Vrije Universiteit Brussel
Faculteit Wetenschappen - Departement Informatica
Academiejaar 2001 - 2002

Transparante Foutentolerantie voor Mobiele Multi-Agent-Systemen

Pieter Verheyden

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

Promotor: Prof. Dr. Theo D'Hondt

Samenvatting

Wanneer het netwerk waarin gedistribueerde applicaties aan het runnen zijn, gedeeltelijk faalt door onverwachte fouten, dan kunnen deze applicaties beginnen haperen en in het slechtste geval stilvallen of volledig crashen. Partial failures van het netwerk kunnen opgevangen worden door gedistribueerde applicaties in een foutentolerantiesysteem te runnen. Een gewenste eigenschap van zo'n foutentolerantiesysteem is dat het transparant werkt. Hierdoor zal enerzijds de applicatieprogrammeur er niet zelf voor moeten instaan voor foutentolerantie in zijn applicaties te voorzien, anderzijds zal de complexiteit van een applicatie niet onnodig worden verhoogd. In deze thesis zullen we onderzoeken hoe we een mobiel multi-agent-systeem kunnen voorzien van een transparant foutentolerantiesysteem. We gebruiken hierbij het mobiel multi-agent-systeem Borg als case study.

Dankbetuiging

Ik dank **Prof. Dr. Theo D'Hondt** die mij de toestemming gegeven heeft mijn thesis aan zijn onderzoekslabo te doen.

Mijn grootste dank gaat naar **Werner Van Belle** die zijn kostbare tijd en geduld in mij (en mijn thesis) heeft willen steken. Zijn ideeën en onze (soms zeer lang durende) discussies waren van onschatbare waarde voor deze thesis.

Mijn grootste dank gaat ook naar **Dirk Van Deun**. Zijn tijd, mening, ideeën (die soms tegengesteld waren aan die van Werner) en de vele schrijftips waren eveneens van onschatbare waarde voor deze thesis.

Dank gaat uit naar **Pieter De Leenheer** voor zijn hulp bij het gebruik van LaTeX, wat het schrijvingsproces van deze thesis zeker heeft versneld.

Dank gaat ook uit naar **Jeannine Lefevre** voor het inbinden van meerdere exemplaren van deze thesis.

Tenslotte wil ik alle PROG-medewerkers bedanken die zo gul waren om naar mijn ideeën te luisteren en mij daarbij al dan niet feedback verleenden.

Inhoudsopgave

Lijst van figuren

Lijst van algoritmen

Hoofdstuk 1

Introductie

1.1 Intentie

Mobiele multi-agent-systemen zijn heden ten dage zeer populair, vooral in academische kringen treft men vele projecten aan. Ondanks deze hype worden echter belangrijke faciliteiten niet concreet uitgewerkt of gewoonweg overzien. Eén van deze faciliteiten is foutentolerantie. Een mobiel multi-agent-systeem is een sterk gedistribueerd systeem en maakt dus gebruik van een netwerk bestaande uit verschillende netwerkcomputers, nameservers en routers. Al deze hardware, de netwerkconnecties die de hardware onderling verbinden en de operating system die op de hardware runt mag men niet zomaar als betrouwbaar beschouwen. Een gedistribueerde applicatie in een mobiel multi-agent-systeem bestaat uit een aantal interagerende, actieve agenten die de mogelijkheid hebben om zich te verplaatsen doorheen het netwerk en daarbij hun code en data met zich meenemen. Meestal is één enkele hardware-failure genoeg om een volledige gedistribueerde applicatie te laten haperen en/of stilvallen. Hierbij gaan in het slechtste geval veel informatie en de berekeningen die de gedistribueerde applicatie tot vlak voor de failure had bereikt, verloren. Het probleem van zulke failures kunnen we opvangen door gedistribueerde applicaties in een foutentolerantiesysteem te runnen. Een gewenste eigenschap van zo'n foutentolerantiesysteem is transparantie. Programmeurs van applicaties zijn namelijk niet noodzakelijk experts in foutentolerantie en een expliciete foutentolerantie in applicaties zou ook onnodig de complexiteit van die applicaties verhogen.

1.2 Doel

Er bestaan reeds technieken om foutentolerantie toe te voegen aan gedistribueerde systemen. Eén van deze technieken is replicatie. Het doel van deze thesis is, vertrekkende van deze replicatietechniek, een transparant foutentolerantiesysteem te ontwerpen voor mobiele multi-agent-systemen. We nemen hierbij het mobiel multi-agent-systeem Borg als case study om dit te verwezenlijken.

1.3 Reisgids

Deze thesis is opgedeeld in 2 delen. Het eerste deel zullen we inleiden met partial failures van het netwerk te definiëren en bekende replicatietechnieken te overlopen. Daarna zullen we 2 bestaande foutentolerantiesystemen bespreken die replicatie hanteren; de eerste werkt volledig transparant, de tweede werkt niet transparant. Hierbij zullen we de voor- en nadelen trachten te bepalen die belangrijk zijn bij het ontwerpen van een transparant foutentolerantiesysteem met gebruik van replicatie. Zo'n ontwerp bespreken we dan in het tweede deel van deze thesis.

Programmeervoorbeelden in Borg zullen steeds in het `Typewriter` font geschreven zijn om Borg-code duidelijk van tekst te onderscheiden.

De meeste vaktermen in de informatica zijn Engelstalige termen. Deze vaktermen zijn in deze thesis zoveel mogelijk naar het Nederlands vertaald. Wanneer de Nederlandstalige term echter onherkenbaar was voor een informaticus, werd de Engelstalige term gebruikt.

Enige voorkennis van gedistribueerde systemen en/of mobiele multi-agentsystemen is een voordeel bij het lezen van deze thesis, maar is niet noodzakelijk. Hetzelfde geldt voor kennis van foutentolerantie in gedistribueerde systemen.

Deel I

Evaluatie van bestaande foutentolerantiesystemen

Hoofdstuk 2

Foutentolerantie door replicatie

2.1 Inleiding

Replicatie is een reeds gekende techniek om gedistribueerde systemen te voorzien van foutentolerantie. We zullen deze replicatietechnieken introduceren [7], maar zullen beginnen met de verschillende klassen van partial failures die in een netwerk kunnen optreden, te overlopen [5].

2.2 Partial failures van het netwerk

Een gedistribueerd systeem runt in een netwerk. Een netwerk is een verzameling van vele verschillende onderdelen, waaronder de belangrijkste: routers, nameservers, netwerkcomputers, en netwerkconnecties die de vele hardware-onderdelen met elkaar verbinden. Bij deze onderdelen kunnen onverwacht *failures* optreden als gevolg van *fouten* de oorzaak van zijn. We zeggen dan dat het netwerk te lijden heeft van *partial failures*, dat het netwerk met andere woorden gedeeltelijk faalt omdat bij één of meerdere onderdelen van dat netwerk failures zijn opgetreden. Hierdoor kunnen gedistribueerde applicaties die aan het runnen zijn in dat netwerk beginnen haperen en/of stilvallen. We kunnen 2 klassen van failures onderscheiden: failures in de communicatie en failures van processen.

2.2.1 Failures in communicatie

Failures in de communicatie kunnen optreden doordat de netwerkconnecties tussen verschillende hardware-onderdelen voor onbepaalde duur verbroken worden. Hierdoor kunnen berichten tussen de componenten van een gedistribueerde applicatie, of dat nu objecten of actieve agenten zijn, vertraging oplopen of gewoonweg verloren gaan. Hierdoor zal de applicatie beginnen haperen en in het slechtste geval stilvallen. Naast het uitvallen van netwerkconnecties kan nog een zwaardere failure in de communicatie optreden, namelijk het uitvallen van één of meerdere routers. Hierdoor zal het netwerk in 2 of meerdere stukken gesplitst worden waardoor delen van een gedistribueerde applicatie afgezonderd worden van elkaar. Totdat de uitgevallen router hersteld wordt, kunnen de afgezonderde delen van de gedistribueerde applicatie niet met elkaar communiceren. Dit leidt vervolgens ook tot het haperen en/of stilvallen van die applicatie.

2.2.2 Failures van processen

Failures van processen kunnen we opdelen in verschillende soorten waarvan de belangrijkste: crash failures, omission failures, timing failures, value failures en arbitrary failures. Met *processen* bedoelen we uitvoerende entiteiten, gaande van bijvoorbeeld gedistribueerde objecten, actieve agenten, de virtuele machine waarin deze runnen tot het operating system van de hardware.

- Een **crash failure** doet zich voor wanneer een goed functionerend proces plotseling wordt beëindigd en er dus geen enkele communicatie meer mogelijk is met dat proces. Een typisch voorbeeld van zo'n crash failure is dat het operating system van bepaalde hardware de geest geeft waardoor in het slechtste geval gereboot moet worden.
- Een **omission failure** doet zich voor wanneer een proces geen verwachte output levert voor een gegeven input. Een typisch voorbeeld van deze failure is wanneer een serverproces geen antwoord geeft op de request van een clientproces.
- Een **timing failure** doet zich voor wanneer een proces wel output levert maar te laat om nog bruikbaar te zijn voor het ontvangend proces.
- Een **value failure** doet zich voor wanneer de output van een proces incorrect is.
- Een **arbitrary failure**, ook wel bekend als een *Byzantine* failure, doet zich voor wanneer één of meerdere processen zich onbegrijpelijk gedragen door bijvoorbeeld output te produceren die ze eigenlijk niet zouden mogen produceren. Deze failures zijn zeer moeilijk te detecteren en zijn nauw verwant met crash failures.

De meeste van deze failures, zowel in communicatie als bij processen, kunnen opgevangen worden met behulp van replicatie. Replicatie is een reeds gekende techniek die ons in staat stelt om fouten in een gedistribueerd systeem te kunnen tolereren. We zullen deze techniek in volgende sectie introduceren.

2.3 Replicatietechnieken

Bij replicatie van componenten (gedistribueerde objecten of actieve agenten bijvoorbeeld) kunnen we 2 klassen onderscheiden: *passieve* replicatie (ook wel *primary backup* genaamd) en *actieve* replicatie.

2.3.1 Passieve replicatie

Passieve replicatie wordt ook wel primary backup genaamd. Bij deze strategie speelt 1 van de replica's een centrale rol. Deze *primaire* replica ontvangt alle berichten/invocaties, voert deze uit en verstuurt ook alle antwoorden. De andere replica's dienen enkel als backup wanneer de primaire replica zou uitvallen. Dit gaat als volgt te werk:

1. De primaire replica ontvangt van een bepaald proces een bericht/invocatie.
2. De primaire replica voert deze uit en heeft op het einde een antwoord klaar liggen. Door het uitvoeren zal de interne toestand van de replica mogelijk aangepast worden.
3. De primaire replica stuurt vervolgens update-berichten naar de andere replica's zodat die hun interne toestand kunnen updaten. Na deze update stuurt elke replica een acknowledgement-bericht naar de primaire replica.
4. Na het ontvangen van alle acknowledgement-berichten, stuurt de primaire replica het antwoord naar het oorspronkelijk sturend proces.

De primaire replica kan op 3 verschillende momenten uitvallen: (1) voordat de update-berichten worden gestuurd, (2) na het sturen van de update-berichten maar voor het versturen van het antwoord naar het sturend proces en (3) nadat het sturend proces het antwoord heeft ontvangen. Op elk van deze momenten zal een nieuwe primaire replica moeten worden aangeduid en zal het bericht/invocatie opnieuw door het sturend proces moet worden gestuurd.

2.3.2 Actieve replicatie

Bij deze strategie is er geen centrale, primaire replica. Bijgevolg ontvangen alle replica's alle berichten/invocaties van een sturend proces, verwerken deze en sturen vervolgens ook allemaal een antwoord naar het sturend proces. Dat proces heeft dan de keuze: ofwel wachten op het eerste antwoord, ofwel wachten op de meerderheid van de antwoorden, ofwel wachten op alle antwoorden. Opdat alle (actieve) replica's de berichten/invocaties in dezelfde volgorde ontvangen, moet er een zogenaamd totaal-geordend systeem van multicasting (geordende 'one-to-many' berichten) voorzien zijn. Dit bespreken we verder in volgende hoofdstukken waar we bestaande implementaties van actieve replicatie in gedistribueerde systemen zullen bespreken.

2.3.3 Passief versus actief

Het voordeel van actieve replicatie is dat deze het uitvallen van een replica sneller kan opvangen omdat er op elk moment een actief uitvoerende replica voorhanden is. Bij passieve replicatie is er maar één enkele uitvoerende replica, de primaire replica. Wanneer deze primaire replica uitvalt, kost de herstelling ervan aanzienlijk meer tijd omdat een nieuwe primaire replica moet worden aangesteld en het sturend proces zijn bericht/invocatie opnieuw moet sturen. Merk op dat hierdoor het uitvallen van (passieve) replica's niet transparant is voor de andere processen. De applicatie zal bijgevolg ook aanzienlijk vertragen bij een herstel. Het nadeel van actieve replicatie is echter dat het meer processorkracht vraagt (alle actieve replica's zijn wel verspreid over de processors van het netwerk maar voeren alle berichten/invocaties uit) en ook meer bandbreedte omdat alle replica's alle invocaties/berichten ontvangen en alle antwoorden versturen.

Het voordeel van passieve replicatie is dat het geen last heeft van niet-deterministische operaties. Een niet-deterministische operatie is een operatie die verschillende resultaten bij de verschillende replica's oplevert. Een random-operatie is hiervan een voorbeeld. Wanneer alle actieve replica's een random-operatie oproepen, zal dit bij elke replica een verschillend resultaat opleveren waardoor deze replica's hun consistentie verliezen. Bij passieve replicatie zal enkel de primaire replica de random-operatie oproepen.

2.3.4 Groepen van replica's

We kunnen alle replica's van één component (een object of actieve agent) abstraheren tot een *groep*. Zo'n groep representeert dus een verzameling van replica's en kan gebruikt worden om een bericht naar deze verzameling te sturen, eerder dan het bericht naar elke replica apart te sturen. Hierdoor wordt ook het aantal replica's van de groep voor andere groepen verborgen.

2.4 Conclusie

In dit hoofdstuk hebben we de verschillende klassen van partial failures waaraan een netwerk kan lijden, besproken. Deze kunnen we in 2 klassen onderscheiden: failures in de communicatie en failures bij processen. Replicatie is een gekende techniek om deze failures op te vangen en om dus mogelijke fouten te kunnen tolereren. We hebben 2 strategieën van replicatie geïntroduceerd, passieve en actieve replicatie, en de voor- en nadelen van beide strategieën besproken.

Hoofdstuk 3

Het Aroma System

3.1 Inleiding

Het Aroma System werd door Nitya Narasimhan ontwikkeld aan het Department of Electrical and Computer Engineering, University of California. Deze middleware voorziet Java Remote Method Invocation (JavaRMI) van transparante foutentolerantie en maakt daarbij gebruik van een interceptiemechanisme voor replicatie van objecten. Haar zus, Priya Narasimhan, deed het haar enkele jaren eerder voor door CORBA te voorzien van transparante foutentolerantie met dezelfde technieken [3]. In dit hoofdstuk zullen we het Aroma System kort introduceren en beginnen met JavaRMI kort te bespreken. Verdere details in verband met JavaRMI kan men vinden op de specificatiewebpagina van Sun Microsystems [4]. Verdere details in verband met het Aroma System en de implementatie ervan kan men vinden in de doctoraatsthesis van Nitya Narasimhan [1] en in de paper over transparante consistente replicatie van JavaRMI-objecten [2].

3.2 Inleiding tot JavaRMI

Java Remote Method Invocation (JavaRMI) is een gedistribueerd objectmodel voor de Java virtuele machine (JVM). Een gedistribueerd object ('remote object') is een object waarvan de methoden kunnen worden opgeroepen vanuit verschillende JVMs die zich al dan niet op dezelfde host bevinden. De methoden van zo'n gedistribueerd object worden gedeclareerd in één of meerdere gedistribueerde interfaces ('remote interfaces'). Het aanroepen van die methoden gebeurt door remote function calls (RMI). Het grote voordeel hierbij is dat het aanroepen van een methode op een gedistribueerd object dezelfde syntax gebruikt als het aanroepen van methoden op lokale objecten. Hierdoor wordt het aanmaken en gebruiken van gedistribueerde objecten voor de applicatieprogrammeur eenvoudiger.

Communicatie in een typische JavaRMI-applicatie gebeurt vanuit een clientproces naar een serverobject toe. De client vraagt bij een registry een bepaald object op basis van de naam op, en krijgt een zogenaamd stub-object terug. Zo'n stub-object fungeert als een 'lokale versie' van een gedistribueerd object. De client roept dan methoden op bij deze lokale stub, en het is de taak van de stub om die method calls over te brengen naar het het serverobject. Het serverobject heeft op zijn beurt een overeenkomstig skeleton. Zo'n skeleton zal de binnenkomende method calls dispatchen naar het eigenlijk serverobject en vervolgens het antwoord terugsturen naar de client zodat die verder kan runnen. Zoals reeds aangehaald, wordt het protocol tussen client en server gedefinieerd als een remote interface.

3.3 Overzicht van het Aroma System

De huidige JavaRMI-infrastructuur bezit aparte faciliteit voor foutentolerantie en kan dus geen betrouwbare applicaties garanderen. De applicatieprogrammeur moet er met andere woorden zelf voor instaan om JavaRMI-apllicaties van foutentolerantiemechanismen te voorzien. Het Aroma System voorziet JavaRMI van foutentolerantie die transparant is voor de applicaties en dit met geen of minimale aanpassing van bestaande applicaties of van het onderliggend systeem. Hiervoor wordt gebruik gemaakt van interceptie voor replicatie van Java-objecten.

Het nadeel van het hebben van één enkel serverobject in een applicatie is dat de communicatie tussen client(s) en server wegvalt wanneer die één enkele server uitvalt waardoor de volledige applicatie zal stilvallen. Door het Aroma System te gebruiken, kunnen zowel de client- als de serverobjecten van een applicatie gerepliceerd worden, verspreid over verschillende processors in het systeem. Deze replica's worden in een consistente toestand gehouden op een manier dat transparant is voor de applicatie. Het Aroma System ondersteunt zowel passieve als actieve replicatie en zal de replica's van een object groeperen. Een objectgroep is een abstractie voor een verzameling van objecten die een gemeenschappelijke interface implementeren. Een objectgroep representeert een gerepliceerd object. De leden van een objectgroep zijn de replica's van dat ene object. De interface van zo'n objectgroep verbergt de details van de replicatie zoals bijvoorbeeld de locatie van de replica's en de stijl van replicatie (actief of passief).

3.3.1 De verschillende componenten

In het Aroma System kunnen we 4 componenten onderscheiden: de Aroma Interceptor, de Aroma Parser, de Aroma Message Handler, en de Aroma Multiplexer.

De Aroma Interceptor maakt zichzelf at runtime vast aan zowel de clientprocessen als de serverprocessen om zo de client-server-communicatie (die uit TCP/IP-calls bestaat) te onderscheppen en om te leiden naar de Aroma Message Handler. De Aroma Message Handler implementeert de meeste mechanismen om een JavaRMI-applicatie van consistente replicatie te voorzien.

Niet alle TCP/IP-calls van een applicatie moeten onderschept worden door de Aroma Interceptor. Een JavaRMI-object kan bijvoorbeeld met een grafische user-interface communiceren. Hiervoor moet een noodzakelijke connectie via TCP/IP gemaakt worden naar een lokale X-server. Het is niet nodig om al deze communicatie langs het Aroma System te laten passeren. De Aroma Interceptor kan met behulp van de Aroma Parser een onderscheid maken tussen met JavaRMI verbonden TCP/IP-calls en andere TCP/IP-calls zoals dus bijvoorbeeld TCP/IP-calls naar een lokale X-server. De Aroma Parser wordt ook door de Aroma Message Handler gehanteerd om buitengaande berichten aan te passen, en om bij binnenkomende berichten de aanpassingen ongedaan te maken. Het aanpassen van berichten is een zeer complex proces en is nodig om niet-determinisme van het JavaRMI-model aan te pakken. Hierop komen we zo dadelijk even op terug.

Het Aroma System onderschept remote method invocations en zal deze doorsturen naar alle replica's van het remote-object. Omdat alle replica's deze invocatieberichten in dezelfde volgorde moeten ontvangen om consistentie te behouden, is een totaal-geordende multicasting van berichten noodzakelijk. De Aroma Multiplexer voorziet een interface voor Totem. Totem is een reeds ontwikkeld betrouwbaar groepcommunicatiesysteem met totaal-geordende multicast-berichten ('one-to-many' berichten). Met behulp van Totem kan de toestand van de replica's van een object consistent gehouden worden, ook wanneer failures zich voordoen.

3.4 Duplicatie van berichten

Bij zowel actieve als passieve replica's kan duplicatie van berichten (invocaties en antwoorden) ontstaan. Elke replica van een actief gerepliceerde server voert elke invocatie uit en geeft een antwoord hierop terug. Een actief gerepliceerde client zal dubbele invocaties genereren en een actief gerepliceerde server zal bijgevolg dubbele antwoorden genereren. Bij passieve replica's kan duplicatie van berichten ontstaan tijdens de periode van herstel na failure(s). Stel bijvoorbeeld dat de primaire server-replica een invocatie van een client uitvoert, maar uitvalt voordat het een antwoord op de invocatie kan teruggeven. Hierdoor zal de client opnieuw een invocatie uitvoeren op de nieuwe, herstelde primaire server-replica voor een request dat al afgehandeld was (maar waarop geen reply was teruggeven).

Duplicatie van berichten zal door de Aroma Message Handler onderschept en tegengehouden worden door aan elke invocatie (en vervolgens aan elk antwoord) een unieke operation identifier mee te geven.

3.5 Niet-determinisme in JavaRMI

Een niet-deterministische operatie is een operatie die verschillende resultaten bij de verschillende replica's oplevert, zoals reeds aangehaald in vorig hoofdstuk. Niet-deterministische operaties leiden bijgevolg tot inconsistentie van replica's. Voorbeelden van niet-determinisme bronnen in JavaRMI zijn: object identifiers, multi-threading, connection timeouts en connecties naar databases via JDBC. Hoe deze bronnen van niet-determinisme door het Aroma System behandeld worden, kan men terugvinden in Hoofdstuk 6 van de doctoraatsthesis van Nitya Narasimhan [1].

3.6 Conclusie

In dit hoofdstuk hebben we het Aroma System kort geïntroduceerd. Deze middleware voorziet JavaRMI van foutentolerantie door gebruik van (actieve of passieve) replicatie en heeft de gewenste eigenschap van transparant te werken. Het onderscheppen van een JavaRMI-applicatie en het repliceren van de gedistribueerde objecten van die JavaRMI-applicatie gebeurt transparant voor de applicatieprogrammeur, waardoor deze er niet zelf voor moet instaan om zijn JavaRMI-applicaties van foutentolerantie.

Hoofdstuk 4

Foutentolerantie in TIB/Rendezvous

4.1 Inleiding

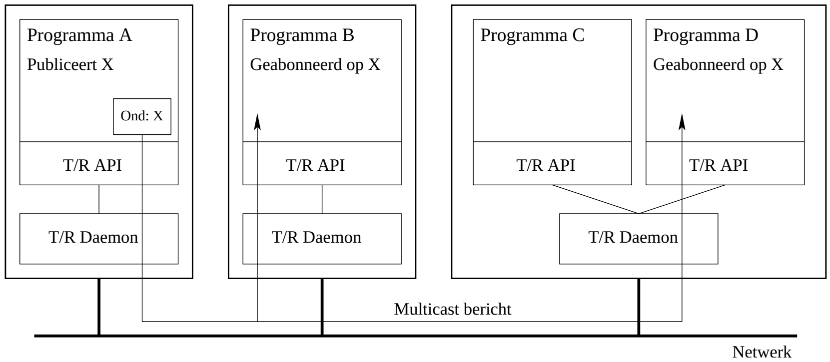
In dit hoofdstuk bekijken we het TIB/Rendezvous systeem, ontwikkeld door TIBCO[1]. We zullen eerst een overzicht geven van wat TIB/Rendezvous doet en hoe het te werk gaat. Hierna zullen we wat dieper ingaan op bepaalde functionaliteiten om vervolgens de foutentolerantie in dit gedistribueerd systeem te bespreken. Verdere details kan men vinden in de handleiding waarin de concepten van TIB/Rendezvous en foutentolerantie ervan uitvoerig besproken worden [6].

4.2 Overzicht

TIB/Rendezvous is een gedistribueerd systeem waarmee data tussen verschillende (gedistribueerde) programma's kan uitgewisseld worden over een netwerk door het sturen en ontvangen berichten. Bij het sturen van deze berichten wordt *onderwerp-gebaseerde adressering* gehanteerd. Wanneer een programma een bericht stuurt, dan geeft dit geen expliciete bestemming op, maar tagt het het bericht met een *onderwerp*. Ontvangers geven van hun kant niet op van welke programma's ze berichten willen ontvangen, maar geven onderwerpen op waarin ze geïnteresseerd zijn. Dit betekent dat programma's (referentieel) ontkoppeld worden van elkaar. Het ontkoppelen van componenten in een gedistribueerd systeem wordt gemotiveerd met voordelen zoals makkelijker onderhoud van uitvoerende systemen en het makkelijker toevoegen van componenten 'on the fly'.

Het sturen van berichten met onderwerp-gebaseerde adressering noemt men ook wel *publiceren*. Een programma dat berichten met bepaalde onderwerpen wil ontvangen, moet *geabonneerd* zijn op deze onderwerpen. Berichten zijn zelfbeschrijvend. Dit betekent dat een programma het bericht moet onderzoeken, hoe dit gestructureerd is en welke data dit bevat. Hierdoor worden berichten

[1]Homepage van TIBCO Sotware Inc. is te vinden op http://www.tibco.com



Figuur 4.1: Een TIB/Rendezvous-omgeving met een voorbeeld van publiceren en abonneren.

veel dynamischer: een programma moet niet bekend zijn met de structuur van de berichten die het ontvangt.

De TIB/Rendezvous-software bestaat voor de applicatieprogrammeur uit 2 delen: een TIB/Rendezvous-API en de TIB/Rendezvous-daemon. Van de TIB/Rendezvous-API bestaan er verschillende versies voor verschillende programmeertalen waaronder C, C++ en Java. Een TIB/Rendezvous-daemon is een achtergrondproces dat op elke deelnemende computer in het netwerk runt. Elke uitwisseling van informatie tussen programma's (die zich op dezelfde computer bevinden of op verschillende hosts) passeert langs een TIB/Rendezvous-daemon.

Figuur 4.1 toont een TIB/Rendezvous-omgeving met een voorbeeld van publiceren en abonneren. Merk op dat er één enkele daemon op elke deelnemende computer in het netwerk runt (en dus geen daemon per programma). In dit voorbeeld publiceert programma A een bericht met onderwerp X. Dit bericht wordt verstuurd (door multicasting) naar alle hosts in het netwerk die een TIB/Rendezvous-daemon runnen. Programma's B en D zijn geabonneerd op onderwerp X bij hun lokale daemon. Een daemon houdt in een tabel bij welke programma's op welke onderwerpen geabonneerd zijn. Wanneer het bericht met onderwerp X binnenkomt, kijkt de daemon in de tabel naar welke programma's hij het bericht moet doorsturen. Als er geen enkel programma geabonneerd is op het onderwerp, dan wordt het bericht onmiddellijk door de daemon vernietigd.

4.3 Berichten

Hardware en operating systems gebruiken verschillende representaties voor data. TIB/Rendezvous zet daarom lokale dataformaten om naar een universeel TIB-Rendezvous-formaat. Zoals reeds aangehaald wordt data over het netwerk uitgewisseld door het sturen en ontvangen van berichten met onderwerp-gebaseerde adressering.

Een bericht moet steeds een *onderwerp* krijgen (en dus geen expliciete bestemming) voordat het kan gestuurd worden. Optioneel kan een onderwerp voor antwoord meegeven worden. Zo'n onderwerp kan door de ontvanger van het bericht gebruikt worden om een antwoord terug te sturen naar het sturende programma. De zender van het bericht moet dan natuurlijk geabonneerd zijn op het eigen onderwerp van antwoord. De naam van een onderwerp is een rij van karakters dat door punten (.) verdeeld is in elementen. Deze naam kan ook wildcards bevatten zodat programma's zich met één enkel abonnement kunnen abonneren op een verzameling van onderwerpen.

Naast onderwerpen bestaan er ook zogenaamde *inboxen*. Zo'n inbox is een speciaal onderwerp en stelt een bestemming voor die uniek is voor een bepaald proces. Wanneer bijvoorbeeld een programma A een bericht publiceert en daarbij de naam van de inbox van een ander programma B gebruikt, dan wordt het bericht direct naar B gestuurd door middel van een point-to-point communicatie in plaats van multicasting. De naam van een inbox wordt niet door het programma gekozen, maar wordt door een TIB/Rendezvous-functie dynamisch aangemaakt.

4.4 Transport

Om berichten te kunnen sturen en ontvangen, maken programma's een zogenaamd *transport* aan. Een transport is een lokaal object dat gebruikt wordt om berichten te versturen naar TIB/Rendezvous daemons en om te luisteren naar berichten op een specifieke poort. Programma's kunnen op elk moment berichten versturen naar andere TIB/Rendezvous programma's. Om een bericht te versturen, roept een programma methoden van een transport op. We hebben 3 verschillende operaties:

- De **send-operatie** verstuurt een buitengaand bericht. Dit gebeurt door het bericht door te geven aan de lokale TIB/Rendezvous-daemon die op zijn beurt ervoor zorgt dat het bericht correct gepubliceerd wordt.
- Met de **sendreply-operatie** wordt een buitengaand bericht verstuurd als antwoord op een binnenkomend bericht.
- Met de **sendrequest-operatie** wordt een buitengaand bericht verstuurd en gewacht op een binnenkomend antwoord. Hiervoor wordt de oproepende thread geblokkeerd totdat het antwoord is ontvangen. Het antwoord wordt teruggestuurd naar een (dynamisch aangemaakte) inbox gebruikmakende van point-to-point communicatie. Deze operatie is met andere woorden een *synchrone* operatie.

Merk op dat er geen receive-operatie bestaat. TIB/Rendezvous maakt namelijk gebruik van events voor het ontvangen van berichten. De sendrequest-operatie is hier een uitzondering op.

4.5 Events

Wanneer een programma zich abonneert op een onderwerp, dan wordt een zogenaamde *listener event* aangemaakt. Dit is een lokaal object dat geassocieerd is met een *transport* en een *specifiek onderwerp* (dat wildcards kan bevatten). Het transport begint onmiddellijk te luisteren naar binnenkomende berichten met overeenkomende onderwerpen. Een listener event bevat ook de referentie naar een *callback functie*. Deze callback functie wordt aangeroepen wanneer een *event* (een binnenkomend bericht bijvoorbeeld) wordt afgehandeld, en wordt door de applicatieprogrammeur gedefinieerd.

Telkens wanneer een bericht binnenkomt dat voldoet aan het onderwerp dat in een listener event gespecifieerd is, dan wordt een overeenkomstig *event object* aangemaakt. Een event object houdt onder andere de referentie naar de callback-functie bij die gebruikt zal worden om het event object in een laatste fase te kunnen afhandelen. Het listener event wordt gebruikt om verder te luisteren naar meer binnenkomende berichten. Een *event driver* zorgt ervoor dat wanneer een event (binnenkomend bericht) zich voordoet, het event object aan een lokale *event queue* wordt toegevoegd waar het wacht om afgehandeld te worden. Met elke event queue is een dispatcher thread geassocieerd die het event van de top van de queue haalt en vervolgens de geassocieerde callback-functie hierop aanroept. De meeste programma's handelen hun events af in een loop. De TIB/Rendezvous-API voorziet ook een oproep dat een aparte dispatcher thread creëert. Om een abonnement op een onderwerp op te zeggen, moet het overeenkomstig listener event vernietigd worden. Hierdoor zullen ook alle wachtende overeenkomstige event objects vernietigd worden. Verder kunnen we opmerken dat wanneer een binnenkomend bericht voldoet aan 2 of meerdere onderwerpen gespecifieerd in 2 of meerdere listener events, voor *elk* listener event een overeenkomstig event object wordt aangemaakt.

Default is er één enkele event queue waarmee alle events worden afgehandeld. Het is ook mogelijk meerdere queues te gebruiken en expliciete events te associëren met bepaalde queues. Dit heeft als voordeel meerdere events tegelijkertijd te kunnen afhandelen aangezien elke event queue over een eigen dispatcher thread beschikt. We kunnen echter nog verdergaan, we kunnen verschillende queues ook groeperen. Aan elke queue in een queue-groep kunnen we dan een prioriteit geven. Een dispatcher voor een queue-groep zal dan eerst de events in de queue met de hoogste prioriteit afhandelen. Wanneer deze queue leeg is, wordt de queue met de tweede hoogste prioriteit behandeld.

4.6 Foutentolerantie

TIB/Rendezvous ondersteunt foutentolerantie voor 2 situaties van failures. Ten eerste voorziet het verschillende faciliteiten voor betrouwbare en gegarandeerde communicatie. Deze faciliteiten zorgen er dus voor dat gepubliceerde berichten in de communicatie niet verloren gaan. Ten tweede voorziet het een faciliteit om processen te groeperen om deze zo te beschermen tegen crash failures van processen. We zullen de verschillende faciliteiten in deze sectie uitvoerig bespreken.

4.6.1 Betrouwbare levering van berichten

In de TIB/Rendezvous-daemon zijn reeds protocollen geïmplementeerd voor betrouwbare levering van berichten. De TIB/Rendezvous-daemon houdt buitengaande berichten tot 60 seconden bij en informeert programma's wanneer data verloren is gegaan. Informatie over deze verloren data is echter niet beschikbaar. Een tweede nadeel is dat na het verstrijken van de tijdslimiet van 60 seconden of bij een beëindiging van het proces van de TIB/Rendezvous-daemon, er geen sprake meer is van betrouwbare levering van berichten.

	Betrouwbare levering	**Gegarandeerde levering**
Locatie van protocol(len)	Geïmplementeerd in de daemon.	Geïmplementeerd in een aparte CM library layer die gebruik maakt van de daemon voor transport van berichten.
Transparantie van protocol	Transparant voor de applicatieprogrammeur.	Applicatieprogrammeur maakt gebruik van CM library calls.
Feedback van protocol	Daemons informeren programma's wanneer data verloren is gegaan, maar er is geen informatie over die verloren data beschikbaar	Extra informatie wordt door de CM library aan het programma voorgelegd.
Bijhouden van berichten	Nee.	Buitengaande berichten worden opgeslagen in een ledger (ledger file of geheugen van het proces).
Tijdslimiet voor het bijhouden van berichten	Daemon houdt buitengaande berichten bij voor 60 seconden.	De CM library houdt buitengaande berichten bij in een ledger totdat de levering ervan succesvol is of totdat de (meegeven) tijdslimiet ervan verstreken is.
Extra gebruik van netwerk	Nee.	Ja, om de levering van elk gegarandeerd bericht te bevestigen.
Extra schijfgeheugen	Nee.	Ja, wanneer gebruik gemaakt wordt van ledger files voor het opslaan van elk buitengaand bericht.

Tabel 4.1: Vergelijken van betrouwbare en gegarandeerde levering van berichten.

4.6.2 Gegarandeerde levering van berichten

Gegarandeerde levering van berichten ('Certified Message Delivery') voorziet TIB/Rendezvouz van een veel sterkere betrouwbaarheid bij het leveren van berichten, samen met extra functionaliteiten zoals rapportering. Tabel 4.1 geeft een vergelijking van de belangrijkste aspecten tussen betrouwbare en gegarandeerde levering van berichten.

CM transport

Om berichten te versturen of te ontvangen met gegarandeerde levering, moet een programma een CM transport aanmaken. Zo'n CM ('Certified Message') transport bevat extra informatie: een *naam* en een *ledger*. De naam wordt onder andere gebruikt om CM transporten te identificeren bij andere CM transporten en kan al dan niet herbruikt worden. Een CM transport maakt gebruik van een ledger om informatie over (verstuurde of ontvangen) gegarandeerde berichten bij te houden. Programma's hebben de keuze hoe zo'n ledger informatie bijhoudt: in een ledger file of in het geheugen van het proces. Het voordeel van een ledger file is dat het ook mogelijk wordt om gegarandeerde levering van berichten aan te bieden bij failures van programma's. Wanneer bijvoorbeeld een ontvangend programma uitvalt, worden alle gemiste berichten opgeslagen in een ledger aan de sturende kant. Wanneer het ontvangend programma zich hersteld heeft, neemt het contact op met de ledger en vraagt om alle gemiste berichten opnieuw te versturen. Een eerste nadeel van een ledger file is dat zo'n file behoorlijk wat schijfgeheugen kan innemen. De grootte van de file groeit namelijk evenredig met het aantal onvolledig geleverde berichten en de data die deze berichten bevatten. Een tweede nadeel is dat het programma zelf vertraagt omdat deze de ledger file (synchroon of asynchroon) moet updaten.

Gelabelde berichten

Naast CM transporten zijn er ook gelabelde berichten. Zo'n gelabeld bericht bevat extra informatie die CM transporten kunnen gebruiken voor gegarandeerde levering van berichten. Deze extra informatie bestaat uit: de naam van het CM transport dat het bericht verstuurt, een rangnummer toegekend door het sturende CM transport en een tijdslimiet na dewelke het sturende programma geen gegarandeerde levering van het bericht meer verwacht door het sturende CM transport. Zowel een gewoon transport als een CM transport kan gelabelde berichten ontvangen. Wanneer een gewoon transport een gelabeld bericht ontvangt, wordt dit bericht als een gewoon bericht behandeld. De extra informatie wordt met andere woorden genegeerd. Wat er gebeurt wanneer een CM transport een gelabeld bericht ontvangt, zal afhangen of dat CM transport *geregistreerd* is voor gegarandeerde levering. Registratie voor gegarandeerde levering wordt zo dadelijk uitgelegd. Wanneer het luisterend CM transport geregistreerd is, dan geeft deze de extra informatie ook door aan de callback-functie. Wanneer het luisterend CM transport niet geregistreerd is voor gegarandeerde levering met het sturend CM transport, dan wordt de naam van het sturend transport doorgegeven aan de callback-functie.

Ontdekking en registratie van gegarandeerde levering

Na het aanmaken van een CM transport, hebben we volgende stappen in het protocol voor gegarandeerde levering van berichten: ontdekking en registratie. Deze stappen volgen elkaar op.

Ontdekking vindt plaats wanneer een luisterend CM transport een gelabeld bericht (met een bepaald onderwerp) ontvangt van een sturend CM transport dat niet in de ledger van het luisterend CM transport voorkomt. Na deze ontdekking zal het luisterend CM transport de naam van het sturend CM transport toevoegen aan zijn ledger. Vervolgens zal dat luisterend CM transport een aanvraag tot registratie naar het sturend CM transport versturen. Deze registratie is nodig voor gegarandeerde levering van berichten met het bepaalde onderwerp. Om de ontdekking te melden, legt het luisterend CM transport een REGISTRATION.DISCOVERY advies voor aan het programma. Tenslotte houdt het luisterend CM transport binnenkomende berichten met het ontdekte onderwerp van het sturend CM transport bij.

Registratie vindt plaats wanneer het sturend CM transport de aanvraag tot registratie van het luisterend CM transport ontvangt. Deze aanvraag wordt automatisch aanvaard. Het is mogelijk om deze aanvraag te weigeren door de naam van het luisterend CM transport expliciet mee te geven met behulp van de disallowlistener-operatie. Bij een weigering van de aanvraag, zal het luisterend CM transport een REGISTRATION.NOT_CERTIFIED advies aan het programma voorleggen. Verder is het ook mogelijk dat door failures de aanvraag of de aanvaarding nooit toekomt of ontvangen wordt door een CM transport. In dat geval zal het registreren meerdere malen opnieuw geprobeerd worden. Wanneer dit uiteindelijk niet lukt, zal het luisterend CM transport een REGISTRATION.NO_RESPONSE advies moeten voorleggen aan het programma. Wanneer de registratie voor gegarandeerde levering van het onderwerp aanvaard wordt door het sturend CM transport, wordt het luisterend CM transport opgenomen in de ledger (voor dat bepaald onderwerp) en wordt het ervan op de hoogte gebracht dat zijn aanvraag aanvaard is. Het sturend CM transport legt een REGISTRATION.REQUEST advies voor om het nieuwe luisterend CM transport aan het sturend programma door te geven. Het luisterend CM transport legt op zijn beurt een REGISTRATION.CERTIFIED advies voor om aan het ontvangend programma mee te delen dat de aanvraag tot registratie gelukt is.

Op dit moment zal het luisterend CM transport de reeds bijgehouden binnenkomende berichten in de juiste volgorde in een queue steken. Nu dat de ontdekking en registratie (met succes) afgerond zijn, moet het sturend CM transport elk buitengaand bericht met het bepaald onderwerp bijhouden in zijn ledger totdat het luisterend CM transport bevestigd heeft dat elk bericht correct is aangekomen of totdat de tijdslimiet van een bericht is verstreken.

Voordat we overgaan naar de laatste stap van het protocol, namelijk het afleveren van gegarandeerde berichten, moeten er 2 zaken opgemerkt worden. Ten eerste: de eigenlijke sturende en ontvangende programma's nemen enkel indirect deel aan dit protocol, namelijk door CM library calls te gebruiken. Ten tweede: gegarandeerde levering na ontdekking en registratie (zoals hier beschreven staat) gebeurt maar in één enkele richting: van zender naar ontvanger. Voor de omgekeerde richting moet er dus een aparte ontdekking en registratie plaatsvinden.

Afleveren van gegarandeerde berichten

Hoe komt een bericht met bepaald onderwerp van de zender nu gegarandeerd terecht bij de ontvanger? Dit gaat als volgt. Een luisterend CM transport steekt het binnenkomend gegarandeerd bericht in een queue voor verdere afhandeling (aanroepen van callback-functie). Hierbij wordt extra informatie van het bericht meegegeven: de naam van het sturend CM transport en het rangnummer. Het rangnummer van een gegarandeerd bericht wordt zo dadelijk besproken. Wanneer de callback-functie het bericht heeft afgehandeld, wordt automatisch het sturend CM transport op de hoogte gebracht dat de levering succesvol is. Deze *automatische* bevestiging kan aangepast worden door het programma naar expliciete bevestiging met behulp van de confirmmsg-operatie en de setexplicitconfirm-operatie. Wanneer deze bevestiging is aangekomen, geeft het sturend CM transport de succesvolle levering door aan zijn ledger en geeft een DELIVERY.COMPLETE advies weer aan het programma. Het is mogelijk dat het gegarandeerd bericht meerdere luisterende CM transporten heeft bereikt. Daarom wacht het sturend CM transport op alle bevestigingen om het bericht uit zijn ledger te kunnen verwijderen.

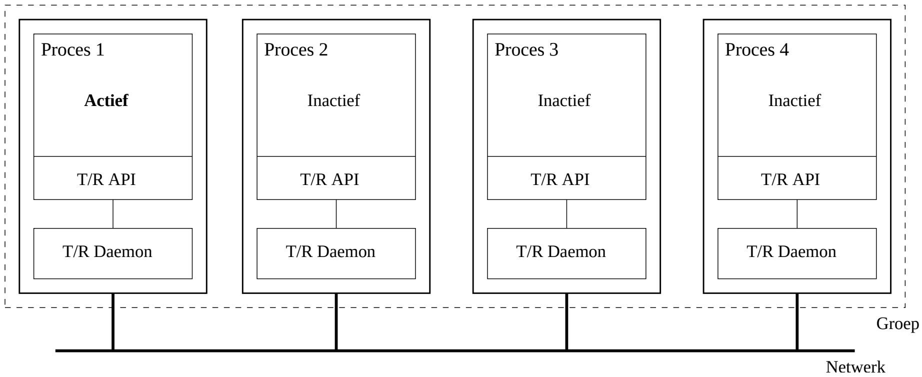
Rangnummer van een gegarandeerd bericht

Een rangnummer wordt als extra informatie aan een gegarandeerd bericht meegegeven. De reden van dit nummer is dat verstuurde berichten niet noodzakelijk in de juiste volgorde bij de ontvanger aankomen, mede door het mogelijk optreden van, al dan niet, tijdelijke failures. Aan de hand van dit nummer kan een luisterend CM transport binnenkomende berichten in de juiste volgorde in de event queue steken. We illustreren dit even met een (informeel) voorbeeld. Een luisterend CM transport ontvangt gegarandeerde berichten met een bepaald onderwerp van een bepaald sturend CM transport. Stel dat het luisterend CM transport juist het bericht met nummer 10 in de queue gestoken heeft en vervolgens bericht 13 ontvangt. Berichten 11 en 12 zijn met andere woorden (nog) niet toegekomen. Het bericht 13 zal daarom *tijdelijk* worden bijgehouden (en dus niet in de queue gestoken worden) totdat berichten 11 en 12 zijn toegekomen. Daarvoor wordt automatisch een aanvraag tot hersturen van berichten 11 en 12 door het luisterend CM transport naar het sturend CM transport gestuurd. Wanneer de tijdslimiet van deze berichten verstreken is, kan het sturend CM transport deze berichten niet meer hersturen omdat die berichten verwijderd werden uit diens ledger. In dit geval zal het luisterend CM transport het `DELIVERY.UNAIVAILABLE` advies teruggeven om aan te geven dat berichten 11 en 12 niet meer beschikbaar zijn. Het kan dan niets anders doen dan het bericht 13 toch in de queue te steken.

4.6.3 Groeperen van processen

Om programma's te beschermen tegen allerhande failures zoals bijvoorbeeld hardware failures en netwerkonderbrekingen, voorziet TIB/Rendezvous een faciliteit om processen van een programma in groepen te organiseren. Zo'n *foutentolerante groep* bestaat uit om het even welk aantal processen. Deze processen runnen op verschillende computers in het netwerk en kunnen ofwel *actief* ofwel *inactief* zijn. Actieve processen reageren op alle binnenkomende berichten. Inactieve processen daarentegen reageren enkel op speciale events. Hierop komen we zo dadelijk terug. Processen kunnen op elk ogenblik lid worden van een groep door een member object aan te maken. Het lidmaatschap van een groep kan op elk moment verbroken worden wanneer een proces (dat is lid is van die groep) beëindigd wordt of wanneer een lid zich terugtrekt door het member object te vernietigen. Een proces kan lid zijn van meerdere groepen. Verder heeft elke groep een unieke naam. Deze naam identificeert alle leden van de groep en wordt gebruikt om berichten die uitgewisseld worden tussen de leden, te labellen.

Wanneer een proces lid wordt van een groep, is deze initieel inactief. Inactieve processen runnen op de achtergrond en dienen als back-up voor slecht functionerende actieve processen. Om alle leden van een groep te kunnen sorteren, krijgt elk proces een *rang* toegewezen. Deze rang wordt bepaald door het *gewicht* van een proces dat manueel als parameter moet worden meegegeven wanneer het proces lid wordt van een bepaalde groep. Het gewicht van een proces is



Figuur 4.2: Een foutentolerante groep die bestaat uit 4 processen en waarvan de active goal gelijk is aan 1.

een positieve integer (verschillend van nul) die door de applicatieprogrammeur bepaald wordt. Bij het bepalen van dat gewicht worden best factoren zoals de snelheid van de hardware waarop het proces zal runnen en de betrouwbaarheid ervan, overwogen. Wanneer 2 processen met hetzelfde gewicht in een groep voorkomen, zal TIB/Rendezvous zelf uitmaken welk proces een hogere rang krijgt (elk proces in een groep heeft een verschillende rang). Programma's hebben ook de mogelijkheid om het gewicht van processen op elk ogenblik te veranderen. Hierbij zal TIB/Rendezvous automatisch de leden van de groep opnieuw sorteren. Wanneer een groep uit een groot aantal processen bestaat, kan deze herberekening tot een verlies van performantie leiden.

Hoeveel processen er steeds actief moeten zijn in een specifieke groep, wordt bepaald door de applicatieprogrammeur. Het aantal actieve processen in een groep noemt men de *active goal* van die groep. In vele gevallen moet duplicatie vermeden worden en is de active goal bijgevolg gelijk aan 1. We illustreren dit met volgend voorbeeld. Figuur 4.2 toont een groep van 4 processen met active goal gelijk aan 1. Dit betekent dus dat er één enkel actief proces is dat bijvoorbeeld berichten met een specifiek onderwerp (bijvoorbeeld voetbaluitslagen) publiceert door multicasting. Andere programma's die geabonneerd zijn op dit onderwerp, luisteren naar berichten met dat specifiek onderwerp en verwerken de data die deze (binnenkomende) berichten bevatten. De 3 overige processen zijn inactief en dienen als back-up. Deze processen doen exact hetzelfde als het actief proces, ze werken met andere woorden parallel met elkaar. Het enige verschil is dat de inactieve processen geen berichten publiceren, dit om duplicatie van berichten te vermijden. Wanneer nu het actief proces stopt met functioneren, zal TIB/Rendezvous het inactief proces met hoogste rang activeren, in dit geval proces met rang 2 (rang n is hoger dan rang n+1). Hiermee wordt vermeden dat berichten meerdere keren over het netwerk worden gestuurd en dus meerdere keren door de geabonneerde programma's ontvangen worden. Belangrijk is op te merken dat dit protocol zeer gelijkaardig is aan actieve replicatie, besproken in het inleidend hoofdstuk van Deel I.

Elk actief proces stuurt regelmatig een zogenaamde *hartslag* naar de andere, inactieve processen van de groep. Dit signaal (bericht) wijst erop dat het proces nog steeds actief is. Het tijdsinterval tussen 2 opeenvolgende (sturende) hartslagen wordt door de applicatieprogrammeur bepaald en wordt als parameter meegeven wanneer een proces lid wordt van een groep. Bij het bepalen van dit interval worden best volgende factoren overwogen: transmissietijd van het netwerk, capaciteit van het netwerk, en de nood aan een niet onderbroken service. TIB/Rendezvous detecteert slecht of niet functionerende actieve processen met behulp van deze hartslagen. Wanneer de hartslag van een actief proces ophoudt, wordt dat proces beschouwd als verloren en wordt het inactief proces met hoogste rang door TIB/Rendezvous geactiveerd. Andere situaties waar een actief proces beschouwd wordt als verloren, kunnen bijvoorbeeld zijn: het terugtrekken van een actief proces uit de groep, het uitvallen van de TIB/Rendezvous-daemon geassocieerd met een actief proces, of het uitvallen van een netwerkconnectie dat de processen met elkaar verbindt. Naast een tijdsinterval voor hartslagen, hebben we ook een gelijkaardig interval voor activeren. Dit interval bepaalt hoe lang een inactief proces wacht om actief te worden nadat een hartslag van een actief proces verloren is. De reden voor dit interval is dat sommige processen tijd nodig hebben om bijvoorbeeld te connecteren met een databank voordat ze actief worden.

Een programma kan het aantal actieve processen van een groep controleren met behulp van een *passieve monitor*. Deze monitor hoeft zelf geen lid te zijn van een groep en is passief in de betekenis dat hij geen enkele invloed heeft op de processen die hij controleert.

In al het bovenstaande hebben we gezien dat de applicatieprogrammeur expliciete foutentolerantie moet voorzien in zijn programma's om foutentolerant te kunnen werken. We kunnen dus niet over *transparante* foutentolerantie spreken. Dit heeft als groot nadeel dat bestaande programma's moeten aangepast worden om foutentolerant te kunnen werken en dat naast de logische code van een nieuw programma ernstig moet worden nagedacht over hoe foutentolerantie te voorzien in dat programma.

4.7 Conclusie

In dit hoofdstuk hebben we TIB/Rendezvous en de voorzieningen voor foutentolerantie ervan besproken. TIB/Rendezvous is een gedistribueerd systeem waarmee data tussen verschillende gedistribueerde programma's kan uitgewisseld worden. Hiervoor worden berichten met een specifiek onderwerp door sturende programma's gepubliceerd. Ontvangende programma's kunnen zich abonneren op bepaalde onderwerpen om de berichten met die onderwerpen te kunnen ontvangen en te verwerken. TIB/Rendezvous voorziet verschillende operaties om berichten asynchroon of synchroon te versturen. Om berichten te ontvangen worden geen operaties voorzien maar wordt gebruik gemaakt van events. De TIB/Rendezvous software bestaat uit 2 delen: een TIB/Rendezvous daemon die op elke computer in het netwerk runt en een TIB/Rendezvous-API. Elke uitwisseling van data passeert langs de daemon en van de API bestaan er verschillende versies voor verschillende programmeertalen.

TIB/Rendezvous kan zowel failures in de communicatie als failures bij processen opvangen. De TIB/Rendezvous-daemon bevat reeds verschillende protocollen voor een betrouwbare levering van berichten. Daarnaast wordt een extra faciliteit voor een gegarandeerde levering van berichten voorzien. Deze gegarandeerde levering verzekert dat gegarandeerde berichten in de juiste volgorde bij elk ontvangend programma aankomt. Zolang de meegegeven tijdslimiet van een gegarandeerd bericht niet verstreken is, zal TIB/Rendezvous ervoor zorgen dat de gegarandeerde berichten correct zullen aankomen. Wanneer de levering uiteindelijk toch mislukt, krijgen zowel het sturend programma als de ontvangende programma's expliciete informatie hierover. Hiervoor wordt gebruikt gemaakt van een file-gebaseerde of proces-gebaseerde ledger die de status van elk bericht bijhoudt. Het meermaals uitwisselen van grote berichten bij failures, het bijhouden van die berichten in ledgers en het updaten van deze ledgers zijn nadelig voor de performantie van deze faciliteit.

TIB/Rendezvous voorziet een tweede faciliteit voor foutentolerantie, namelijk het organiseren van processen in groepen. In zo'n groep treft men 1 of meerdere actieve processen aan. De overige processen zijn inactief en dienen als back-up wanneer de actieve processen ophouden met correct te functioneren. Deze techniek is gelijkaardig aan actieve replicatie. Het grootste nadeel van deze faciliteit én van gegarandeerde levering van berichten, is dat deze faciliteiten expliciet door de applicatieprogrammeur moeten voorzien worden tijdens de ontwikkeling programma's. Hierdoor moeten bestaande programma's aangepast worden om foutentolerant te kunnen werken en wordt van de applicatieprogrammeur geëist dat die over een sterke kennis van foutentolerant programmeren in TIB/Rendezvous beschikt.

Hoofdstuk 5

Conclusie

We zijn dit deel begonnen met het overlopen van de verschillende failures die in een gedistribueerd systeem kunnen optreden. We kunnen deze in 2 klassen opdelen: failures in de communicatie en failures bij processen. Met processen bedoelen we uitvoerende entiteiten, gaande van bijvoorbeeld gedistribueerde objecten, actieve agenten, de virtuele machine waarin deze runnen tot het operating system van de hardware. Beide klassen van failures kunnen ervoor zorgen dat een gedistribueerde applicatie kan haperen en/of stilvallen. Zulke failures kunnen we opvangen door de componenten van een gedistribueerde applicatie (actief of passief) te repliceren.

We hebben vervolgens 2 bestaande foutentolerantiesystemen besproken die gebruik maken van replicatie om fouten te tolereren. Het eerste systeem was het Aroma System dat Java Remote Method Invocation (JavaRMI) van foutentolerantie voorziet en daarbij zowel actieve replicatie als passieve replicatie ondersteunt. Deze middleware heeft de gewenste eigenschap van transparant te werken. Het onderscheppen van een JavaRMI-applicatie en het repliceren van de gedistribueerde objecten van die JavaRMI-applicatie gebeurt transparant voor de applicatieprogrammeur. Dit heeft 2 voordelen. De applicatieprogrammeur moet er niet zelf voor instaan om foutentolerantie te voorzien in zijn JavaRMI-applicaties, deze taak wordt gedelegeerd naar het Aroma System. Van de applicatieprogrammeur wordt met andere woorden geen kennis van foutentolerantie geëist en deze kan JavaRMI-applicaties ontwikkelen als voorheen. Een tweede voordeel is dat de complexiteit van de applicatie niet onnodig verhoogd wordt door expliciete foutentolerantiecode.

Het tweede systeem was de TIB/Rendezvous-software van TIBCO. Met deze middleware kan men met behulp van berichten data uitwisselen tussen verschillende gedistribueerde applicaties. TIB/Rendezvous bestaat uit 2 delen, een API en een daemon, en voorziet 3 faciliteiten voor foutentolerantie. De daemon bevat ingebakken protocollen voor een, eerder beperkte, betrouwbare levering van berichten. Gegarandeerde levering van berichten wordt als tweede faciliteit voorzien om failures in de communicatie op te vangen. Deze faciliteit is echter niet aangeraden wanneer berichten grote hoeveelheden data bevatten. Er wordt namelijk gebruik gemaakt schijfgeheugen of geheugen van het proces om verzonden berichten tijdelijk bij te houden totdat de levering ervan succesvol is. Een tweede nadeel is dat de applicatieprogrammeur deze faciliteit moet hanteren door expliciete calls naar de library die de protocollen van deze faciliteit implementeert. Hierdoor is deze faciliteit allerminst transparant en wordt bijgevolg van de applicatieprogrammeur een degelijke kennis van het gebruik ervan verwacht. De derde faciliteit is het groeperen van processen wat gelijkaardig is aan actieve replicatie. Deze faciliteit moet ook expliciet geprogrammeerd worden waardoor we ook niet van transparantie kunnen spreken.

Uit dit deel kunnen we enerzijds besluiten dat we foutentolerantie kunnen voorzien met behulp van gekende replicatietechnieken, anderzijds kunnen we opmerken dat transparantie een gewenste eigenschap is om de applicatieprogrammeur niet onnodig op te zadelen met het extra werk van foutentolerantievoorziening in gedistribueerde applicaties.

Deel II

Ontwerp van een foutentolerantiesysteem in Borg

Hoofdstuk 6

Borg

6.1 Inleiding

In dit hoofdstuk zullen we het mobiel multi-agent-systeem Borg introduceren. We zullen enerzijds laten zien hoe men agenten in Borg kan aanmaken, klonen, onderling laten communiceren en migreren, anderzijds zullen we uitleggen hoe Borg intern werkt. Bij de uitleg van de interne werking gaan we ons vooral richten op informatie die relevant is voor volgende hoofdstukken die over transparante foutentolerantie voor Borg handelen.

6.2 Wat is een mobiel multi-agent-systeem?

De ontwikkelaars van Borg hanteren volgende definities voor een mobiele multi-agent en een mobiel multi-agent-systeem [8, 13, 14]:

mobiele multi-agent

> Een mobiele multi-agent is een actieve, autonome software-component die kan communiceren met andere agenten over een wide-area netwerk. Met de term *mobiel* wordt gerefereerd naar het feit dat een runnende agent (proces) kan migreren naar andere agentsystemen (processors) en zijn programmacode en data met zich meeneemt.

mobiel multi-agent-systeem

> Een mobiel multi-agent-systeem is de architectuur in dewelke deze agenten geschreven en uitgevoerd kunnen worden. Het moet de programmeur zo veel mogelijk flexibiliteit aanbieden en daarbij zoveel mogelijk de onderliggende architectuur van het netwerk verbergen.

In de kringen van de AI wordt de term 'multi-agent-systeem' daarentegen gekoppeld aan een software-systeem waarmee het gedrag van (grote) groepen van interagerende agenten gesimuleerd wordt, zonder het gedistribueerde aspect in acht te nemen. Wij focussen ons dus op een mobiele multi-agent-systeem als een *gedistribueerde* omgeving.

6.3 Borg als experimentele omgeving

Het multi-agent platform Borg werd ontwikkeld op het Programming Technology Lab (PROG) aan de Vrije Universiteit Brussel (VUB). De licentiaatsthesis van Werner Van Belle lag in 1996-1997 aan de basis ervan [12]. Bij de meeste programmeeromgevingen zoals bijvoorbeeld Java, en met name JavaRMI, is het niet mogelijk om een proces naar een andere processor te laten migreren. Dit betekent dat het onmogelijk is om een Java-thread te migreren naar een andere Java virtuele machine. Hierdoor moesten de ontwikkelaars uitkijken naar een andere oplossing en is men bij de ontwikkeling van Borg [8] vertrokken van een reeds bestaande virtuele machine die aangepast werd aan de noden van de ontwikkelaars. Deze virtuele machine van de programmeertaal Pico is geschreven door Prof. Dr. Theo D'Hondt. Pico is een educatieve programmeertaal die aan de VUB vooral gehanteerd wordt om de concepten van programmeertalen aan niet-informatica-studenten aan te leren. Pico heeft daarom een zeer eenvoudige syntax, die sterk aanleunt bij de syntax van C. Borg is een uitbreiding en aanpassing van Pico, er werden bijvoorbeeld nieuwe primitieven voor het aanmaken en migreren van agenten toegevoegd.

Borg bestaat uit 2 grote delen. Het ene deel is de core-engine (de borg-core) die de agenten interpreteert. Het andere deel bestaat uit verschillende user-interfaces voor verschillende platformen: KDE1 (Linux), MS Windows, Palm OS en een zeer eenvoudige text-interface.

Borg is een *experimentele* omgeving. Dit betekent dat continu gewerkt wordt om de huidige toestand van Borg te verbeteren. Dit omvat zowel het verbeteren van de core-engine als het uitbreiden van de verschillende user-interfaces voor meerdere platformen.

6.4 Programmeren in Borg

In deze sectie zullen we laten zien hoe we met agenten in Borg kunnen werken. Dit houdt in: agenten creëren, laten communiceren, migreren en doden. We zullen ook eerst laten zien hoe men objecten in Borg kan gebruiken.

Agenten zijn actief en autonoom. De interne toestand van een agent kan enkel gewijzigd worden door berichten naar de agent te sturen; de data van een agent is private. Objecten zijn daarentegen passief en delen data-space.

6.4.1 Korte inleiding tot Pico

Zoals reeds aangehaald is Borg een uitbreiding van de zeer eenvoudige programmeertaal Pico. Pico is dynamisch getypeerd en een variabele kan het volgende bevatten: getallen, tekst (gelijkaardig aan strings in andere programmeertalen), de booleans `true` en `false`, de native value `void`, tabellen en functies. In Borg komt daarbij nog dat een variabele ook de referentie naar een agent kan bijhouden. Een variabele wordt gedefinieerd met de `:` operator, een variabele toewijzen doen we met de `:=` operator en de `::` operator gebruiken we om constanten en constante functies te definiëren. We treffen in Pico ook de volgende primitieve functies aan: rekenkundige functies, if-then-else constructies, relationele functies, functies op tekst, I/O functies en andere[1]. Zeer eenvoudige voorbeelden (met resultaat na evaluatie) kunnen we als volgt geven:

```
a:0
:0
a:=3
:3

b[a]:5
:[5 5 5]

factorial(x)::
   { if(x>1,
        x*factorial(x-1),
        1) }
:<function factorial>

factorial(5)
:120
```

6.4.2 Aanmaken van objecten

We beschouwen de code van een eenvoudig counter-object [9]:

```
makeCounter()::
   { c:0;
     get()::display(c);
     increment()::c:=c+1;
     clone() }
```

Merk op dat we hier niet over klassen spreken maar over objecten. Pico is namelijk uitgebreid met prototype-gebaseerde OO om objecten te ondersteunen. Deze hanteert een andere benadering dan klasse-gebaseerde OO: we maken een counter-object aan door de interne toestand ervan te construeren en dan die omgeving te *klonen*, vandaar de clone-operatie in de laatste regel van de code. Wanneer we nu een (tweede) counter-object willen aanmaken, klonen we het eerste.

We kunnen een counter-object aanmaken, de waarde van de teller opvragen en ophogen:

```
t:makeCounter()
:<dictionary>
t.get()
:0
t.increment()
:1
t.get()
:1
```

[1]Een uitvoerige beschrijving van de syntax van Pico (met voorbeelden) is te vinden op http://pico.vub.ac.be

Inheritance kan ook toegepast worden bij deze objecten, verdere informatie hierover kan men vinden in de tutorial van Borg [9].

6.4.3 Creëren van agenten

Borg heeft 2 primitieve functies voor het aanmaken van agenten: de agentclone-functie en de clone2agent-functie. We bespreken nu beide primitieve functies.

De agentclone-functie

De primitieve agentclone-functie heeft volgende syntax:

```
agentclone(object,agtname)
```

Deze functie neemt als argumenten een object en een naam voor een agent. Het *object* zal omgevormd worden tot een agent, die uniek geïdentificeerd wordt met *agtname*. Dit gebeurt door de volledige omgeving (variabelen en functies) te kopiëren. Na evaluatie geeft de functie de referentie naar de gecreëerde agent weer, of 'void' als er reeds een agent bestaat met de gegeven *agtname*. We hernemen het vorig counter-object en gaan ervan uit dat de naam Pieter nog niet gebonden is op machine mypc:

```
a:makecounter()
:<dictionary>
agentclone(a,"Pieter")
:<reference mypc/Pieter>
agentclone(a, "Pieter")
:<void>
```

De clone2agent-functie

De primitieve clone2agent-functie heeft maar 1 argument, in tegenstelling tot de agentclone-functie. De syntax van deze functie is de volgende:

```
clone2agent(agtname)
```

Deze functie gebruikt de omgeving (dictionary) van de oproepende agent om tot een agent te klonen. Als *agtname* reeds gebonden is, dan wordt een random post-fix toegevoegd aan *agtname*. Merk op dat hierdoor de functie steeds een nieuwe agent zal creëren en dus nooit 'void' zal teruggeven. We geven een voorbeeld: wanneer `clone2agent` opgeroepen wordt door mypc/Alice (opnieuw gaan we ervan uit dat naam Pieter nog niet gebonden is op machine mypc):

```
clone2agent("Alice")
:<reference mypc/Alice-510164114>
clone2agent("Pieter")
:<reference mypc/Pieter>
clone2agent("Pieter")
:<reference mypc/Pieter-762035609>
```

Referenties naar agenten kunnen aangemaakt worden met de primitieve functie `agent`. We kunnen bijvoorbeeld de variabele a de referentie naar mypc/Alice laten bevatten:

```
a:agent("mypc/Alice")
:<reference mypc/Alice>
```

Verder treffen we ook de variabelen `agentself` en `agentname` aan. De variabele `agentself` geeft de huidige agent (de omgeving) weer, en de variabele `agentname` geeft de naam van de huidige agent (de referentie) weer. De agent mypc/Alice roept deze variabelen bijvoorbeeld op:

```
agentself
:<dictionary>
agentname
:<reference mypc/Alice>
```

6.4.4 Communicatie tussen agenten

Agenten communiceren met elkaar door het sturen en ontvangen van berichten. Dit kan in Borg op 3 manieren, die we nu zullen bespreken [9, 14].

Message passing

We kunnen 2 agenten met elkaar laten communiceren door message passing. Voor zo'n message passing gebruiken we de `->` operator. We kunnen dit eenvoudig demonstreren als volgt:

mypc/Alice	mypc/Bob
	fac(x):: if(x>1, x*fac(x–1), 1)
	:<function fac>
b:agent("mypc/Bob")	
:<reference mypc/Bob>	
b−>fac(5)	
:<void>	120

In dit voorbeeld wordt getoond dat de fac-functie niet enkel kan opgeroepen worden binnen agent mypc/Bob waar de functie gedefinieerd is, maar ook door een andere agent mypc/Alice. Merk op dat message passing geen resultaat teruggeeft aan de oproepende agent mypc/Alice. Remote function calls gebeuren *asynchroon* en geven altijd 'void' terug aan de oproepende agent. Wanneer mypc/Alice resultaat terug moet krijgen, zullen we moeten gebruikmaken van een techniek met callback. We kunnen bijvoorbeeld mypc/Bob een faculteit laten berekenen voor mypc/Alice als volgt:

mypc/Alice	mypc/Bob
	fac(x):: if(x>1, x*fac(x–1), 1)
	:<function fac>
	getfac(x,agt):: agt−>display(fac(x))
	:<function getfac>
b:agent("mypc/Bob")	
:<reference mypc/Bob>	
b−>getfac(5,agentname)	
:<void>120	

Asynchrone send en receive

Borg bevat 2 primitieve functies voor asynchrone communicatie: send en receive. Wanneer de primitieve send-functie door een agent wordt opgeroepen, wordt een bericht (data) *asynchroon* naar een ontvangende agent gestuurd. Dit betekent dat de sturende agent niet op een antwoord van de ontvangende agent wacht en gewoon verdergaat met zijn berekening. Aan de ontvangende kant wordt het bericht in een (wachtende) receive-queue gestoken. Wanneer tijdens (verdere) berekening een primitieve receive-functie wordt bereikt, dan wordt het bericht van de receive-queue gehaald. De syntax van de 2 primitieven is de volgende:

```
send(agtname,message)
recv(agtname,pattern)
```

De send-functie heeft 2 argumenten. Het eerste argument is de referentie naar de ontvangende agent en het tweede argument is het bericht zelf. Het bericht wordt door deze functie na oproep teruggeven. Het argument *message* kan ook een tabel zijn. Hierdoor kunnen we meerdere berichten in 1 keer versturen of het bericht taggen.
Bijvoorbeeld:

```
a:agent("mypc/Bob")
:<reference mypc/Bob>
send(a, "This is a message")
:This is a message
send(a,["tag1","This is a tagged message"])
:<table>
```

De recv-functie heeft 2 argumenten. Het eerste argument is de agent waarvan we berichten wensen te ontvangen. Het argument *agtname* kan hier één enkele referentie naar een agent zijn, een tabel van referenties naar agenten of een wildcard. Als geen enkel bericht van deze geldige agenten beschikbaar is, dan wordt 'nomesg' teruggeven. Het tweede argument is een patroon waaraan het bericht moet voldoen.
Bijvoorbeeld:

`recv(agent("mypc/Alice"),any)` : elk bericht van agent mypc/Alice ontvangen
`recv([agent("mypc/Alice"),agent("mypc/Bob")],["tag",any])` : een tabel

die als eerste element ”tag”moet bevatten, ontvangen van agent mypc/Alice of agent mypc/Bob

Synchrone send en receive

Voor synchrone communicatie werd een derde primitieve functie `sync` toegevoegd aan Borg. Deze functie synchroniseert 2 (of meer) agenten. Volgende (informeel) voorbeeld illustreert het gedrag van de functie door agenten Alice en Bob te synchroniseren.

Alice	Bob
sync(Bob)	...
... waiting for sync ...	...
...	sync(Alice)
execution continues	execution continues

De argumenten van deze sync-functie zijn dezelfde als die van de recv-functie. Het tweede argument is echter optioneel. We kunnen als tweede argument een patroon meegeven dat restricties oplegt op wat agenten mogen synchroniseren.

Door de 3 primitieve functies send, recv en sync te combineren, kunnen we een synchrone send en receive simuleren in Borg. Dit ziet er als volgt uit:

```
ssend(agtname,message):{
   send(agtname,message)
   sync(agtname,message) }

srevc(agtname,message):{
   sync(agtname,message)
   recv(agtname,message) }
```

6.4.5 Migreren van agenten

Het grote voordeel van Borg is de *sterke* migratie van agenten naar andere agentsystemen. Dit betekent dat de migratie op *elk moment* kan plaatsvinden, dus ook tijdens de uitvoering van agenten. Bij een sterke migratie wordt de berekening van een uitvoerende agent tijdelijk onderbroken. Vervolgens wordt de agent verplaatst naar een andere (opgegeven) host en tenslotte wordt de berekening van de agent hervat op die nieuwe host. We maken hiervoor gebruik van de primitieve agentmove-functie. Hoe we deze functie kunnen gebruiken, zullen we illustreren met volgend voorbeeld [10]. We laten een agent opnieuw een faculteit berekenen, maar we verplaatsen de agent vóór de berekening en na elke recursiestap naar een andere (random-bepaalde) host.

```
{
  hosts:[ agent("mypc2/Bob"),
          agent("mypc3/Christine"),
```

```
          agent("mypc4/Dirk") ];
  fac(x)::{
     display("factorial ",text(x));
     host:hosts[(abs(random())\\size(hosts))+1)];
     display("moving to ",text(host),eoln);
     agentmove(host);
     if(x>1, x*fac(x-1), 1)
  }
}
```

We evalueren deze code nu bijvoorbeeld in agent mypc1/Alice en roepen de functie met argument 5 op:

```
fac(5)
:factorial 5 moving to mypc3/Christine
:factorial 4 moving to mypc4/Dirk
:factorial 3 moving to mypc3/Christine
:factorial 2 moving to mypc2/Bob
:factorial 1 moving to mypc3/Christine
:120
```

Merk op dat op elke opgegeven host een Borg omgeving moet runnen en dat de opgegeven agenten moeten bestaan. De naam van agent mypc1/Alice verandert *niet* wanneer deze zich verplaatst naar andere hosts.

6.4.6 Doden van agenten

Een agent doden doen we met de primitieve functie `agentdie`. Deze functie heeft geen argumenten. Een agent kan zelfmoord plegen door deze functie (lokaal) op te roepen of kan andere agenten doden door message passing. Wanneer een agent sterft, wordt diens berekening beëindigd en is geen enkele communicatie met deze agent meer mogelijk. Het nadeel is dat agenten die de referentie naar een gestorven agent bijhouden, niet weten dat die agent er niet meer is. De programmeur kan dit bijvoorbeeld oplossen door eerst al deze agenten te verwittigen dat een bepaalde agent zal sterven, zodanig dat die agenten de referentie naar die stervende agent mogen verwijderen (deze zal toch in de nabije toekomst ongeldig worden). Merk op dat deze oplossing geen garanties biedt. Agenten kunnen namelijk ook sterven ten gevolge van hardware failures.

6.4.7 Gedistribueerde applicaties

Een gedistribueerde applicatie in Borg bestaat uit een aantal interagerende agenten die zich op verschillende computers in het netwerk bevinden en die zich ook kunnen migreren over dat netwerk. Voor concrete voorbeelden maken we gebruik van het traditionele 'Hello World' programma en geven hiervan 2 variaties: eentje dat peer-to-peer werkt en eentje met een client-server hiërarchie.
We beschouwen de code van de eerste variatie die peer-to-peer werkt:

```
sayHello(agttable)::{
   display("Hello World",eoln);
```

```
   i:0;
   while(i<size(agttable),
      { i:=i+1; agttable[i]->display("Hello World",eoln) }
   )
}
```

Agenten kunnen na evaluatie van deze functie elkaar de boodschap 'Hello World' meedelen. De sayHello-functie heeft slechts 1 argument: een tabel met referenties naar agenten. Na het oproepen van deze functie wordt de string 'Hello World' teruggeven aan de oproepende agent en aan elke agent in de parameter. Merk op dat hiervoor de primitieve display-functie lokaal wordt opgeroepen, en remote met message passing. (Merk ook op dat, in tegenstelling tot vele andere programmeertalen, het eerste element in een primitieve tabel een index gelijk aan 1 heeft.)

Deze functie werkt peer-to-peer, we hebben geen client-server-hiërarchie waar alle communicatie tussen agenten (clients) via een centrale server gaat. Dit heeft als voordeel dat alle agenten op een gelijk niveau met elkaar communiceren. Bij een client-server-hiërarchie speelt de server de belangrijkste rol. Wanneer die server stilvalt, dan ligt de volledige applicatie onmiddellijk stil. We kunnen het gebruik van deze functie als volgt illustreren:

pc1/Alice	pc2/Bob	pc3/Christine
b:agent("pc2/Bob")		
:<reference pc2/Bob>		
c:agent("pc3/Christine")		
:<reference pc3/Christine>		
sayHello([b,c])		
:Hello World		
:<void>	:Hello World	:Hello World

We beschouwen nu de code voor de tweede variatie van het 'Hello World' programma. Deze variatie heeft een client-server-hiërarchie en is daarom opgedeeld in client-code en server-code.

```
‘ Client-code
{
  refserver:void;

  loginToServer(agtserver)::
     agtserver->clientLogin(agentname);

  loginSuccess(agtserver)::{
     refserver:=agtserver;
     display("Login to server successful")
  };

  sayHello()::
     if(is_void(refserver),
        display("First login to a server"),
        refserver->sayHello()
```

```
    )
}


‘ Server-code
{
  clients[10]:void;
  nbrclients:0;

  clientLogin(agtclient)::
    if(nbrclients = size(clients),
      agtclient->display("Server has reached max.  number of clients"),
      { nbrclients:=nbrclients + 1;
        clients[nbrclients]:=agtclient;
        display("New client");
        agtclient->loginSuccess(agentname) }
    )
  sayHello()::
    for(i:1, i <= nbrclients, i:=i+1,
      clients[i]->display("Hello World",eoln)
    )
}
```

Agenten die als clients zullen fungeren, evalueren de client-code en de agent die als server zal fungeren de server-code. We kunnen nu de communicatie tussen client(s)-server als volgt illustreren (pc2/Bob fungeert als server en de andere agenten als clients):

pc1/Alice	pc2/Bob	pc3/Christine
s:agent(”pc2/Bob”)		
:<reference pc2/Bob>		
loginToServer(s)		
:<void>Login to server successful	:New client	
sayHello()		
:<void>Hello World		
		s:agent(”pc2/Bob”)
		:<reference pc2/Bob>
		loginToServer(s)
	:New client	:<void>Login to server successful
		sayHello()
:Hello World		:<void>Hello World
sayHello()		
:<void>Hello World		:Hello World

De code van zowel client als server is voor illustratie zeer eenvoudig gehouden. Deze code zou kunnen uitgebreid worden met een logoutFromServer-functie aan de kant van de client en een clientLogout-functie aan de kant van de server.

6.5 Interne werking

In de vorige sectie werd er al af en toe kort aangehaald wat er intern gebeurt. In deze sectie zullen we de belangrijkste punten aangaande de interne werking van Borg bespreken [14, 15].

6.5.1 Naming

De naam van een agent in Borg identificeert deze agent uniek en verandert niet wanneer deze agent migreert naar andere hosts in het netwerk. Deze naam wordt met andere woorden *altijd* gebruikt om naar die specifieke agent te refereren. Een naam bestaat uit 2 delen, gescheiden door een slash (/). Het eerste deel is de naam van het agentsysteem (de naam van de lokale machine) waar de agent gecreëerd is, het tweede deel is een vrije string. Als bijvoorbeeld het agentsysteem mypc.vub.ac.be (de naam van onze machine) een agent Alice aanmaakt, dan is de naam van deze agent mypc.vub.ac.be/Alice. Deze naam is uniek, dit betekent dat zolang mypc.vub.ac.be/Alice leeft, er geen tweede Alice door mypc.vub.ac.be/Alice zal kunnen worden aangemaakt. Merk op dat deze naam leesbaar is voor de applicatieprogrammeur.

6.5.2 Sturen van berichten

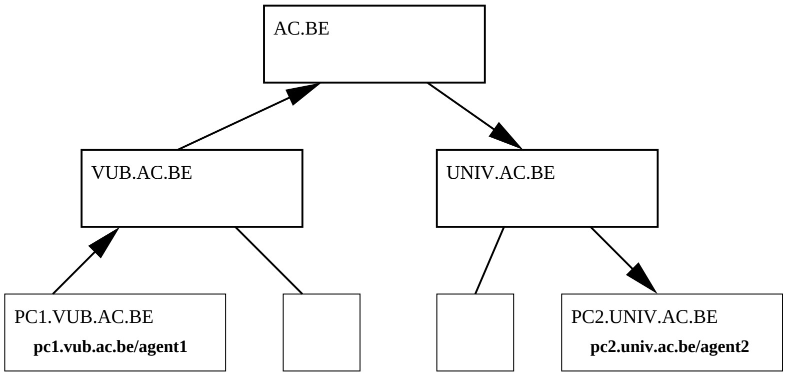
Bij het sturen van berichten gebruiken we de naam van een agent en niet de plaats (het adres) waar die agent zich bevindt in het netwerk. Hierdoor moet de sturende agent het netwerkadres van de ontvangende agent niet weten om berichten te sturen. Figuur 6.1 is een voorbeeld van een hiërarchisch wide-area netwerk waarin 2 agenten met elkaar communiceren door het versturen en ontvangen van berichten. Nameservers/routers worden aangeduid door rechthoeken met een dikke rand, en netwerkcomputers waarop agenten runnen, worden aangeduid door rechthoeken met een dunnere rand. Merk op dat we niet spreken van gescheiden nameservers en routers, een nameserver fungeert ook als router en omgekeerd. Hoe komt bijvoorbeeld een bericht verstuurd door agent pc1.vub.ac.be/agent1 nu terecht bij agent pc2.univ.ac.be/agent2? Dit gaat als volgt te werk:

1. Agent pc1.vub.ac.be/agent1 stuurt een bericht met als bestemming agent pc2.univ.ac.be/agent2

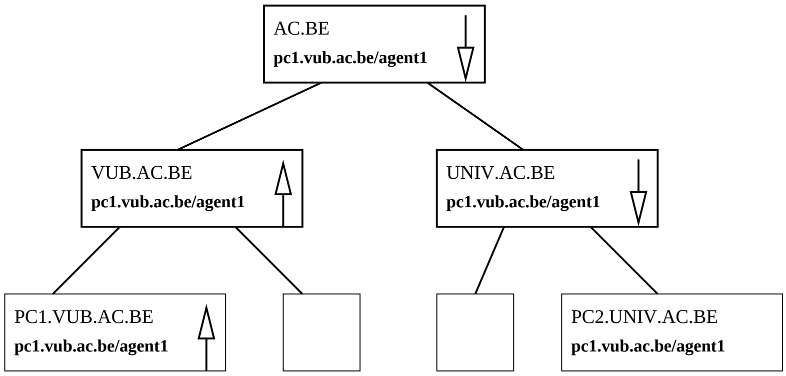
2. Het volledige bericht wordt doorgestuurd naar de lokale nameserver/router vub.ac.be; deze nameserver/router stuurt het bericht op zijn beurt door naar de nameserver/router ac.be

3. De nameserver/router ac.be ziet dat het bericht bestemd is voor zijn rechter link naar beneden en stuurt het bericht dus verder door de nameserver/router univ.ac.be

4. De nameserver/router univ.ac.be stuurt tenslotte het bericht door naar pc2.univ.ac.be waar agent2 het bericht uiteindelijk ontvangt.



Figuur 6.1: Sturen van berichten tussen 2 agenten over een wide-area netwerk.



Figuur 6.2: Migratie van een agent in een wide-area netwerk.

6.5.3 Migreren van agenten

Het migreren van een agent gebeurt op eenzelfde manier als het versturen van berichten. Het verschil is dat een nameserver/router zijn routing-tabel moet updaten telkens wanneer een migrerende agent langs die nameserver/router passeert. Deze update bestaat uit het updaten van de link (naar beneden of naar boven) die de (nieuwe) richting van de (migrerende) agent voorstelt. Een routing-tabel houdt dus geen exacte locaties van agenten bij. Dit heeft als voordeel dat er geen update moet gebeuren wanneer een agent opnieuw migreert en mogelijk niet langs dezelfde nameservers/routers passeert (als bij de eerste migratie). Figuur 6.2 toont hetzelfde wide-area netwerk als Figuur 6.1. Wat gebeurt er nu bijvoorbeeld als agent pc1.vub.ac.be/agent1 migreert naar pc2.univ.ac.be? Dit gaat als volgt te werk:

1. We laten agent pc1.vub.ac.be/agent1 migreren naar pc2.univ.ac.be door de toestand van deze agent als een bericht door te sturen naar de lokale nameserver/router vub.ac.be

2. De nameserver/router vub.ac.be stuurt de agent door naar nameserver/router ac.be (zoals bij het versturen van berichten) en update zijn routing-tabel door de link naar boven te laten verwijzen naar de migrerende agent.

3. De nameserver/router ac.be stuurt de agent door zijn naar zijn rechter link naar beneden en update ook zijn routing-tabel.

4. De nameserver/router univ.ac.be stuurt tenslotte de agent door naar pc2.univ.ac.be en update daarbij ook zijn routing-tabel.

Wanneer een agent migreert terwijl berichten naar deze migrerende agent gestuurd worden, volgen deze berichten gewoon het migratiepad van de agent en worden uiteindelijk ontvangen door deze agent. Het hiervoor gebruikte algoritme voorkomt dat verstuurde berichten de migrerende agenten zullen voorbij steken.

6.6 Conclusie

In dit hoofdstuk hebben we het mobiel multi-agent-systeem Borg besproken. Op dit platform kunnen we op een eenvoudige manier met mobiele agenten werken. We kunnen agenten met unieke namen creëren en laten communiceren. Deze communicatie gebeurt met asynchrone of synchrone berichten of met behulp van message passing. Borg bevat het grote voordeel van sterke migratie, we kunnen namelijk uitvoerende agenten op elk moment laten migreren naar andere agentsystemen in het netwerk. Bij deze migratie verandert de naam van de migrerende agent niet. Dit heeft als voordeel dat een agent de locatie van andere agenten niet moet weten om ermee te kunnen communiceren. Eén van de belangrijke kenmerken die echter nog in Borg ontbreekt, is foutentolerantie. Een ontwerp voor een mogelijk (transparant) foutentolerantiesysteem volgt in volgende hoofdstukken.

Hoofdstuk 7

Foutentolerantie voor Borg

7.1 Inleiding

We hebben in vorig hoofdstuk aangehaald dat Borg geen faciliteiten voorziet voor foutentolerantie. In dit hoofdstuk zullen we een overzicht geven van een ontwerp van een foutentolerantiesysteem in Borg en de eisen overlopen waaraan dit moet voldoen. Dit foutentolerantiesysteem zal gebruikmaken van actieve replicatie, werkt volledig transparant en kan geïmplementeerd worden op high level. In volgende hoofdstukken zullen we in detail de verschillende fases, componenten en functionaliteiten van dit transparant foutentolerantiesysteem in Borg bespreken.

7.2 Partial failures in Borg

In Hoofdstuk 2 hebben we gezien dat failures te verdelen zijn in 2 klassen: failures in communicatie en failures van processen. Met *processen* bedoelen we uitvoerende entiteiten zoals bijvoorbeeld de mobiele Borg-agenten en de virtuele machine van Borg waarin deze runnen. Zoals reeds aangehaald in vorig hoofdstuk, bestaat een gedistribueerde Borg-applicatie uit agenten die met elkaar communiceren met behulp van berichten. Deze agenten bevinden zich op verschillende netwerkcomputers[1] en kunnen ook migreren over het netwerk tijdens het runnen van de applicatie.

Stel dat een gedistribueerde applicatie bestaat uit een bepaald aantal agenten waaronder agenten a en b die zich respectievelijk op netwerkcomputer A en B bevinden. Failure in communicatie kan dan optreden doordat bijvoorbeeld de netwerkconnectie tussen A en B uitvalt op het moment dat a een bericht van b verwacht. Hierdoor splitst het netwerk en zal a blijven wachten op een bericht van b en niet weten waarom het niet komt, totdat de netwerkconnectie hersteld is. Hierdoor zal de applicatie beginnen haperen. Failure van een proces kan bijvoorbeeld optreden wanneer netwerkcomputer B uitvalt door een bepaalde fout. Een bijkomend probleem bij deze failure is dat agent b sterft waardoor de interne toestand van die agent verloren gaat en onmogelijk hersteld kan worden.

[1]Nameservers/routers worden enkel gebruikt voor naming/routing-taken, agenten runnen bijgevolg enkel op netwerkcomputers.

In vorig hoofdstuk hebben we gezien dat een gedistribueerde Borg-applicatie zowel peer-to-peer (P2P) als met een client-server-hiërarchie geïmplementeerd kan worden. We bekijken nu elke geval apart met een voorbeeld.

7.2.1 Failures bij P2P-applicaties

In een P2P-applicatie communiceren alle applicatie-agenten op een gelijk niveau, alle agenten zijn met andere woorden even belangrijk. MeetingPlanner is een voorbeeld van een Borg-applicatie die peer-to-peer werkt [11]. In deze P2P-applicatie hebben we 3 soorten agenten die met elkaar interageren door message passing:

- **MP-agent** die een gebruiker (organisator of deelnemer van een meeting) of een plaats voorstelt omdat een plaats functioneel ongeveer equivalent is aan een gebruiker (geen 2 meetings op hetzelfde moment bijvoorbeeld). Deze agent bevat logische code om meetings op te stellen, af te zeggen, aan te passen en op te vragen.
- **Meeting-agent** die een meeting voorstelt en wordt gecreëerd door MP-agenten. De taak van deze agent is het plannen (beheren) van de meeting waarvoor hij gecreëerd is.
- **GUI-agent**, de grafische user-interface (onder KDE1/Palm) voor een MP-agent.

Een meeting heeft volgende gegevens: organisator, datum, begin- en eindtijdstip, onderwerp, plaats, gastenlijst, en een aanwezigheidspercentage dat bepaalt hoeveel gasten er minimum moeten deelnemen aan de meeting opdat die effectief doorgaat. Het plannen van een meeting (we abstraheren van de GUI) gaat als volgt te werk. De organiserende MP-agent creëert een meeting-agent met de verschillende gegevens als parameters. Na creatie contacteert de meeting-agent onmiddellijk de plaats (een bepaalde MP-agent) voor reservatie. Wanneer deze plaats niet beschikbaar is, zal de meeting-agent dit melden aan de organiserende MP-agent. De organisator heeft dan 2 mogelijkheden: een alternatieve plaats doorgeven of de meeting afzeggen (door de meeting-agent te doden). Wanneer een plaats gereserveerd is, zal de meeting-agent voldoende deelnemers proberen op te trommelen. Een door de organiserende MP-agent meegegeven aanwezigheidspercentage van bijvoorbeeld 100 betekent dat alle gasten moeten deelnemen aan de meeting of deze gaat niet door. Gasten die vrij zijn wanneer de meeting zal plaatsvinden, worden automatisch gereserveerd door de meeting-agent. Deelnemers hebben nadien de mogelijkheid een meeting af te zeggen, net zoals de plaats en de organisator zelf. Wanneer een deelnemer afzegt, zal de meeting-agent aan de hand van het aanwezigheidspercentage kijken of de meeting nog kan doorgaan (indien niet moet de meeting afgezegd worden). In het geval dat de plaats afzegt, wordt dit door de meeting-agent aan de organiserende MP-agent gemeld zodat deze opnieuw de 2 mogelijkheden heeft. Wanneer de organiserende meeting afzegt, moeten plaats en deelnemers door de meeting-agent hiervan verwittigd worden en wordt de meeting-agent nadien door de organiserende MP-agent gedood.

Zoals blijkt uit de hierboven uitgelegde werking van MeetingPlanner, gebeurt er redelijk wat communicatie tussen de verschillende applicatie-agenten wanneer een agent een meeting plant met een grote gastenlijst (van bijvoorbeeld 20 MP-agenten). Door het optreden van failures kan de applicatie beginnen haperen en voor onbepaalde duur stilvallen. We geven enkele mogelijke situaties waar dit het geval is.

In het slechtste geval valt een meeting-agent uit door een failure. Hierdoor gaat de volledige interne toestand van die agent verloren en kan geen enkele MP-agent (plaats, deelnemer of organisator) deze gestorven meeting-agent contacteren. Het afzeggen, opvragen of aanpassen van deze meeting wordt nu onmogelijk. Wanneer een MP-agent van een gebruiker uitvalt, dan gaan al zijn (referenties naar) geplande en deelnemende meetings (meeting-agenten) verloren en kan deze gestorven MP-agent niet meer gecontacteerd worden door bestaande en nieuwe meeting-agenten.

Stel dat een MP-agent automatisch gereserveerd is door een meeting-agent. Stel nu dat de communicatie tussen die MP-agent en de meeting-agent verbroken wordt op het moment dat die MP-agent zijn deelneming aan de meeting wil afzeggen. Hierdoor gaat deze afzegging verloren en kan de deelnemende MP-agent de meeting pas afzeggen wanneer deze opnieuw verbonden wordt met de meeting-agent.

7.2.2 Failures bij client-server-applicaties

Bij een client-server-hiërarchie speelt de centrale server-agent de belangrijkste rol. Het uitvallen van deze enkele server-agent heeft het volledig stilvallen van de applicatie tot gevolg ('total failure'). In vorig hoofdstuk hebben we een zeer eenvoudig voorbeeld van een client-server-applicatie in Borg gezien. Een client-agent moet in deze applicatie inloggen bij een server (die een maximum aantal clients toelaat) om het 'Hello World'-bericht te sturen. De server-agent zal dit bericht ontvangen en doorsturen naar alle ingelogde clients. De clients ontvangen dit bericht en printen deze uit op het scherm.

Het is duidelijk dat wanneer de centrale server-agent sterft, de volledige applicatie zal stilvallen omdat geen enkele communicatie tussen de client-agenten meer mogelijk is. Wanneer de connectie tussen server-agent en één of meerdere client-agenten uitvalt, dan valt de communicatie met die client-agenten voor onbepaalde tijd weg waardoor deze 'Hello World'-berichten kunnen missen.

Het uitvallen van een client-agent kan met behulp van expliciete mechanismen door de server-agent opgevangen worden. We zullen echter stellen dat foutentolerantie transparant moet plaatsvinden. Dit betekent dat de applicatie zelf gevrijwaard blijft van expliciete foutentolerantiemechanismen (zoals dus bijvoorbeeld het opvangen van een uitgevallen client-agent door de server-agent). Het plotseling uitvallen van een client-agent betekent echter ook dat de interne toestand van die client-agent verloren gaat.

7.2.3 Opvangen van partial failures

Elke agent in een gedistribueerde applicatie is normalerwijze uniek. Wanneer één van die unieke applicatie-agenten uitvalt of wanneer de communicatie tussen 2 unieke applicatie-agenten onderbroken wordt, heeft dit tot gevolg dat de applicatie zal haperen en/of stilvallen. Om deze problemen ten gevolge van failures op te vangen, kunnen we ervoor zorgen dat er meerdere exemplaren van elke applicatie-agent bestaan, verspreid over het netwerk, waardoor applicatie-agenten niet meer uniek zijn. Deze techniek van *replicatie* is een gekende techniek om foutentolerantie in een gedistribueerd systeem te voorzien. Zowel client-agenten, server-agenten als P2P-agenten zullen door het foutentolerantiesysteem gerepliceerd worden.

Het foutentolerantiesysteem zal partial failures volledig opvangen of geheel niet opvangen. Dit betekent dat partial failures ofwel tot een total crash herleid worden (dit gebeurt door het uitvallen van 1 referentiepunt in het netwerk) ofwel volledig opgevangen worden. Dit heeft als voordeel dat een applicatie in het foutentolerantiesysteem nooit verder blijft runnen met onbepaald gedrag.

7.3 Actieve replicatie van applicatie-agenten

In Hoofdstuk 2 hebben we twee stijlen van replicatie geïntroduceerd: passieve en actieve replicatie. Bij passieve replicatie heeft men 1 runnende, primaire replica en de overige replica's dienen als backup voor wanneer de primaire replica zou uitvallen. Bij actieve replicatie zijn alle replica's actief aan het runnen. Een groot voordeel bij actieve replicatie is dat het herstel van replica's sneller gebeurt dan bij passieve replicatie, waardoor de applicatie ook minder lang hapert. Bij actieve replicatie is het namelijk zo dat op elk moment een actief runnende agent voorhanden is. Wanneer de primaire replica (bij passieve replicatie) uitvalt, moet eerst een nieuwe primaire replica aangesteld worden, vervolgens moet de failure door een andere reeds communicerende agent gedetecteerd worden en tenslotte moet deze agent zijn berichten waarvoor hij nog geen antwoord heeft ontvangen, opnieuw sturen (naar de nieuwe primaire replica). Dit heeft als nadeel dat het uitvallen van replica's bij passieve replicatie zichtbaar is voor andere agenten.

Een nadeel bij actieve replicatie is dat het meer processorkracht vraagt en meer bandbreedte omdat alle (actieve) replica's alle berichten ontvangen, verwerken en tenslotte alle antwoorden versturen. Een tweede nadeel is dat bij de implementatie ervan, niet-deterministische operaties moeilijk of niet behandeld kunnen worden. We komen hier zo dadelijk op terug. Niettegenstaande zullen we toch actieve replicatie gebruiken omdat de snelheid waarmee applicaties kunnen hersteld worden, aanzienlijk hoger ligt dan bij passieve replicatie.

7.3.1 Uitzonderingen

Het is mogelijk dat niet alle agenten van een applicatie in aanmerking komen om gerepliceerd te worden. Applicatie-agenten met I/O-taken mogen niet actief gerepliceerd worden. Een voorbeeld van zo'n agent is een agent die schijftoegang heeft. Voor deze agenten is passieve replicatie een betere oplossing omdat bij deze strategie enkel de primaire agent toegang heeft tot bijvoorbeeld het lezen en schrijven van bepaalde I/O-files. Een tweede voorbeeld van zo'n agent is een agent die een grafische user-interface (GUI) implementeert, bijvoorbeeld de GUI-agent bij MeetingPlanner. Het is niet aangewezen dat een GUI van een gebruiker tegelijk op verschillende netwerkcomputers verschijnt tengevolge van replicatie.

Alle functies moeten *deterministisch* zijn. Een voorbeeld van een niet-deterministische functie is de random-functie. Wanneer verschillende replica's van een applicatie-agent deze functie aanroept bij het uitvoeren van een binnenkomend bericht, dan zal dit verschillende waarden bij de verschillende replica's opleveren. (In Borg neemt de random-functie een random-seed en als die dus geïnitialiseerd is op dezelfde waarde, dan levert die dezelfde resultaten op. Het is een pseudo-random-functie). Voor agenten met niet-deterministische functies is passieve replicatie een betere oplossing.

7.4 Consistentie van replica's

Het belangrijkste aspect bij replicatie van agenten is het consistent houden van replica's. Dit betekent dat, vertrekkende van dezelfde initiële toestand, na het reageren op berichten en het produceren van consistente antwoorden, de interne toestand van de replica's dezelfde moet zijn. De vereisten waaraan voldaan moet worden om deze consistentie te behouden, zijn de volgende:

1. Alle replica's van een applicatie-agent moeten alle berichten in *dezelfde volgorde* ontvangen en behandelen. De binnenkomende berichten moeten met andere woorden totaal-geordend zijn.

2. Berichten worden naar *elke* replica van een applicatie-agent gestuurd, of naar *geen enkele*. Een bericht dat bijvoorbeeld maar door één enkele replica ontvangen wordt en vervolgens door die replica wordt behandeld, is dus niet toegestaan.

3. Berichten mogen maar *1 keer* door een replica behandeld worden. Actieve replica's ontvangen alle berichten, behandelen deze en sturen ook allemaal dezelfde antwoorden terug waardoor ontvangende agenten meerdere keren hetzelfde bericht zullen ontvangen. Deze binnenkomende berichten mogen door de ontvangende agenten maar 1 keer behandeld worden.

4. Het behandelen van bericht kan de *creatie van een nieuwe agent* tot gevolg hebben. Nieuwe agenten mogen maar 1 keer gecreëerd worden en moeten dan gerepliceerd worden om foutentolerant te kunnen runnen.

5. Replicatie van agenten moet ook consistent gebeuren tijdens de *periode van herstel*. Wanneer een replica uitvalt, zal een nieuwe replica gecreëerd moeten worden en zal de interne toestand ervan gesynchroniseerd moeten worden met die van de andere replica's. Gedurende deze periode moeten nieuwe binnenkomende berichten gequeued worden totdat het herstel van de uitgevallen replica voltooid is en het proces verder kan met het behandelen van berichten.

Deze vereisten moeten door het foutentolerantiesysteem gerealiseerd worden. In volgende hoofdstukken zullen we in detail behandelen hoe dit concreet gebeurt.

7.5 Transparante replicatie

Een belangrijk gewenst aspect van replicatie ten opzichte van de applicatie en de applicatieprogrammeur is transparantie. Stel bijvoorbeeld dat een gedistribueerde Borg-applicatie uit 5 interagerende applicatie-agenten bestaat, die zich op verschillende netwerkcomputers bevinden. We repliceren nu elke applicatie-agent bijvoorbeeld 4 keer en verspreiden vervolgens de replica's over verschillende netwerkcomputers. Na deze replicatie onstaat er interactie tussen de verschillende gerepliceerde applicatie-agenten, in dit geval zijn er dat 25. Het is niet aangewezen om de applicatieprogrammeur op te zadelen met deze replicatie en spreiding (door migratie over het netwerk). Het repliceren van applicatie-agenten en vervolgens verspreiden van de replica's ligt trouwens buiten het domein van die applicatie en zou alleen maar de complexiteit ervan verhogen. Daarom moet de replicatie en de spreiding van de verschillende replica's over het netwerk, transparant gebeuren voor de applicatie (en de applicatieprogrammeur). Dit wordt concreet gerealiseerd door een overname- en spreidingsalgoritme dat we in volgend hoofdstuk zullen bespreken.

7.5.1 Groepen van replica's

We zullen de verschillende replica's van één applicatie-agent groeperen. Als we het vorig voorbeeld hernemen, kunnen we de 25 applicatie-agenten in 5 groepen verdelen. Elke groep bevat dan 1 oorspronkelijke applicatie-agent en 4 replica's ervan. Elke groep is geassocieerd met een unieke referentie waardoor replica's van verschillende replicagroepen niet rechtstreeks met elkaar communiceren maar er enkel communicatie tussen de groepen bestaat via de groepsreferenties. Hierdoor worden ook de verschillende eigenschappen van de ene replicagroep verborgen voor de andere: het aantal replica's van de replicagroep, de namen (referenties) van die replica's, mogelijk uitgevallen replica's en de interactie binnen de groep.

De *groepsreferentie* stelt concreet een agent voor, namelijk de *replicamanager* van een replicagroep. Deze agent zal onder andere de multicasting van binnenkomende berichten naar de verschillende replica's van de groep verzorgen en zal er ook voor zorgen dat er telkens maar 1 antwoord wordt teruggestuurd. Met de termen "groep" en "replicagroep" bedoelen we vanaf nu de groep die bestaat uit x aantal replica's van één applicatie-agent, en een replicamanager die deze replica's beheert. Replicagroepen en hun verschillende functionaliteiten zullen in Hoofdstuk 9 verder besproken worden.

7.6 Conclusie

In dit hoofdstuk hebben we een overzicht gegeven van een mogelijk foutentolerantiesysteem in Borg dat gebruik maakt van replicatie. Deze replicatie zal actief en transparant gebeuren en zal partial failures ofwel volledig opvangen ofwel geheel niet opvangen, namelijk wanneer volledig herstel onmogelijk is. Dit transparant foutentolerantiesysteem moet aan bepaalde eisen voldoen. We sommen deze nog eens kort op:

- De replica's moeten op elk moment consistent zijn.
- De replicatie van applicatie-agenten en de daarop volgende spreiding van de replica's moet transparant gebeuren ten opzichte van de applicatie zelf.
- De replica's worden logisch gegroepeerd. Voor elke replicagroep staat een replicamanager in als referentie van die groep.

Hoofdstuk 8

Overname en spreiding

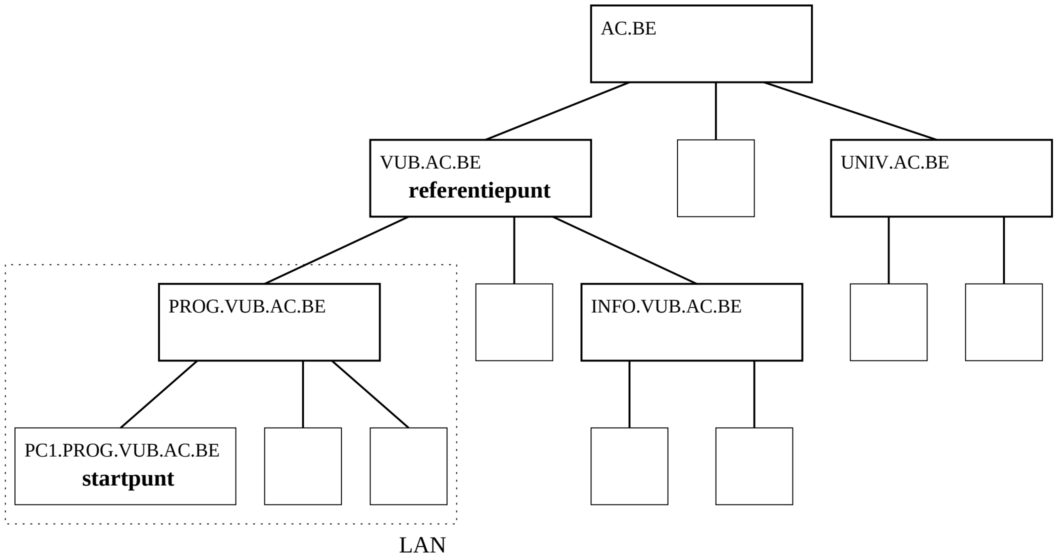
8.1 Inleiding

In het foutentolerantiesysteem kunnen we 2 opeenvolgende fasen onderscheiden: het starten van de applicatie in het foutentolerantiesysteem en het runnen van die applicatie. In dit hoofdstuk zullen we de eerste fase bespreken. Deze fase bestaat uit de opeenvolgende stappen van (1) overname van de applicatie-agenten door replicamanagers en (2) de spreiding van de replica's over het netwerk. We zullen beginnen met de plaats van de overname van de applicatie-agenten in het netwerk te bepalen.

8.2 Plaats van overname

De plaats van overname van de applicatie-agenten wordt bepaald door de plaats waar de applicatie in het netwerk wordt gestart. Dit is meestal in een local-area netwerk (LAN). We nemen immers aan dat de meeste applicatie-agenten in het LAN runnen daar waar ook de meeste gebruikers van de applicatie zich bevinden. Figuur 8.1 toont een voorbeeld van een hiërarchisch wide-area netwerk (WAN). Zoals in Hoofdstuk 6 worden netwerkcomputers voorgesteld door een rechthoek met dunne rand en nameservers/routers door een rechthoek met een dikke rand. Wanneer bijvoorbeeld in het LAN met nameserver/router prog.vub.ac.be een applicatie in het foutentolerantiesysteem gestart wordt, dan is het *startpunt* (plaats van overname) een netwerkcomputer in dat LAN, bijvoorbeeld netwerkcomputer pc1.prog.vub.ac.be.

Naast het startpunt duidt Figuur 8.1 de nameserver/router vub.ac.be als centraal *referentiepunt* aan. Dit referentiepunt is per definitie een nameserver/router die 2 stappen hoger in de WAN-hiërarchie ligt ten opzichte van het startpunt en zal gebruikt worden als "goede" referentie bij een splitsing van het netwerk. Merk op dat we in dit voorbeeld over *de* nameserver/router vub.ac.be spreken die 2 stappen hoger in de WAN-hiërarchie ligt. Het is namelijk mogelijk dat nameserver/router prog.vub.ac.be verbonden is met meerdere nameservers/routers. In dat geval mag maar één enkele nameserver/router als centraal referentiepunt gekozen worden. Dit zal de nameserver/router zijn die het snelst op een ping van de nameserver/router een niveau lager in de hiërarchie reageert. Agenten (replica's en replicamangers) die na een splitsing van het



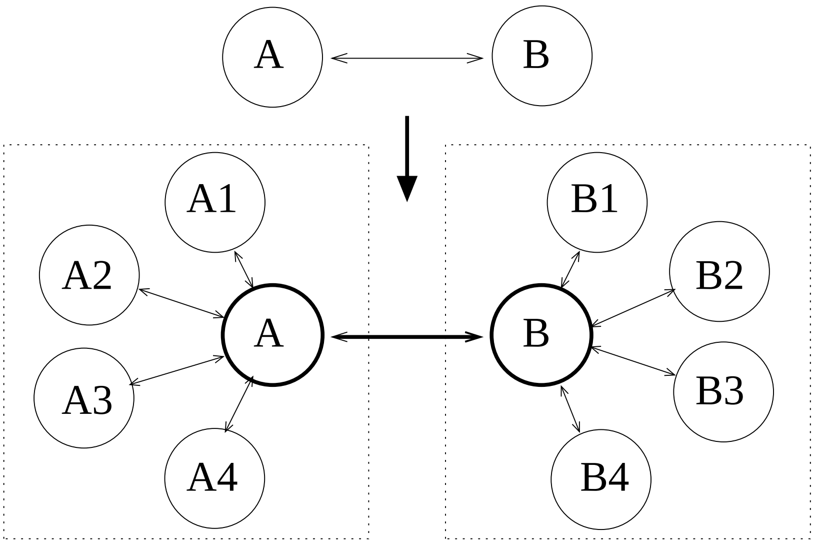
Figuur 8.1: Voorbeeld van de plaats van overname en van de bepaling van het centraal referentiepunt in een wide-area netwerk.

netwerk dit referentiepunt kunnen contacteren, moeten blijven leven. De agenten die na de splitsing dit referentiepunt niet meer kunnen contacteren zijn gedoemd om te sterven. Dit referentiepunt bepaalt met andere woorden de "goede" kant van het netwerk. We komen hier in volgend hoofdstuk op terug.

8.3 Overname door replicamanagers

Nu we de plaats van overname van applicatie-agenten hebben bepaald, kunnen we overgaan tot de overname door replica-managers zelf. Door de overname zullen replicagroepen gevormd worden, 1 voor elke applicatie-agent. Aan het hoofd van zo'n groep staat een replicamanager. Dit is een agent die de replica's (leden) van een groep zal beheren. Hij bevat management-code en is makkelijk te herconstrueren na failure. Elke replica bevat code om een nieuwe replicamanager te creëren. De interne toestand van de replicamanager wordt door zijn replica's bijgehouden waardoor deze na failure hersteld kan worden. De communicatie die tussen de applicatie-agenten zou ontstaan tijdens het runnen van de applicatie, zal in het foutentolerantiesysteem tussen de replicamanagers gebeuren. De communicatie tussen replicamanager en replica's van een groep wordt hierdoor voor de andere groepen verborgen gehouden.

Figuur 8.2 toont een applicatie bestaande uit 2 interagerende agenten, de applicatie-agenten A en B. We duiden applicatie-agenten en de replica's ervan aan met cirkels met een dunne rand. Replicamanagers duiden we aan met cirkels met een dikke rand. Merk op dat we de unieke namen van agenten vereenvoudigd hebben door ze te vervangen door unieke hoofdletters. Door de overname van applicatie-agenten door replicamanagers en de replicatie van die applicatie-agenten, ontstaan replicagroepen. In dit voorbeeld vertrekken we van 2 applicatie-agenten, A en B. Na de overname hebben we 2 replicagroepen.



Figuur 8.2: Twee interagerende applicatie-agenten A en B voor en na de overname door replicamanagers.

Elke replicamanager, die als groepsreferentie fungeert, zal de oorspronkelijke naam van applicatie-agent behouden. De overname van de applicatie-agenten houdt dus ook de overname van hun namen in. De naam van een replica van een groep bestaat uit de naam van de replicamanager van die groep en een rangnummer dat uniek is binnen de groep. In dit voorbeeld hebben we in de replicagroep van replicamanager A, 4 replica's: $A1$, $A2$, $A3$ en $A4$. In de replicagroep van replicamanager B treffen we eveneens 4 replica's aan: $B1$, $B2$, $B3$ en $B4$. De functie van dit rangnummer is enerzijds elke replica van een unieke naam voorzien, anderzijds zal dit rangnummer bepalen welke replica van de groep de uitgevallen replicamanager mag herstellen. Hierop komen we in volgend hoofdstuk terug.

We kunnen deze overname opsplitsten in opeenvolgende stappen:

1. Voor elke applicatie-agent wordt een replicamanager aangesteld. Zo'n replicamanager zal in volgende stappen zijn applicatie-agent een meegegeven x aantal keer repliceren.

2. Elke replicamanager repliceert zijn applicatie-agent een eerste maal en geeft deze het rangnummer 1 mee (agenten repliceren in Borg gebeurt door kloning).

3. Elke replica-manager verwisselt van naam met zijn applicatie-agent. Hierdoor neemt een replicamanager de naam van zijn applicatie-agent over.

4. Elke replicamanager doodt zijn applicatie-agent. Deze is trouwens volledig overgenomen en niet meer nodig (de naam ervan is overgenomen door de replicamanager en er bestaat reeds een eerste replica).

5. De replicamanager injecteert extra code in body van de replica. Deze

code zal nodig zijn om: (1) alle buitengaande berichten naar de replicamanager te sturen, (2) creatie van nieuwe agenten over te laten aan de replicamanager en (3) de uitgevallen replicamanager te kunnen herstellen.

6. De eerste replica wordt $x - 1$ aantal keer gerepliceerd. Elke nieuwe replica krijgt een uniek rangnummer mee. Alle rangnummers in een groep volgen elkaar op.

We kunnen dit algoritme (en volgende algoritmes) formeel beschrijven met behulp van pseudo-code. De pseudo-code die we hiervoor hanteren gelijkt sterk op Borg. De syntax voor message passing en variable assignment is bijvoorbeeld rechtstreeks uit Borg overgenomen. Een implementatie van dit algoritme kan men vinden in Bijlage E. Het overname-algoritme neemt 2 inputs: een

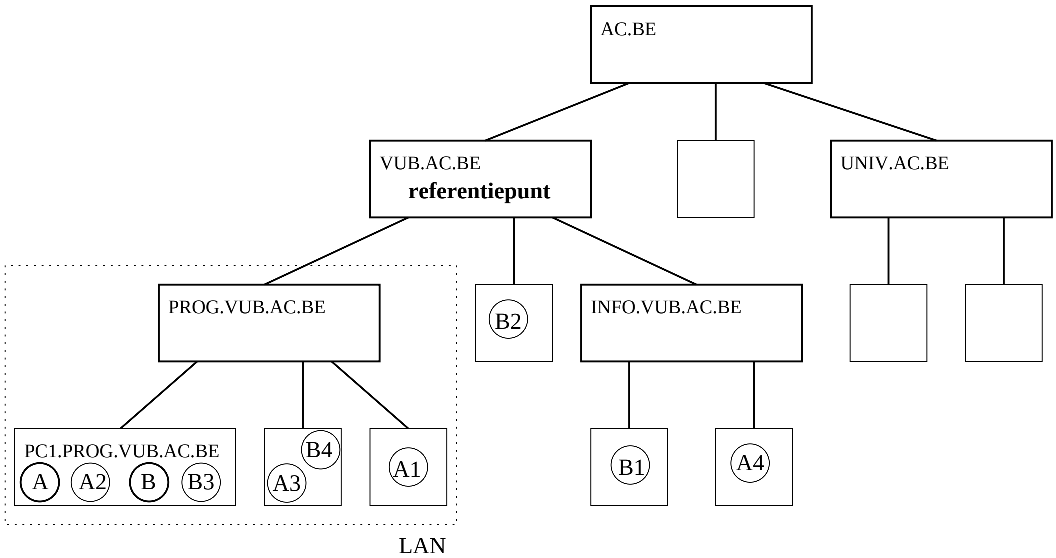
Algoritme 1 Overname door replicamanagers

```
takeover (agentlist, numberofreplicas)
for each agent A in agentlist do
  RMA : createreplicamanager()
  nameforreplica : A1
  RMA → replicate(A, nameforreplica)
  swapnames(RMA, A)
  A → killagent(RMA)
  A1 → load(forwardoutgoingmessages_code)
  A1 → load(agentcreation_code)
  A1 → load(recoveringreplicamanager_code)
  for i : 2 to numberofreplicas do
    nameforreplica : Ai
    A → replicate(A1, nameforreplica)
  end for
end for
```

lijst van applicatie-algoritmen die moeten worden overgenomen en het gewenst aantal van replica's per replicagroep. Merk op dat het aanduiden van applicatie-agenten voor replicatie en het gewenst aantal replica's bepaald wordt door de gebruiker van het foutentolerantiesysteem. Dit staat echter los van de applicatie zelf en we kunnen bijgevolg nog steeds over transparante replicatie spreken. Het gewenste aantal replica's is sterk afhankelijk van de uitgebreidheid van het netwerk. Wanneer we dit algoritme toepassen op de beginstituatie in het voorbeeld in Figuur 8.2, dan hebben we de inputs (A, B) en 4. Merk op dat de tijdelijke naam van de replicamanager bestaat uit de string "RM" met de naam van de applicatie-agent daar achter. Naast uit logische code bestaat de body van een replica ook nog uit extra code die ingeladen werd in een eerste replica (de andere replica's zijn replica's van die eerste replica). De inhoud van deze extra code zullen we in volgend hoofdstuk bespreken. Een implementatie in Borg van dit idee kan men vinden in bijlage D.

8.4 Spreiding van replica's

Na de overnamestap staan alle replica's en replicamanagers waaruit de applicatie in het foutentolerantiesysteem bestaat, op één enkele netwerkcomputer



Figuur 8.3: Voorbeeld van spreiding van replica's over een wide-area netwerk.

(het startpunt van de applicatie). Een volgende stap is het verspreiden van deze agenten over een deel van het netwerk. We willen met deze spreiding een interessante begintoestand verkrijgen waarin de applicatie in het foutentolerantiesysteem zal beginnen runnen. We werken echter met actieve, mobiele agenten die op elk moment tijdens executie kunnen migreren over het netwerk. De spreiding zal daarom eerder willekeurig ('at random') gebeuren. Hierbij volgen we deze regels:

1. Elke replicamanager blijft op het startpunt.

2. Voor snelheidsperformantie blijft van elke replicagroep één random-gekozen replica op het startpunt. Het rangnummer van de replica is niet van belang voor de spreiding ervan.

3. De ene (random-gekozen) helft van overige replica's worden random verdeeld over de netwerkcomputers in het LAN van het startpunt.

4. De andere helft van de overige replica's wordt random verspreid over de overige netwerkcomputers onder het referentiepunt.

Figuur 8.3 toont een voorbeeld van spreiding. Hierbij worden de replicagroepen A en B uit Figuur 8.2 verspreid in het wide-area netwerk van Figuur 8.1. Op het startpunt pc1.vub.ac.be blijven de replicamanagers A en B runnen, net als één random-gekozen replica van elke replicagroep, in dit voorbeeld zijn dat replica's $A2$ en $B3$. De overige 6 replica's worden gemigreerd naar andere netwerkcomputers. Drie random-gekozen replica's migreren random binnen het LAN waar het startpunt zich bevindt, dit zijn de replica's $A1$, $A3$ en $B4$. De overige replica's worden random verspreid over de andere netwerkcomputers onder het referentiepunt. In dit voorbeeld zijn dat de replica's $A4$, $B1$ en $B2$ die onder de nameserver/router vub.ac.be worden verspreid. Op dit moment is de spreiding voltooid en kan de applicatie in het foutentolerantiesysteem starten met runnen.

We kunnen dit spreidingsalgoritme formeel beschrijven met pseudo-code. Algoritme 2 bestaat uit twee delen: een hulpalgoritme voor het bepalen van

Algoritme 2 Verspreiden van replica's

listnetworkcomputers ($nameserverrouter$)

```
create empty list lst
let nameserverrouter perform a ping to all its machines underneath
nameserverrouter starts receiving pongs from network computers and adds
  them to lst in the order they come in
return lst
```

migratereplicas
($startpoint$, $localnsr$, $referencepoint$, $listreplicagroups$)

```
for each replica group in listreplicagroups do
   select randomly 1 replica r
   leave r at startpoint and ignore r in next steps
end for
listlocalhosts : listnetworkcomputers(localnsr) - startpoint
listotherhosts : listnetworkcomputers(referencepoint)
divide replicas in listreplicagroups randomly in part1 and part2
migrate replicas of part1 randomly to network computers of listlocalhosts
migrate replicas of part2 randomly to network computers of listotherhosts
```

de netwerkcomputers onder een meegeven nameserverrouter als input, en het eigenlijke spreidingsalgoritme voor replica's. Het spreidingsalgoritme neemt 4 inputs: het startpunt, de nameserverrouter van het LAN waarin het startpunt zich bevindt, het referentiepunt, en de lijst van replicagroepen waaruit de applicatie in het foutentolerantiesysteem bestaat. Merk op dat replicamanagers niet in het algoritme voorkomen, deze blijven op het startpunt en moeten dus niet verspreid (gemigreerd) worden. Dit is ook het geval voor één random-gekozen replica van elke groep.

8.5 Conclusie

We hebben in dit hoofdstuk de eerste fase van het transparante foutentolerantiesysteem besproken. Deze fase kunnen we opdelen in 2 opeenvolgende stappen. Tijdens de eerste stap zullen applicatie-agenten overgenomen worden door replicamanagers. Een replicamanager zal ervoor zorgen dat er replica's van een bepaalde applicatie-agent worden aangemaakt, en beheert, als leider van de replicagroep, deze replica's. Elke replica van een replicagroep kan zijn replicamanager volledig herstellen na failure. In een tweede stap zal het grootste deel van de replica's uit de verschillende replicagroepen verspreid worden over het netwerk om zo een interessante beginsituatie te creëren voor de applicatie die dan kan starten met runnen in het foutentolerantiesysteem. In volgend hoofdstuk zullen we de verschillende functionaliteiten van zo'n replicagroep bespreken.

Hoofdstuk 9

Replicagroepen

9.1 Inleiding

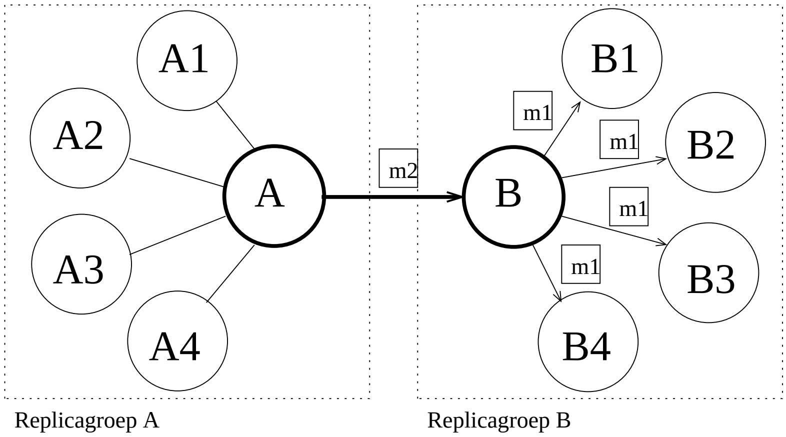
Na overname en spreiding bestaat een applicatie in het foutentolerantiesysteem uit replicagroepen. Een replicagroep is een logische groepering van één replicamanager (de groepsreferentie) en meerdere actieve replica's van één applicatie-agent. In dit hoofdstuk zullen we de verschillende functionaliteiten van een replicamanager en van actieve replica's bespreken en hoe de vereisten voor consistentie van deze replica's (zie sectie 7.4) gerealiseerd worden. Zoals in vorig hoofdstuk zullen we de verschillende algoritmen formeel met pseudo-code beschrijven. Implementaties in Borg kan men terugvinden in de verschillende bijlagen van deze thesis.

9.2 Replicamanager

Een replicamanager is een agent die tijdens de overname van applicatie-agenten wordt gecreëerd. Elke replicamanager krijgt één applicatie-agent toegewezen. Een eerste taak van de replicamanager bestaat erin die applicatie-agent over te nemen en te repliceren, zoals reeds besproken in vorig hoofdstuk. Door deze overname krijgt de replicamanager de naam van de applicatie-agent. De replicamanager zal extra code in de applicatie-agent laden die nodig is voor de replica's om (1) alle buitengaande berichten naar de replicamanager door te sturen, (2) het creëren van nieuwe agenten over te laten aan de replicamanager, en (3) de replicamanager te herstellen wanneer die door failure is uitgevallen. Wat de inhoud deze extra code is zullen we in dit hoofdstuk zien.

Na de overname en de daarop volgende spreiding van de verschillende replica's, gebeurt de interactie tijdens het runnen van de applicatie in het foutentolerantiesysteem niet meer tussen de oorspronkelijke applicatie-agenten[1], maar tussen de replicamanagers van de verschillende replicagroepen, en tussen de replicamanager zijn replica's onderling. We bespreken nu de functionaliteiten van een replicamanager en hoe deze de verschillende vereisten voor consistentie van zijn replica's realiseert.

[1] Het mogelijk is dat niet alle applicatie-agenten overgenomen en gerepliceerd worden, zie subsectie 7.3.1.



Figuur 9.1: Voorbeeld van het doorsturen van elk binnenkomend bericht door de replicamanager naar zijn replica's.

9.2.1 Doorsturen en tegenhouden van berichten

De taak van de replicamanager als groepsreferentie bestaat uit het doorsturen van elk bericht dat bij hem binnenkomt naar al zijn replica's in dit in dezelfde volgorde voor elke replica. Het doorsturen van ontvangen berichten door de replicamanager in dezelfde volgorde naar al zijn replica's noemt men ook wel *totaal-geordende multicasting* van berichten. Merk op dat hierdoor aan vereiste 1 voor consistentie van replica's voldaan wordt (zie sectie 7.4).

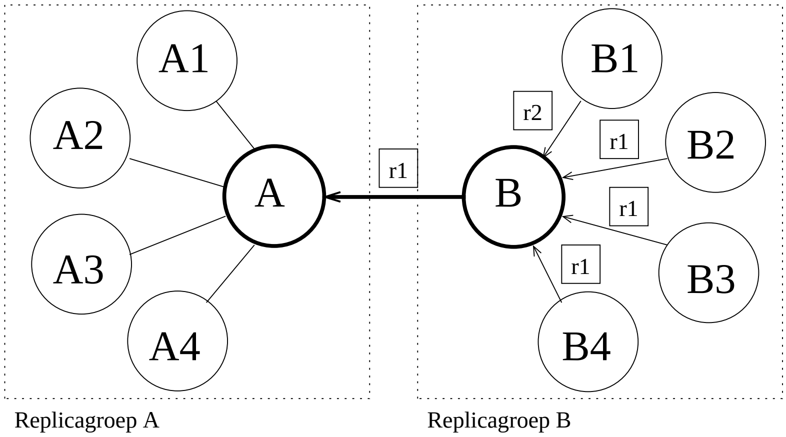
Figuur 9.1 toont een voorbeeld van het ontvangen en doorsturen van binnenkomende berichten door een replicamanager. Replicamanager B ontvangt van replicamanager A een eerste bericht $m1$ en stuurt deze door naar al zijn replica's: $B1$, $B2$, $B3$ en $B4$. Wanneer B het tweede bericht $m2$ ontvangt, gebeurt de multicasting van dat bericht analoog. Een replicamanager moet hiervoor alle referenties naar zijn replica's bijhouden als interne toestand.

Stel bijvoorbeeld dat een agent 3 verschillende berichten op op ongeveer hetzelfde moment ontvangt. Intern worden deze berichten in een receive-queue gestoken en *één voor één* afgehandeld door de agent. Hierdoor worden binnenkomende berichten één voor één doorgestuurd naar de replica's in dezelfde volgorde als ze bij de replicamanager zijn binnengekomen. Elke replica ontvangt hierdoor de berichten in dezelfde volgorde waardoor we over een totaal-geordende multicasting van berichten kunnen spreken. We moeten bijgevolg geen extra mechanismen voor *totaal-geordende* multicasting voorzien.

Algoritme 3 Doorsturen van binnenkomende berichten

multicast (*inmessage*, *listreplicas*)
for each replica r in *listreplicas* **do**
    $r \rightarrow$ receiveandprocess(inmessage)
end for

Algoritme 3 beschrijft formeel het doorsturen van binnenkomende berichten naar replica's. Een implementatie in Borg van dit idee kan men vinden in Bijlage C.

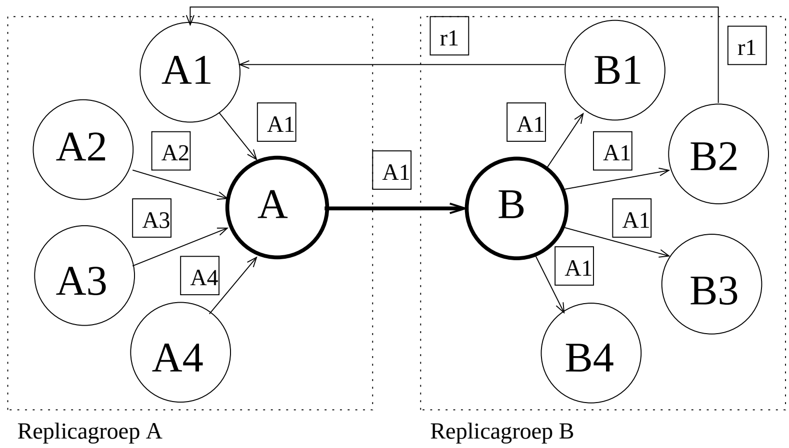


Figuur 9.2: Voorbeeld van het doorsturen van één enkel bericht van de snelste replica door de replicamanager. De andere berichten worden door de replicamanager tegengehouden.

Elke replica ontvangt doorgestuurde berichten van de replicamanager en behandelt deze. Als antwoord hierop kunnen berichten teruggestuurd worden naar de replicamanager. Elke replica zal dezelfde berichten naar de replicamanager sturen. Deze mag echter maar één van die berichten doorsturen naar de ontvanger om aan vereiste 3 voor consistentie van replica's te voldoen.

We hernemen het voorbeeld van Figuur 9.1 met Figuur 9.2. Stel dat op de berichten $m1$ en $m2$ respectievelijk de berichten $r1$ en $r2$ door elke replica doorgestuurd worden naar de replicamanager. Replicamanager B heeft 4 replica's. Dit betekent dat hij voor elk doorgestuurd bericht, 4 dezelfde berichten van zijn 4 replica's mag terug verwachten. Van deze 4 berichten mag maar één doorgestuurd worden naar replicamanager A. De 3 andere berichten moeten door B tegengehouden worden opdat de replica's van replicamanager A niet 4 keer hetzelfde bericht zullen ontvangen en behandelen. In het voorbeeld van Figuur 9.2 is replica $B1$ de snelste met het doorsturen van bericht $r1$ naar zijn replicamanager B, waardoor B op zijn beurt dit bericht van $B1$ zal doorsturen naar A. De andere berichten van zijn replica's ($B2$, $B3$, en $B4$) zal B tegenhouden. Een replicamanager wacht niet op een meerderheid of op alle berichten om er vervolgens één door te sturen. Hierdoor zal de ontvanger zijn antwoord op de snelst mogelijke manier ontvangen.

Voor het doorsturen en tegenhouden van berichten van replica's, moet de replicamanager een berichtenteller voor elke replica bijhouden als interne toestand. Zo'n berichtenteller wordt opgehoogd wanneer de replicamanager een bericht van de bepaalde replica ontvangt. Wanneer de opgehoogde berichtenteller groter is dan de berichtentellers van de andere replica's, dan wordt het bericht door de replicamanager doorgestuurd. Indien niet, betekent dit dat het bericht al is doorgestuurd en wordt het door de replicamanager tegengehouden opdat de ontvangende agent hetzelfde bericht niet meerdere keren zou ontvangen en behandelen.



Figuur 9.3: Voorbeeld van wat er misloopt wanneer buitengaande berichten niet door de replicamanager geparst worden op replicanamen.

Algoritme 4 beschrijft formeel het doorsturen of tegenhouden van berichten van de replica's door de replicamanager. Een implementatie in Borg van dit idee kan men vinden in Bijlage D.

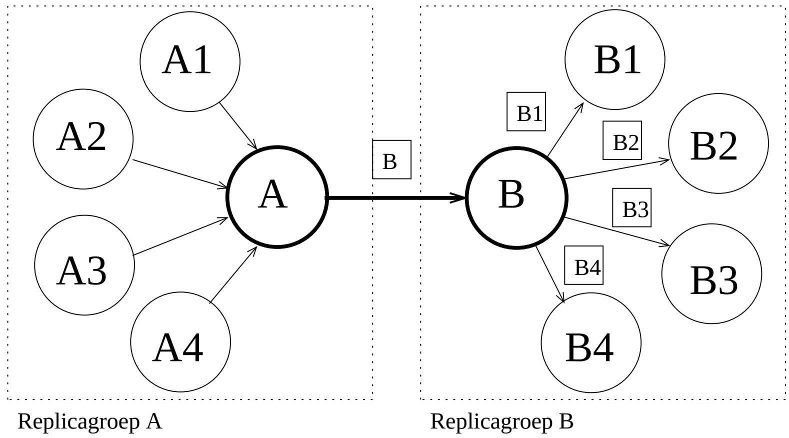
Algoritme 4 Doorsturen of tegenhouden van buitengaande berichten

handlereplicamessage ($message$, $fromreplica$, $toagent$)

increment message counter ctr of $fromreplica$
if $ctr >$ message counters of the other replicas **then**
  toagent $\rightarrow$ revceiveandprocess($message$)
end if

9.2.2 Parsen en aanpassen van berichten

De naam van een replica is het enige verschil dat replica's van een replicagroep met elkaar hebben. Een probleem kan zich voordoen wanneer replica's hun eigen naam meegeven in berichten die ze doorsturen naar hun replicamanager. In Figuur 9.3 beschouwen we opnieuw dezelfde replicagroepen A en B. Stel nu bijvoorbeeld dat de replica's van A, door het behandelen van een bepaald bericht bijvoorbeeld, elk een bericht naar A sturen en hierop een antwoord terug verwachten en daarvoor hun naam in het bericht meegeven. Replicamanager A zal het eerst binnenkomend bericht doorsturen naar B en de andere tegenhouden. In dit voorbeeld wordt het bericht van replica $A1$ door A naar B gestuurd. Replicamanager B zal op zijn beurt het bericht doorsturen naar al zijn replica's. De replica's van B behandelen het bericht en zullen allen een antwoordbericht $r1$ naar replica $A1$ sturen doordat de naam van $A1$ in het bericht dat ze hebben behandeld, is meegegeven. Hierdoor worden aan vereisten 2 en 3 voor de consistentie van replica's niet voldaan. De replica's van replicagroep B communiceren rechtstreeks met een replica van replicagroep A, in dit voorbeeld met replica $A1$. Omdat de communicatie niet via de replicamanagers gebeurt, zal replica $A1$ ook 4 dezelfde berichten van de replica's van replicagroep B ont-



Figuur 9.4: Voorbeeld hoe een binnenkomend bericht door een replicamanager geparst wordt op eigen naam en indien nodig aangepast wordt naar de naam van elke replica.

vangen en uitvoeren. Dit probleem kan opgelost worden door replicamanagers elk buitengaand bericht van hun replica's te laten parsen. Indien buitengaande berichten namen van replica's bevatten, moeten deze namen veranderd worden door de naam van de replicamanager.

Dit probleem doet zich ook voor in de omgekeerde richting aangezien een replicamanager een verschillende naam heeft dan zijn replica's. We beschouwen figuur 9.4. Replicamanager A stuurt een bericht naar B en dat bericht bevat de naam van replicamanager B. Wanneer B binnenkomende berichten ontvangt, dan zal hij deze moeten parsen op eigen naam. Telkens wanneer zijn naam in het bericht voorkomt, dan zal deze naam veranderd moeten worden naar de naam van de ontvangende replica voordat het bericht doorgestuurd wordt naar die replica. In dit voorbeeld parst B het binnenkomend bericht en verandert indien nodig zijn naam door de naam van de ontvangende replica, namelijk $B1$, $B2$, $B3$, of $B4$.

Algoritme 5 beschrijft formeel het parsen en mogelijk aanpassen van binnenkomende en buitengaande berichten. Een implementatie in Borg van dit

Algoritme 5 Parsen en mogelijk aanpassen van binnenkomende of buitengaande berichten

parsemessage
($message$, $fromagent$, $toagent$, $namereplicamanager$)

1: if not at end of $message$, continue traversing $message$;
when an agent name $name$ is found, go to step 2
2: **if** $name = namereplicamanager$ **then**
3: $name := toagent$
4: **else if** $name = fromagent$ **then**
5: $name := namereplicamanager$
6: **else**
7: go back to step 1
8: **end if**

idee kan men vinden in Bijlage A en B. Merk op dat Algoritme 3 en 4 moeten worden aangepast zodanig dat een bericht eerst geparst wordt voordat het doorgestuurd wordt.

9.2.3 Onderscheppen van agentcreatie

Het behandelen van berichten door de replica's van een replicagroep kunnen creaties van nieuwe agenten tot gevolg hebben. Een nieuwe agent mag maar één enkele keer gecreëerd worden. Om te voorkomen dat elke replica eenzelfde nieuwe agent creëert, zal de taak van agentcreatie overgelaten worden aan de replicamanager.

Het mechanisme van onderscheppen van agentcreatie gebeurt gelijkaardig aan het doorsturen/tegenhouden van buitengaande berichten. Een replicamanager houdt voor elke replica een agentcreatie-teller bij. Zo'n agentcreatie-teller wordt opgehoogd wanneer de replicamanager een agentcreatie-aanvraag van de bepaalde replica ontvangt. Wanneer de opgehoogde agentcreatie-teller groter is dan de agentcreatie-tellers van de andere replica's, dan wordt de agent door de replicamanager gecreëerd en vervolgens overgenomen en gerepliceerd door een andere, nieuwe replicamanager. Indien niet, betekent dit dat de agent reeds gecreëerd is en wordt de agentcreatie-aanvraag afgewezen.

Algoritme 6 Onderscheppen van agentcreatie door de replicamanager

handleagentcreation
$(namenewagent, fromreplica, numberofreplicas)$

```
increment agent creation counter ctr of fromreplica
if ctr > agent creation counters of the other replicas then
   createagent(namenewagent)
   takeover((namenewagent), numberofreplicas)
end if
```

Algoritme 6 beschrijft formeel het onderscheppen van agentcreatie door de replicamanager. Merk op dat hierdoor aan vereiste 4 voor consistentie van replica's wordt voldaan.

9.2.4 Detecteren en herstellen van uitgevallen replica's

Wanneer een applicatie in het foutentolerantiesysteem start met runnen, bestaat elke replicagroep van de foutentolerante applicatie uit x aantal replica's. Elke replicamanager zal ervoor moeten zorgen dat tijdens het runnen van de applicatie er steeds x aantal replica's up-and-running zijn. Een replicamanager moet met andere woorden in staat zijn om uitgevallen replica's in zijn replicagroep te *detecteren* en te *herstellen*. Replica's vallen uit ten gevolge van failures in de communicatie of failures van processen, zoals reeds besproken.

Tijdens de periode van herstel moeten nieuwe binnenkomende berichten wachten om afgehandeld te worden totdat alle uitgevallen replica's hersteld zijn. Dit is vereiste 5 voor consistentie van replica's.

Detectie

Een replicamanager stuurt om de 60 seconden een ping-bericht naar elke replica. Wanneer een replica zo'n ping-bericht ontvangt, stuurt deze onmiddellijk een pong-bericht naar de replicamanager terug. Wanneer de replicamanager na 60 seconden (na het versturen van het eerste ping-bericht) geen pong-bericht ontvangt van de replica, dan zal de replicamanager een tweede ping-bericht sturen en opnieuw 60 seconden wachten op een pong-bericht. Wanneer uiteindelijk geen pong-bericht ontvangen wordt, dan beschouwt de replicamanager deze replica als zijnde uitgevallen. Algoritme 7 beschrijft formeel het detecteren van een uitgevallen replica.

Algoritme 7 Detecteren van uitgevallen replica's

alivecheck (*replica*)

```
send a ping message to replica and wait for a pong message
if no pong message is received after 60 seconds then
    send a second ping message to replica
end if
if still no pong message is received after 60 seconds then
    return false
end if
```

De eerste stap die een replicamanager hierna onderneemt, is het verwijderen van de naam van de uitgevallen replica uit de referenties naar al zijn replica's. Deze uitgevallen replica kan ofwel gestorven zijn, ofwel voor onbepaalde duur afgesloten zijn van de replicamanager doordat de connectie tussen beiden verbroken werd door bepaalde failure(s) in de communicatie. In het tweede geval is het mogelijk dat, na herstelling van de connectie tussen replicamanager en replica, de replica opnieuw contact opzoekt met zijn replicamanager. Wanneer een replicamanager een bericht ontvangt van een replica dat niet in zijn lijst staat, dan zal hij deze replica doden (deze replica is reeds als uitgevallen beschouwd).

In een volgende stap zal de replicamanager het referentiepunt pingen (op eenzelfde manier als het pingen van replica's). Wanneer de replicamanager het referentiepunt niet kan contacteren, betekent dit dat het netwerk gesplitst is en dat de replicamanager zich aan de "slechte" kant bevindt en gedoemd is om zichzelf te doden[2]. Indien de replicamanager het referentiepunt wel kan contacteren, bevindt hij zich aan de "goede" kant en gaat hij over tot de herstelling van de uitgevallen replica om weer x aantal replica's up-and-running te krijgen.

Herstelling

De herstelling van een replica gebeurt door replicatie van een replica dat niet is uitgevallen. Dit kan enkel gebeuren wanneer replica's geen berichten aan het behandelen zijn omdat het repliceren (klonen) van replica's die op dat moment berichten aan het behandelen zijn, tot inconsistentie kan leiden. De replicamanager houdt volgende, extra tellers bij.

[2] We kunnen nu al opmerken dat wanneer het referentiepunt (zie sectie 8.2) zelf uitvalt, de volledige applicatie gecontroleerd zal worden afgebroken. Alle replicamanagers en replica's kunnen immers het referentiepunt niet contacteren en zullen bijgevolg collectief zelfmoord plegen.

- Een **doorgestuurde-berichten-teller**. Deze teller wordt opgehoogd telkens wanneer de replicamanager een binnenkomend bericht doorstuurt naar al zijn replica's.

- Een **afgehandelde-berichten-teller voor elke replica**. Wanneer een replica een bericht volledig heeft afgehandeld, zal deze de replicamanager hiervan op de hoogte brengen door het sturen van een acknowledge-bericht. Wanneer de replicamanager het acknowledge-bericht ontvangt, wordt de afgehandelde-berichten-teller voor die replica opgehoogd.

Met behulp van deze 2 tellers kan de replicamanager bepalen welke replica's gerepliceerd mogen worden. Wanneer de doorgestuurde-berichten-teller gelijk is aan de afgehandelde-berichten-teller van een replica, dan betekent dat die replica op dat moment geen berichten aan het behandelen is en dus veilig door de replicamanager gerepliceerd kan worden.

De naam van de nieuwe replica bestaat uit de naam van de replicamanager en het eerst volgend, lager rangnummer. Merk op dat hierdoor de naam van de nieuwe replica uniek is.

Algoritme 8 Detecteren en herstellen van een uitgevallen replica door de replicamanager.

detectandrecover
($listreplicas$, $referencepoint$, $lowestrankingroup$)

for each 60 seconds **do**
  perform alivecheck on each replica in $listreplicas$
  if a dead replica dr is detected, remove dr from $listreplicas$ and perform a ping to $referencepoint$
  if pong received from $referencepoint$, go to next step, otherwise kill yourself
  wait for a replica lr where forwarded message counter = processed message counter
  replicate lr to create new replica with name = name of replicamanager + ($lowestrankingroup$ + 1)
end for

Algoritme 8 beschrijft formeel het detecteren en herstellen van een uitgevallen replica door de replicamanager.

9.3 Actieve replica's

Een replica is een kloon van één bepaalde applicatie-agent en neemt de taak ervan over, namelijk het runnen van de applicatie. Naast de logische code van de applicatie-agent, bevat de body van de replica nog extra code, ingeladen door de replicamanager tijdens de overnamefase. Deze extra code zal gebruikt worden voor de volgende *functionaliteiten*, die we grotendeels al besproken hebben in vorige sectie.

- Alle berichten van een replica worden naar de replicamanager gestuurd.

- Wanneer een bericht volledig is afgehandeld, moet de replicamanager hiervan op de hoogte gebracht worden door het sturen van een acknowledge-bericht.
- Creatie van nieuwe agenten wordt overgelaten aan de replicamanager door het sturen van een agentcreatie-aanvraag.
- Replica's moeten in staat zijn een uitgevallen replicamanager te detecteren en volledig te herstellen. Deze functionaliteit zullen we zo dadelijk verder bespreken.

Als *extra interne toestand* houdt elke replica volgende data bij:

- Referentie naar de replicamanager.
- Het referentiepunt.
- Referenties naar de andere replica's van de replicagroep.
- Een doorgestuurde-berichten-teller. Deze teller wordt opgehoogd telkens wanneer de replica een bericht naar de replicamanager doorstuurt (acknowledge-berichten worden hier niet meegerekend).
- Een agentcreatie-teller die opgehoogd wordt telkens wanneer een agentcreatie-aanvraag naar de replicamanager wordt gestuurd.
- Een afgehandelde-berichten-teller. Deze teller wordt opgehoogd nadat de replicamanager op de hoogte is gebracht van een afgehandeld bericht (door het sturen van een acknowledge-bericht).

Belangrijk is op te merk dat deze data ook door de replicamanager wordt bijgehouden. Hierdoor kunnen de replica's hun replicamanager volledig herstellen.

9.3.1 Detecteren en herstellen van uitgevallen replicamanager

Zoals reeds besproken in vorige sectie, voert de replicamanager alive-checks uit op zijn replica's door het regelmatig sturen van ping-berichten. Deze ping-berichten zullen door de replica's gehanteerd worden om de (mogelijk) uitgevallen replicamanager te kunnen detecteren. Wanneer een replica gedurende 120 seconden geen ping-bericht van de replicamanager heeft ontvangen, dan zal deze replica de replicamanager als zijnde uitgevallen beschouwen.

De eerste stap die de replica na het detecteren van de uitgevallen replicamanager onderneemt, is het pingen van het referentiepunt om te bepalen of hij mag overleven, of zichzelf zal moeten doden. Wanneer het referentiepunt met succes gepingd wordt, dan zal de replica de andere replica's van de replicagroep proberen te contacteren. De replica met het hoogste rangnummer en die zich aan de "goede" kant van het netwerk bevindt, zal de replicamanager moeten herstellen. Deze regel verzekert dat de replicamanager maar door één enkele replica zal worden hersteld. Wanneer de replica zo'n replica niet kan pingen, dan weet die replica dat het zijn taak is om de replicamanager te herstellen. Dit gaat als volgt te werk. De replica maakt een nieuwe replicamanager aan met dezelfde naam als die de uitgevallen replicamanager. Vervolgens moet de interne toestand van die nieuwe replicamanager worden upgedate naar de toestand waarin de vorige replicamanager is uitgevallen. Zoals reeds aangehaald is dit mogelijk omdat elke replica de data bijhoudt die de replicamanager ook bijhoudt.

9.4 Toegevoegde acknowledge-communicatie

In deze sectie zullen we een acknowledge-communicatie aankaarten die nodig is opdat berichten tussen replicamanagers niet verloren zouden gaan. We hernemen de replicagroepen A en B uit vorige voorbeelden. Stel nu bijvoorbeeld dat B een bericht van A ontvangt en dat B uitvalt voordat hij het bericht heeft kunnen doorsturen zijn replica's. Hierdoor gaat het bericht verloren. We kunnen dit probleem opvangen met behulp van het sturen van een acknowledge-bericht tussen B en A.

Wanneer A een bericht naar B stuurt, dan zal B het bericht doorsturen naar al zijn replica's. De replica's behandelen het bericht en sturen elk een acknowledge-bericht naar B wanneer het bericht volledig is afgehandeld, zoals reeds besproken in subsectie 9.2.4. Wanneer B een eerste acknowledge-bericht van één van zijn replica's ontvangen heeft, dan zal hij A hiervan op de hoogte brengen door het sturen van een acknowledge-bericht. Replicamanager A houdt op zijn beurt het verstuurde bericht bij totdat hij een acknowledge-bericht hiervoor van B heeft ontvangen.

9.5 Conclusie

In dit hoofdstuk hebben we de functionaliteiten van een replicagroep, bestaande uit een replicamanager en replica's, besproken en hoe de vereisten voor consistentie van replica's gerealiseerd worden. We hebben tenslotte aangekaart dat een toegevoegde acknowledge-communicatie tussen replicamanagers noodzakelijk is opdat berichten tussen replicagroepen niet verloren zouden gaan. In volgend conclusiehoofdstuk zullen we nog eens een kort overzicht geven van het transparant foutentolerantiesysteem dat we in dit deel uitvoerig besproken hebben.

Hoofdstuk 10

Conclusie

10.1 Inleiding

Het doel van deze thesis was om een foutentolerantiesysteem te ontwerpen voor mobiele multi-agent-systemen om hiermee partial failures van het netwerk op te vangen tijdens het runnen van gedistribueerde applicaties. We zijn hierbij vertrokken van gekende replicatietechnieken en hebben het mobiel multi-agent-systeem Borg als case study gehanteerd. Na een evaluatie van 2 bestaande foutentolerantiesystemen ondervonden we dat transparantie een gewenste eigenschap is van zo'n foutentolerantiesysteem. Enerzijds moet de applicatieprogrammeur dan niet zelf instaan voor foutentolerantie in zijn applicaties, anderzijds wordt de complexiteit van een applicatie niet onnodig verhoogd.

10.2 Ons ontwerp

Het foutentolerantiesysteem dat we ontworpen hebben, werkt volledig transparant en gebruikt actieve replicatie. Partial failures van netwerk worden door dit foutentolerantiesysteem volledig opgevangen of geheel niet opgevangen. Hierdoor vertonen applicaties die in het foutentolerantiesysteem runnen nooit een onbepaald gedrag: ofwel laten we de applicatie gecontroleerd afbreken wanneer volledig herstel onmogelijk is, ofwel herstellen we de applicatie volledig.

Het foutentolerantiesysteem is onder te verdelen in 2 fasen. De eerste fase is onder te verdelen in 2 stappen. Tijdens de eerste stap worden applicatie-agenten volledig overgenomen door replicamanagers. Een replicamanager zorgt er voor dat van de applicatie-agent actieve replica's worden aangemaakt, en beheert, als leider van de replicagroep, deze replica's. Elke replica van een replicagroep kan zijn replicamanager volledig herstellen na failure en omgekeerd. In een tweede stap wordt het grootste deel van de replica's uit de verschillende replicagroepen verspreid over het netwerk om zo een interessante beginsituatie te creëren voor de applicatie. De tweede fase bestaat dan in het werkelijk runnen van de applicatie onder dit systeem.

10.3 Voortzetting van deze thesis

Het huidig ontwerp ondersteunt enkel actieve replicatie van applicatie-agenten. Agenten met I/O-taken en/of niet-deterministische functies, en GUI-agenten kunnen niet actief gerepliceerd worden. Voor deze agenten is passieve replicatie aangewezen. Deze agenten kunnen ook in het foutentolerantiesysteem opgenomen worden wanneer passieve replicatie ondersteund wordt.

Nu we een ontwerp voor een foutentolerantiesysteem in Borg hebben uitgewerkt, kunnen we verdergaan met de volledige implementatie ervan.

Na implementatie kunnen performantietesten uitgevoerd worden. De uitgebreidheid en latentie van het netwerk, het aantal applicatie-agenten en het aantal replica's per applicatie-agent, de processorkracht van de verschillende netwerkcomputers, en de partial failures zelf zijn allemaal factoren die hierin meespelen en het schetsen van een algmeen beeld van de performantie complexer maken.

Bijlage A

AgTraversal.borg

```
{
‘ Author:  Werner Van Belle
‘ Reference:  borg/examples/MetaAgentProtocol/AgTraversal.borg

load("AvlTree.borg");

compare(ptr1,ptr2)::
   if(mem_ref(ptr1[1]) < mem_ref(ptr2[1]), 1,
      if (mem_ref(ptr1[1]) = mem_ref(ptr2[1]), 0, -1));
printidx(ptr)::printexp(ptr[1]);

traverse(expression, replacer)::
   {
    changes:MakeAvlTree(compare, printidx);
    traverse_aux(expression, replacer, changes)
   };
traverse_aux(expression, replace, changes)::
   {
   gointo(expression)::
      {
      enter()::
         {
         if(is_nbr(expression),
            replace(expression),
         if(markers.Exist([expression]),
            {
            result:markers.Search([expression]);
            result[2]
            },
            {
            replaced:replace(expression);
            markers.Add([expression,replaced]);
            process(replaced)
            }))
         };
```

```
      process(expression)::
         {
         siz:mem_siz(expression);
         if(siz=0,expression,
            {
            while(siz>0,
               {
               before:expression[siz];
               after:gointo(before);
               if(before!~after,
                  {
                  display("CHANGING !!!"+printexp(before)+"to "+printexp(after)
                  +"["+text(tag(before))+"]->["+text(tag(after))+"]"+eoln);
                  expression[siz]:=after;
                  changes.Add([expression,siz,before])
                  });
               siz:=siz-1
               });
            expression
            })
         };
      enter()
      };
   markers:MakeAvlTree(compare,printidx);
   gointo(expression)
   };

sortapplications(function)::
   {
   compareword(w1,w2)::if (w1>w2, 1, if(w1=w2, 0, -1));
   printword(w)::display(w);
   applications:MakeAvlTree(compareword,printword);
   storeit(exp)::
      {
      if(is_apl(exp),
      if(applications.Exist(text(get_apl_nam(exp))),void,
        applications.Add(text(get_apl_nam(exp)))));
      exp
      };
   traverse(function[3],storeit);
   applications.Print()
   };

printit(exp)::
   {
   print(exp);
   print(text(mem_ref(exp))+eoln);
   exp
   };
```

```
replaceapplication(exp)::
   if(is_apl(exp),
   if(exp[1]~quote(apply),
      exp,
      make_apl(quote(apply),[exp[1],exp[2]])),exp);

"traverse(printit[3],printit)"; "sortapplications(traverse_aux)"
"sortapplications(traverse_aux)"
}
```

Bijlage B

Masquerading.borg

```
{
‘ Author:  Werner Van Belle
‘ Reference:  borg/examples/MetaAgentProtocol/Masquerading.borg

load("AgTraversal.borg");

masquerade(exp, from, to)::
   {
   fromagent:text(rdc_agent(from));
   toagent:text(rdc_agent(from));
   changer(exp)::
      {
      if(!is_ref(exp), exp,
      if(strstr(text(exp), fromagent) != 1, exp,
         {
         texp:text(exp);
         newexp:toagent+substr(texp,size(fromagent)+1,size(texp));
         display("Changed "+ texp + -> "+ newexp);
         agent(newexp)
         }))
      };
   traverse(exp, changer)
   }
}
```

Bijlage C

ForwardIncoming.borg

```
{
‘ Author:  Werner Van Belle
‘ Reference:  borg/examples/MetaAgentProtocol/ForwardIncoming.borg

‘ to forward incoming messages:
‘ - we must override acceptmessage to send
‘   the message to the actual target
‘ - hereby we have to change all references to
‘   ourselve to the name of the masqueraded
‘   agent.
‘ If we want to reach the forwarder itself, eg,
‘ for stopping the forwarding,
‘ - we look at the content of a message and
‘   accept it with the old acceptmessage

load("Masquerading.borg");

realacceptmessage::acceptmessage; forwardto:void;
setforwardto(agt)::forwardto:=agt;
acceptmessage(msg)::
   {
   if(is_void(forwardto),
      realacceptmessage(msg),
      forwardto=>masquerade(msg,agentname,forwardto))
   };

}
```

Bijlage D

ForwardOutgoing.borg

```
{
‘ Author:  Werner Van Belle
‘ Reference:  borg/examples/MetaAgentProtocol/ForwardOutgoing.borg

‘ to forward outgoing messages (to a gateway
‘ for example):
‘ - we must override sendmessage to send the message
‘   to another target
‘ - hereby we don’t masquerade anything

load("Masquerading.borg");

realsendmessage::  =>;
forwardto:void;
setforwardto(agt)::forwardto:=agt;
=>(target,msg)::
   {
   if(is_void(forwardto),
      realsendmessage(target,msg),
      realsendmessage(forwardto,msg))
   };

}
```

Bijlage E

FilterInAndOut.borg

```
{
‘ Author:  Werner Van Belle
‘ Reference:  borg/examples/MetaAgentProtocol/FilterInAndOut.borg

load("Masquerading.borg")
realacceptmessage::acceptmessage;
forwardto:void;
setforwardto(agt)::forwardto:=agt;
‘ - if a message is from forwardto, we masquerade it
‘   and send it trough
‘ - if a message is targetted at ourselve, we masquerade
‘   it and send it to forwardto.

acceptmessage(msg)::
   {
   if(is_void(forwardto),
      realacceptmessage(msg),
   if(!is_async_msg(msg),
      display("Non asynchronous message !"+eoln),
      {
      from:async_msg_from(msg);
      target:async_msg_to(msg);
   if(from~forwardto,
      {
      msg:=masquerade(msg,forwardto,agentname);
      target:async_msg_to(msg);
      display("Outgoing message to "+ text(target)+eoln);
      target=>msg
      },
   if(target~agentname,
      { display("Incoming message from "+text(from)+eoln);
      msg:=masquerade(msg,agentname,forwardto);
      forwardto=>msg
      },
   ))}))
```

```
   };


createagent(A)::
   {
   clone2agent(A)
   };

takeover(A)::
   {
   B:agentname;
   `-----------------
   `  A         B
   `Alogic   rmanager
   `-----------------
   logger("Swapping names, first time"+eoln);
   swap_names(A);
   `-------------------
   `  B         A
   `Alogic   rmanager/A
   `-------------------
   logger("Injecting gateway logic in old shell");
   B->load("ForwardOutgoing.borg");
   `-------------------
   `  B         A
   `Alogic   rmanager/A
   `-------------------
   logger("Creating agent"+eoln);
   A1:createagent(A);
   `------------------------------
   `  B         A         A1
   `Alogic   rmanager/A
   `------------------------------
   logger("Sending swapnames to the original"+eoln);
   B->swap_names(A1);
   `-----------------------------
   `  A1        A          B
   `Alogic/A1  rmanager/A
   `-----------------------------
   logger("The shell may die"+eoln);
   B->agentdie()
   `------------------------------
   ` A1         A
   `Alogic/A1  rmanager/A
   `------------------------------

   `CreateAgent(A2)
   `------------------------------
   ` A1         A          A2
   `Alogic/A1  rmanager/A
```

```
    `-------------------------------
    ` A1->duplicateto(A2)
    `    (getstacklessbody)
    `    (A2->injectbody())
    `-----------------------------------
    ` A1         A          A2
    `Alogic/A1  rmanager/A  ALogic/A1
    `-----------------------------------
    `A2->injectbody(b)
    `    dit wordt gedaan door te deserialiseren
    `    en dan te renamen van de oude naam
    `-----------------------------------
    ` A1         A          A2
    `Alogic/A1  rmanager/A  ALogic/A2
    `-----------------------------------
    ` B->setforwardto(B);
    };
}
```

Bibliografie

[1] N. Narasimhan, "Transparent Fault Tolerance for Java Remote Method Invocation", Ph.D Dissertation, Department of Electrical and Computer Engineering, University of California, Santa Barbara, June 2001.

[2] N. Narasimhan, L. E. Moser, P. M. Melliar-Smith, "Transparant Consistent Replication of Java RMI Objects", Proceedings of the International Symposium on Distributed Objects and Applications (DOA 2000), pages 17–26, Antwerp, Belgium, September 2000.

[3] P. Narasimhan, "Transparent Fault Tolerance for CORBA", Ph.D Dissertation, Department of Electrical and Computer Engineering, University of California, Santa Barbara, December 1999.

[4] Sun Microsystems Inc., "Java Remote Method Invocation (RMI) Specification", 1997–2001,
http://java.sun.com/j2se/1.4/docs/guide/rmi/spec/rmiTOC.html

[5] A. S. Tanenbaum, M. van Steen, *Distributed Systems: Principles and Paradigms*, pages 364–366, Prentice Hall, 2002.

[6] TIBCO Software Inc., *TIB/Rendezvous Concepts*, Software Release 6.7, Palo Alto, CA, July 2001.

[7] R. Guerraoui, A. Schiper, "Fault-Tolerance by Replication in Distributed Systems", Proceedings of Conference on Reliable Software Technologies (Ada-Europe '96), pages 38–57, Montreux, Switzerland, June 1996.

[8] W. Van Belle et al., "The Borg Research Page", 2002,
http://borg.rave.org/research.html

[9] J. Pockele, "The Borg Tutorial", 2000,
http://borg.rave.org/tutorial/

[10] W. Van Belle, "JumpingFactorial", 2001,
http://borg.rave.org/examples/JumpingFactorial.html

[11] P. Verheyden, "MeetingPlanner", 2001,
http://borg.rave.org/examples/MeetingPlanner/index.html

[12] W. Van Belle, "Ontwikkeling van een Mobiele Multi Agent Architectuur", Licentiaatsthesis, Programming Technology Lab, Vrije Universiteit Brussel, 1996–1997.

[13] W. Van Belle, J. Fabry, K. Verelst, T. D'hondt, "Experiences in Mobile Computing: The CBorg Mobile Multi-Agent System", Proceedings of Tools Europe 2001, Zürich, Switzerland, March 2001.

[14] W. Van Belle, K. Verelst, T. D'hondt, "The CBorg Mobile Multi-Agent System", Accepted by Agents 2001 (but sent in too late), October 2001.

[15] W. Van Belle, K. Verelst, T. D'Hondt, "Location Transparant Routing in Mobile Agent Systems, Merging Name Lookups with Routing", Presented on Future Trends of Distributed Computer Systems, Tunisia, South Africa, December 1999.