde is wel dat deze zelf gemaakte klassen de `Summonable` interface implementeren. We kunnen dus een interne Java representatie maken op de lokale machine van ongeveer elk denkbaar remote object, eventueel zelfs niet-Java programma's.

6.7 Naming & meta data

Naming in de Infosphere bestaat uit de hostname met daarachter de naam van de Djinn en het poortnummer waarop men hem kan bereiken. Deze naam is globaal uniek en wordt de DjinnName genoemd. Een Djinn heeft verder een DjinnTrueName die meer informatie bevat over de Djinn. Hierin treffen we de auteur en de organisatie van de Djinn aan. Het emailadres, de releasedate, het versienummer en een leesbare name zijn eveneens beschikbaar. Uiteindelijk moeten we ook de URL waar de Djinn leeft meegeven. Deze info is niet meer echt relevant om een Djinn te benoemen maar ze is wel nuttig om een bepaalde Djinn terug te vinden op het WWW met behulp van search engines.[10] Een extreem voordeel van de Djinntruename is dat we in staat zijn Djinns te mirrorren en te dupliceren. Een aspect waar andere agent systemen vaak te kort schieten.

Nu we toch in de buurt van meta data beginnen komen kunnen we onmiddellijk ook interface beschrijvingen toevoegen aan elke Djinn. Dit kan een beschrijving zijn op mailbox niveau, of dit kan een beschrijving zijn op methode niveau. Meer info hieromtrent vind de lezer in 'Leveraging the World Wide Web for the Worldwide Component Network' [13]. Een typisch voorbeeld vind u hieronder:

```
<META name="component_identifier"
content="http://components.cs.caltech.edu/calendar/kiniry/1.0/">
<META name="description"
content="This is the standard Calendar component used at Caltech.
It provides for interactive and automatic scheduling of meetings
between arbitrary groups of participants at arbitrary locations.">
<META name="keywords"
content="calendar, scheduling, distributed, component, infospheres,
caltech, java, tcl, corba">
```

6.8 Conclusie

De Infosphere is een framework waarbinnen sterk nagedacht is over communicatie tussen agents onderling. Het is tot nu toe het enige communicatiesysteem aangetroffen binnen agent frameworks dat toelaat berichten te typeren en te differentieren op basis van het type.

De meta data die men toevoegd aan Djinns laat toe snel bepaalde services op te zoeken. Het nadeel is wel dat Djinns niet verplaatsbaar zijn, wat wil zeggen dat

[10] Hiervoor moeten we wel de nodige meta data toevoegen aan de webpage waar de Djinn op verblijft met behulp van de META tag.

we geen last hebben om proxies te creëren en location transparency te voorzien, juist omdat Djinns niet kunnen verhuizen.

7 Conclusie

In het eerste deel van deze thesis hebben we besproken wat belangrijk is aan agentframeworks en waar er problemen optreden. Het blijkt zo te zijn dat Java in deze context een onverwacht handig hulpmiddel is om te redeneren over en te experimenteren met mobiele agents. Om ons in te werken in deze materie hebben we een drietal frameworks grondig uitgespit. De eerste noemt de Aglet Workbench, de tweede heet Mole en de laatste is de Infosphere. De eigenschappen van elk van de drie systemen staan hieronder samengevat.

	Aglets	Mole	Infosphere
Mobiliteit	Looping model	Looping model	Niet mobiel
Naming	1. Onmogelijk zelf namen toe te kennen 2. Agent wijzigt van naam na een verplaatsing	Agent verandert van naam na een verplaatsing.	Globaal unieke naming
Location transparency	Geen en het is bijna onmogelijk ze te implementeren	Niet aanwezig maar zelf gemakkelijk implementeerbaar	Ja, in die zin dat een agent niet kan verplaatsen. Nee, in de zin dat *als* we een agent verplaatsen we problemen hebben.
Communicatie	Asynchroon, één in-queue per agent, één out-queue per agent	Asynchroon, één in-queue per agent, één out-queue per agent	Asynchroon, meerdere getypeerde in/out queues per agent
Meta	Geen	Er is de mogelijkheid tot het opvragen van agents voor een bepaalde functie.	Zowel algemene meta data als interface description (IDL) in html-pages
Proxies	Aanwezig (AgletProxy)	Niet aanwezig, zelf gemakkelijk implementeerbaar	Aanwezig (Lamp)
Persistentie	Ja, impliciet.	Nee	Ja, zeer expliciet. Elke agent komt overeen met 1 file.

We kunnen concluderen dat location control het grootste gebrek is aan bestaande agent systemen. Een ander enorm probleem is dat van security en resource misbruik. Het eerste probleem zullen we trachten oplossen. Het probleem van security is er een probleemdomein op zich vanwege zijn complexiteit. We zullen hier dan ook niet verder over uitwijden.

Deel II

Ontwerp van een agent systeem

Samenvatting

Nu we het eerste deel reeds achter de rug hebben, kennen we de nadelen van bestaande agent systemen. In dit hoofdstuk zullen we trachten al de aangehaalde aspecten zo goed mogelijk te benaderen en eventueel op te lossen. Dit houdt in dat we eerst en vooral een stevige location control moeten verwerven, met liefst zoveel mogelijk location transparency. Het zal blijken dat we daar inderdaad in slagen. Nadien zullen we kijken naar de communicatie tussen agents. Uiteindelijk bespreken we het maken, verplaatsen en verwijderen van agents.

8 Location control

Zoals beschreven in sectie 2.6 werken we in een gedistribueerde omgeving waarbinnen agents op elk moment van de ene machine naar de andere kunnen verhuizen. De nodes in het netwerk zijn verbonden door middel van een LAN/WAN, zoals de internet omgeving waarbij de wachttijden tussen de verschillende hosts serieus kan oplopen en de throughput geregeld enorm laag kan liggen. In deze uitgebreide omgeving treffen we gigantisch veel agents aan die op ireguliere basis van de ene machine naar de andere machine kunnen verplaatsen.

We hebben reeds uitgelegd dat het programmeren van agents in Java, best gedaan kan worden door elke agent voor te stellen als een instance van een Java-klasse. De voordelen hiervan worden beschreven in 3. In het kader van deze thesis, waarin we een architectuur voor impliciet mobiele agents willen bouwen, mogen we dus verwachten dat agents volledig location transparant bereikbaar zijn. Dit houdt effectief in dat we elk agent object op elke aanwezige host een globaal unieke naam moeten geven. Meer specifiek willen we dat

- Elke agent een voor de gebruiker verstaanbare naam heeft.
- De gebruiker zelf identifiers kan toekennen.
- Deze naam globaal uniek is.
- Als we berichten willen sturen naar een agent moeten we deze naam kunnen gebruiken. Ook na het verplaatsen van een agent.
- Het verplaatsen van een agent moet zo weinig mogelijk performantie-impact hebben op het sturen van berichten naar de agent.

8.1 Naming

Een **naam** is een symbolische representatie van een entiteit. Dat ding kan een actie zijn om uitgevoerd te worden of een object van een bepaalde soort, bijvoorbeeld een agent. Entiteiten kunnen meerdere namen hebben. Een naam heeft enkel betekenis in een **naam-context**, in dit geval het refereren naar agents. Een **name-space** is de verzameling van alle mogelijke geldige namen. Het **naam-domein** is de verzameling van alle mogelijke dingen die benoemd kunnen worden in een bepaalde context. Een entiteit kan aan een naam **gebonden** worden. **Herbinden** is het toekennen van een nieuwe entiteit aan een eerder gebruikte naam.

Resolutie (ook wel name-lookup genaamd) is het omzetten van een naam naar het ding dat hij representeert. Als resolutie zorgt voor het teruggeven van een nieuwe naam die gebruikt kan worden in een andere/verschillende context spreekt men over een **address**. [4]

8.1.1 Naming van virtuele machines

Aangezien we het in deze thesis hebben over mobiele agents is het noodzakelijk elke verblijfplaats voor een agent een unieke naam te geven. Een agent verblijft altijd in een virtuele machine. Dus als we ooit een agent een unieke naam willen geven moeten we al zeker zijn dat de virtuele machine waarop een agent runt een unieke naam heeft. Dit kunnen we doen door de standaard hostname te nemen en er een poortnummer achter te plakken. Het poortnummer is nodig om meerdere virtuele machines te kunnen laten runnen op één host. Een voorbeeld van dergelijke naam is ketchup.rave.org:8080

We mogen er vanuit gaan dat eens de naam van een VM gekend is deze niet meer gewijzigd mag worden. Dit is een reële eis omdat het veranderen van IP-nummer en hostname altijd pijnlijk is als we een continu, vaak gebruikt, werkend systeem hebben. In het extreme geval dat de naam van de machine toch gewijzigd moet worden dient dit volledig transparant te gebruiken voor iedereen die ooit die naam opgevraagd heeft. We kunnen hier met bestaande middelen reeds zorgen voor persistente namen, bijvoorbeeld door gebruik te maken van DNS. In dit systeem geeft men elke naam opnieuw een naam. Bijvoorbeeld ketchup.rave.org is het adres igwe8.vub.ac.be en deze laatste is een naam voor 134.184.49.8 die op zijn beurt het hardware-adres 0xE8CFF580 refereert. Door gebruik te maken van het Handle-system [5] kunnen we hier nog extra indirecties aan toevoegen die namen toelaat, nog veel langer te bestaan.

In deze thesis zullen we ons beperken tot gewone hostnames omdat de experimenten toch nooit lang genoeg duren.

8.1.2 Naming van de instances

Nu elke virtuele machine een unieke persistente naam heeft, moeten de instances die er runnen ook een aparte naam krijgen. Een instance kan een agent zijn of zelfs een gewoon object. Een instance is een lokale entiteit, het maakt niet uit wat, die een naam moet krijgen. Omdat virtuele machines uniek benoemd zijn en de instances op die machine dit ook zijn is de combinatie van beide, bijvoorbeeld ketchup.rave.org:8080:instance1, een globaal unieke naam.

We streven ernaar slechts één naam en niet meer dan één naam toe te kennen per instance. Mochten we toch toelaten een instance meerdere namen te geven schept dit problemen wat betreft het vergelijken van instances. Nu zijn we in staat instances te vergelijken op basis van hun naam. Als we toch meerdere namen voor hetzelfde object willen kunnen we dit doen door een 'alias' te introduceren die refereert naar de echte instance name. Stel dat we de agent 'Werner' ook 'Ditmar' willen noemen. Dan kunnen we beter 'Ditmar' een alias laten zijn voor 'Werner'. Dus als we naar 'Ditmar' refereren krijgen we onmiddellijk het adress voor Ditmar weer, zijnde Werner. Raarmee weten we perfect wat we moeten weten: over wie het gaat. Het is duidelijk dat we voldoende hebben aan exact één unieke naam per instance.

Ter vergelijking: binnen Java RMI is instance naming ook gebasseerd op de unieke hostname met daarachter een lokaal unieke naam. Bijvoorbeeld: rmi://wextpc2.vub.ac.be/demoServer. De host zorgt hierbij zelf voor het toekennen van unieke namen aan instances door het lokaal draaien van een naming service, rmiRegistry genaamd. Voor de virtuele machine zelf is het hier simpel. Hij heeft een namespace die hij volledig vrij kan gebruiken. De uniciteit naar buiten toe wordt gegarandeerd door de unieke VM-name.

Dergelijke host qualified namen specifieren dus wel degelijk een lokatie. Als men de naam vast heeft weet men direct over welk object op welke machine het gaat. Dit geld natuurlijk enkel als we met niet mobiele objecten werken. Als objecten mobiel moeten zijn zullen we ons moeten afvragen of deze naam mag wijzigen van bijvoorbeeld: igwe8.vub.ac.be:8080:demoServer hernoemen naar igwe1.vub.ac.be:8080:demoServer. Het zal blijken dat dit niet wenselijk is.

8.1.3 Naming van klassen

Tot nu toe was het betrekkelijk simpel. Zwaar steunend op DNS hebben we ervoor gezorgd dat elke machine, en de objecten erop, een unieke naam hebben. Objecten en andere entiteiten op een virtuele machine zijn helaas niet het enige wat we nodig hebben. We moeten ons ook zorgen maken over de naming van de klassen die we instantiëren. Omdat men in Java een classloader heeft die klassen van overal zonder specifieke naam kan laden kan men dus klasse-name-clashes tegenkomen als we een klasse instantiëren. Bijvoorbeeld als in een applet `new Test()` staat moet de klasse 'test' geladen worden. Eerst zal de omgeving dit remote proberen doen en als dat niet gaat zal de lokale klasse genomen worden. Men heeft dus geen zekerheid over de klasse die overeen komt met de naam 'Test': Test kan resolven naar http:/igwe8/~werner/Test.class of naar file:/home/Werner/java/Test.class Dit geeft zeker problemen op het ogenblik dat men een agent van de ene VM naar de ander laat verhuizen, waar men dus eigenlijk geen klassen remote wil laden, of men allesinds niet wil weten van waar ze komen. We willen dat één klasse naam globaal overeen komt met één klasse.

Het is duidelijk dat het idee van een ClassLoader niet orthogonaal staat op de design van de klasse naming in Java. Al de naming wordt gedaan volgens een filesystem (effectief op disk). Het laden van klassen van ergens anders, een netwerk bijvoorbeeld, met een ander naamschema, zoals de classloader toelaat, is een sterke doorbreking van de oorspronkelijke directory-structuur.[11]

Eigenlijk willen we naming niet overdragen aan de classloader maar nog naar een stadium vroeger. Een oplossing is dus classloaders te verbieden of zo veel mogelijk te vermijden en zich enkel te beperken tot de SystemClassLoader. Om dergelijk systeem te kunnen implementeren hebben we een virtueel filesystem nodig dat buiten Java staat en zorgt dat alle nodige klassen op tijd beschikbaar zijn. Hierdoor worden al de class-namingconflicten buiten de virtuele machine gebracht en het bied mogelijkheden tot optimalisatie (die absoluut noodzakelijk zullen zijn). Bijvoorbeeld het mirrorren van de standaard java.lang.*; Een voorbeeldfilesystem kan er als volgt uit zien:

```
/
/java          is ftp://java.sun.com/pub/jdk1.02/classes.zip
/java/lang/*
/java/io/*
/dnx           is ftp://dnx.lr.com/pub/lr1.02/dnx/*
/dnx/lr/*
/dnx/lr/node/*
/vub.we.ig/Collections           staan locaal op disk
```

Het speciale aan dit globale filesysteem is dat er een mounting point opgelegd wordt. Als gebruiker van het systeem zouden we niet in staat mogen zijn ooit ftp://java.sun.com/pub/jdk1.02/classes.zip ergens anders te mounten dan onder /java. Dit kunnen we verwezelijken door op dit filesysteem een juiste permissie-controle en direct aan de basis te zorgen dat elke klassenaam uniek is.

Wat betreft de beschikbaarheid van dergelijke systemen hebben we niets aangetroffen dat voor dit specifieke geval ontworpen is. Desalnietemin kunnen we ons behelpen door op Unix stations gebruik te maken van NFS en onder Windows gebruik te maken van de standaard shared services. Een andere mogelijkheid is het schrijven van een classloader die dit wwfs (world wide filesystem) zelf implementeerd. Dit zou helaas niet toegankelijk genoeg zijn en is veel minder algemeen. Voor de mensen die verder willen met dergelijke filesystemen is Prospero [6] zeer nuttig. We gebruiken het helaas niet omdat het filesysteem

[11] Het CLASSPATH is de meest gruwelijke uitvinding op dit gebied

niet transparant gemount kan worden, wat wil zeggen dat het programma niet interfacebaar is met het 'fysieke' filesysteem dat door de Java runtime benadert wordt.

8.1.4 Naming van agents

Nu we reeds zo ver zijn moeten we zien hoe we agents een naam gaan geven. In de Infosphere heeft elke Djinn een willekeurige naam, zonder direct verbonden VM-name. In de plaats daarvan voegt men een hoop meta data toe die in html-files beschreven wordt. [13] Rekenend op de goodwill van search engines zou men zo een Djinn kunnen terugvinden. Het probleem hiermee is dat als een agent eens van plaats wijzigt de search engines direct outdated zijn. Deze benadering is dus danig ontoereikend.

Het enige waar we op dit niveau, waar we trachten naamconflicten op te lossen, zekerheid over willen hebben is dat elke agent die in de wereld rondloopt een unieke naam heeft. Hoe we deze agent dan terugvinden is een zorg voor later. De voor de hand liggende oplossing is elke agent een instance name te geven (deze zijn reeds globaal uniek). Let wel op ! De naam van een agent wijzigt niet als hij van virtuele machine verspringt. Een Cookie agent gecreëerd op igwe8 met de naam igwe8.vub.ac.be:8080:Cookie heeft dus als hij fysiek verblijft op igwe1 nog steeds de naam igwe8.vub.ac.be:8080:Cookie

Het wijzigen van een agent zijn naam als hij van plaats verhuist is op zich al een vreemd idee. Als we een pot confituur uit kast halen en op tafel zetten verandert die ook niet van naam. Als die toch van 'confituur' naar 'gelei' zou wijzigen kunnen we argumenteren dat we met twee verschillende entiteiten zitten, zijnde de confituur–in–de–kast en de gelei–op–de–tafel. Op het ogenblik dat we de pot verhuizen van de kast naar de tafel houd de confituur–in–de–kast op met bestaan en ontstaat er een volledig nieuwe entiteit gelei–op–de–tafel. Deze hoogst bizarre wijze van redeneren is blijkbaar heden ten dage gemeengoed in agent systemen.

8.2 Routing

In de besproken systemen werd er weinig of geen aandacht geschonken aan het verzenden van berichten. En zeker niet aan –hoe– ze er geraken. Doch hier moeten we toch voorzichtig zijn. We werken nu in een dynamische omgeving waar een agentnaam plots kan overeenkomen met een andere lokatie. De meeste mobiele-agent–systemen gaan er vanuit dat dit zorgen zijn voor de programmeur van de agent en bieden soms ad hoc oplossingen aan. We zullen aantonen dat deze benadering nodeloos blijkt te zijn en dat we een performant routing algoritme kunnen ontwikkelen voor mobiele agents.

In het voorgaande hoofdstuk hebben we een globaal uniek namingsysteem opgezet. Hierbij geldt dat agents hun naam niet mag wijzigen na een verplaatsing, omdat we anders geen persistente referenties naar agents kunnen doorgeven. Een voorbeeld van dergelijke naam was igwe8.vub.ac.be:8080:Cookie. Wat we nu willen is dat we naar deze agent, door enkel gebruik te maken van zijn naam, een bericht kunnen sturen. Als we dus een bericht sturen naar de agent igwe8.vub.ac.be:8080:Cookie moet die, eender waar de agent zich bevindt, toekomen. Het probleem dat hierbij natuurlijk ontstaat is dat we in de naam igwe8.vub.ac.be:8080:Cookie geen enkele informatie hebben van waar de agent zich op het ogenblik bevind.

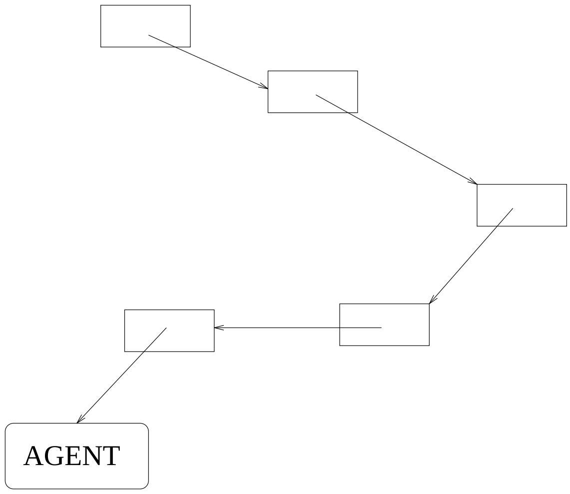
Een eerste gedachte die optreed om dit probleem te benaderen zijn stubs.

8.2.1 Stub

Een stub is een dummy-agent die een bericht dat binnenkomt doorstuurt naar de juiste agent. Soms wordt dit ook een proxy genoemd, hoewel een proxy meer functionaliteit kan bieden, zoals het activeren en deactiveren van een agent, of het monitoren van de berichten die gezonden worden naar een agent. We zullen, als we het over het location-aspect van een proxy hebben, steeds spreken van een stub.

Met stubs is het mogelijk, telkens als een agent verplaatst, een stub[12] (forwarder) achter te laten. De stub is in dit geval het steentje dat Klein Duimpje heeft laten vallen.

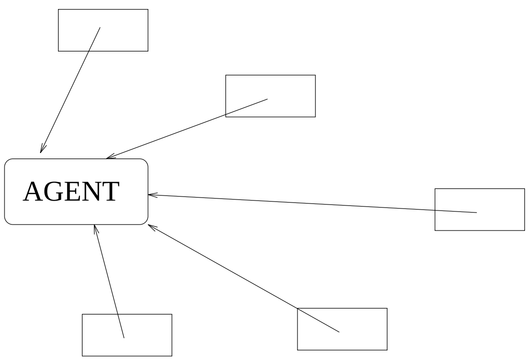
[12] Zoals te zien is op de figuur zouden we in dit geval beter van 'Pipes' spreken dan van 'Stubs'.



De nadelen van dit systeem:

- Dit slorpt resources op als een agent vaak verplaatst
- Het gaat traag als een agent verplaatst over grote afstanden: De masterlokatie moet telkens opnieuw geraadpleegd worden omdat de naam geen enkele informatie bevat wat betreft zijn eventueel nieuwe lokatie.
- Er moet maar één steentje verdwijnen en men kan de agent niet meer bereiken.

Wat men ook kan doen met stubs is ervoor zorgen dat de agent ingelicht wordt telkens een nieuwe stub gecreëerd wordt. Als de agent dan verplaatst kan hij al de gekende stubs inlichten.



Ook dit stub-systeem heeft een paar nadelen:

- Naarmate een agent vordert sleept hij meer en meer balast mee. Hij moet steeds meer stubs inlichten over zijn verplaatsen

- Dit systeem is absoluut niet fouttolerant. Het is zeer goed mogelijk dat stubs invalid worden terwijl de agent ze eigenlijk niet kon bereiken met zijn 'position-changed' boodschap. Dit kan ofwel de agent tegen te houden te verplaatsen, ofwel die stub invalid maken.
- Het is mogelijk dat er positie-updates verstuurd worden die niet meer nodig zijn omdat de stub in kwestie niet meer gebruikt wordt.

8.2.2 Source Routing

Een andere klasse van positiebepalende systemen is het gebruik maken van name servers. Bij deze zoekt men, voor men een paket kan versturen, het adres van de bestemming op door aan het een name server te vragen. Dergelijke routingmethode noemt men sourcerouting. [13]

Het is denkbaar dat sourcerouting bruikbaar is in kleine systemen waar één centrale name server vlot bereikbaar is. In wide area networks is de overlast van het opvragen van een adres helaas te groot om bruikbaar te zijn. Dit kan geremedieerd worden door een reeks lokale name servers te gebruiken en die aan elkaar te koppelen: een gedistribueerde name server. Hier treden echter enorme concurentie problemen op die dit systeem onhandelbaar maken.

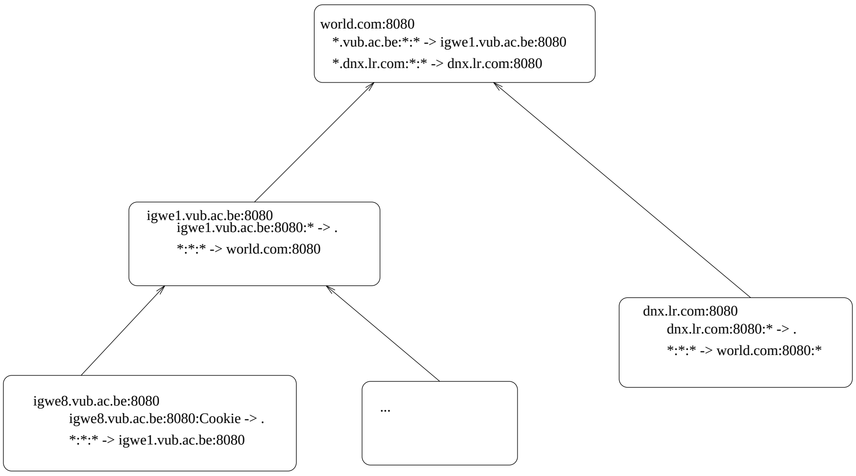
Een andere enorme moeilijkheid is dat tussen het opvragen van de naam, de namelookup, en het gebruiken van de naam, de agent die we willen bereiken misschien niet meer bestaat. Het opvragen van een naam en het routen van een bericht moet als atomaire operatie beschouwd worden.

8.2.3 Hop by Hop Routing

Bij hop by hop routing stuurt men het bericht dat men wil versturen naar de name server. Deze kijkt wat ermee aangevangen moet worden en stuurt het bericht verder naar de volgende name server.

Bijvoorbeeld: We hebben op een bepaalde plaats twee agent systemen runnen. Zijnde igwe8.vub.ac.be:8080 en igwe1.vub.ac.be:8080. Aan de andere kant van de aardkloot is er een systeem dnx.lr.com:8080. Verder zit er nog een hele hoop name-servers en hosts tussen. Voor het gemak doen we alsof er maar één tussen zit, genaamd world.com:8080. Op onderstaande figuur staan de routingtabellen.

[13] Zowel sourcerouting, hop by hop routing als stubs worden beschreven in [14]



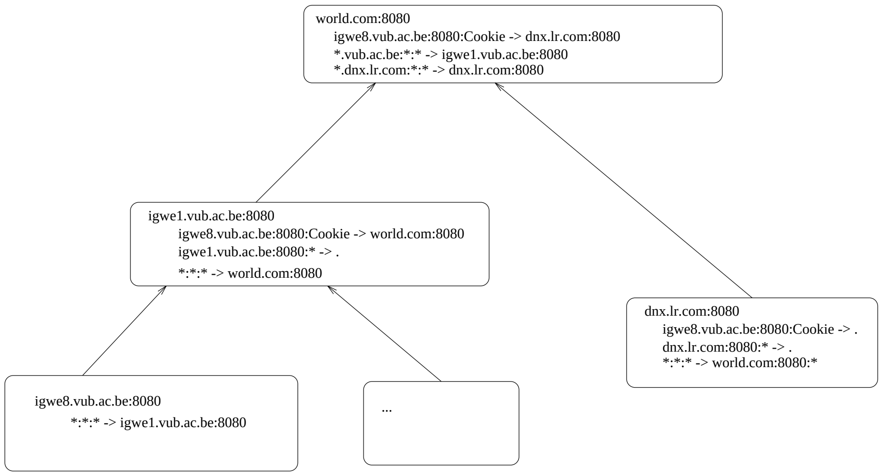
Als nu igwe8.vub.ac.be:8080:Cookie een message wil sturen naar dnx.lr.com:8080:YoYo, dan zal het agent systeem op igwe8 zien dat hij niet weet waar deze zit. Het bericht wordt doorgestuurd naar igwe1, die op zijn beurt niet weet waar de doelagent zich bevind. Igwe1 zal dan het bericht doorsturen naar world.com. Deze ziet dat dnx.lr.com:8080:YoYo te vinden is binnen *.dnx.lr.com:*:* en stuurt het bericht door naar die machine.

8.2.4 Agent gevoelige routing

Als we die hop by hop routing een tikje aanpassen komen we bij een systeem dat zeer bruikbaar is. Eerst zullen we het informeel beschrijven. Nadien zullen we het idee in pseudocode uitgewerkt geven.

Als een agent zich verplaatst van zijn originele positie gebeurt dit via de name server. De name server ziet dat en verandert de regel die naar deze agent verwees. Hij laat hem naar de lokatie wijzen naar waar de agent gestuurd is. Als de agent zo verder uit het domain verplaatst, wordt dit doorgegeven aan de volgende name server die weer de agentpositie update.

Na het verplaatsen van de agent zien de routers er als volgt uit.



De voordelen van dergelijke systeem :

- Als de agent verplaatst van het ene domain naar het andere is hij in het nieuwe domain voor andere agents snel bereikbaar.
- Men kan een agent volledig location transparant uniek benoemen.
- Een agent kan verplaatsen terwijl een bericht verzonden wordt. Het bericht loopt dan wel achter de agent aan maar zal toekomen.

8.3 Correctheid

We zullen om twee redenen formeel aantonen dat dit algoritme werkt. De eerste reden is dat we zo het algoritme formaliseren, de tweede reden is dat we dan zeker van zijn dat het inderdaad werkt in algemene gevallen. Het bewijs verloopt een beetje zoals men in VDM correctheidsbewijzen geeft.

8.3.1 Verzamlingen

Eerst moeten we de nodige verzamelingen introduceren. Dit is vrij eenvoudig en straight-forward. Zoals te verwachten is, hebben we eerst een verzameling agents $\mathbb{A}$ nodig. Deze verzameling is een reeks 'mogelijke' agents. De agents die effectief gebruikt worden zitten in de verzameling $\wp\mathbb{A}$. Aangezien we agents willen benoemen hebben we ook een verzameling namen, zijnde $\mathbb{R}$ nodig. Net zoals bij de agents hebben we hier de verzameling $\wp\mathbb{R}$ die al de namen bevat die gebruikt worden. Agents leven altijd op een of andere host die uit de verzameling $\mathbb{H}$ komt.

Nu we al de mogelijke entiteiten in verzamelingen hebben onderverdeeld dienen we een aantal relaties te definiëren waarmee we allerhande aspecten van een agent kunnen opvragen. We hebben de naam en de host van een agent nodig en op een bepaalde host willen we weten naar waar we onze berichten kunnen routen. Deze relaties zijn zo gekozen dat we dezelfde restricties hebben als in gedistribueerde systemen. Om die reden voeren we geen functie $\mathsf{gethost}$ in die een agentnaam als invoer neemt.

Van een agent kunnen we zijn naam opvragen. Deze functie moet injectief zijn, we willen niet dat twee verschillende agents eenzelfde naam kunnen hebben. Het is duidelijk dat deze functie niet op alle mogelijke startwaarden gedefinieerd is. Daarom kunnen we $\mathbb{e}\mathbb{R}$ (= $\mathbf{dom}(\mathsf{getname})$) beschouwen als al de bruikbare refereerbare agents. We kunnen over een referentieel transparant framework spreken als we $\mathsf{getname}$ niet moeten wijzigen van de ene statuswijziging naar de andere.

$$\mathsf{getname} : \mathbb{A} \rightarrow \mathbb{R}$$

Van een agent kunnen we ook opvragen op welke host de agent zich bevind met behulp van de functie $\mathsf{gethost}$. In tegenstelling tot $\mathsf{getname}$ moet deze functie niet injectief zijn. Het is mogelijk dat twee agents op dezelfde machine runnen.

$$\mathsf{gethost} : \mathbb{A} \rightarrow \mathbb{H}$$

Om op een lokale machine te weten naar waar we een bericht moeten sturen hebben we een router-functie nodig. Deze functie wordt $\mathsf{resolve}$ genoemd en neemt een host en een agentname als parameter. De host is de machine waarop men wil weten hoe men de agent met naam agentname kan bereiken. Als antwoord wordt een host weergegeven naar waar het bericht gestuurd zou moeten worden.

$$\mathsf{resolve} : \mathbb{H} \times \mathbb{R} \rightarrow \mathbb{H}$$

Met deze functies hebben we voldoende om aan te tonen dat het beschreven algoritme correct werkt, maar voor we daartoe komen definiëren we de staat van het globale systeem

8.3.2 Status van het systeem

De staat van het globale systeem is een 3-tupel bestaande uit een $\mathsf{resolve}$-functie, een $\mathsf{getname}$-functie en een $\mathsf{gethost}$-functie.

$$\Theta : (\mathbb{A} \rightarrow \mathbb{R}, \mathbb{A} \rightarrow \mathbb{H}, \mathbb{H} \times \mathbb{R} \rightarrow \mathbb{H})$$

Telkens als we $\mathsf{resolve}$, $\mathsf{getname}$ of $\mathsf{gethost}$ schrijven refereren we naar het eerste, tweede en respectievelijk derde element van het oude tupel. Primes worden gebruikt om een nieuwe configuratie aan te duiden.

We mogen wensen dat elke host een route heeft naar elke mogelijke andere host. Doch dit willen we niet opleggen in de definitie van een configuratie. We zullen aantonen dat, indien we met een goede startconfiguratie vertrekken en we slechts één transitie toelaten, dit systeem consistent blijft.

8.3.3 Verplaatsen van een agent

Deze transitie is het verplaatsen van een agent naar een naburige host. Om de notatie te vereenvoudigen gebruiken we een dag (†). $f\dagger(a \to b)$ geeft een nieuwe functie weer die de oude overridden heeft. De nieuwe ziet er dan als volgt uit:

$$f\dagger(a \to b) = \begin{cases} f(x) & x \neq a \\ b & x = a \end{cases}$$

De transitie ziet er als volgt uit: $C_1 \longmapsto^a_H C_2$ met $C_1, C_2 \in \Theta$, $a \in \mathbb{R}$ en $H \in \mathbb{H}$.

$\mathsf{resolve}' := \mathsf{resolve}\dagger(\mathsf{gethost}(a) \times \mathsf{getname}(a) \to H)\dagger(H \times \mathsf{getname}(a) \to H)$
$\mathsf{gethost}' := \mathsf{gethost}\dagger(a \to H)$

Bovenstaande transitie is enkel geldig als $\mathsf{resolve}(\mathsf{gethost}(a), \mathsf{getname}(a)) = h$

De eerste regel zorgt dat de routing anders verloopt. De enige twee elementen die hier gewijzigd worden zijn te vinden op de nieuwe host en de oude host. De tweede regel zorgt dat de agent inderdaad verplaatst is.

De restrictie laat enkel toe de transitie te gebruiken om een agent te verplaatsen naar een naburige host; als we nu een agent van een host naar een niet naburige host $H \in \mathbb{H}$ willen verplaatsen gebruiken we het volgend algoritme:

repeat $c := \longmapsto^a_H (c)$ **until**$(c = H)$

8.3.4 Invariantie

Hierin definiëren we aan wat het systeem steeds moet voldoen. Hetgeen waarvan we al zeker willen zijn is dat we elke agent die bestaat kunnen bereiken zonder dat er iets gezegd wordt over performantie, noch over concurrentie. Vanuit elke startconfiguratie die we willen gebruiken moeten we bewijzen dat het onderstaande waar is initieel en waar blijft na een transitie.

$$\forall h_0 \in \mathbb{H}, \forall r \in \mathbb{R}, \exists (h_0, h_1, h_2, \ldots, h_n) \quad \text{such that}$$

$$\begin{aligned}
\nexists i, j, i \neq j : h_i &= h_j && (1)\\
\mathsf{gethost}(\mathsf{getname}^{-1}(r)) &= h_n && (2)\\
\mathsf{resolve}(h_i, r) &= h_{i+1} && (3)
\end{aligned}$$

De eerste regel zegt dat we geen onperformante lussen willen zien, de tweede zegt dat het eindpunt van de route inderdaad de agent moet bevatten en de laatste geeft ons de constructie van het pad.

8.3.5 Startconfiguratie

De startconfiguratie waarmee we verder zullen werken is diegene waarbij elke agent verblijft op zijn host die vermeld staat in zijn naam. $\mathsf{hostpart}$ is op dit gebied een hulpfunctie die de host weergeeft die vermeld staat in een agent-name. In de startconfiguratie is reeds een routing voorzien die zorgt dat we elke mogelijke host kunnen bereiken.

$$\begin{gathered}
\mathsf{gethost}(a) := \mathsf{hostpart}(\mathsf{getname}(a)) \qquad \forall a \in \mathbb{e}\mathbb{A}\\
\mathsf{resolve}(h, r) := \mathsf{hostpart}(r) \qquad \forall h \in \mathbb{H}, r \in \mathbb{R}
\end{gathered}$$

$$\begin{gathered}
\forall r \in \mathbb{A}, \quad \forall h_0 \in \mathbb{H} \quad \exists (h_0, \ldots, h_n) \quad \text{such that}\\
\mathsf{resolve}(h_i, r) = h_{i+1}, h_n = \mathsf{hostpart}(r)
\end{gathered}$$

8.3.6 Inductiebasis

Nu zullen we bewijzen dat deze startconfiguratie aan de gestelde eisen voldoet:

Aangezien elke agent een naam heeft gelegen op de host zelf kunnen we een pad maken van h_0 tot h_n met $h_n = \mathsf{hostpart}(r)$. Met de gegeven start weten we dat al de hosts verbonden zijn via de start-resolvefunctie:

$$\begin{gathered}
\forall r \in \mathbb{A}, \quad \forall h_0 \in \mathbb{H}, \quad \exists (h_0 \ldots h_n) \quad \text{zodat}\\
\mathsf{resolve}(h_i, r) = h_{i+1}, h_n = \mathsf{hostpart}(r)
\end{gathered}$$

De rij die hier uit komt nemen we gewoon over als route. (3) is dan onmiddellijk bewezen. (2) kan bewezen worden als volgt:

$$\begin{aligned}
\mathsf{gethost}(\mathsf{getname}^{-1}(r)) &= \mathsf{hostpart}(\mathsf{getname}(\mathsf{getname}^{-1}(r)))\\
&= \mathsf{hostpart}(r)\\
&= h_n
\end{aligned}$$

Dat er geen dubbels voorkomen wordt bewezen als volgt: Stel dat we een lus kunnen creeren: $h_1 \ldots h_i \ldots h_j \ldots h_n$ met $h_i = h_j$. Dan moet er een h_k en $h_l, (k \neq l)$ zijn zodat $\mathsf{resolve}(h_k, r) \neq \mathsf{resolve}(h_l, r)$ wat niet kan aangezien $\mathsf{resolve}$ een functie is en dus geen twee beelden kan hebben.

8.3.7 Inductiestap

Stel dat we van C_i naar C_{i+1} zijn gegegaan via $\longmapsto_H^a$ dan zullen we een rij construeren die voldoet aan de gewenste eigenschappen. Binnen C_i bestond reeds een rij om aan onze bestemming te geraken: $(h_1 \ldots h_n)$.

$\longmapsto_H^a$ kan voldoen aan een paar eigenschappen. ofwel is a onze doelagent, ofwel is a een andere onafhanekelijke agent. Laat ons beginnen met a volledig onafhankelijk:

a onafhankelijk We hebben een transitie $\longmapsto_H^a$ waarbij $\mathsf{getname}(a) \neq r$ en we hebben reeds een rij vanuit de vorige stap C_i. Deze rij kunnen we in zijn geheel overnemen.

(1) is waar omdat, als er in de vorige stap geen lussen in zaten er nu nog steeds geen zullen inzitten aangezien we geen wijzigingen aanbrengen.

$$\begin{aligned}
\mathsf{gethost}'(\mathsf{getname}^{-1}(r)) &= \mathsf{gethost}\dagger(a \to H)(\mathsf{getname}^{-1}(r)) \\
&= \mathsf{gethost}(\mathsf{getname} - 1(r)) \\
&= h_n
\end{aligned}$$

$$\begin{aligned}
\mathsf{resolve}'(h_i, r) &= \mathsf{resolve} \\
&\quad \dagger(\mathsf{gethost}(a) \times \mathsf{getname}(a) \to H) \\
&\quad \dagger(H \times \mathsf{getname}(a) \to H)(h_i, r) \\
&= \mathsf{resolve}(h_i, r) = h_{i+1}
\end{aligned}$$

a is onze bestemming We hebben in dit geval een transitie $\longmapsto_H^a$ die geschreven kan worden als $\longmapsto_{H_k}^{a=\mathsf{getname}^{-1}(r)}$. Hierbij is H_k ofwel een nieuwe host ofwel één die reeds in de rij voorkwam. Als hij reeds in de rij stond knippen we de rij af op H_k, in het andere geval nemen we $k = n + 1$. Omwille van deze constructie staan er geen lussen in de rij en is (1) bewezen.

(2) wordt bewezen aan de hand van het gebruik van de injectiviteit van $\mathsf{getname}$:

$$
\begin{aligned}
\mathsf{gethost}'(\mathsf{getname}-1(r)) &= \mathsf{gethost}\dagger(a \rightarrow h_k)(\mathsf{getname}-1(r)) \\
&= \mathsf{gethost}\dagger(a \rightarrow h_k)(a) \\
&= h_k
\end{aligned}
$$

De laatste eis wordt ook voldaan:

$$
\begin{aligned}
\mathsf{resolve}'(h_i, r) &= \mathsf{resolve}\dagger(gethost(a) \times r \rightarrow h_k)\dagger(h_k \times r \rightarrow h_k)(h_i, r) \\
&= \mathsf{resolve}\dagger(gethost(a) \times r \rightarrow h_k)\dagger(h_k \times r \rightarrow h_k)(h_i, r) \\
&= \begin{cases} \mathsf{resolve}(h_i, r) = h_{i+1} \\ h_k \end{cases}
\end{aligned}
$$

Het eerste geval geeft inderdaad h_{i+1} zoals gewenst. Het tweede geval komt h_k uit. We weten dat dit ligt aan het koppel $(\mathsf{gethost}(a), r)$ dat afgebeeld wordt op h_k. Nu moeten we bewijzen dat dit enkel kan als $\mathsf{gethost}(a) = h_{k-1}$. En inderdaad dit kan niet anders. Eer de transitie $\longmapsto_{h_k}^{\mathsf{getname}^{-1}(r)}$ geldig is moet $\mathsf{resolve}(\mathsf{gethost}(a), r) = h_k$, dit kan enkel als in de voorgaande C_i $\mathsf{gethost}(a) = h_{k-1}$ vanwege $\mathsf{resolve}(h_i, r) = h_{i+1}$

We hebben hier eigenlijk bewezen dat als een agent zich verplaatst het pad dat een bericht volgt ofwel vanachter één element korter wordt, ofwel één element langer wordt.

8.4 Referenties

Dit algoritme werd onafhankelijk van andere research-labo's ontworpen. Voor de volledigheid geven we een aantal referenties naar andere bestaande papers, die pas laattijdig doorgekomen zijn. Als we zoeken naar andere hierarchische systemen zijn referenties [36], [37], [38] en [39] nuttig. De lezer die een gedetailleerde in-depth studie van de gebruikte algoritmes wil uitvoeren neemt het best [40] vast. De location transparency beschreven in deze papers maakt deel uit van een architecturaal groter systeem, waarvan meer informatie gevonden kan worden in [34] en [35].

Een andere benadering waarbij men de statische toer opgaat wordt besproken in 'Efficient Location Transparency in Concurrent Object Oriented Languages' [33], waarin men een Actor computationeel model als geschikt formalisme hanteert.

8.5 Conclusie

In dit hoofdstuk hebben we een routingalgortime ontworpen dat een performante manier om agents te bereiken aanreikt. Eer dit mogelijk was hebben we grondig naar de manier van naamgeven gekeken. In het kort komt het erop neer dat, als men agents en instanties binnen een virtuele machine een *globaal* unieke naam wil geven men moet beginnen met elke virtuele machine een unieke naam geven. Een belangrijk probleem dat daarbij optrad was dat niet alle Java-klassen uniek benoemd zijn. De basis van deze naamconflicten ligt essentieel in het feit dat men meerdere virtuele machines heeft, die elk hun eigen klasse definiëren en gebruiken. Als droom hebben we een filesysteem geformuleerd dat deze klasse–name–clashes buiten de virtuele machine bracht, maar aangezien dit het best mogelijk te wensen geval is, zullen we ons moeten beperken tot het gebruik van classloaders en shared filesystems.

Eens we deze naam-conflicten opgelost hadden, hebben we een aantal routing technieken onderzocht (zoals stubs, name servers en hop by hop routing). Geen enkele methode voldeed perfect aan de gestelde eisen, zijnde een performante location transparante routing voor mobiele objecten. Uiteindelijk hebben we zelf een algoritme geschreven dat op basis van hop by hop routing rekening houdt met de agents die verplaatsen. Om dit hoofdstuk af te sluiten hebben we bewezen dat het algoritme werkt, zonder rekening te houden met performantiecriteria of concurentie.

9 Communicatie

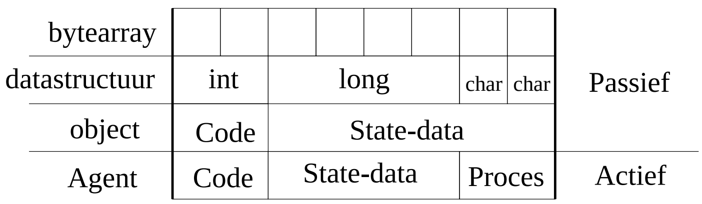
Nu we het gehad hebben over de routing van de data flow tussen agents wordt het tijd dat we een blik werpen naar hoe we dit zullen implementeren en aanbieden aan de programmeur van de agent. Vragen die hier rijzen zijn hoe we berichten voorstellen, hoe we berichten verplaatsen van de ene machine naar de andere, welke beperkingen we opleggen en hoe we berichten afhandelen. In hetgeen volgt betekend "bericht" een hoeveelheid informatie die van de ene agent naar de andere gestuurd moet worden.

Eerst zullen we ons plaatsen aan de zijde van de agent programmeur, die gemakkelijk agents wil kunnen programmeren. In het tweede onderdeel zullen we behandelen hoe we dergelijke interface implementeren.

9.1 Interface

9.1.1 Actieve/Passieve berichten

We kunnen overwegen berichten voor te stellen als byte-arrays zoals UDP-paketjes, doch dit is een low level benadering die niet vaak meer aangetroffen wordt. Als we ons iets hoger plaatsen kunnen we er aan denken datareeksen te gebruiken waarbij we er een expliciete onderverdeling op plakken, ze dus als "structuur" door te sturen. Waarbij rekening gehouden wordt met de volgorde van de bytes. Een hoger niveau van abstractie treffen we aan in objectgerichte talen waarbij het doorgeven van berichten neerkomt op het doorgeven van objecten. Als we dit in een agent systeem willen steken zijn we verplicht code mee op te sturen, maar dat vormt geen enkel probleem meer.



Dit brengt ons in de verleiding agents te gebruiken als message passing tussen agents, dus agents gebruiken als actieve berichten. Doch dit kunnen we niet doen omdat we dan de kern van het probleem ontwijken. Het bericht (de agent dus) is dan nog steeds niet in staat zichzelf aan te melden bij de ontvanger. Verder kan de vraag gesteld worden welk nut gevonden wordt in het actief maken van elk bericht.

Bijgevolg zullen we berichten voorstellen als passieve data, onder de vorm van een instance van een Java-klasse.

9.1.2 Typering van berichten

Altijd en overal waar we berichten doorsturen zijn deze expliciet of impliciet getypeerd. Het type van het bericht bepaalt wat ermee aangevangen kan of moet worden. Dus, of we nu willen of niet, berichten zijn getypeerd. Het is dus niet misplaats ons zorgen te maken over de *uniciteit* van de typering. We willen bijvoorbeeld niet dat twee verschillende data typen ontvangen worden als hetzelfde bericht-type. We willen dus niet dat een float en een double beiden als double worden behandelt.

Omdat we een bericht tussen agents voorstellen als een object hebben we de mogelijkheid de klassesignatuur van het object te gebruiken als typeinformatie. Hierbij moeten we, net zoals bij het laden van code, weer kijken naar de klasse-loading van Java. Een agent kan bijvoorbeeld een bericht van het type Agents.test sturen. Als de ontvanger daarvoor lokaal een andere klasse heeft staan zal de interpretatie van het type, zijnde "dit is een test bericht", wel juist zijn maar het gebruik ervan niet meer. De ontvanger zal ervan uitgaan dat bepaalde andere methoden gedefinieerd zijn op het bericht. Als we een globaal filesystem hebben treffen we hier geen problemen aan.

Nu we hebben uitgelegd dat we een unieke typering hebben zullen we uitleggen dat single inheritance vaak te kort schiet om types te overriden. Wat we nog extra nodig hebben zijn interfaces. Stel dat een bepaald bericht moet verstaan worden door meerdere ontvangers in een andere context, bijvoorbeeld een bericht dat naar de spooler gestuurd wordt bedoelt voor één of andere printer. De spooler krijgt berichten binnen die behandeld worden als 'printjobs' terwijl elke printer op zich zijn eigen berichttype verstaat, bijvoorbeeld 'postscript formaat 1' of 'postscript formaat 2'. Dit kan met behulp van het typesysteem van Java uitgevoerd worden als we twee interfaces definiëren, zijnde: `PrintAble` en `PostScriptAble`. Als we dan een afstammeling maken van het message type dat beide interfaces implementeerd kunnen we het bericht op twee verschillende manieren behandelen.

Het is duidelijk dat als we deze interfacebenadering toelaten we er op voorhand rekening mee moeten houden. We hebben een methodologie nodig om berichten samen te stellen zodat nieuwe gebruikers gemakkelijk hun eigen interfaces kunnen toevoegen. Dit wil zeggen dat *elk* bericht behandelt moet kunnen worden als een interface van een of andere soort. Inheritance is iets dat we in deze context niet kunnen toelaten omdat dit de uitbreidmogelijkheden van objecten restricteerd (hoe raar dat ook mag klinken). Laat ons even een voorbeeld geven van deze boude uitspraak.

Stel dat fabricant A postscriptprinters maakt die berichttypes `PostScript` verstaan, waarbij `PostScript` een afstammeling van `Object` is. De spooler verstaat enkel `SpoolBare` berichten (ook een afstammeling van `Object`). In dit geval zijn we niet in staat een bericht te construeren dat zowel geïnterpreteerd kan worden als `SpoolBare` en `PostScript`

Hetgeen we eigenlijk doen door te zorgen dat elke klassenaam en interfacenaam uniek is, is het opheffen van het lokale Java typesysteem naar een globaal typesysteem. Aangezien we agents in Java programmeren en het bewezen is dat het systeem flexiebel genoeg is, is dit een goede keuze.

9.1.3 Eén/meerdere argumenten

We hebben nu reeds gesproken over het doorgeven van één getypeerd bericht, we zullen nu overwegen om meerdere parameters, samen met hun type, door te geven aan een agent. Dit in overeenstemming met objectgerichte talen waar we in staat zijn meerdere argumenten door te geven aan een methode. Het voordeel van het doorgeven en ontvangen van meerdere parameters is dat we onbenoemde ad-hoc structuren kunnen definiëren. Het nadeel is dan ook dat we een kettingreactie aantreffen als we ooit een parameter moeten toevoegen.

Het is natuurlijk zo dat het gebrek aan meerdere parameters doorgeven gemakkelijk te remediëren is door een berichttype te ontwerpen dat andere berichten bevat, zoals hieronder gedemonstreerd staat.

```
public class ParametersMessage extends Message
  {private Message parameters[];
  public ParametersMessage() {super();}
  public ParametersMessage(Message m1, Message m2)
    {parameters=new Message[2];
    parameters[0]=m1;
    parameters[1]=m2;}
  public ParametersMessage(Message m1, Message m2, Message m3)
    {parameters=new Message[3];
    parameters[0]=m1;
    parameters[1]=m2;
    parameters[2]=m3;}
  public Message get(int n) {return parameters[n];}
  public int Aantal() {return parameters.length;}}
```

Samengevat hebben we genoeg aan het doorsturen van slechts één object, maar zou het handig zijn mocht het inpakken en het uitpakken van meerdere argumenten automatisch afgehandeld worden.

9.1.4 Differentiatie

Differentatie van binnenkomende berichten houd in hoe we binnenkomende structuren en objecten dispatchen naar een bepaalde methode. De gemakkelijkste wijze is geen automatische dispatching te voorzien en alles te laten opvangen door een `handleEvent(M)` dispatcher.

Een beetje meer differentiatie kunnen we introduceren als we de dispatching van meerdere argumenten reeds automatisch afhandelen. Hierbij zouden we methoden `accept1Message(M)`, `accept2Message(M, M)`, `accept3Message(M, M, M)` hebben.

Dit heeft nogaltijd het grote nadeel dat we zelf nog moeten dispatchen op het type van de binnengekomen berichten. Hetgeen echt gewenst is is een automatische dispatching waarbij rekening gehouden wordt met het aantal parameters en de types van de parameters.

Een extra mogelijkheid bestaat eruit aan elk bericht een extra string-veld toe te kennen. Dit stringveld kan dan bepalen welke functie eigenlijk echt aangeroepen moet worden. Bijvoorbeeld de caller roept procedure `doSomething` op met 3 parameters α, β en γ. Dit komt overeen met Remote Procedure Calling.

Hoe we dit binnenkomende berichten ontrafelen en doorgeven aan de agent is een ander probleem. We zijn op het ogenblik niet in staat een bericht om te zetten naar een method-call. Om dit tekort op te vangen kan een meta level glue hoogst aangenaam ervaren worden. Doch op dit ogenblik is deze binnen Java nog niet voorhanden zoals beschreven op pagina 17.

Hieronder treft u een stuk voorbeeldcode aan dat aantoont hoe we dergelijke glue kunnen implementeren door gebruik te maken van de bestaande middelen. Eerst en vooral worden de nodige methoden zoals `sendMessages` en `handleMessage` toegevoegd. De eerste zal zijn parameters encapsuleren en doorsturen. De tweede zal het geëncapsuleerde bericht uitpakken en de juiste methode aanroepen.

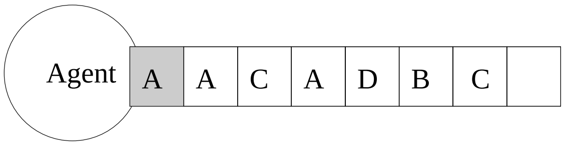
```
void SendMessages(Message m1, Message m2, AgentName Target)
  {SendMessage(new ParametersMessage(m1,m2),Target);}
void SendMessages(Message m1, Message m2, Message m3, AgentName Target)
  {SendMessage(new ParametersMessage(m1,m2,m3),Target);}


void handleMessage(Message m)
  {if (m instanceof ParametersMessage)
     {ParametersMessage p=(ParametersMessage)m;
     if (p.Aantal()==2)
        acceptMessages(p.get(0),p.get(1));
     else if (p.Aantal()==3)
        acceptMessages(p.get(0),p.get(1),p.get(2));}
  else acceptMessage(m);}
```

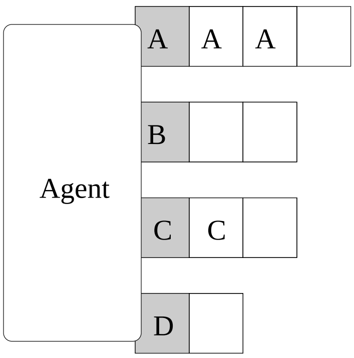
9.1.5 Synchronisatie

We hebben er de voorkeur aangegeven de message passing asynchroon te laten verlopen omdat we grote fluctuaties mogen verwachten in het bereiken van andere agents. We willen hiermee de schrijver van de agent verplichten zoveel mogelijk requests tegelijk uit te vaardigen en niet onmiddellijk te wachten op een antwoord. Hier bovenop kan natuurlijk synchrone message passing gelegd worden indien dit nodig mocht blijken.

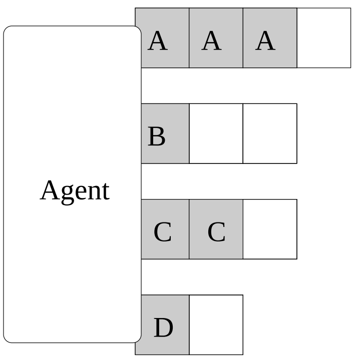
Het afhandelen van berichten hangt deels af van hoeveel threads we tegelijk willen starten en hoe we deze willen synchroniseren. We kunnen één queue maken waarvan steeds het topelement afgehaald wordt, waarbij we niet in staat zijn de eerste elementen over te slaan en onmiddellijk een element van type B op te vragen. Hierbij hebben we slechts één thread die message per message afhandelt.



We kunnen anderzijds ook toelaten meerdere berichttypes tegelijk af te handelen, wat inhoud dat we ook meerdere threads moeten starten. Door op voorhand te specifiëren welke types verwacht kunnen worden zijn we toch in staat in te schatten hoeveel threads maximaal gelijktijdig kunnen runnen. Een bovengrens is het aantal methoden gedefinieerd op de agent klasse.



Een derde en laatste mogelijkheid is elk binnenkomend bericht (onafhankelijk van het type) te laten afhandelen in een eigen thread. Het grootste nadeel van deze methode is dat we hierbij onbepaald veel threads kunnen runnen.



Omdat het meest gebruikte systeem vandaag de dag de enkele queue zonder typering is zullen we dit systeem overnemen. Des te meer de twee andere systemen zonder problemen zelf gemakkelijk geïmplementeerd kunnen worden. De standaard event-handling routine van een agent ziet er uiteindelijk als volgt uit:

```
public void run()
  {try {
    while(keeponrunning)
      {Message m;
      if (!receivebox.empty())
         {m=receivebox.receive();
         handleMessage(m);}}}
    catch (Exception e) {}}
```

9.1.6 Conclusie

Omdat de lijst van gewenste eigenschappen ver van exhaustief was en omdat we niet op voorhand kunnen bepalen wat "nodig" is (in die zin dat het onmisbaar handig is), zullen we vanuit het basisframework enkel toelaten berichten op te sturen en te ontvangen die één voor één afgehandeld worden op basis van het aantal argument-objecten.

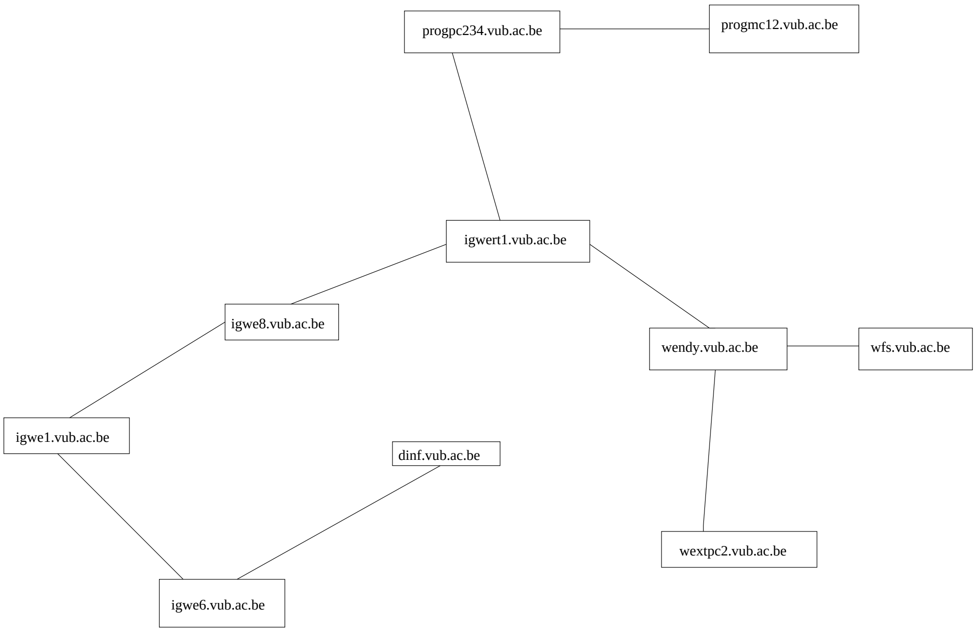
Onderstaande tabel beschrijft met wat we genoeg hebben, wat we zullen implementeren en wat we in de limiet kunnen verwachten.

	Voldoende	Is geïmplementeerd	In de limiet
Representatie	rauwe byte-arrays	Objecten	Actieve objecten/agents
Typering	Klasse-signatuur	Klasse-signatuur & interfaces	Interface beschrijvingen (IDL)
Parameters	Eén argument	Eén argument	Meerdere argumenten
Synchronisatie	Single threaded, asynchroon	Single threaded, asynchroon	Multithreaded, zowel synchroon als asynchroon
Differentiatie	`eventHandler`	`acceptMessage` met meerdere parameters	Methodenaam, aantal parameters en parametertype

9.2 Implementatie

De implementatie bestaat uit twee lagen. De eerste zorgt voor het interne transport van berichten binnen een mailsysteem en maakt gebruik van bestaande netwerk-voorzieningen. Hierbij trachten we een interconnectie tussen agent systemen te verwezelijken. De tweede laag situeerd zich daarboven en creëert een eigen hierarchische topologie nodig voor de name-services. Hierdoor kunnen agents met elkaar communiceren.

9.2.1 Laag 1: Berichtenafhandeling tussen agent systemen

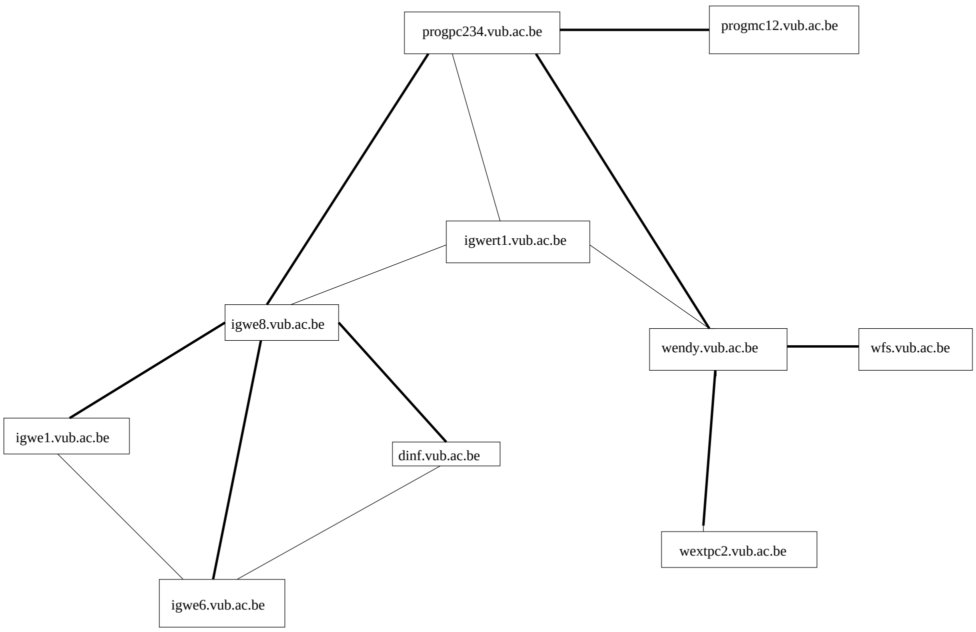


De berichtenafhandeling tussen de agent systemen wordt gedaan met behulp van de info.net package die bijgeleverd is bij de Infosphere van Caltech. Zowel het versturen van agents als het versturen van berichten gebeurt hierlangs.

Het versturen van berichten gebeurt asynchroon. Dit wil zeggen dat men gebruik moeten maken van queueing. Verder zal elk bericht dat verstuurd wordt zal toekomen binnen een eindige tijd. Dit is het allom bekende fairness-begrip. Het voordeel van het gebruiken van deze package is dat deze fout-tolerant is, gebruik maakt van serialisatie en zijn berichten zelf reeds typeerd. Als de ontvanger niet meer bestaat wordt automatisch een foutboodschap gestuurd naar de zender.

Door gebruik te maken van deze package kan een interconnectie verwezenlijkt worden die niet noodzakelijk overeen moet komen met bestaande routers en routings-algoritmen. Dit is exact hetgeen we ermee zullen aanvangen. Dit ziet u in volgende figuur.

9.2.2 Laag 2: Berichtenafhandeling tussen agents



Elk agentsytsteem heeft één info.net.mailDaemon draaien naar waar al de requests (zowel administratieve als andere) naar gestuurd worden. De routing gebeurt zoals beschreven in het voorgaande hoofdstuk en ligt bovenop de info.net package.

Het agent systeem maakt voor elke agent een standaard mailbox aan die gebruikt kan worden om berichten naar de agent te sturen. Na de (her)constructie van een agent kent de agent zijn eigen mailbox. Het verplaatsen van een agent van de ene machine naar de andere levert hier geen problemen. Het systeem vraagt eerst aan het doel-systeem een nieuwe mailbox en stuurt dan de agent in kwestie er naartoe. Het is dus mogelijk om berichten te sturen naar een agent die net nog niet toegekomen is.

10 Levensloop van een Agent

In dit hoofdstuk bespreken we hoe we een agent maken en vernielen, hoe een agent kan uitvoeren en verplaatsen, en hoe processen behandelt worden. Dit wordt vaak als de levensloop van een agent omschreven. Verder bespreken we van welke services agents gebruik kunnen maken en hoe ze die services moeten bereiken.

Dit hoofdstuk bevat een hele reeks voorbeelden die aantonen hoe een agent systeem gebruikt kan worden. We zullen vaak praten over actieve files zijn in plaats van passief. Een *actieve file* is er een waarbij een proces geassocieerd wordt met een hoop data. [14]

10.1 Creëren van agents

Een agent kunnen we maken door aan het lokaal runnende agent systeem te vragen een agent te instantiëren. Elke agent die gecreëerd wordt heeft dus een agentnaam waarvan de instance name naar de lokale machine verwijst en het agent field zelf gekozen kan worden. Deze naam zal hij steeds blijven behouden, zelfs na het verplaatsen naar andere machines. Onderstaande maakt een agent van het type “Agents.fileAgent” en geeft hem de naam “TestFile”.

```
agentSystem.createNewAgent("Agents.fileAgent","TestFile");
```

Het agent systeem zal als hij de agent `Agents.fileAgent` wil maken een standaard instantie van deze klasse aanmaken door `new()` aan te roepen. Nadien zal hij de nodige fields, zoals de agentnaam igwe8.vub.ac.be:8080:TestFile, invullen en mailboxen creëren. We zijn niet in staat parameters mee te geven bij de constructie van een agent. Dit omdat we evengoed de agent een bericht kunnen sturen dat als initialisatie kan doorgaan. Dit maakt de scheiding agent laag en Java laag duidelijker. De initialisatie van een agent is iets dat expliciet moet gebeuren en niet op basis van een taal-feature, zoals constructors.

Onderstaand is een deel van de FileAgent. Hierin ziet u hoe we berichten kunnen versturen naar andere agents en hoe we zelf berichten moeten dispatchen. De creatie verloopt zoals hierboven beschreven. De initialisatie kan gebeuren door het bericht ‘LOCALFILE’ op te sturen. Data kunnen we opvragen met behulp van ‘GETDATA’ waarachter de naam van de ontvanger-agent moet komen. ‘SETDATA’ herschrijft de data die de agent bij zich draagt.

```
public class FileAgent extends Agent
{private String fileName;
public void localFile(String filename)
```

[14] Eigenlijk is dit de omgekeerde wereld. Eerst willen we geen data gebruiken en willen we enkel met agents en processen werken die de bewerkingen op de data zelf uitvoeren. Maar nu beginnen we onmiddellijk met het schrijven van een actieve file om ‘data’ te kunnen modelleren.

```
    {fileName=filename;}
void SetData(DataMessage dm)
    {if (fileName!=null)
    dm.writeToFile(fileName);
    else fileName=dm.writeToFile();}
void GetData(AgentName m)
    {DataMessageImpl dm;
    dm=new DataMessageImpl();
    dm.readFromFile(fileName);
    SendMessages(new StringMessage("DATAOF"),getAgentName(),dm,m);}
public void acceptMessages(Message m1, Message m2)
    {if (((StringMessage)m1).data().compareTo("GETDATA")==0)
      GetData(AgentName.parseString(((StringMessage)m2).data()));
    if (((StringMessage)m1).data().compareTo("SETDATA")==0)
      SetData((DataMessage)m2);
    if (((StringMessage)m1).data().compareTo("LOCALFILE")==0)
      localFile(((StringMessage)m2).data());}}
```

Een eerste drawback die we tegenkomen is dat we een agent niet zomaar remote kunnen aanmaken. Stel bijvoorbeeld dat we een file 'remotefile' hebben, waarvan we weten dat hij op een andere host, bijvoorbeeld igwe1.vub.ac.be:8080, aan te treffen is. Indien we voor deze file een remote agent willen maken zijn we daartoe niet in staat. We moeten altijd een lokale agent maken en die dan verhuizen naar het andere agent systeem. Dit wil zeggen dat we geen remote execution hebben omdat het overbodig wordt geacht.

Er treed natuurlijk wel een ander probleem op. Als een agent op verplaatsing gaat werken en op een andere host agents begint te spawnen, hebben we het probleem dat er name-clashes kunnen optreden. Bijvoorbeeld: als agents igwe8.vub.ac.be:8080:creator en igwe1.vub.ac.be:8080:god beide op igwe7.vub.ac.be:8080 toekomen en daar allebei een agent willen creëeren met de naam igwe7.vub.ac.be:8080:childish hebben we een geweldig naam-conflict. Met andere woorden: als we toelaten dat agents op andere hosts dan hun lokale machine zomaar willekeurig benoemde agents kunnen aanmaken wordt onze name-space aan flarden gerukt.

Om de namespace op een bepaalde machine niet uit te putten zijn we dus genoodzaakt de eigenaar en de creatie-machine beide in één agentname te steken. Hiervoor kunnen we nog steeds de agentnames gebruiken die we beschreven hebben. Zijnde creatiemachine:poort:lokaleagentname. We moeten enkel de lokale agentname expanderen naar eigendommachine:poort:somefield. Het toevoegen van deze extra "created-by" informatie gebeurt automatisch door het agent systeem.

Als nu agent cm:cp:em:ep:agentfield[15] een andere agent creëert op een remote machine om:op dan zal de gecreëerde agent het volgende patroon hebben: om:op:em:ep:somefield.

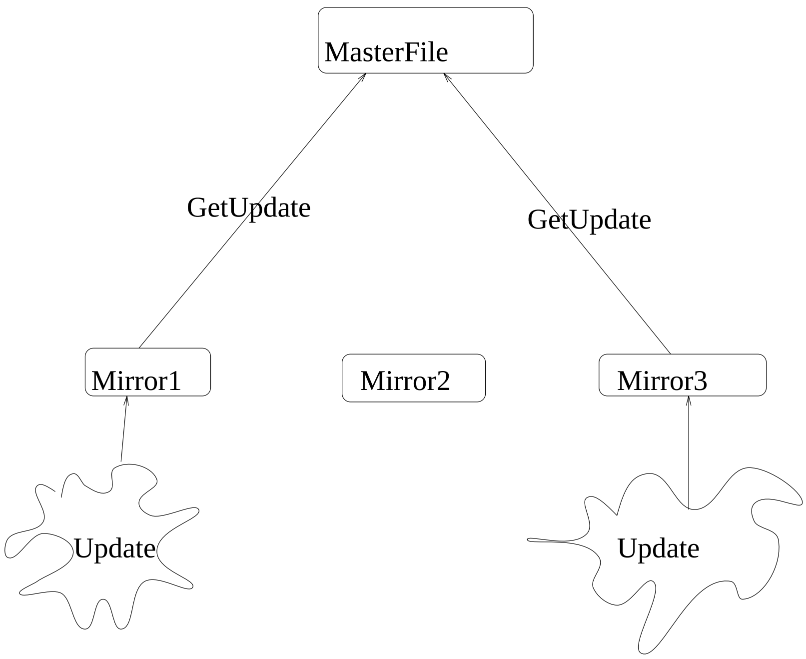
[15] cm staat voor creatiemachine, cp voor creatie-poort, em voor eigendom van machine, ep is de eigendompoort en agentfield is een zelf gekozen identifier

10.2 Wijzigen van "self"

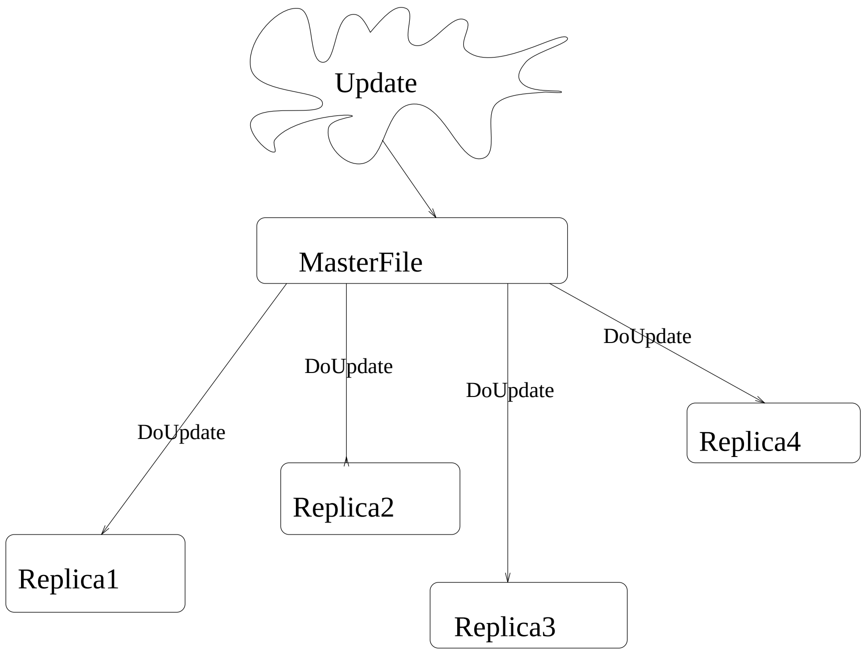
Het agent systeem zorgt dus dat aan een naam een agent gebonden wordt. We kunnen nu ook toelaten een agent zijn naam te herbinden. Dit stelt ons in de mogelijkheid een agent zijn gedrag radicaal te wijzigen. Aangezien een agent zelf verplicht is binnenkomende calls af te handelen kan zijn levensloop op transparante wijze verdergezet worden. Het nut van dergelijke "self"-wijziging zal duidelijk worden in de paragraaf over proxy agents.

We zullen nu een voorbeeld geven dat het nut van deze herbinding duidelijk maakt. Binnen een groot netwerk is het vanwege efficiëntieredenen niet ver gezocht bepaalde files te dupliceren. Hier bestaan grosso-modo twee benaderingen voor. Enderzijds het mirrorren van een file en anderzijds het repliceren door een master.

Het mirrorren van een file is een methode waarbij de client updates vraagt aan een server. Het synchronizerern van de mirror met zijn master gebeurt dus volledig aan client-zijde.



Bij een replica worden de updates zelf opgestuurd door een master-file. Het repliceren van een file bied de mogelijkheid dat de eigenaar van de master alle mirrors van een file update indien cruciale aanpassingen gebeuren.



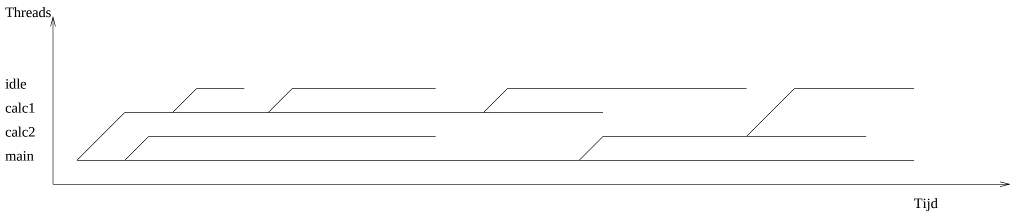
De owner van een replica komt overeen met de owner van de master. Dit in tegenstelling tot een een mirror waarbij de replicatie zelf gecontroleerd kan worden. Nu kan het nuttig zijn op een bepaald ogenblik een replica om te zetten naar een mirror om zelf "owner" te worden van een file. Hiervoor blijk een self-rebind zeer nuttig te zijn. Hieronder staat de methode `becomeMirror` uit de klasse `ReplicatedFileAgent` die demonstreert hoe dit gedaan kan worden.

```
public void becomeMirror()
  {
  // loskoppelen van de master-file
  unregisterReplica();
  // een mirror maken
  MirroredFileAgent mfa=new MirroredFileAgent(MasterFile);
  // mezelf bevriezen en alle processen stoppen
  Freeze();
  // self wijzigen
  changeSelf(mfa);
  // en de nieuwe agent leven inblazen
  mfa.Thaw();
  }
```

10.3 Processen

Nu we reeds agents kunnen maken en herbinden zijn we bijna in staat agents te verplaatsen. We hebben enkel nog een beschrijving nodig van de processen die zich afspelen binnen een agent. Voordat een agent namelijk kan verplaatsen moeten we in de mogelijkheid zijn al zijn gestartte processen te stoppen. We zouden de processen zelf prompt kunnen afbreken en een perfect migratie uitvoeren, maar dat is iets dat buiten het bereik van de Java VM ligt. Dus zijn we verplicht de agent te betrekken in het stoppen en starten van zijn processen.

De `Freeze` methode is er een die voor het stoppen van de agent moet zorgen. Eens de controle teruggegeven wordt van Freeze mogen we alle processen killen en de agent als serialiseerbaar beschouwen. Als we een agent weer willen ontdooien, na het stilleggen van zijn processen, kunnen we de `Thaw` methode aanroepen. In deze methode kan een agent zelf initiatief nemen om meerdere processen te starten, zoals gevisualiseerd staat in volgende figuur.



Zowel `freeze()` als `thaw()` moeten de controle teruggeven eens hun werk verricht is. Bij `thaw()` wil dit zeggen dat de nodige agentprocessen gecreëerd zijn, bij `freeze()` wil dit zeggen dat de agent beschouwd mag worden als stilgelegd.

Ter illustratie tonen we hoe de refresh-loop van de MirroredFileAgent wordt bevroren en ontdooit. Deze refresh-loop is nodig om bijvoorbeeld om het uur de data weer te updaten.

```
public void Thaw()
     {
     super.Thaw();
     refresh=new RefreshLoop(10000);
     refresh.start();
     }
public void Freeze()
     {
     refresh.stopasap();
     super.Freeze();
     }
```

10.4 Verplaatsen van agents

Nu zijn we aanbeland aan de sectie waarin we bespreken hoe een agent verplaatst. Essentieel komt het erop neer dat we de Java serialisatie gebruiken om een agent te serialiseren, eens al zijn processen gestopt zijn. Om de bovenbeschreven fileagent te verplaatsen kunnen we volgende code schrijven:

```
AgentSystem.MoveAgent(Agent,new VMName(ïgwe8.vub.ac.be",8080));
```

We hebben ooit gekozen berichten niet voor te stellen als agents (9.1.1), nu zullen we ook verbieden agents door te sturen als berichten, juist omwille van de moeilijkheden die we hebben om de processen stil te leggen en te herstarten. Het verbod tot het verplaatsen van agents als bericht treffen we ook aan in het Actor computationele model waarbinnen we geen actors mogen versturen als bericht. (Zie [32])

Dit niet kunnen doorsturen van agents als bericht is op zich geen probleem omdat we steeds de naam van een agent kunnen doorsturen. Hetgeen wel een probleem vormt is hoe een agent zich moet loskoppelen van zijn omgeving. Als bijvoorbeeld een actieve webpage (een actieve .html file) een lokale mailer gebruikt dan zal deze webpage na een verplaatsing naar een andere host een andere mailer moeten gebruiken om zijn doel te bereiken.

Dit herbinden van services is iets wat we niet transparant kunnen aanbieden. We zullen, telkens voordat een agent verplaatst wordt, hem inlichten dat hij verhuist gaat worden door de methode `preMove()` aan te roepen. Hierbij kan hij zich ontrekken aan de omgeving waarbinnen hij zich geïntegreerd had. Als een agent aankomt op een host wordt de methode `postMove()` aangeroepen zodat de agent zichzelf weer kan integreregen binnen het nieuwe systeem.[16] Een agent wordt nooit geactiveerd op zijn tussenstations, en dus worden zijn `preMove` en `postMove` methoden ook niet aangeroepen. De enige impact die hij daar heeft is een lokale wijziging van de routing-tabellen.

Ter illustratie van deze `preMove` en `postMove` tonen we hoe binnen deze methoden een message gestuurd kan worden naar een algemene tracer die bijhoud waar de agent zich op dat ogenblik bevind.

```
public class FileAgent extends Agent
{
public void preMove(VMName from, VMName to)
  {
  SendMessage(new StringMessage("LOCATION -> "+ to),
              new AgentName("igwe8.vub.ac.be",8080,"Printer"));
  }
public void postMove(VMName now)
  {
  SendMessage(new StringMessage("-> LOCATION "+ now),
              new AgentName("igwe8.vub.ac.be",8080,"Printer"));
  }
}
```

[16] Over hoe deze integratie moet verlopen willen we niets zeggen, omdat dat een veel te complex domein is waarbinnen het afbakenen van grenzen nogal durft vastlopen.

10.5 Proxy-agent

We kunnen nu agents verplaatsen en aan procescontrole doen. Hetgeen we nu nog willen is dat we aan planning kunnen doen en op een hoger niveau beslissen wanneer een agent naar waar moet verhuizen. We willen dit op een flexiebele wijze aanbieden zonder de distributie te doorbreken. De idee van proxy agents is hier uitermate geschikt voor. (waarbij proxy *enkel* gebruikt wordt als interface-omleiding en niet meer om location transparency te voorzien).

Stel bijvoorbeeld dat we een actieve webpage hebben, die afhankelijk van de binnenkomende requests, verhuist naar de meest vragend host. Enerzijds moeten we in staat zijn een proxy agent op transparante wijze tussen de agent en het omliggende systeem plaatsen. Hiervoor kunnen we de self-rebind gebruiken, zoals geïllustreerd in onderstaand code fragment.

```
public class FileAgent extends Agent
{
  public void setProxy()
     {
     ProxyFileAgent pfa=(ProxyFileAgent)
       createAgent("Agents.agent3.ProxyFileAgent",MyName+"_nonproxy_");
     pfa.setContact(MyName);
     swapNames(this,pfa);
     }
}
```

En anderzijds moet de proxy agent een zekere planning verwezenlijken. De proxy moet aan de hand van het omleiden van berichten bepalen wanneer de agent moet verhuizen. Onderstaande is een hyperactieve manier waarbij de agent, bij elke request die binnenkomt, naar de oorsprong van de request verhuist.

```
public class ProxyFileAgent extends Agent
{
   void GetData(AgentName m)
     {
     SendMessages(new StringMessage("MOVETO"),
       new VMName(m.getHost(),m.getPort()),contact);
     }
   public void acceptMessages(Message m1, Message m2)
     {
     if (m1=="GETDATA") GetData((AgentName)m1);
     SendMessages(m1,m2,contact);
     }
}
```

Met behulp van dergelijke proxy agent kunnen we dus aan caching van resultaten doen, replicatie management kan erin voorzien worden en planning wanneer een agent naar waar moet verhuizen is mogelijk.

10.6 Garbage collection

Het ontbinden is de laatste fase in de levensloop van een agent. Hierbij moet de agent zorgen dat hij al zijn resources die hij gebruikt weer vrij geeft en al de nodige instanties inlicht. Een lokaal werkende agent heeft hier gene probleem mee. Die licht zijn agent systeem in dat hij niet langer meer hoeft te bestaan. Een op afstand werkende agent heeft ook het pad naar hem weg te werken. Al de name servers en routingstabellen die naar hem wijzen moeten geupdated worden.

Om dit te verwezelijken zouden we kunnen opleggen dat de agent enkel mag sterven op de machine waar hij gecreëerd werd, doch we zullen dit niet doen omdat dit een aspect is van de location control dat door het agent systeem zelf afgehandelt moet worden. Agents moeten dus steeds verwijdert kunnen worden door aan het agent systeem `deleteAgent` te vragen.

Het definiëren wanneer een agent de geest moet geven is een aspect dat nader besproken dient te worden. Het blijkt onmogelijk te zijn een garbagcollector te schrijven die zegt wanneer een agent verwijdert mag worden. Dit enerzijds omdat we geen vast gedefinieerde root hebben en anderzijds omdat de agent zijn eigen verantwoording vormt voor zijn bestaan.

10.7 Conclusie

In dit hoofdstuk hebben we expliciet de levensloop van een agent in ons framework beschreven. Samengevat wordt een agent geboren, waarna hij zijn processen kan beheren en aan zijn taakuitvoering werken. Indien nodig kan hij verplaatsen of verplaatst worden. Uitdeindelijk kan hij ook nog vernield worden.

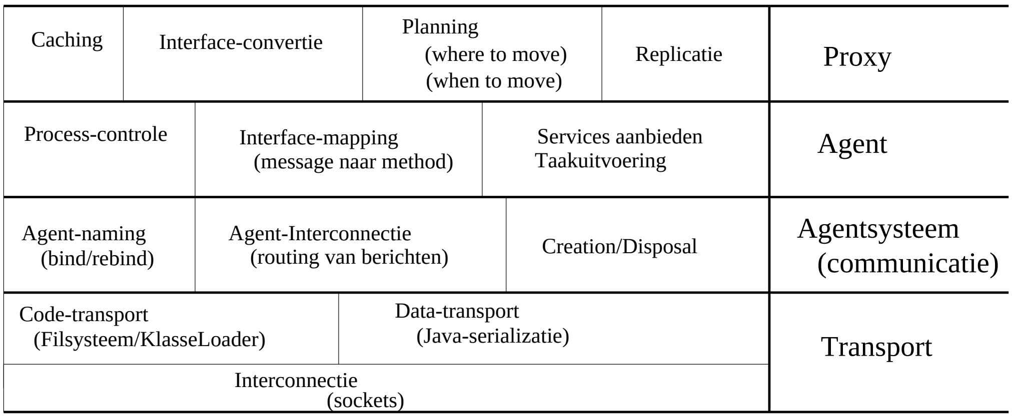
We hebben laten aanvoelen dat het onmogelijk is gebruiker resources en resource binding transparant te maken omdat de gebruiker vaak zelf in de pap wil brokken. Daartoe hebben we `preMove` en `postMove` methoden geïntroduceerd. We hebben ook duidelijk gemaakt dat we onmogelijk een garbage collector kunnen schrijven voor agents, zonder extra voorzieningen toe te voegen aan deze agents. Het bepalen wanneer een agent moet sterven is iets dat door de auteur van de agent zelf moet worden beschreven.

11 Conclusie

In deze thesis hebben we getracht een agent systeem te ontwikkelen op basis van een reeks bestaande agent systemen. Als voertaal hebben we Java genomen, hoofdzakelijk omdat deze voldoende technisch support bied om op hoger niveau te kunnen redeneren over mobiele code. Zonder Java als backbone zou deze thesis er heel anders uitzien, heel wat dikker, omdat we verloren zouden lopen in allerhande low-level technische details.

Om een eigen framework te kunnen ontwikkelen hebben we een reeks agent systemen zoals Mole, de Aglet Workbench en de Infosphere onderzocht. Elk framework had zijn eigen voor en nadelen die we samengevat hebben in Hoofdstuk 7. Uitgaande hiervan hebben we een agent systeem geconstrueerd dat er als volgt uit ziet.

11.1 Resultaat



We treffen op het laagste niveau de transportlaag aan, waarbinnen gefocuseerd wordt op het verplaatsen van data en datastructuren van de ene machine naar de andere. (dit werd behandeld in Hoofdstuk 3). Een niveau hoger treffen we het agent systeem aan dat zorgt voor een unieke globale naming strategie (Paragraaf 8.1) en een globale agent interconnectie. (Paragraaf 8.2) Verder zorgt het agent systeem ook voor het aanmaken en vernielen van agents.

Tot daar is het werk achter de schermen verricht. Vanaf nu moet men als schrijver van een agent tussenbeide komen. Dit houd in dat men zelf de process controle van de agent moet doen. Een aspect waarmee we nog verveeld zitten is de interface mapping[17], die ook nog moet gedaan worden door de auteur van de agent.

Een niveau hoger kan een auteur zijn eigen agent monitoren en bijsturen door een proxy agent te installeren die zich niet meer bezig houd met *wat* er moet

[17] Dit wil zeggen, het vertalen van binnenkomende berichten naar method calls.

gedaan worden maar wel met *wanneer* iets uitgevoerd moet worden. We kunnen verwachten dat caching, interface convertie, planning en replicatie hieronder zullen gebracht worden. De idee van zulk een proxy agent is enkel mogelijk als we toelaten de self pointer te wijzigen, wat er in een agent systeem op neerkomt dat we een agentname moeten kunnen herbinden.

Eén zaak die zeer sterk naar voor kwam in deze tekst is dat we dringend moeten nadenken hoe we geschikte routingtopologiën ontwerpen die toelaten dat multi agents met elkaar kunnen communiceren op een positie onafhankelijke basis. We hebben aangetoond dat dergelijke algoritmen gemaakt kunnen worden en hebben er ook zelf één ontworpen. Nadien zijn we verder gaan kijken naar huidige bestaande gedistribueerde mobiele object systemen. Het is blijkbaar een tendens bij de ontwerpers van agent systemen dat men alle bestaande kennis van gedistribueerde systemen vergeet en moedwillig weigert deze ter hand nemen. Aangezien het duidelijk is dat agent systemen sterke analogieën vertonen met gedistribueerde object systemen kan men er duidelijk in de leer gaan.

Een gedachte die naar voor kwam is dat we, indien we in een Java agent omgeving werken, we geen klasse inheritance willen gebruiken om berichten tussen agents door te geven. Om berichten voor te stellen maken we beter gebruik van interfaces. De idee hierachter bestond eruit dat we niet noodzakelijk in staat zijn af te stammen van het berichttype dat één van de ontvangers verwacht. We hebben dus een stelregel uitgevaardigd die zegt dat elk bericht volledig bereikbaar moet zijn door een interface erop gedefinieerd.

11.2 Gebreken

We zullen nu trachten een aantal gebreken van dit agent systeem op te sommen. De meeste onvolkomenheden zijn te wijten aan de beperkte tijd en aan het veel te open domein van agents. We hebben onszelf op vele plaatsen dus moeten beperken.

Een eerste zaak die verloren gelopen is, is replicatie van agents. De hoofdreden om hiermee geen rekening te houden, is dat replicatie inhoud dat we weten wat het doel van een agent is, dat we dus zijn services kunnen ontdubbelen, en dit op zulk een wijze dat alle duplicaten semantisch consistent blijven. Dit is een topic dat niet onmiddellijk haalbaar is.

Het aspect security is er één waar we zwaar over gekeken hebben. Dit is een domein dat zo verschrikkelijk complex is dat we zelfs niet getracht hebben een minimale security te voorzien. Het probleem van security doet ons trouwens licht denken aan het deadlock probleem. Systeem A ken perfect deadlockvrij zijn, systeem B kan dat ook zijn, doch de combinatie van beide kan een deadlock veroorzaken. Hetzelfde geld voor security. Systeem A kan perfect secure zijn, systeem B kan dat ook zijn, maar systeem C (vertrouwd door beide) kan de security van beide grondig doorbreken.

Andere zaken waarmee we in deze thesis geen rekening gehouden hebben, zijn het specifiëren van protocols en de interfacing met andere talen dan Java. Bij de routing hadden we misschien kunnen nadenken over een mogelijke multi cast feature. We hebben ook maar één routing-topologie bestudeert en niet verder gezocht op mogelijke betere topologiën. De voorgestelde hierarchische stervorm is nogal beperkt.

11.3 Voortzetting van deze thesis

Alhoewel de fysieke verandering van lokatie van software (zijnde programmatuur en toestand) technisch reeds mogelijk is, blijft de migratie strategie een open vraag. In de huidige stand van zaken ligt deze beslissing volledig bij de agent zélf: de agent zal tot migratie overgaan op basis van beschikbare hardware resources, interactie met andere agents of de eigen inwendige toestand. Het gedrag van zo een agent is dus volledig vastgebakken binnen de code en omwille van zijn autonomiteit is deze zijn enige migratiestrategie.

Deze zuiver autonome voorgeprogrammeerde migratiestrategie zal meestal aanleiding geven tot een alles behalve optimale globale verdeling van agents over een computernetwerk. Dit zal dan ook tot gevolg hebben dat de lokale performantie van de agent hieronder lijdt. Dit is analoog aan de klassieke spanning tussen bijvoorbeeld micro- en macro-economie en wordt ook geïllustreerd door het bekende prisonner's dilemma. In beide gevallen moet het gebrek aan globale kennis gecompenseerd worden door een leerproces. Voor onze agents betekent dit dat lokale waarnemingen via verwerving van kennis dienen omgezet te worden in een negotiatie- en migratiestrategie. Klassieke computernetwerken bezitten dit soort van strategie op het niveau van de host machines, van routers of bridges. In ons geval moet deze vervat zijn binnen individuele softwarecomponenten die totaal onafhankelijk van de onderlinge hardware evolueren. Op dit ogenblik ontbreken de formalismen of programmatie-idiomen om dergelijke nieuwe vormen van routing en load-balancing te bestuderen of te verwezenlijken.

Een allereerste doel van een eventueel verder onderzoek zal bestaan uit een uitputtende studie van de parameters die het probleem bepalen: welke factoren bepalen de performantie van een potentieel migrerende agent? Hierbij kan men zowel denken aan performantieaspecten van de beschikbare hardware (geheugen, schijfruimte, cpu-tijd, netwerk toegangstijden...), als aan samengestelde performatiebegrippen zoals de gemiddelde doeltreffendheid van agents of een sommatie van verschillende formele performantie–eigenschappen van agents [24]. Deze laatste moeten toelaten om op een macroscopische schaal formeel te kunnen redeneren over de performantie van agents of clusters van agents.

Eens de relevante criteria vastgelegd kunnen worden, dienen deze gekoppeld aan de handelingen die kunnen genomen worden om zowel de lokale als globale performantie op te drijven. De vraag die zich hier stelt is welke agent wanneer [22] waarheen moet of mag migreren. Om het verband tussen deze criteria en handelingen beter te begrijpen dient de speltheorie zich aan als kandidaat-formalisme. Deze reeds dikwijls toegepaste benadering geeft een goed inzicht in de verbanden tussen het lokale en globale niveau wanneer een reeks wedijverende agents handelingen dienen te ondernemen op basis van een reeks vooropgestelde beslissingscriteria. Om het verband tussen de criteria en de handelingen echter computationeel op te lossen, is een klassiek-algoritmische benadering niet aangewezen: zelfs in het statische geval (begrensde en niet evoluerende topologie) is het probleem NP-compleet. Zodoende zullen we onze toevlucht moeten zoeken tot heuristische benaderingstechnieken zoals genetische algoritmen [25] en/of neurale netwerken [26].

Werner Van Belle
16/4/1997

Referenties

[1] Java Remote Method Invocation, *Sun Microsystems,* http://chatsubo.javasoft.com/current/

[2] JavaObject Serialization Specification PreBeta Release Revision 1.1 *Sun Microsystems,* 11 November 1996 http://chatsubo.javasoft.com/current/

[3] JavaSpaces, *Sun Microsystems,* http://chatsubo.javasoft.com/javaspaces

[4] Gordon Irlam, gordoni@base.com: Naming, verzamelde hyperlinks (23 Mei 1995) http://www.base.com/gordoni/naming.html,

[5] Robert Kahn, Robert Wilensky: The Handle System *CNRI,* (13 Mei 1995) http://www.handle.net/gordoni/naming.html,

[6] Clifford Neumann: The Virtual System Model/ A Scalable Approach to Organizing Large Systems (1992) *the Information Sciences Institute of the University of Southern California,* Prospero(TM) is te vinden op http://gost.isi.edu/info/prospero/

[7] Gihan V. Dias, Graham Cope, Ravi Wijayaratne (gihan@cse.mrt.ac.lk, gacope@cse.mrt.ac.lk, raviwija@cse.mrt.ac.lk) : A smart Internet Caching System *Dept. of Computer Science and Engineering, University of Moratuwa, Sri Lanka*

[8] Dejan S. Milojicic, Fred Douglis, Yves Paindaveine, Richard Wheeler and Songnian zhou: Process Migration *OSF Research Institute, AT&T Laboratories, University of Toronto and Platform Computing,* 10/8/96

[9] Markus straßer, Joachim Baumann, Frits Hohl: Mole - A Java Based Mobile Agent System *IPVR (Institute for Parallel and Distributed Computer Systems), University of Stuttgart,* 23 october 1996

[10] Frederick Colville Knabe: Language support for mobile agents *School of computer sciece, Carnegie Mellon University,* Pittsburg, PA 15213 December 1995

[11] Joseph R. Kiniry and K. Mani Chandy: The Caltech Infospheres Project *the Caltech Infosphere Group,* http://www.infospheres.caltech.edu/

[12] K. Mani Chandy Caltech Infospheres Project Overview: Information Infrastructure for Task Forces *Computer Science 256-80; California Institute of Technology, Pasadena California 91125,* mani@cs.caltech.edu http://www.infospheres.caltech.edu/ 14 November 1996

[13] Joseph R. Kiniry and K. Mani Chandy: Leveraging the World Wide Web for the Worldwide Component Network *the Caltech Infosphere Group,* (October 1996) gepresenteerd op OOPSLA

[14] A. Goscinski: Distributed Operating Systems, The Logical Design *Addison Wesley,* ISBN 0 201 41704 9

[15] Don Gilbert, Manny Aparicio, Betty Atkinson: Intelligent Agent Strategy *IBM Corporation, Research Triangle Park, NC USA*

[16] David L.Tennehouse & David. J. Wetherall: Towards an Active Network Architecture *Laboratory for Computer Science, MIT*

[17] T.Magedanz, K.Rothermel, S.Krause: Intelligent Agents: An emerging Technology for Next Generation Telecommunications ? *TU Bezrlin/GMD Fokus/Berlin; University of Stuttgart,* INFOCOM '96; 24-28 Maart 1996; San Francisco

[18] Danny B. Lange: Agent Transfer Protocol ATP/0.1 Draft *IBM Tokyo Research Laboratory,* 29 July 1996 aglets@yamato.ibm.co.jp http://www.trl.ibm.co.jp/aglets

[19] Leslie L. Daigle, Sima Newell: Intelligent Agents and the Internet Information Infrastructure *Bunyip Information Systems, Inc. Departement of Electrical Engineering, McGill University, Montreal Canada*

[20] Thomas Kreifelts: Social Agents for the Web *GMD – German National Research Centre for information technology; Germany,* http://orgwis.gmd.de:80/projects/SAW

[21] Jeeves: Java Servlets and the Servlets architecture. *Sun Microsystems,* http://www.javasoft.com:80/products/java-server

[22] Richard Goodwin: Reasoning about when to start acting *School of computer sciece, Carnegie Mellon University,* Pittsburg, Pennsylvania 15213-3890

[23] Sven Koenig, Richard Goodwin, Reid G. Simmons: Robot navigation with Markov models: a framework for path planning and learning with limited computational resources *School of computer sciece, Carnegie Mellon University,* Pittsburg, Pennsylvania 15213-3890

[24] Richard Goodwin: Formalizing properties of agents *School of computer sciece, Carnegie Mellon University,* Pittsburg, PA 15213

[25] Melanie Mitchell: Lectures in Complex Systems, Lecture Volume 5: Genetic Algoritms *Santa Fe Institute,* Addison Wesley Publishing Company

[26] John Hertz, Anders Krogh, Richard G.Palmer: Introduction to the theory of neural computation *Santa Fe Institute,* Addison Wesley publishing company

[27] K.Mani Chandy and Adam Rifkin: Systematic Composition of Objects in Distributed Internet Applications: Processes and Sessions *Computer Science 256-80; California Institute of Technology; Pasadena,* 9 Juni 1996

[28] *Autonomy Corporation & Cambridge Neurodynamics UK Office, St Johns innovation Centre,* Cambridge CB4 4WS http://www.agentware.com:80/company.htm

[29] Robert S. Gray: Agent Tcl: A Transportable Agent System *Departement of Computer Science Dartmouth College,* Hanover, New Hampshire 03755 robert.s.gray@darthmouth.edu

[30] Paolo Ciancarini: Coordination Models, languages and applications *University of Bologna, Italy,* 19-21 Februari 1997

[31] Rajive L. Bagrodia, Wen-Toh Liao: Maisie: A language for the design of efficient discrete-event simulations *Computer Science Departement, University of California,* Los Angeles, CA 90024

[32] G. Agha: Actors: A Model of Concurrent Computation in Distributed Systems. *MIT Press* 1986

[33] Woo Young Kim and Gul Agha: Efficient Support of Location Transparency in Concurrent Object Oriented Languages *University of Illinois, Urbana Champaign,* 16 Augustus 1995

[34] Philip Homburg, Maarten van Steen, Andrew S. Tanenbaum: An Architecture for A Scalable Wide Area Distributed System *Vrije Universiteit, Amsterdam,* {philip, steen, ast}@cs.vu.nl

[35] Philip Homburg, Maarten van Steen, Andrew S. Tanenbaum: The Architectural Design of Globe: A Wide-Area Distributed System *Vrije Universiteit, Amsterdam,* Internal report IR-422 steen@cs.vu.nl

[36] Franz J. Hauck, Maarten van Steen, Andrew S. Tanenbaum: A Location Service for Worldwide Distributed Objects *Departement of Mathematics and Computer Science,* Vrije universiteit, Amsterdam, Nederland

[37] Maarten van Steen, Franz J. Hauck, Andrew S. Tanenbaum: A Scalable Location Service for Distributed Objects *Vrije Universiteit, Amsterdam*

[38] Maarten van Steen, Franz J. Hauck, Philip Homburg, Andrew S. Tanenbaum: Location Objects in Wide-Area Systems *Vrije universiteit, Amsterdam*

[39] Maarten van Steen, Franz J. Hauck, Andrew S. Tanenbaum: A Model for Worldwide Tracking of Distributed Objects *Vrije Universiteit, Amsterdam*

[40] Maarten van Steen, Franz J. Hauck, Andrew S. Tanenbaum: Algorithmic Design of the Globe Location Service *Vrije Universiteit, Amsterdam,* Internal Report IR-413

Index