Katholieke
Universiteit
Leuven

**Faculteiten Wetenschappen
en Ingenieurswetenschappen**

DYNAMISCH GEDISTRIBUEERDE WERKLASTBALANCERING VAN SOFTWARECOMPONENTEN OP MOBIELE SYSTEMEN

Jeroen TRAPPERS

Eindverhandeling aangeboden tot het behalen van de graad van licentiaat in de informatica

2005–2006

Promotor : Prof. Dr. ir. Y. BERBERS

Faculteiten Wetenschappen
en Ingenieurswetenschappen

K.U. Leuven
Academiejaar 2005–2006

Naam en voornaam student: Trappers Jeroen

Titel:

Dynamisch Gedistribueerde Werklastbalancering van Softwarecomponenten op Mobiele Systemen

Engelse vertaling:

Dynamic Distributed Load Balancing of Software Components on Mobile Systems

ACM Classificatie: J.7, D.1.3, D.4.8

Korte inhoud:

We ontwikkelen een methode die het mogelijk maakt om een applicatie, opgebouwd uit softwarecomponenten, vlot te gebruiken op een toestel met beperkte systeembronnen in een mobiele omgeving. Concreet willen we het componentensysteem Draco uitbreiden, zodat Draco-applicaties op een PDA automatisch verspreid worden via het draadloze netwerk over hardwareknopen in de omgeving. We willen dat de componentenuitrol [deployment] van de applicatie zich dynamisch aanpast aan veranderingen van werklast in de omgeving. De beslissingsmethode moet rekening kunnen houden met de systeembronnen die aangeboden worden door de hardware, en de vereisten die de software aan systeembronnen stelt. Omdat het niet wenselijk is een centraal punt te gebruiken waar alle informatie samenkomt, maken we een gedistribueerde methode.

In deze thesis houden we voor de werklastverdeling enkel rekening met geheugenvereisten van componenten en de geheugenbeperkingen van hardwareknopen. We houden ook rekening met de beperkingen die de bandbreedte tussen de hardwareknopen oplegt aan het plaatsen van componenten op de hardwareknopen.

We passen de methode *Diffusie door zenderinitiatief* toe. Dit is een gedistribueerde lokale aanpak. Er wordt gebruik gemaakt van de werklastinformatie van de nabije buren om een surplus aan werklading van zwaar beladen rekenknopen toe te wijzen aan onderbeladen buur-rekenknopen. Globaal evenwicht wordt bereikt doordat de werklast van zwaar beladen omgevingen verspreid wordt over de licht beladen omgevingen.

Eindverhandeling aangeboden tot het behalen van de graad van licentiaat in de informatica

Promotor: Prof. Dr. ir. Y. Berbers

Assessoren: Dr. E. Truyen
ir. Peter Rigole

Begeleider: ir. Peter Rigole

640k ought to be enough for anybody.
– Bill Gates, 1981 –

Dankwoord

Ik wil alle mensen bedanken die rechtstreeks of onrechtstreeks hebben meegewerkt aan het tot stand komen van deze thesis.

In de eerste plaats gaat mijn dank uit naar mijn ouders die deze studies mogelijk maakten. Vooral mijn moeder wil ik bedanken voor het nalezen van deze thesistekst. Het is altijd fijn een tweede mening te hebben voor de formulering van bepaalde moeilijkere passages.

Daarnaast wil ik in het bijzonder mijn begeleider Peter Rigole danken. Ik mocht wekelijks langskomen om de evolutie van mijn thesis te bespreken. Hij moedigde me aan waar nodig en stond steeds klaar met goede raad wanneer ik niet verder kon. Peter zorgde ook voor het grondig nalezen van deze thesistekst.

Nog vele anderen uit mijn omgeving verdienen mijn dank: mijn vrienden in Leuven, mijn broers, mijn zus en mijn vriendin voor de nodige afwisseling die ze brachten tijdens de soms zeer drukke periodes.

Jeroen Trappers

Inhoudsopgave

Lijst van figuren

Hoofdstuk 1

Inleiding

1.1 Ubiquitous computing – context

We geven een situering van de context waarin we deze thesis schrijven en gaan na wat we onder 'Ubiquitous computing' verstaan. 'Ubiquitous computing' kan vertaald worden door 'alomtegenwoordig computerwerk', maar wordt soms ook 'pervasive computing' – 'doordringend computerwerk', 'calm technology' – 'stille technologie' of 'things that think' – 'dingen die denken' genoemd. We lichten toe hoe dit begrip met deze thesis verband houdt.

Figuur 1.1: Kleine draagbare computers zijn alomtegenwoordig

Kleine draagbare computers zijn alomtegenwoordig

Draagbare telefoons, muziek- en multimediaspelers, PDA's, digitale camera's, draagbare DVD-spelers,... zijn enkele voorbeelden van apparaten die tegenwoordig niet meer uit het straatbeeld weg te denken zijn. Het digitale leven heeft vorm gekregen door een veelheid aan consumentenelektronica. Het aantal toestellen en gadgets dat we gebruiken blijft groeien. We komen dagelijks in contact met deze apparaten, ze maken een belangrijk deel uit van onze maatschappij.

We spreken van *ubiquitous computing* wanneer deze toestellen op een intelligente manier hun verwerkingskracht bundelen en samenwerken om een gemeenschappelijk doel te bereiken. Gebruikers van deze embedded devices zullen kunnen profiteren van allerlei diensten die aangeboden worden, die ons helpen dagelijkse taken efficiënter uit te voeren.

1.2 Uitdagingen voor ubiquitous computing – probleemstelling

Omdat we beschikken over toestellen die slechts beperkte verwerkingskracht hebben, moeten we aangepaste software gebruiken die kan omgaan met de beperkte systeembronnen [resources] (geheugen, CPU kracht, bandbreedte, batterijlading,...) Dit kan gedaan worden door abstractie te maken van de onderliggende hardware.

De aangeboden diensten kunnen gerealiseerd worden door gebruik te maken van meerdere toestellen in de omgeving. Deze gedistribueerde aanpak veronderstelt een opsplitsing van het werk. Een toestel kan inspelen op systeembronnen die zich in de omgeving bevinden door de aangeboden functionaliteit uit te breiden. Omdat verschillende rekeneenheden het werk uitvoeren, kan dit tegelijkertijd gebeuren – in parallel.

De omgeving waarin een gebruiker zich bevindt, verandert wanneer die zich verplaatst. De software moet overweg kunnen met deze veranderingen in systeembronnen. Er kunnen ook diensten aangeboden worden die afhankelijk zijn van de locatie van de gebruiker, of andere informatie die door de context aangeleverd wordt. De software wordt zich bewust van zijn omgeving.

Om aan deze uitdagingen tegemoet te komen, moet bij het ontwerpen van software over de grenzen van de verschillende domeinen van de computerwetenschappen gekeken worden. (embedded systemen, netwerken, middleware, agentsystemen, ad hoc gedistribueerde toepassingen, beveiliging, enz.)

In deze thesis willen we ervoor zorgen dat we vlot een uitgebreide applicatie kunnen gebruiken in een mobiele omgeving op een toestel met beperkte systeembronnen. Het probleem dat hierbij optreedt is dat grote, veeleisende applicaties wegens de beperkingen van de beschikbare systeembronnen, moeilijk uit te voeren zijn op zulke apparaten. Binnen deze thesis zoeken we een oplossing voor dit probleem.

Ongeveer 80% van de hedendaagse software bestaat uit embedded software, hierdoor kan de impact[1] van *ubiquitous computing* in de toekomst fenomenaal groot zijn.

[1] 5 november 2005, Nokia CEO Jorma Ollila voorspelt op de *Nokia Mobility Conference* in Barcelona dat er tegen 2010 3 miljard gsmgebruikers zullen zijn. Tegen dan zullen de meeste toestellen dezelfde of meer functionaliteit bieden dan huidige pda's. Dit wil zeggen dat de invloed op het dagelijkse leven van miljarden

1.3 Resource aware computing spaces – doelstellingen

In deze thesis willen we een methode ontwikkelen die het mogelijk maakt om een applicatie, opgebouwd uit softwarecomponenten, vlot te gebruiken op een toestel met beperkte systeembronnen in een mobiele omgeving. Hiervoor maken we gebruik van systeembronnen die beschikbaar zijn in de omgeving. Softwarecomponenten werken samen om een bepaalde functioniteit te verzekeren, en vormen zo de applicaties die we willen gebruiken. Om veeleisende applicaties te kunnen uitvoeren, vertrouwen we enkele van de componenten van lokale toepassingen toe aan naburige verwerkingseenheden. We verplaatsen met andere woorden een deel van het werk naar een andere knoop in het netwerk. Dit doen we door na te gaan of het binnen de beperkingen van de systeembronvereisten van de individuele componenten, en de beperkingen van de aangeboden systeembronnen van de omgeving, opportuun is om de uitvoering van componenten uit te besteden aan naburige knopen. Het systeem wordt zich bewust van de systeembronnen in de (reken-) omgeving.

Concreet willen we het componentensysteem Draco[2] uitbreiden, zodat Draco-applicaties op een PDA automatisch verspreid worden via het draadloze netwerk over hardwareknopen in de omgeving. We willen dat de componentenuitrol[3] [deployment] van de applicatie zich dynamisch aanpast aan veranderingen van werklast in de omgeving. De beslissingsmethode moet rekening kunnen houden met de systeembronnen die aangeboden worden door de hardware, en de vereisten die de software aan systeembronnen stelt. We willen dit probleem proberen op te lossen, zonder gebruik te maken van een centraal punt waar alle informatie samenkomt. Een gecentraliseerde aanpak kan voordelen hebben, omdat een oplossing gezocht kan worden met alle beschikbare informatie. Hiervoor zal een van de knopen in het netwerk die speciale functie moeten vervullen. Deze knoop wordt typisch gekozen door een virtuele verkiezing te organiseren en alle deelnemende knopen in het netwerk te laten stemmen. In deze thesis beschouwen we alle knopen in het netwerk als gelijkwaardig, en kiezen we voor een gedistribueerde aanpak, hierdoor vermijden we de overhead van het organiseren van virtuele stemmingen. Een verder voordeel van de gedistribueerde aanpak is dat bij grote veranderingen in het netwerk, niet telkens een nieuwe centrale knoop gekozen moet worden. De schaalbaarheid van een gedistribueerde aanpak is ook beter. Het rekenwerk dat gedaan moet worden om een nieuwe componentenuitrol te berekenen wordt bovendien zelf ook verdeeld.

In deze thesis houden we voor de werklastverdeling enkel rekening met geheugenvereisten van componenten en de geheugenbeperkingen van hardwareknopen. We houden ook rekening met de beperkingen die de bandbreedte tussen de hardwareknopen oplegt aan het plaatsen van componenten op de hardwareknopen. Deze twee systeembronnen zijn de belangrijkste beperkende factoren voor applicaties in een mobiele context. Er is duidelijk een afweging [trade-off] tussen de twee systeembronnen. Enerzijds is het goed om het geheugengebruik van een applicatie over de verschillende hardwareknopen in het netwerk te spreiden. Anderzijds houdt deze spreiding een extra kost in voor het bandbreedtegebruik.

---

mensen beïnvloed zal worden door het gebruik van embedded software en *ubiquitous computing*.

"There's a good chance we'll be watching 2008 Bejing Olympics on a Nokia device."

http://hardware.silicon.com/pdas/0,39024643,39153869,00.htm

[2] zie Hoofdstuk 3 voor een uitgebreide beschrijving van de concepten en werking van Draco

[3] Dit is de toewijzing van componenten aan hardwareknopen. In hoofdstuk 2 wordt hier uitgebreid op ingegaan.

We passen de methode *Diffusie door zenderinitiatief* toe. Dit is een gedistribueerde lokale aanpak. Er wordt gebruik gemaakt van de werklastinformatie van de nabije buren om een surplus aan werklading van zwaar beladen rekenknopen toe te wijzen aan onderbeladen buur-rekenknopen. Globaal evenwicht wordt bereikt doordat de werklast van zwaar beladen omgevingen verspreid wordt over de licht beladen omgevingen.

1.4 Structuur van de tekst

In hoofdstuk 2 geven we een overzicht van relevante bestaande technieken rond componenttoewijzing en werklastverdeling. We lichten toe wat de invloed van de componentuitrol van applicaties is op bepaalde kwaliteitskenmerken. We bespreken een aanpak, gebaseerd op de veronderstelling dat er een centrale knoop is, waar alle informatie verzameld wordt. In het stuk over werklastverdeling geven we een situering waar deze techniek toegepast wordt. We geven voorbeelden van toepassingen en we geven aan wat de huidige stand van zaken is rond gedistribueerd rekenen en rekennetten.

Vervolgens geven we in hoofdstuk 3 aan waar we ons werk op verder bouwen. We geven een overzicht van de functionaliteit, opbouw en werking van de architecturale middleware Draco. Lezers die thuis zijn in de wereld van Draco kunnen dit hoofdstuk links laten liggen.

Hoofdstuk 4 is het belangrijkste voor deze thesis. In dit hoofdstuk werken we de strategie uit die we toepassen om tot een gedistribueerde werklastverdelingsmethode te komen. We maken duidelijk hoe we aan systeembrongegevens komen en hoe we Draco uitgebreid hebben om deze informatie te kunnen gebruiken. We beschrijven welke methode we gebruiken om met de verkregen informatie aan werklastherverdeling te doen. We bouwen dynamisch een nieuwe componentenuitrol op, transparant voor de originele applicatie, om de geheugenlast van de applicatiecomponenten te spreiden over beschikbare systeembronnen in de omgeving.

In hoofdstuk 5 verifiëren we de ontwikkelde methode door enkele niet-triviale opstellingen te testen. We geven er tevens een beschrijving van een demo-applicatie en lichten toe hoe het systeem opgezet kan worden en hoe de applicatie ingeladen kan worden. In hoofdstuk 6 formuleren we het besluit. We kijken welke doelstellingen bereikt zijn, en lichten toe aan hoe we tot de oplossing gekomen zijn. We geven ook een kritische kijk op het geleverde werk.

In de nabeschouwing geven we een chronologisch overzicht van de ervaringen die we opdeden tijdens het werken aan deze thesis. Hierbij geven we enkele persoonlijke commentaren.

Hoofdstuk 2

State-of-the-art

In dit hoofdstuk bespreken we de huidige stand van zaken op gebied van componentenuitrol [deployment] en op het gebied van werklastverdeling [load-balancing].

We lichten toe wat de invloed van de componentuitrol van applicaties is op bepaalde kwaliteitskenmerken. In sectie 2.1.4 bespreken we een aanpak, gebaseerd op de veronderstelling dat er een centrale knoop is, waar alle informatie verzameld wordt. In sectie 2.2 geven we een situering waar de techniek van werklastverdeling toegepast wordt. We geven voorbeelden van toepassingen en bespreken de huidige stand van zaken rond gedistribueerd rekenen en rekennetten.

2.1 Componentenuitrol

2.1.1 Wat is het?

Wanneer men applicaties opbouwt uit verschillende componenten die samenwerken om aan bepaalde functionele en niet-functionele vereisten te voldoen, wordt men in het geval van een gedistribueerde omgeving geconfronteerd met de vraag hoe de componenten optimaal toegewezen moeten worden aan de beschikbare hardwareknopen, om zo een optimale componentenuitrol te bekomen.

De toewijzing van componenten aan de verschillende knopen kan een significante invloed hebben op de werking van het uiteindelijke systeem in het licht van de vereisten. Twee componenten die veel boodschappen uitwisselen, en dus een grote interactiefrequentie hebben, worden indien mogelijk best aan dezelfde hardwareknopen toegewezen. Wanneer groepen van zulke componenten dan onderling minder communiceren, maar wel grote eisen stellen aan de centrale verwerkingseenheid en geheugen, kunnen deze groepen best aan verschillende knopen toegewezen worden.

2.1.2 Wat moet er gebeuren?

De afweging tussen samenhouden en verdelen van componenten kan worden uitgedrukt in een toewijzingsfunctie die de verschillende conflicterende belangen quantificeert. De functie drukt

voor elke configuratie van toewijzingen van componenten aan hardwareknopen uit hoe goed de keuze van die configuratie is. Het maximum van de berekende waardes is dus gerelateerd aan de optimale toewijzing.

Wanneer de onderliggende hardwareconfiguratie verandert of wanneer er extra functionaliteit toegevoegd wordt aan een actieve applicatie tijdens de werking van het softwaresysteem kan het gebeuren dat de waarde voor die functie verandert voor de verschillende configuraties. De nieuwe toewijzing van componenten aan hardwarenodes wordt dan dynamisch berekend en toegepast.

2.1.3 Hoe wordt het ondersteund?

Deze dynamische beslissing komt niet uit de lucht vallen. Tussen de hardware en applicatie zit buiten het besturingssysteem nog een laag software die bepaalde diensten aan de applicatie levert, onderandere de mogelijkheid om componenten tijdens de levensduur van de applicatie van locatie te veranderen. Ook meer basisfunctionaliteit zoals het sturen van boodschappen en het uitvoeren van de code aanwezig in de componenten wordt ondersteund.

De *Middleware*[1], zoals deze tussenlaag genoemd wordt, kan onderliggende hardware en de netwerkverbindingen in de gaten houden door online metingen uit te voeren. Er zijn vele toepassingen te bedenken die gebruik maken van de diensten die de middleware aanbiedt. Bijvoorbeeld kan getracht worden om de reactietijd van het systeem te optimaliseren. Anderzijds is het mogelijk om de middleware in te schakelen om de beschikbaarheid van het systeem te verhogen. Dit kan bijvoorbeeld bewerkstelligd worden door zogenaamde *Disconnected operation* toe te laten. Wanneer de communicatiemogelijkheden tussen twee hardwareknopen, waarop samenwerkende componenten uitgevoerd worden, wegvallen, kunnen de afgezonderde delen van het systeem autonoom verder hun diensten verzorgen, mogelijk met verminderde kwaliteit van de aangeboden diensten. Dit is in sommige gevallen wenselijk, boven het volledig stoppen van de aangeboden diensten.

In de volgende sectie werken we het voorbeeld van een beschikbaarheidsverhogende aanpak uit.

2.1.4 Case: Beschikbaarheid verbeteren via redeployment

In [10] wordt beschreven wat de invloed van deployment van componenten is op de beschikbaarheid van een applicatie. In deze sectie geven we een overzicht van de gebruikte technieken.

Beschikbaarheid wordt gedefinieerd als de verhouding van het aantal succesvol verzonden boodschappen tussen componenten, ten opzichte van het totaal aantal interacties (interactiepogingen) over een tijdsperiode.

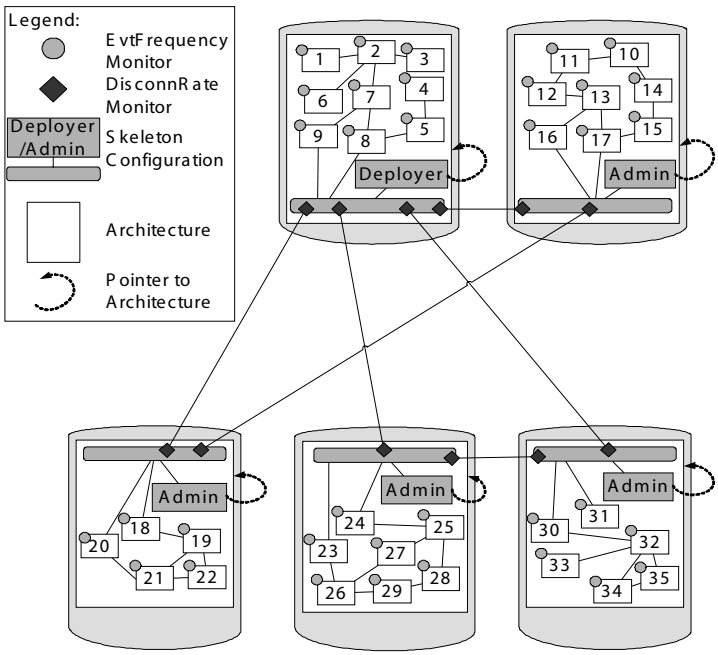
Door componenten die een grote onderlinge interactie hebben samen op eenzelfde knoop te plaatsen kan de beschikbaarheid van een applicatie verbeterd worden. De betrouwbaarheid van verbindingen tussen de knopen kan tijdens de levensduur van een applicatie veranderen.

[1] Tegenwoordig wordt middleware gebruikt om webservers, applicatieservers en content management systemen te beschrijven. Enkele commerciële middlware producten zijn: JBoss, PrismTech, IBM Websphere, Microsoft BizTalk Server, Oracle Fusion Middleware. De term *middleware* kan dus vrij breed geïnterpreteerd worden.

Om de beschikbaarheid van een applicatie te verhogen kan het nuttig zijn om een nieuw deploymentplan [redeployment] te bepalen. Het bepalen van een optimale plaatsing van componenten op de hardwareknopen (dit is de *optimale* deployment) is een exponentieel complex probleem. In het algemene geval is de complexiteit k^n, waarbij k het aantal knopen is en n het aantal softwarecomponenten.

[10] stelt een geautomatiseerde, dynamische oplossing voor die de beschikbaarheid van een applicatie verhoogt. Wanneer de communicatie tussen hardwareknopen volledig uitvalt, wordt teruggevallen op *Disconnected operation*. De twee belangrijkste waarnemingen waar [10] op steunt zijn: ten eerste veranderen de parameters van het systeem tijdens de levensduur van de applicatie. Ten tweede is slechts een *beperkte* tijd beschikbaar om een nieuwe plaatsing van componenten te bewerkstelligen nadat de veranderingen in het systeem zich hebben voorgedaan.

De voorgestelde oplossing verhoogt zelfstandig de beschikbaarheid door *actief metingen* uit te voeren op het systeem, een nieuw deploymentplan te *berekenen* en de berekende componentenuitrol *toe te passen*. De metingen gebeuren op elke knoop, maar de resultaten worden centraal verwerkt. Om de oplossing mogelijk te maken wordt een lichtgewicht, efficiënte en aanpasbare architecturale middleware gebruikt, Prism-MW genaamd. Deze middleware maakt het mogelijk softwarearchitecturen te implementeren, uit te voeren, te controleren en opnieuw op te stellen, dit alles in een omgeving van hoogst gedistribueerde en mobiele hardwareknopen.



Figuur 2.1: Een componentenuitrol met 5 knopen en 35 componenten. Bron: [10]

Het gedrag van softwarecomponenten in het licht van communicatiefrequentie kan mogelijk moeilijk vastgesteld worden voordat de applicatie in gebruik genomen is. Ook de beschikbaarheid van de netwerkverbindingen kan in deze context moeilijk vooraf bepaald worden. Hierdoor is het mogelijk dat een initieel bepaalde deploymentconfiguratie voor de applicatie

ongeschikt blijkt te zijn. Dit betekent dat een nieuwe componentenuitrol van het softwaresysteem nodig is om de beschikbaarheid te verbeteren. We tonen een componentenuitrol in figuur 2.1.

Probleemdefinitie

We bepalen formeel welke de beste deploymentconfiguratie is, in het licht van beschikbaarheid. Naast de communicatiefrequentie en de betrouwbaarheid van de netwerkverbindingen zijn er andere beperkingen aan het deploymentplan van een softwaresysteem. Hiertoe behoren het beschikbare geheugen op elke knoop, het vereiste geheugen voor elke softwarecomponent en mogelijke beperkingen op het samenplaatsen van componenten. Aangenomen wordt dat de belangrijkste bron van aftakeling van de beschikbaarheid van het systeem het verlies van netwerkconnectiviteit is.

Gegeven

Een verzameling C van n componenten ($n = |C|$) en twee functies *freq*: $C \times C \rightarrow \mathbb{R}$ en mem_{comp}: $C \rightarrow \mathbb{R}$ met

$$freq(c_i, c_j) = \begin{cases} 0 & \text{als } i = j \\ \text{communicatiefrequentie tussen } c_i \text{ en } c_j & \text{als } i \neq j \end{cases}$$

en

$$mem_{comp}(c) = \text{vereist geheugen voor } c$$

Een verzameling H van k hardwareknopen ($k = |H|$) en twee functies *rel*: $H \times H \rightarrow \mathbb{R}$ en mem_{knoop}: $H \rightarrow \mathbb{R}$ met

$$rel(h_i, h_j) = \begin{cases} 1 \text{ als } i = j \\ 0 \text{ als } h_i \text{ niet verbonden met } h_j \\ \text{betrouwbaarheidvan verbinding tussen } h_i \text{ and } h_j \text{ als } i \neq j \end{cases}$$

en

$$mem_{knoop}(k) = \text{beschikbaar geheugen voor knoop } k$$

Twee functies die beperkingen opleggen aan de locatie van softwarecomponenten *loc*: $C \times H \rightarrow \{0, 1\}$ en *colloc*: $C \times C \rightarrow \{-1, 0, 1\}$ met

$$loc(c_i, h_j) = \begin{cases} 1 & \text{als } c_i \text{ op } h_j \text{ geplaatst kan worden} \\ 0 & \text{als } c_i \text{ niet op } h_j \text{ geplaatst mag worden} \end{cases}$$

en

$$colloc(c_i, c_j) = \begin{cases} -1 & \text{als } c_i \text{ niet op dezelfde knoop als } c_j \text{ geplaatst mag worden} \\ 0 & \text{als er geen restricties op collocatie zijn tussen } c_i \text{ en } c_j \\ 1 & \text{als } c_i \text{ op dezelfde knoop als } c_j \text{ geplaatst moet worden} \end{cases}$$

Probleem

Zoek een functie $f\colon C \to H$ zodat de beschikbaarheid A van de gehele deploymentconfiguratie van de applicatie gemaximaliseerd wordt, waarbij

$$A = \frac{\sum_{i=1}^{n}\sum_{j=1}^{n}\left(freq(c_i,c_j) * rel\big(f(c_i),f(c_j)\big)\right)}{\sum_{i=1}^{n}\sum_{j=1}^{n}\left(freq(c_i,c_j)\right)}$$

en aan volgende drie beperkingen voldaan is:

$$\begin{aligned}
&(1)\ \forall i \in [1,k] : \left\{\forall j \in [1,n] : f(c_j) = h_i | \sum_j mem_{comp}(c_j) \leq mem_{knoop}(h_i)\right\}\\
&(2)\ \forall j \in [1,n] : loc\,(c_j, f(c_j)) = 1\\
&(3)\ \forall k \in [1,n], \forall l \in [1,n] : \begin{cases} \text{if}\big(colloc(c_k,c_i) = 1\big) \Longrightarrow \big(f(c_k) = f(c_l)\big)\\ \text{if}\big(colloc(c_k,c_i) = -1\big) \Longrightarrow \big(f(c_k) \neq f(c_l)\big)\end{cases}
\end{aligned}$$

In het algemene geval is het aantal mogelijke functies f k^n. Merk echter op dat er dan deploymentplannen berekend worden die mogelijk niet aan de drie bovenstaande beperkingen voldoen.

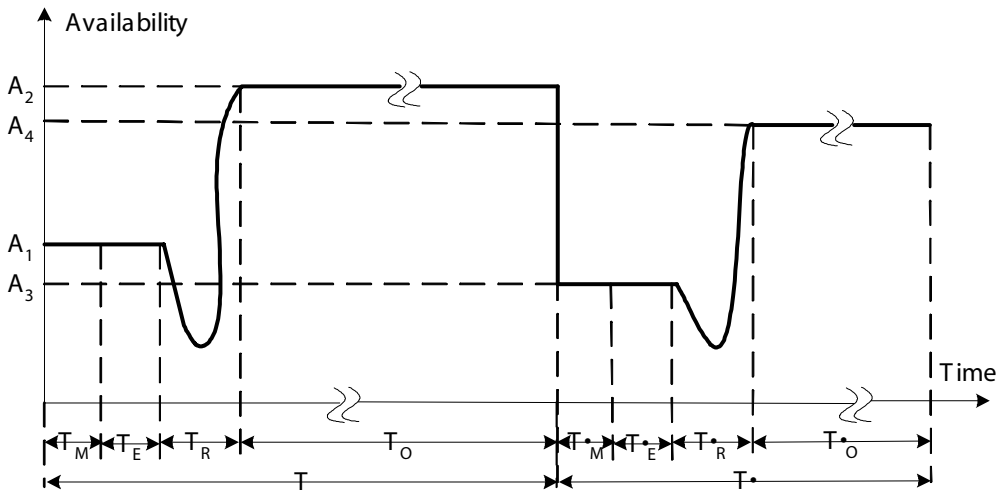
Aanpak

De aanpak die in [10] voorgesteld wordt bestaat uit drie fasen in de tijd. T_M is de periode waarin metingen op het systeem worden uitgevoerd, gedurende die periode blijft de beschikbaarheid onveranderd. Vervolgens wordt tijdens periode T_E een beter deploymentplan bepaald, ook tijdens deze periode blijft de beschikbaarheid constant. Tijdens de periode T_R wordt de nieuwe componentenuitrol bewerkstelligd om de beschikbaarheid van het systeem te verhogen. Tijdens deze periode is er een tijdelijke daling in beschikbaarheid, omdat tijdens de verplaatsing van componenten, deze componenten tijdelijk onbruikbaar zijn. Deze aanpak wordt vervolgens herhaald wanneer de parameters van het systeem veranderen. In figuur 2.2 tonen we de beschikbaarheid in functie van de tijd voor twee periodes T en T'. De beginsituatie is A_1. Tijdens T_R van de eerste periode wordt door bewerkstelligen van een nieuwe deploymentconfiguratie de beschikbaarheid verhoogd tot A_2. Tijdens T_O is het systeem operationeel.

De resulaten van deze aanpak zullen goed zijn wanneer de tijd die nodig is om een nieuw deploymentplan te berekenen en bewerkstelligen klein is ten opzichte van T (dwz. $T_M + T_E + T_R << T$). Anders zouden de parameters van het systeem te frequent wijzigen, waardoor het systeem nooit lang in een stabiele toestant verkeert. In zulke situaties zijn andere technieken (bv. caching[2] of hoarding[3]) aangewezen. Beschikbaarheid zal in zulke onstabiele situaties echter een probleem blijven, onafhankelijk van de gebruikte technieken.

[2]Caching: niet-lokale gegevens lokaal bewaren, anticiperend dat deze gegevens nog nodig zullen zijn.

[3]Hoarding: gegevens waarvan vermoed wordt dat ze nodig zullen zijn, ophalen voordat een verbinding wegvalt.



Figuur 2.2: Grafische voorstelling van de beschikbaarheidsfunctie over twee periodes. Bron: [10]

Redeployment bepalen

Om een nieuw deploymentplan te bepalen worden drie algoritmen voorgesteld. De algoritmen gebruiken de informatie die tijdens de metingsfase verworven werd.

Exact algoritme. Dit algoritme probeert elk mogelijk deploymentplan en selecteert datgene waarvoor de beschikbaarheid maximaal is en waarvoor alle beperkingen voldaan zijn. Dit algoritme garandeert een optimale oplossing. De complexiteit is in het algemene geval k^n, waarbij k het aantal hardwareknopen is, en n het aantal softwarecomponenten. Voor realistische applicaties is dit algoritme niet geschikt, omdat het niet tijdig een oplossing kan aanbieden. Ter illustratie: voor een applicatie van 15 componenten die geplaatst moet worden op 4 knopen, heeft het algortime 8 uur nodig op een mid-range PC.

Stochastisch algoritme. Dit algoritme ordent eerst de knopen en componenten op een willekeurige manier. Vervolgens selecteert het de eerste knoop en plaatst het een maximaal aantal componenten op deze knoop. De componenten worden geselecteerd uit de geordende lijst, en er wordt nagegaan of aan alle beperkingen voldaan is. Wanneer een knoop volledig beladen is, wordt de volgende knoop geselecteerd, en herhaalt het proces zich met de overige componenten, tot alle componenten toegewezen zijn aan een knoop.

Het algoritme herhaalt dit een vooropgesteld aantal keer en kiest het beste deploymentplan. De complexiteit is polynomiaal. We berekenen voor elk bekomen deploymentplan de beschikbaarheid, en dit kost $O(n^2)$ tijd.

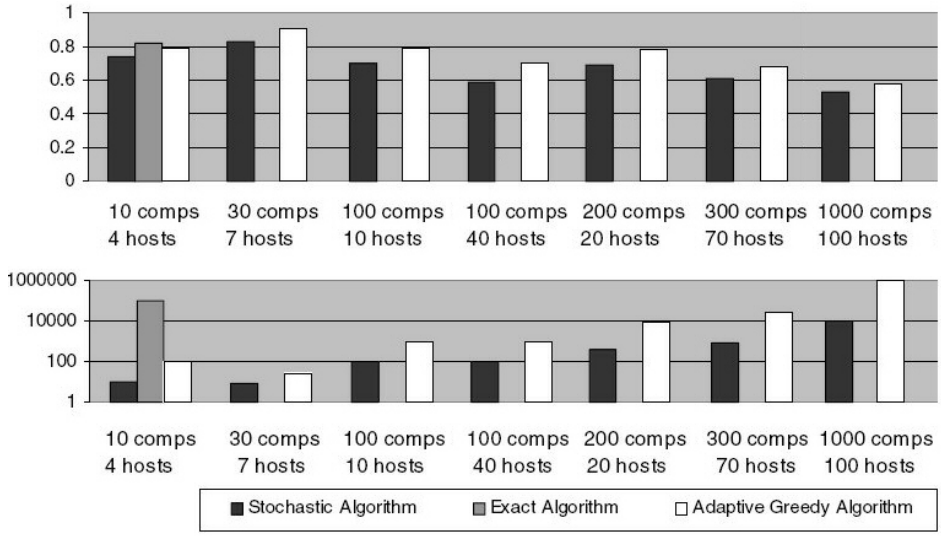
Adaptief-gulzig algoritme. Dit algoritme wijst incrementeel componenten toe aan hardwareknopen. Op elk punt in het algoritme waar een component toegewezen wordt aan een knoop, wordt geprobeerd om de toewijzing te kiezen die het meest bijdraagt aan de beschikbaarheidsfunctie. Dit wordt bereikt door de “beste” knoop en “beste” component in elke stap te selecteren. De beste hardwareknoop is diegene die de hoogste som aan netwerkbetrouwbaarheid met de andere knopen in het systeem heeft, en de hoogste geheugencapaciteit.

De beste component is diegene met de hoogste interactiefrequentie met andere componenten in het systeem, en de laagste geheugenvereisten. De verhouding tussen de onderdelen in het bepalen van de "beste" component en knoop kunnen bijgeregeld worden door afstemmingsparameters op te geven. Dit proces gaat verder tot de beste knoop vol is, vervolgens selecteert het algoritme de volgende beste knoop en gaat verder met het toewijzen van componenten aan knopen tot alle componenten toegewezen zijn. Bij de toewijzing worden telkens componenten uitgesloten waarvoor de beperkingen niet voldaan zijn. In het algemene geval is de complexiteit van dit algoritme $O(n^3)$, wanneer er meer componenten dan knopen zijn.

Prestatieanalyse

Een aantal willekeurig gegenereerde hardwareconfiguraties werden getest met een aantal willekeurig gegenereerde applicaties. Als parameter kon opgegeven worden hoeveel knopen en componenten respectievelijk gegenereerd moesten worden. Vervolgens werden de verschillende algoritmen losgelaten op deze gevallen, en werden metingen uitgevoerd, zowel op de tijd die nodig was om een nieuw deploymentplan te bepalen als op de kwaliteit van de bekomen spreiding in termen van beschikbaarheid.

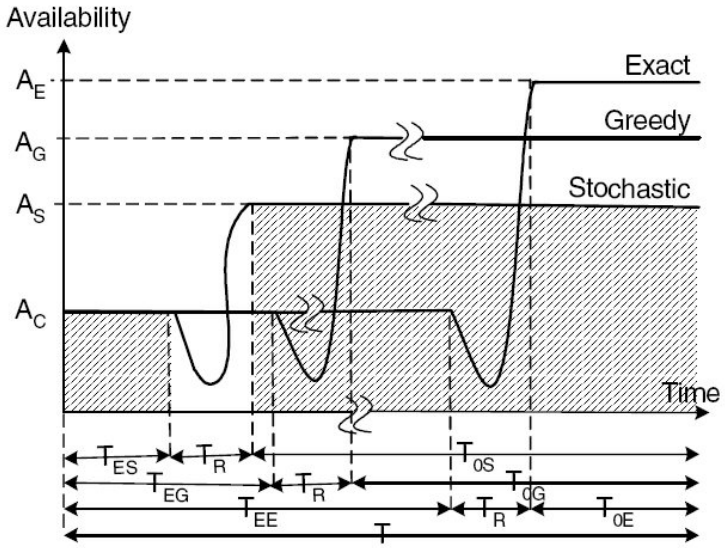
Op figuur 2.3 tonen we de resultaten.



Figuur 2.3: Prestatieresultaten voor de drie algoritmen (gebaseerd op [10]). De bovenste grafiek toont de bereikte beschikbaarheid. De onderste grafiek toont de tijd (in ms) die nodig was om het nieuwe deploymentplan te bepalen. De metingen werden uitgevoerd op een Intel Pentium III, 850Mhz, 128MB RAM geheugen, JVM 1.4 op Windows XP. Merk op dat voor het exacte algoritme slechts voor de kleinste configuratie resultaten gerapporteerd worden. Grotere configuraties zijn niet haalbaar, wegens te slechte complexiteit.

Om te kiezen tussen de drie algoritmes moeten we een afweging maken tussen de kwaliteit van de gevonden oplossing en de tijd die nodig is om de oplossing te berekenen. Op figuur 2.4 tonen we een samenvatting van de 3 methodes op 1 grafiek. Het systeem start bij een beschikbaarheid A_C en de metingsperiode is net afgelopen. T_{ES} is de tijd die het stochatische algoritme nodig heeft om een nieuw deploymentplan te bepalen. A_S is de beschikbaarheid

die door deze berekening bereikt wordt. T_{ES} is de kortste tijd, maar levert ook het slechtste resultaat. T_{EG} is de tijd die het adaptief-gulzige algoritme nodig heeft om een nieuw deploymentplan te bepalen. De beschikbaarheid A_G wordt bereikt. Het exacte algoritme heeft T_{EE} nodig om een nieuw deploymentplan te berekenen. Een beschikbaarheid A_E wordt bereikt. Dit algoritme rekent het langst, maar bereikt wel het beste resultaat. Het adaptief-gulzige algoritme is een compromis tussen de twee. T_R duurt voor elk algoritme even lang, dit is de tijd die nodig is om componenten van knoop te veranderen.



Figuur 2.4: Selectiecriteria voor de algoritmen. Bron: [10]

Evaluatie

De gebruikte methode is nuttig om de beschikbaarheid van een applicatie te verhogen door de componentenuitrol van de applicatie uit te buiten. Het grootste nadeel van de voorgestelde methode is de manier waarop de berekeningen gebeuren. Alle metingsgegevens worden centraal verzameld, waar een algoritme het nieuwe deploymentplan bepaalt. Vervolgens worden verplaatsingsopdrachten gedelegeerd naar de verschillende hardwareknopen in het netwerk. Wanneer de beschikbaarheid van netwerkverbindingen tussen knopen in het gedrang komt, is het realistisch aan te nemen dat ook de verbinding naar de centrale knoop in het gedrang kan komen. De methode kan op dit ogenblik geen antwoord bieden aan dit probleem.

In deze thesis maken we gebruik van een gedistribueerde aanpak om een nieuw deploymentplan te bepalen. Elke knoop maakt gebruik van beperkte informatie van zijn eigen omgeving (bv. enkel zijn directe buren) om de beslissing te nemen welke component verplaatst zal worden. Op deze manier hebben we minder informatie beschikbaar om de beslissing op te baseren – en zullen we mogelijk een sub-optimale componentenuitrol bekomen – maar de methode zal beter schalen naar grotere hardwarconfiguraties, en heeft geen 'single point of failure'.

2.2 Werklastverdeling

Werklastverdeling [Load Balancing] is een techniek om werk over vele computers, processen, schijven of andere systeembronnen te verdelen om een optimaal systeembrongebruik te verkrijgen en de rekentijd te minimaliseren. Werklastverdeling wordt in verschillende contexten gebruikt. Bijvoorbeeld wordt bij druk bezochte internet websites vaak werklastverdeling gebruikt op het niveau van het netwerk en besturingssysteem om de prestaties en toegankelijkheid naar bepaalde systeembronnen te verbeteren.

2.2.1 Werklastverdeling op server niveau

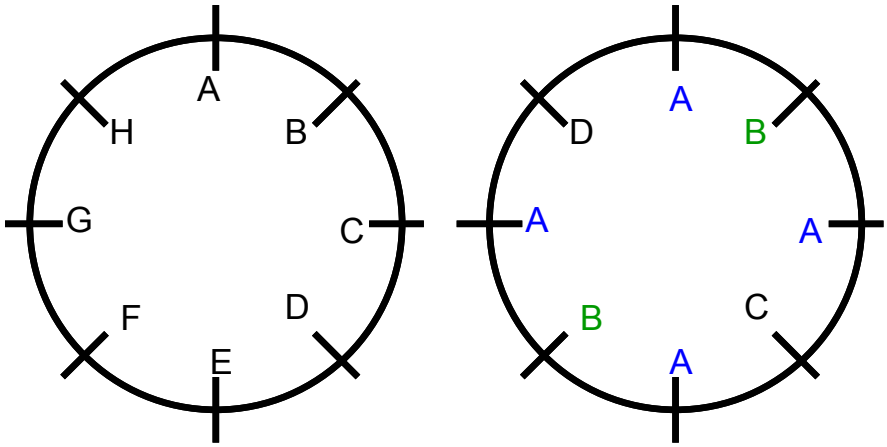
In deze context is de werklast steeds een aanvraag voor een dienst die vanuit een client komt. Als er veel aanvragen zijn, en de werklast te hoog wordt om afgehandeld te worden door een enkele machine is er nood aan werklastverdeling.

Een werklastverdeler [load balancer] kan gebruikt worden om de capaciteit van een server uit te breiden door meerdere machines samen te laten werken. Het kan ook continuïteit van de aangeboden diensten verzekeren wanneer een van de machines faalt of uitgeschakeld moet worden voor onderhoud.

Typisch bestaat een werklastverdeler uit een virtuele server, die aangesproken wordt via één adres (`ip-adres:poort`). Deze virtuele server is gebonden aan een aantal fysieke diensten en draait op een aantal fysieke machines (server farm). Deze fysieke machines hebben elk een eigen adres. Een client stuurt zijn aanvraag naar de virtuele server, deze kiest vervolgens een fysieke machine en wijst de aanvraag door. Bij client-server toepassingen is er soms nood aan persistentie van de toestand van een sessie. Eens een fysieke machine gekozen is voor een client, kunnen aanvragen die afkomstig zijn van die client steeds naar dezelfde fysieke machine gestuurd worden. Een andere methode is om toestandsinformatie in een gedeelde databank op te slaan. Die databank zal dan door alle fysieke machines gebruikt worden om de toestand van de sessie voor een specifieke client op te vragen. De client kan bijvoorbeeld geïdentificeerd worden door middel van een klein token, zoals een cookie, dat met elke aanvraag van de client meegestuurd wordt.

De werklastverdelingsstrategie bestaat erin een goede keuze te maken om de aanvraag door te geven aan een fysieke machine. Hiervoor bestaan verschillende technieken. We sommen er enkele op. We maken onderscheid tussen adaptieve en niet-adaptieve strategieën, afhankelijk of er al dan niet gebruik wordt gemaakt van de huidige werklastinformatie die beschikbaar is in het systeem.

- **Round robin**: dit is een niet-adaptieve strategie waarbij een (circulaire) lijst met mogelijke doel-adressen wordt afgelopen. Elke nieuwe aanvraag wordt aan de volgende in de lijst afgeleverd. Deze eenvoudige techniek wordt in vele contexten gebruikt. Bijvoorbeeld in de context van besturingssystemen kan de techniek gebruikt worden om aan planning te doen. Zo kunnen tijdssloten verdeeld worden over de beschikbare processen. De techniek kan eenvoudig uitgebreid worden met prioriteiten door een proces met hogere prioriteit vaker in de lijst op te nemen. Op figuur 2.5 geven we een illustratie van deze techniek.



Figuur 2.5: Er zijn 8 tijdssloten. In de linker figuur heeft elke proces (A-H) een gelijke uitvoeringstijd. In de rechter figuur heeft A vier tijdssloten, B twee en C en D elk 1. De prioriteit van A is 2 maal hoger dan die van B en 4 maal hoger dan die van C en D.

- **Willekeurig**: dit is een niet-adaptieve strategie waarbij voor elke aanvraag willekeurig een doel-adres gekozen wordt om de aanvraag te behandelen.
- **Laagste werklast** Deze adaptieve strategie laat een fysieke machine aanvragen afhandelen tot een bepaalde drempelwaarde in werklast bereikt wordt. Volgende aanvragen worden naar de fysieke machine gestuurd waarvan de werklast op dat moment het laagst is.
- **Minimum werklast** Deze adaptieve strategie berekent de gemiddelde werklast van de beschikbare machines. Wanneer de werklast op een bepaalde fysieke machine hoger geworden is dan de gemiddelde werklast, en een bepaalde migratie-drempelwaarde hoger ligt dan de fysieke machine met de laagste werklast zullen volgende aanvragen naar het doel-adres gestuurd worden waarvan de werklast op dat moment het laagst is.

2.2.2 Parallel rekenen

Er zijn tegenwoordig heel wat verschillende benaderingen om zoveel mogelijk rekenwerk gedaan te krijgen. We zien rekenkracht stilaan opduiken als een infrastructurele dienst – een nutsvoorziening – die we kunnen aanspreken wanneer er nood aan is[4]. Algemeen is een parallel rekensysteem een computer met meer dan een verwerkingseenheid. We kunnen deze systemen opdelen volgens hun geheugenarchitectuur. Gedeeld geheugen parallelle computers hebben meerdere verwerkingseenheden die al het beschikbaar geheugen aan kunnen spreken als één grote adresseringsruimte[5]. Verdeeld geheugen parallelle computers hebben ook meerdere ver-

[4] "The Sun Grid" is een commerciële dienst van het bedrijf Sun Microsystems. Deze dienst kan gebruikt worden door andere bedrijven, maar ook door eender wie er nood aan heeft. Voor de prijs van een dollar per processor en een dollar per gigabyte per maand kan men systeembronnen op de grid van Sun aankopen. Op deze manier moet superrekenkracht ook beschikbaar komen voor diegenen die zich geen supercomputer kunnen veroorloven.

[5] Deze systemen kunnen we verder opdelen volgens geheugentoegangstijden: *Uniform memory access* (UMA), waarbij elke geheugentoegang even lang duurt, of *Non-Uniform memory access* (NUMA), waarbij

werkingseenheden, maar elke processor kan enkel zijn eigen lokaal geheugen aanspreken. Er bestaat een grote verscheidenheid in parallelle computerarchitecturen.

Clusters

Een van de momenteel populairste architecturen is de zogenaamde *cluster*[6]. Een computer cluster is een groep van computers die via het netwerk verbonden zijn en nauw samenwerken, zodat ze in veel opzichten aanzien kunnen worden als een enkele computer. Vaak worden algemeen beschikbare computerelementen gebruikt om de kosten te drukken[7].

Meestal wordt in deze context aan werklastverdeling gedaan op het niveau van de rekentaken [jobs] zelf. Een planner [scheduler] bepaalt op basis van een vooropgesteld beleid, dynamische prioriteiten en reservaties welke taken uitgevoerd zullen worden. Meestal is het de bedoeling taken op zo'n manier over de knopen te verdelen dat het objectief zo snel mogelijk bereikt wordt. Er wordt rekening gehouden met de fysieke topologie van de verbonden knopen. Processen met hoge interactiefrequentie worden best op fysiek verbonden knopen geplaatst. Hier is de snelheid vaak belangrijker dan de doorvoer. Hoe sneller resultaten bereikt worden, hoe beter.

Gedistribueerd rekenen

Bij *gedistribueerd rekenen* [Distributed Computing] wordt de werklastverdeling op gigantisch grote schaal toegepast. Meerdere onafhankelijke computers worden gebruikt. Ze communiceren over een netwerk (het internet) en proberen een gemeenschappelijk doel te bereiken. De gebruikte systeembronnen kunnen drastisch uiteenlopend zijn, zoals bij clusters. Het verschil is dat de systemen bij gedistribueerd rekenen vaak over grote geografische afstand met elkaar verbonden zijn. Voorbeelden van toepassingen zijn: analyseren van radiosignalen uit de ruimte op zoek naar tekenen van buitenaardse intelligentie (SETI@home); het vouwen van eiwitten op basis van genetische evolutietechnieken, op zoek naar stoffen die actief zijn tegen bepaalde aandoeningen (Rosetta@home); klimaatgegevens worden geanalyseerd om significante veranderingen in de toestand van het klimaat op aarde te ontdekken en voorspellen (Climateprediction.net).

de toegangstijden afhangen van de locatie van gegevens in het geheugen.

[6]Zulke hoog-performante clusters gebaseerd op goedkope pc-hardware worden *Beowulf*-clusters genoemd. Oorspronkelijk ontwikkeld aan NASA, maar nu worden Beowulf-systemen wereldwijd ingezet als ondersteuning voor, voornamelijk, wentenschappelijk onderzoek. Meestal werken de machines op een open-source UNIX-achtig besturingssysteem zoals Linux of BSD. De systemen worden onderling verbonden via een TCP/IP-netwerk (vaak wordt hierbij gekozen voor InifiBand of Myrinet omdat deze technologieën een veel lagere opstarttijd hebben bij het opzetten van een verbinding. Hierdoor kan de latency bij communicatie laag gehouden worden.) Interprocescommunicatie wordt meestal geregeld met behulp van bibliotheken als MPI (Message Passing Interface) en/of PVM (Parallel Virtual Machine). Dit wil zeggen dat applicaties die ontwikkeld worden om optimaal te draaien op een cluster speciaal geparallelliseerd worden, met de onderliggende architectuur in gedachten. http://www.beowulf.org

[7]Een voorbeeld van een krachtige Linux Cluster vinden we terug binnen deze universiteit. Sinds mei 2005 is 'VIC' in gebruik. 174 nodes, waarvan 138 onderling verbonden via een Infiniband netwerk. Het systeem heeft met 360 verwerkingseenheden (AMD Opteron), en 488 GB RAM in de rekennodes, een theoretische piek-performantie van 1,8 Tflop. De totale diskruimte bedraagt meer dan 80TB, verspreid over de rekennodes, en een aantal centrale opslag-servers. Meer informatie op http://ludit.kuleuven.be/hpc/

Men probeert zoveel mogelijk rekenwerk gedaan te krijgen door het probleem in kleinere rekentaken op te splitsen en te verdelen over beschikbare systeembronnen. Een bekend voorbeeld van gedistribueerd rekenen is BOINC (Berkeley Open Infrastructure for Network Computing), een platform van de universiteit van Berkeley waar verschillende toepassingen gebruik van maken. BOINC laat vrijwilligers toe rekenkracht ter beschikking te stellen van wetenschappelijke onderzoeksprojecten. Op het ogenblik van schrijven zijn er een tiental projecten beschikbaar waaruit gekozen kan worden. Vrijwilligers schrijven zich in, en installeren op hun systeem een applicatie die samenwerkt met een centrale server die het werk verdeelt. De applicatie gebruikt de ongebruikte systeembronnen om berekeningen te doen. Bij deze rekenprojecten is het vooral van belang om zo'n groot mogelijke zoekruimte te dekken. De snelheid waarmee de berekeningen worden uitgevoerd is van ondergeschikt belang ten opzichte van de doorvoer. Wetenschappers zelf kunnen projecten die grote rekenkracht vereisen uitwerken en indienen.

Het verdelen van de taken gebeurt op een gecentraliseerde manier. Wanneer het systeem van een vrijwilliger rekenkracht ter beschikking heeft, contacteert het de centrale server van het project en haalt daar een rekentaak op. Deze communicatie is kortstondig. Het systeem kan vervolgens uren rekenen en nadat dit gedaan is worden de resultaten teruggestuurd naar de centrale server. Vervolgens wordt een nieuwe rekentaak opgehaald, mogelijk van een ander project waar de vrijwilliger aan wil meewerken. Deze kan zelf een verhouding van zijn ongebruikte systeembronnen toekennen aan projecten naar zijn keuze[8].

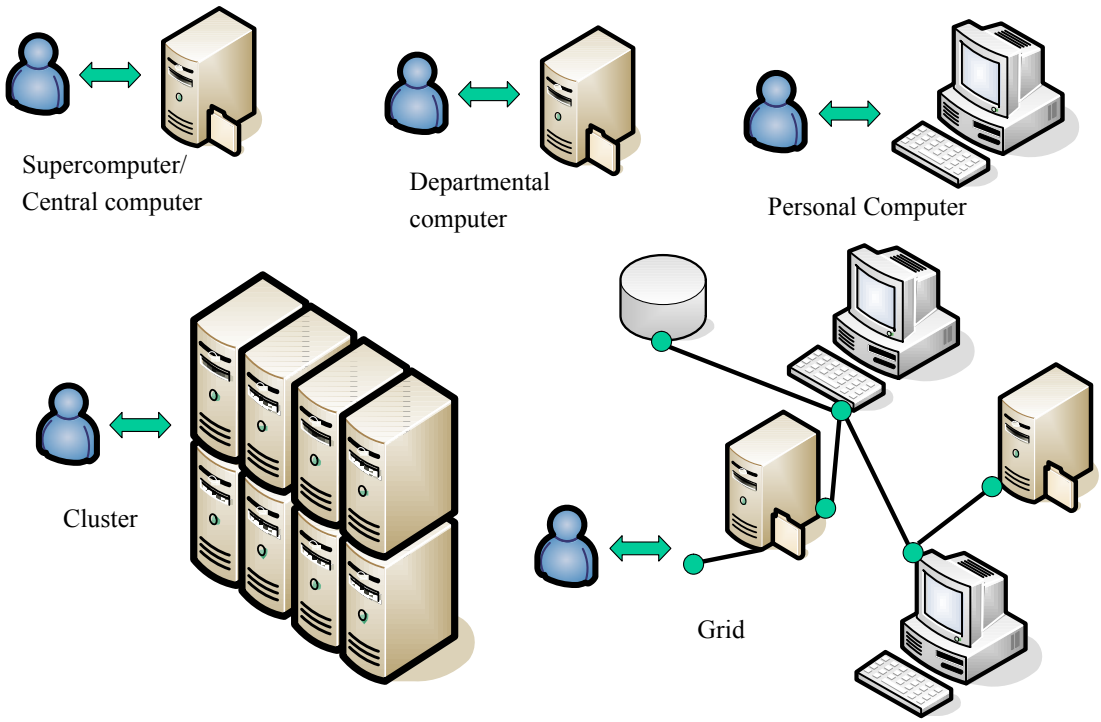
Reken-net

Een reken-net [Computing grid] is een opkomend rekenmodel dat de mogelijkheid aanbiedt om grote doorvoer te bereiken door vele verbonden computers samen te laten werken om zo een virtuele computerarchitectuur te vormen. Deze computerarchitectuur maakt het mogelijk om processen uit te voeren op een parallelle infrastructuur. Een reken-net gebruikt de systeembronnen van vele computers, meestal verbonden via het internet, om rekenproblemen van grote schaal op te lossen. De grote rekentaak wordt opgesplitst en verdeeld over de verschillende systemen.

De term net [grid] komt van de metafoor om rekenkracht zo eenvoudig als een nutsvoorziening aan te bieden, het elektriciteits*net*. Het verschil met gedistribueerd rekenen is vooral dat het beheer van het reken-net niet centraal gebeurt. Open standaarden moeten worden gebruikt om interoperabiliteit te garanderen en een niet-triviale kwaliteit van de aangeboden dienst (quality of service) moet verzekerd zijn. Op figuur 2.6 tonen we de evolutie in computergebruik.

Een verzameling systeembronnen selecteren en toewijzen aan een rekentaak is de taak van het planningsmechanisme. Omdat aanzienlijke hoeveelheden data nodig zijn om een rekentaak uit te voeren en omdat er slechts beperkte bandbreedte beschikbaar is tussen de systeembronnen, moet bij het maken van de selectie rekening gehouden worden met netwerkbelasting en datalokaliteit. Als de volgorde gekend is waarin een rekentaak invoergegevens nodig heeft,

[8] Binnen de gemeenschap van vrijwilligers is er vaak een grote concurrentie. Vrijwilligers groeperen zich en willen met hun groep zo veel mogelijk rekenwerk verzetten. Het is een kwestie van eer om de meeste rekenkracht te kunnen tonen. Dit wordt door BOINC ook ondersteund door uitgebreide statistieken te onderhouden voor de vrijwilligers. Elke dag wordt een vrijwilliger in de bloemetjes gezet door een vermelding op de hoofdpagina. Meer informatie over BOINC en de projecten op http://boinc.berkeley.edu.



Figuur 2.6: Rond 1964 was er binnen sommige bedrijven 1 centrale computer of mainframe. Rond 1965 begonnen departementen binnen bedrijven en onderzoekstinstellingen hun eigen computer te krijgen. In 1976 werden de eerste Personal Computers geïntroduceerd, bv. de Apple I-II, Tandy TRS-80 en Commodore PET. De grote doorbraak voor de PC kwam in 1981 met IBM PC, Intel 8088 en Microsoft DOS. In 1980 begonnen de eerste clusters te verschijnen, mainframes en supercomputers hadden schaalbaarheidsproblemen. Meestal werden homogene rekenelementen gebruikt, die door een lokaal netwerk met elkaar verbonden waren. Rond de jaren 1990 begonnen ook clusters last te krijgen van schaalbaarheidsproblemen en begon men te denken aan grids van hetereogene rekeneenheden.

kan de uitvoering gestart worden wanneer het eerste blok gegevens beschikbaar is. Tegelijk kunnen de volgende gegevens verzonden worden. De uitvoeringssnelheid zal echter beperkt kunnen worden door de bandbreedte van het aanvoerkanaal.

In deze thesis proberen we aan werklastverdeling te doen op het niveau van software-componenten. We maken hiervoor gebruik van de Draco-middleware (zie hoofdstuk 3 voor meer informatie) die op bijna eender welk apparaat kan gebruikt worden. We kunnen de architectuur van de middleware over de fysieke architectuur heenleggen, en op deze manier abstractie maken van de onderliggende hardware. We voegen een module aan Draco toe die informatie uit de onderliggende laag haalt en deze beschikbaar maakt. Deze informatie gebruiken we om beslissingen in verband met het opstellen van een deploymentplan in het licht van lastenspreiding te kunnen nemen. De onderliggende architectuur kunnen we als een grid beschouwen, waarvan we de systeembronnen kunnen gebruiken. In deze thesis willen we een applicatie opsplitsen en verspreiden over de beschikbare systeembronnen om de werklast te verdelen.

Hoofdstuk 3

Componentgebaseerde applicaties

In dit hoofdstuk bekijken we wat we verstaan onder componenten en hoe we met deze componenten een applicatie kunnen bouwen. We bekijken hoe deze componenten in DRACO werken en hoe we applicaties kunnen opbouwen met deze componenten.

DRACO[1] [15] is een lichtgewicht architecturale middleware die ontwikkeld werd binnen de onderzoeksgroep Distrinet[2] van het departement computerwetenschappen[3] van de faculteit toegepaste wetenschappen aan de KULeuven[4].

> "Het DRACO componentenframework is een in-house middleware omgeving dat op een flexibele manier componentgebaseerde applicaties ondersteunt. DRACO is verantwoordelijk voor de afhandeling van communicatie tussen componenten, voor het creëren van component-instanties en voor interactie met de gebruiker. De communicatiesemantiek tussen de componenten is volledig booschap-gedreven, wat de composities ideaal maakt om ze in een gedistribueerde omgeving te deployen."

Er werd gekozen voor het gebruik van JAVA om een maximale draagbaarheid[5] [portability] te bekomen. Bijna elk apparaat waar JAVA-1.3 op aanwezig is kan gebruikt worden om DRACO te draaien. DRACO is uitermate geschikt in een onderzoeksomgeving omdat het eenvoudig uitgebreid kan worden met modules.

Een groot voordeel van het gebruik van de DRACO middleware is de mogelijkheid om gedistribueerde applicaties te bouwen. We bekijken hoe DRACO deze mogelijkheid ondersteunt. We leggen ook kort uit hoe zwakke mobiliteit in DRACO geïmplementeerd werd.

3.1 Componenten

Componenten zijn software-entiteiten die een aantal specifieke functies vervullen. Om taken uit te voeren in een applicatie werkt een component samen met andere componenten door

[1] http://www.cs.kuleuven.be/~yvesv/?q=Draco **D**istriNet **R**eliable and **A**daptive **CO**mponents
[2] http://www.cs.kuleuven.be/~distrinet/
[3] http://www.cs.kuleuven.be/
[4] http://www.kuleuven.be/
[5] Met draagbaarheid wordt hier de mogelijkheid de software te gebruiken op verschillende systemen (zowel hardware als software) bedoeld.

boodschappen uit te wisselen. Een component ontvangt boodschappen, doet berekeningen (die eventueel neveneffecten hebben) en verstuurt boodschappen.

Draco-componenten zijn conceptueel eenvoudig. Ze zijn losjes aan elkaar gebonden en sturen asynchrone boodschappen. Bij het ontvangen van een boodschap wordt in de component een methode uitgevoerd. Door de manier van communiceren is er een automatische synchronisatie. Wanneer een boodschap verzonden wordt naar een poort voor andere boodschap, kunnen we er zeker van zijn dat die eerste boodschap eerst aankomt.

Componentblauwdruk

Een component heeft een blauwdruk. Dit is een herbruikbare entiteit die het type van de component beschrijft en de implementatie van de component specificeert. Het is een statische constructie die geen runtime betekenis heeft. Deze definitie kan vergeleken worden met een klasse van een software-object in de context van objectgeoriënteerde programmeertalen.

Component

Wanneer van een componentblauwdruk een concretisering gemaakt wordt spreken we over de component zelf. Een component is een entiteit die actief is tijdens de uitvoering van de applicatie en heeft een toestand.

Een component kan functionaliteit aanbieden via een interface en kan ook afhankelijk zijn van de interface van een andere component. De interface van een component wordt gespecificeerd door poorten. Een poort kan een aantal verschillende boodschappen behandelen. Voor elk type boodschap kan de poort een andere specificatie hebben.

Poort

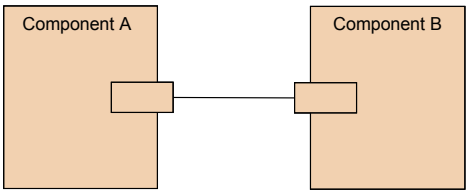
Een poort is de communicatiedoorgang voor componenten. Een component gebruikt poorten om te kunnen communiceren met andere componenten. Om poorten te kunnen verbinden moeten de interfaces die ze voorstellen overeenkomen. De boodschappen die aan de ene kant verstuurd worden, moeten aan de ontvangerskant een zinvolle betekenis hebben. De naam en het type van de boodschap zijn bepalend voor deze betekenis. In Draco zijn er drie verschillende soorten poorten.

Single Port

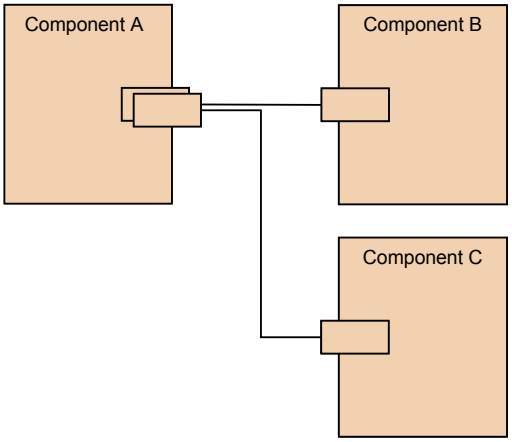
Een enkele poort laat een-tegen-een communicatie toe.

Multiport

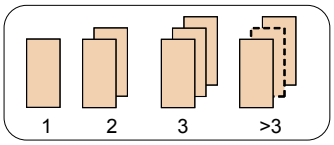
Een multipoort bestaat uit één poortdefinitie waaruit meerdere identieke enkele poorten kunnen ontstaan. Een multipoort heeft een maximum aantal verbindingen. Wanneer de verbinding opgezet is, werkt deze zoals een enkele poort met een-tegen-een communicatie.



Figuur 3.1: Single poort



Figuur 3.2: Multiport



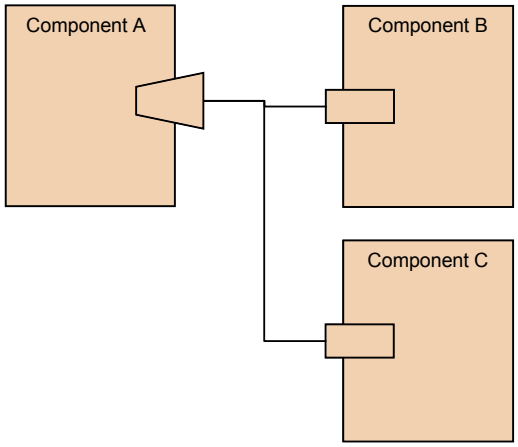
Figuur 3.3: Multiport symbool is afhankelijk van de dimensie van de poort.

Multicast Port

Een multicast poort met dimensie n kan n verbindingen accepteren. De multicast poort kan enkel boodschappen versturen. Alle boodschappen die door een multicast poort gestuurd worden zullen door alle verbonden componenten ontvangen worden.

3.2 Componentencomposities

De grafische voorstelling van een applicatie bestaat uit de voorstelling van de componenten. De communicatie tussen de componenten wordt aangeduid door een verbinding tussen de poorten. Wanneer we de componenten in Draco willen laden, zullen we moeten specificeren welke componenten geladen moeten worden. Daarvoor moeten we namen kiezen voor de concretisering van bepaalde componentblauwdrukken. Vervolgens moeten we aangeven welke poorten verbonden moeten worden. Dit alles moet gebeuren via de commandolijn van Draco.

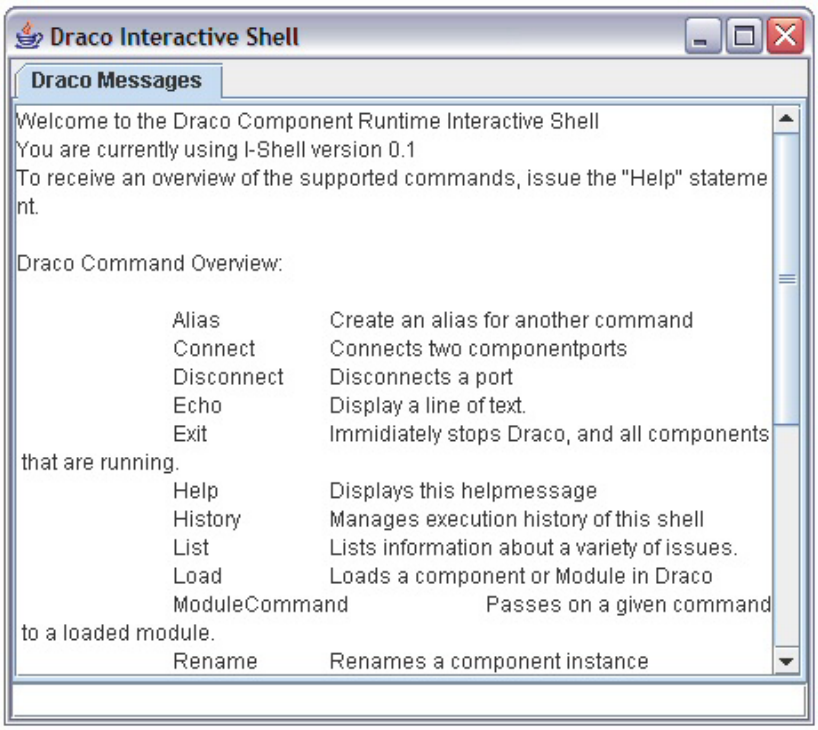


Figuur 3.4: Multicast Port

Shell

Draco zelf heeft geen vorm van gebruikersinteractie. Hiervoor moeten we een module laden die een schil [shell] vormt rond de kern van Draco waardoor we commando's kunnen geven. Op figuur 3.5 tonen we de grafische schil, waarop we het commando `Help` gaven. De commando's voor deze schil zijn hoofdlettergevoelig.

```
java -jar Draco.jar -s Shell.jar
```



Figuur 3.5: De interactieve shell van Draco

Load

Om componenten in te laden in DRACO gebruiken we het `Load`-commando.

```
Load {component|module} InstanceName from SourceJar.jar
```

Met het sleutelwoord `component` of `module` kiezen we welk type entiteit we willen inladen. De `InstanceName` kan vrij gekozen worden, maar mag nog niet bestaan binnen het systeem. Met het sleutelwoord `from` geven we aan welke code ingeladen moet worden (een `jar`-bestand).

```
Unload {component|module} InstanceName [force]
```

Het uitladen van componenten gebeurt met het `Unload`-commando. Ook hier geven we aan of het over een `component` of een `module` gaat. De unieke naam van de entiteit volgt. Een optionele booliaanse parameter (`true` of `false`) kan opgegeven worden waarmee we aanduiden of de entiteit geforceerd uitgeladen moet worden. Wanneer deze parameter niet opgegeven wordt, zal de entiteit enkel uitgeladen worden wanneer deze geen verbindingen meer heeft.

Connect

Om een verbinding te leggen tussen twee componenten gebruiken we het `Connect`-commando.

```
Connect ComponentName1/Source with ComponentName2/Destination [using f.xml]
```

We specificeren twee poorten die verbonden moeten worden. Een poort wordt aangeduid met `ComponentName/PortName`. We verbinden hier de bronpoort `(ComponentName1/Source)` met de doelpoort `(ComponentName2/Destination)` via het sleutelwoord `with`. Optioneel kan de specificatie voor een flexibele connector opgegeven worden met het sleutelwoord `using`. Voor een uitgebreidere specificatie van dit xml-bestand verwijzen we naar de DRACO handleiding [16].

Send

Dit commando gebruiken we om boodschappen te sturen naar componenten die verbonden zijn met de *shell*.

```
Send MsgName[id:message|parm:4] to ComponenName/PortName
```

Om een boodschap te sturen specificeren we de naam van de boodschap `MsgName` gevolgd door vierkante haken. Binnen de vierkante haken kunnen we de parameters van de boodschap specificeren, met de naam van de parameter door een dubbelpunt gescheiden van de waarde van die parameter. Meerdere parameters moeten we scheiden met een vertikale streep. De boodschap sturen we naar (`to`) een poort.

Run

Om het opstarten van een applicatie te vereenvoudigen is het mogelijk om een vooraf opgestelde lijst van commando's na elkaar uit te voeren. Dit doe je met het `Run`-commando.

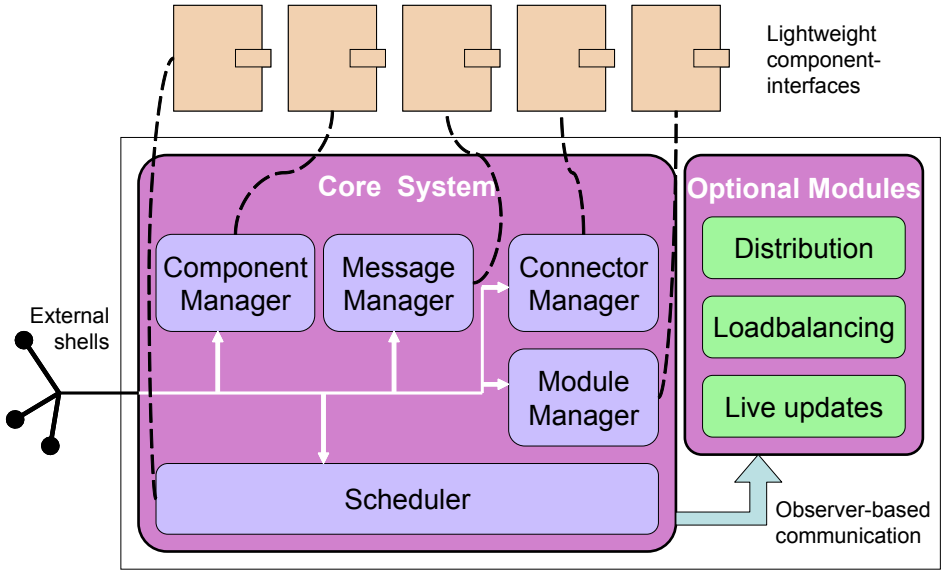
```
Run script
```

3.3 Draco

In deze sectie beschrijven we de architectuur van Draco. We beschrijven de verschillende interne eenheden waaruit Draco is opgebouwd en hoe ze samenwerken om de basisfunctionaliteit te verzorgen.

3.3.1 Architectuur

De kern van Draco is opgebouwd uit 5 eenheden. 5 vaste interfaces beschrijven deze eenheden. De eigenlijke implementatie van deze eenheden gebeurt door over te erven van een aantal vaste basisklassen. Op figuur 3.6 tonen we een overzicht van de architectuur van Draco.



Figuur 3.6: Overzicht van de Draco architectuur

Draco start door een configuratiebestand te lezen en haalt daaruit de specifieke implementatie voor elk van de kerneenheden. Het vervangen of veranderen van een van de kerneenheden is hierdoor gemakkelijk en heeft geen invloed op de rest van het systeem. Wanneer het systeem ingeladen is, kan het niet meer dynamisch veranderd worden. De runtime configuratie staat vast.

Component Manager

De componentmanager is verantwoordelijk voor het aanmaken van componenten en poorten. Hij beheert verwijzingen naar alle componenten die zich op de lokale knoop bevinden. We kunnen via de componentmanager referenties vragen naar componenten via hun componentblauwdruk.

Scheduler

De planner is verantwoordelijk voor het afleveren van de boodschappen. Daarbij is het belangrijk dat de volgorde van aflevering voor elke poort in acht genomen wordt. In een gedistribueerde context is er altijd precies één planner beschikbaar per componentensysteem. De planner is ook de enige entiteit in de componentenarchitectuur die threads behandelt. De planner zorgt ervoor dat er nooit meer dan een uitvoeringsthread een component tegelijkertijd raadpleegt.

Message Manager

De boodschappenbeheerder is verantwoordelijk voor het afleveren van boodschappen. Hij werkt samen met de connectormanager om te achterhalen welke connectors met de poort waarvan de boodschap afkomstig is geassocieerd zijn.

Connector Manager

De connectormanager is verantwoordelijk voor het aanmaken en beheren van de connectors tussen de poorten van componenten. Hij beheert ook de behandeling van boodschappen die geassocieerd zijn met deze connectors.

Module Manager

De modulemanager maakt het mogelijk om de Draco-kern uit te breiden met extra modules. Het is mogelijk dat deze modules niet altijd gebruikt zullen worden. Daarom kunnen we via de modulemanager de extra modules at runtime in- en uitladen. Uitbreidingsmodules kunnen zich inschrijven op een aantal gebeurtenissen die veroorzaakt worden door de Draco kerneenheden. Dit is geïmplementeerd door gebruik te maken van het *Observer*patroon

Component-Interfaces

Elke kerneenheid van Draco brengt een multipoort naar buiten. Componenten kunnen zich verbinden met deze poort om informatie op te halen uit de runtime.

Gebruikersinteractie

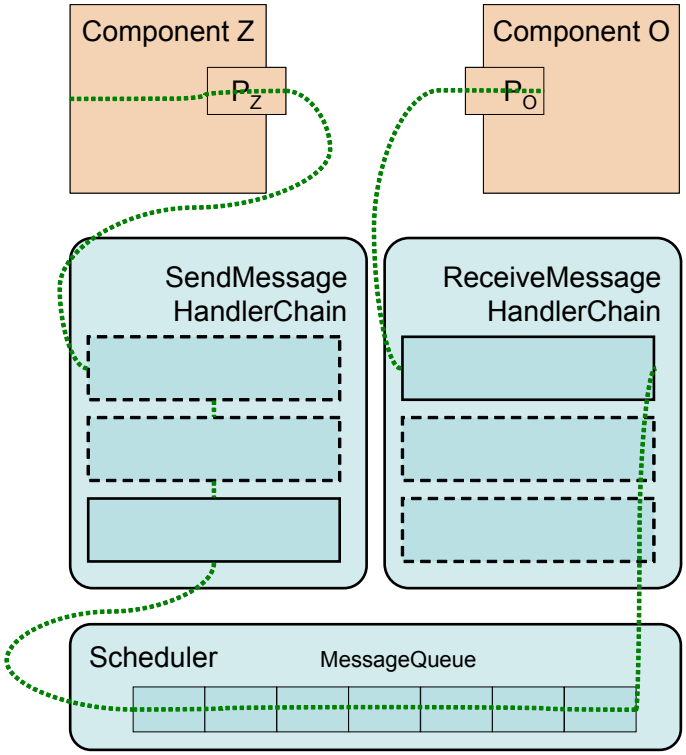
De gebruikersinteractie met Draco wordt verzorgd door de *shell*. De keuze welke shell we willen, gebeurt tijdens het opstarten van Draco, met een opstartparameter. De shell is ondergebracht in een apart bestand. Afhankelijk van de situatie kunnen we kiezen welke shell het beste geschikt is. Een grafische shell kan gebruikt worden op een hoogperformante desktopcomputer. Een afgeslankte versie kan gebruikt worden in een systeembronarme omgeving.

3.3.2 De reis van een boodschap

In Draco worden boodschappen tussen componenten asynchroon verzonden. Om een boodschap te versturen moeten een aantal stappen doorlopen worden: de doelpoort zoeken, de boodschap plannen voor aflevering en uiteindelijk het afleveren van de boodschap, wat uitvoering van de code aan de ontvangerskant als reactie op het ontvangen van de boodschap met zich meebrengt. In een gedistribueerde context zijn meer stappen nodig om de systeemgrens te overschrijden.

Uitbreidingsmodules kunnen zich inschrijven op gebeurtenissen die door deze stappen veroorzaakt worden. Ze hebben de mogelijkheid om in te grijpen in de weg die de boodschap aflegt. Verder moet kunnen ingespeeld worden op het afleveren van een boodschap om verscheidene effecten te kunnen verkrijgen. Dit kan bijvoorbeeld nodig zijn om een component at runtime te vervangen door een andere. Hiervoor zal het afleveren van boodschappen tijdelijk stilgelegd moeten worden, om de component stil te leggen en hem te vervangen. In Draco gebeurt dit door een ketting van boodschapbehandelingen.

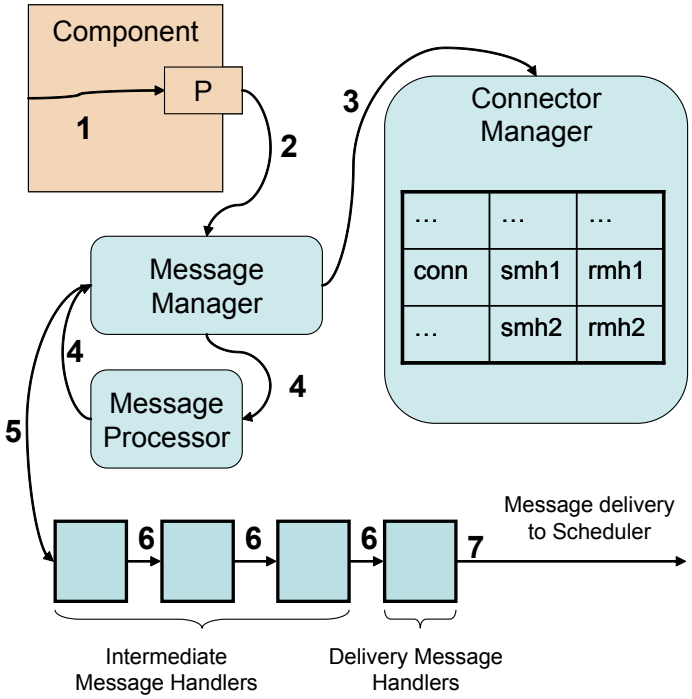
Het pad dat gevolgd wordt, kan opgesplitst worden in drie delen. Het versturen van de boodschap, het plannen van de aflevering en het ontvangen van de boodschap. In figuur 3.7 wordt een overzicht van dit proces getoond.



Figuur 3.7: Schematisch overzicht van de reis van een boodschap

Versturen van een boodschap

De ketting van handelingen die we nodig hebben om een boodschap te versturen begint vanaf het moment dat de boodschap door de poort van de zendende component gaat, en eindigt op het moment dat de boodschap gepland wordt voor uitvoering door de scheduler. Op figuur 3.8 tonen we de gedetailleerde implementatie in Draco.



Figuur 3.8: Details van het versturen van een boodschap

In de eerste stap contacteert de component de poort waar de boodschap door verzonden moet worden. Dit is een lokale oproep, omdat poorten in Draco geïmplementeerd zijn als innerklassen. De poort geeft de boodschap door aan de messagemanager (Pijl 2). Deze vraagt aan de connectormanager referenties naar de verbonden connectors (Pijl 3). De messagemanager voert eventueel transformaties uit op de boodschap, die gespecificeerd kunnen zijn door een flexibele connector (Pijl 4). Elke connector heeft vier boodschap-handlers. De messagemanager haalt twee handlers op die geassocieerd zijn met de richting van de boodschap. De ontvang-handler wordt gebruikt nadat de boodschap gepland werd voor aflevering door de scheduler. Hiervoor wordt deze handler meegegeven met de boodschap (Pijl 5). Elke handler heeft de mogelijkheid om boodschappen te onderscheppen en te veranderen. Daarna wordt de boodschap doorgegeven aan de volgende handler (Pijl 6). De laatste handler is verantwoordelijk voor aflevering aan de scheduler.

Ontvangen van een boodschap

Na ontvangst van de boodschap met de bijhorende ontvang-handler door de scheduler, wordt de boodschap in de rij gezet tot ze klaar is om uitgevoerd te worden. Het exacte rij-mechanisme

hangt af van de gebruikte scheduler. In elk geval moet de volgorde van de boodschappen over een gegeven connector bewaard worden. Een boodschap die voor een andere boodschap verzonden is, moet ook voor die laatste boodschap afgeleverd worden.

Wanneer de scheduler een boodschap selecteert die uitgevoerd mag worden, zal deze een thread reserveren en de boodschap doorgeven aan de ontvangst-handler. Het principe achter de ontvangstsequentie is gelijk aan het verzenden. Er is een ketting van boodschap-handlers die de boodschap behandelen en daarna doorgeven. De laatste handler levert de boodschap af aan de poort op het einde van de connector. Deze poort zal de boodschap afleveren bij de eigenlijke methode die instaat voor de behandeling van de boodschap. Nadat de boodschap is uitgevoerd, wordt de controle terug overgedragen aan de scheduler.

3.4 Component mobiliteit

Het grootste voordeel van het gebruik van middleware is dat we ondersteuning krijgen voor het bouwen van gedistribueerde applicaties. Zowel de deployment van, als de communicatie tussen componenten wordt door de middleware afgehandeld. In Draco wordt deze functionaliteit toegevoegd door gebruik te maken van de distributie module. Op deze manier voelt Draco zich thuis in een gedistribueerde omgeving, zonder de eenvoud en kracht van de kernconcepten te verlaten.

3.4.1 Zwakke mobiliteit

Mobiliteitsconcepten

De distributiemodule van Draco biedt ondersteuning voor zwakke mobiliteit. Dit wil zeggen dat componenten die in gebruik zijn naar een andere knoop in het netwerk verplaatst kunnen worden, transparant voor de kern van het systeem.

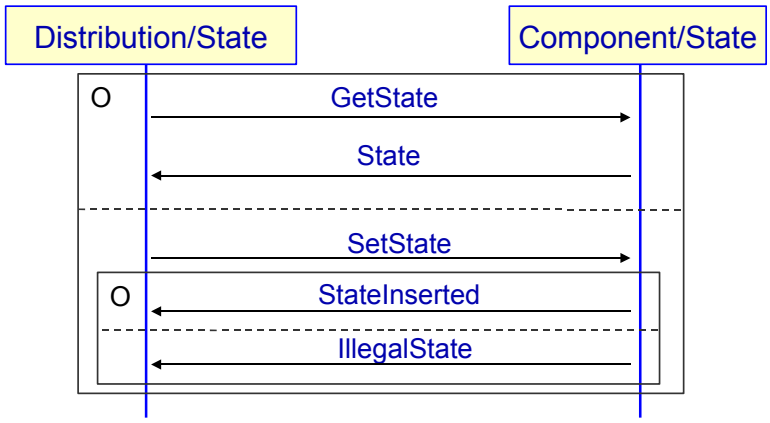
Zwakke mobiliteit wil zeggen dat de threads die de componenten uitvoeren hun toestand niet behouden. Dit is echter geen probleem als we de verplaatsing van een component uitvoeren wanneer deze niet in een actieve toestand is. Er mogen tijdens de verplaatsing dus geen transacties aan de gang zijn en er mogen ook geen nieuwe transacties beginnen met uitvoering tijdens het verplaatsen. Om dit te bekomen kunnen de connectors bevroren worden, zodat alle aankomende boodschappen in de wachtrij blijven. We moeten wachten tot de uitvoerende boodschappen afgelopen zijn, zodat een inactieve toestand van de component verkregen wordt. Daarna kan de toestand uit de component geëxtraheerd worden en de code van de component kan gereloceerd worden.

Mechanismen voor zwakke mobiliteit

Nadat een verbinding is opgezet, kunnen componenten verplaatst worden door het `move`-commando te geven op de shell.

```
move <component name> <connection name>
```

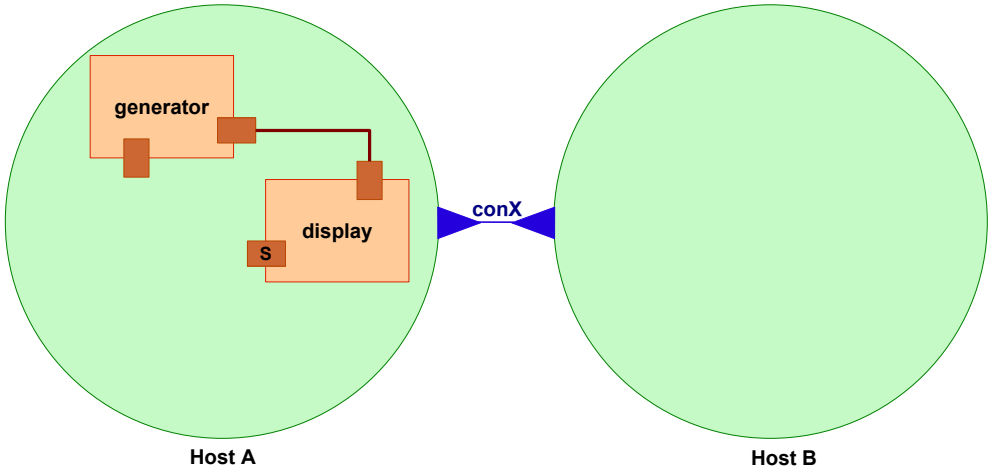
De communicatie met de component wordt bevroren. Dit gebeurt door in te grijpen in de connector, zodat de boodschappen in een buffer geplaatst worden. De component komt in



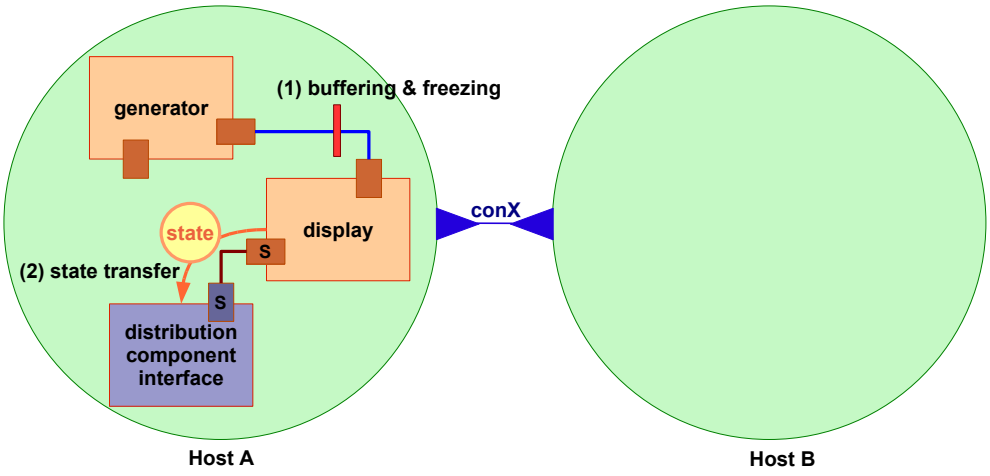
Figuur 3.9: De State poort van de distributie module wordt verbonden met de State poort van een component. De boodschap SetState of GetState wordt uitgewisseld om de toestand uit de component te halen of deze terug te herstellen.

een inactieve toestand. De toestand wordt uit de component gehaald door de GetState boodschap te sturen. De programmeur van de mobiele component moet beslissen welke toestand de verplaatsing van de component moet overleven. Alle relevante informatie moet in een containerobject geplaatst worden. Nadat de component verplaatst is naar een andere knoop in het gedistribueerde systeem, wordt de toestand terug in de component gebracht door middel van de SetState boodschap. Daarna wordt de connector ontvroren en de boodschappen die ondertussen in de buffer verbleven, worden afgeleverd. Om de communicatie te verzorgen, worden proxies van de verbonden componenten aangemaakt. Wanneer een boodschap verstuurd wordt tussen twee lokale componenten, dan zal die boodschap in de situatie waar een component zich op een andere knoop bevindt, gestuurd worden naar de proxy die voor de ontvangende component gegenereerd werd. De boodschap wordt over het netwerk gestuurd en de proxy van de versturende component, die op de andere knoop in het netwerk gegenereerd werd, stuurt de boodschap opnieuw uit. De proxies die aangemaakt worden, moeten enkel de boodschap doorsturen. Ze wegen dus niet zwaar op de systeembronnen. In figuren 3.10 tot 3.14 tonen we de werking van het mechanisme in vijf stappen voor een klein voorbeeld.

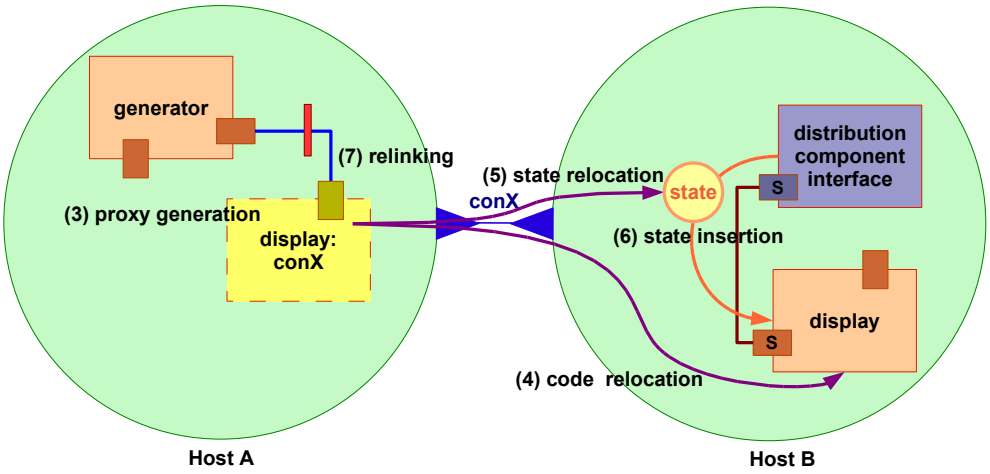
In deze thesis maken we gebruik van de zwakke mobiliteit die door Draco aangeboden wordt, om componenten tussen knopen te laten migreren, met het oog op werklastverdeling. Omdat Draco component-geöriënteerde applicaties gebruikt, kunnen we vrij eenvoudig componenten van deze applicaties op andere knopen in het netwerk onderbrengen, zonder de functionaliteit van de applicatie te veranderen. We moeten in de applicatie zelf geen veranderingen aanbrengen. De enige vereiste die we stellen is dat voor applicaties waar we de werklastverdelingsmethode op willen toepassen, de componenten zwakke mobiliteit ondersteunen. (d.w.z. SetState en GetState boodschappen implementeren)



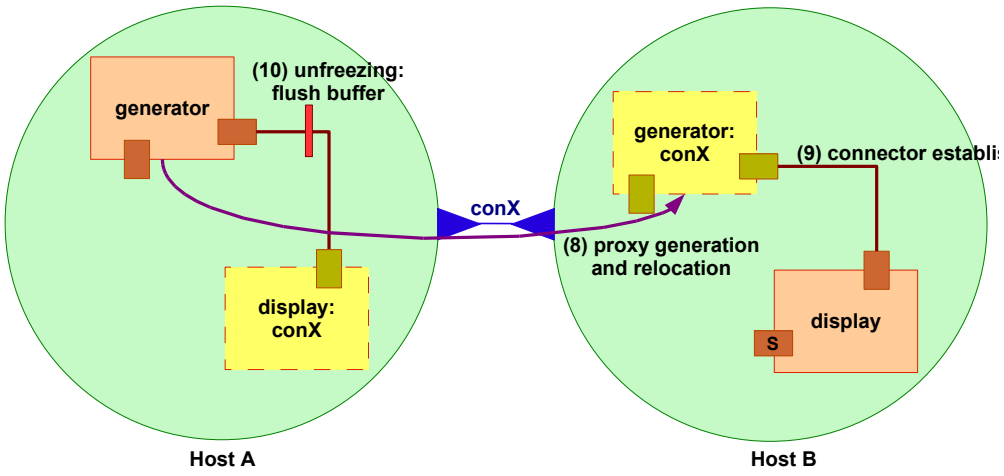
Figuur 3.10: Beginsituatie: de *generator*-component genereert boodschappen die door de *display*-component getoond worden. We willen de display-component verplaatsen over verbinding *conX*.



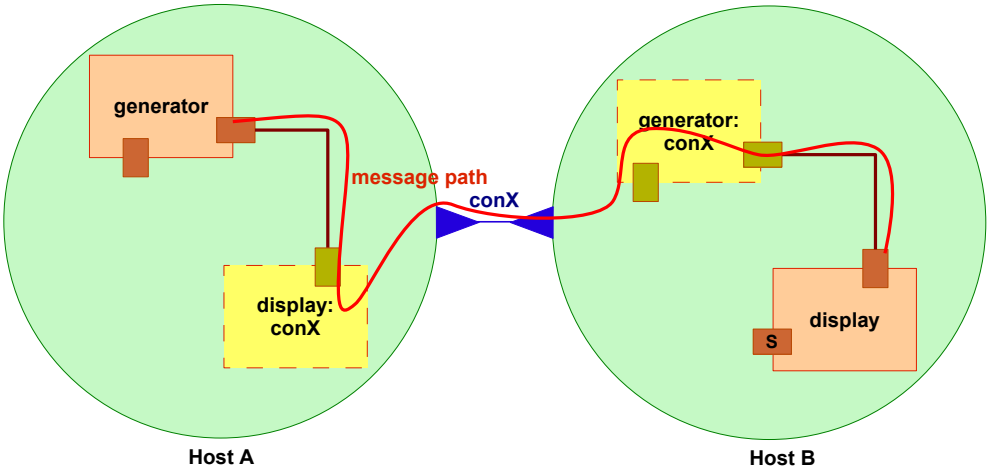
Figuur 3.11: In de eerste fase wordt de *display*-component in een inactieve toestand gebracht. Dit gebeurt door alle boodschappen naar en van de poorten van *display* te blokkeren en te bufferen (1). Nadat de component bevroren is, wordt de toestand opgevraagd door de componentinterface van de distributiemodule (2).



Figuur 3.12: Vervolgens genereert de methode een proxy (3) voor *display* en wordt de code (4) en toestand (5) verplaatst. Daarna wordt de toestand die in (2) opgehaald werd, terug in de *display*-component gebracht (6) door de componentinterface van de distributiemodule. Daarna kan de losse connector verbonden (7) worden met de proxycomponent.



Figuur 3.13: (7) zorgt ervoor dat er ook voor de *generator*-component een proxy aangemaakt wordt (8). Die wordt over *conX* gestuurd. De originele connector wordt gedupliceerd (9) op doelknoop B. Op dit moment kunnen de originele connectors ontvroren worden. De gebufferde boodschappen zullen afgehandeld worden (10).



Figuur 3.14: Met de volle lijn wordt in de resulterende situatie de weg die een boodschap aflegt aangeduid. Een boodschap verzonden door de *generator*-component wordt door de proxy van *display* doorgestuurd. Op de doelknoop wordt deze boodschap opnieuw uitgezonden om de originele zending na te bootsen.

Hoofdstuk 4

Dynamische gedistribueerde werklastbalancering

In dit hoofdstuk beschrijven we de aanpak die we gehanteerd hebben om de doelstellingen van deze thesis – zoals ze in de inleiding (zie sectie 1.3) beschreven zijn – te bereiken. In een eerste sectie beschrijven we hoe we aan systeembrongegevens komen en hoe we Draco uitgebreid hebben om deze informatie te kunnen gebruiken. In een volgende sectie leggen we uit hoe we deze informatie gebruiken om aan werklastherverdeling te doen. We bouwen dynamisch een nieuwe componentenuitrol op, transparant voor de originele applicatie, om de geheugenlast van de applicatiecomponenten te spreiden over beschikbare systeembronnen in de omgeving. Tijdens deze fase moeten we rekening houden dat communicatiebandbreedte tussen applicatiecomponenten de beschikbare bandbreedte tussen hardwareknopen niet overschrijdt.

4.1 Systeembrongegevens

Enkele systeembronvereisten waar we rekening mee zouden kunnen houden zijn

- **Timing:** fragmenten uitvoerbare code moeten binnen een vooropgestelde tijd uitgevoerd worden. Typisch gaat het over een enkele vaak terugkerende taak, bv. een stroom muziekgegevens die gecodeerd moet worden voor verzending.
- **Geheugen:** geheugen is nodig om code, gegevens en runtime variabelen onder te brengen.
- **Bandbreedte:** communicatie tussen componenten vereist dat er een afdoende hoeveelheid gegevens doorgestuurd kan worden tussen deze componenten. Hiervoor kan een bepaalde hoeveelheid bandbreedte gereserveerd worden.
- **Opslagruimte:** de nood om gegevens weg te schrijven naar een blijvend medium [persistent storage]. Dit mogelijk tegen een vastgestelde snelheid.
- **Batterijcapaciteit:** een neveneffect dat het gevolg is van het gebruik van systeembronnen is vaak het afnemen van resterende batterijcapaciteit.

- **Financiële kost:** een ander neveneffect dat in rekening gebracht kan worden is de kost van bandbreedtegebruik of van batterijlading.

In deze thesis houden we voor de werklastverdeling enkel rekening met geheugenvereisten van componenten en de geheugenbeperkingen van hardwareknopen. We houden ook rekening met de beperkingen die de bandbreedte tussen de hardwareknopen oplegt aan het plaatsen van componenten op de hardwareknopen. Deze twee systeembronnen zijn de belangrijkste beperkende factoren voor applicaties in een mobiele context. Er is duidelijk een afweging tussen de twee systeembronnen. Enerzijds is het goed om het geheugengebruik van een applicatie over de verschillende hardwareknopen in het netwerk te spreiden. Anderzijds houdt deze spreiding een extra kost in voor het bandbreedtegebruik. Andere systeembrongegevens kunnen ook opgenomen worden in de beslissingsprocedure (zie sectie 4.4.3). Om aan te tonen dat de ontwikkelde methode werkt, hebben we voldoende aan het gebruik van de twee gekozen systeembronnen.

4.1.1 Configuratie

Om te weten te komen wat de systeembronvereisten van componenten zijn, maken we gebruik van configuratiebestanden. Voor elke component bepalen we vooraf, na berekeningen of metingen, welke de waarden zijn voor de verschillende resourcevereisten. Het systeem leest die waarden uit het componentconfiguratiebestand *ComponentStructure.xml*. In de `<metadata>`-tag voegen we de `<resource>`-tag toe waarin alle relevante waarden gestructureerd opgeslagen worden.

```
1  <ComponentStructure>
2      <component type="atomic">
3          <componentclass>components.dummy.DummyComponent</componentclass>
4      </component>
5      <metadata>
6          <resource>
7              <memory>1024540</memory> <!-- 1MB -->
8              <bandwidth>
9                  <connector name="Information">100</connector>
10             </bandwidth>
11         </resource>
12     </metadata>
13 </ComponentStructure>
```

Listing 4.1: ComponentStructure.xml-configuratie bestand voor een dummy-component met 1 MB vereist geheugen, en een poort Information met een bandbreedtegebruik van 100 bps

In listing 4.1 tonen we een voorbeeld van een configuratiebestand. De geheugenvereisten voor de component worden opgegeven binnen de `<memory>`-tag in bytes. De bandbreedtevereisten (in bps) worden voor elke connector opgegeven, waarbij de naam van de connector gespecificeerd wordt. Dit gebeurt binnen de `<bandwidth>`- en `<connector>`-tag.

Ook voor elke knoop in het netwerk worden de beschikbare resources uit het configuratiebestand *Config.xml* gelezen. Dit bestand dient aangepast te zijn aan de onderliggende

hardwareconfiguratie, zodat het de werkelijk beschikbare resources zo goed mogelijk benadert. In listing 4.2 tonen we een voorbeeldconfiguratie voor de PDA. Het beschikbaar geheugen wordt binnen de `<Memory>`-tag gespecificeerd in bytes. De beschikbare bandbreedte wordt in de `<Bandwidth>`-tag opgegeven. We geven slechts een bandbreedte op per DRACO-instantie.

```
1 <Draco nodeID="PDA">
2   <Resources>
3       <Memory>15728640</Memory> <!-- 15 MB  -->
4       <Bandwidth>33554432</Bandwidth> <!-- 4mbps -->
5   </Resources>
6 </Draco>
```

Listing 4.2: Config.xml voor PDA

We gaan ervan uit dat elke hardware-knoop slechts 1 fysieke verbinding met het netwerk heeft. Over deze netwerkverbinding kunnen wel meerdere TCP/IP-verbindingen opgezet worden met andere DRACO-instanties. De veronderstelling dat er slechts 1 fysieke netwerkverbinding is per toestel is geldig in de huidige standaarden voor Wireless-LAN. Wireless-LAN wordt ook wel Wi-Fi of 802.11b/g genoemd. In deze thesis gebruiken we alleen de Wi-Fi-verbinding van de PDA. De bekomen netwerktopologie heeft een sneeuwvlokstructuur. De uiteinden zijn knopen die we in deze thesis beschouwen. De tussenliggende netwerkapparatuur zoals routers, switches en hubs laten we buiten beschouwing. De netwerkverbindingen tussen de DRACO-instanties kunnen we zien als een overlaynetwerk[1].

Er kunnen echter wel andere (ad-hoc[2]) netwerkverbindingen gemaakt worden, bijvoorbeeld door gebruik te maken van Bluetooth. Bluetooth is in veel opzichten te vergelijken met de Wi-Fi techniek. Er zijn echter belangrijke verschillen. Via Bluetooth kunnen apparaten elkaar herkennen en een beperkte hoeveelheid gegevens met elkaar uitwisselen over korte afstanden. Het biedt bovendien een korte aanmeldingsprocedure. Synchronisatie van agendas of adressenbestanden is een goed voorbeeld. Bovendien is Bluetooth meer geschikt voor ad-hoc netwerken. Wi-Fi daarentegen, is bedoeld voor meer data-intensief gebruik met een hogere snelheid (11 of 54 Mbps) en over grotere afstanden, tot zo'n 100 meter, met een meer permanent karakter. Een draadloze netwerkadaptor kan met de huidige technologie slechts een verbinding maken met één accesspoint. We gaan er vanuit dat een mobiel toestel niet meer dan één zulke netwerkadapter aan boord heeft[3]. In de toekomst zullen ook voor Wi-Fi betere ad-hoc protocollen uitgewerkt en gestandaardiseerd worden, maar hier hebben we nog niet mee kunnen experimenteren.

[1] Een overlaynetwerk is een netwerk op applicatie-niveau dat gebruik maakt van het beschikbare netwerk op een lager niveau in de protocolstapel. De knopen in een overlaynetwerk zijn logisch of virtueel verbonden. Het pad dat verkeer tussen deze knopen volgt, is niet noodzakelijk gelijk aan de rechtstreekse logische verbinding tussen deze knopen. Vele peer-to-peer netwerken zijn bijvoorbeeld overlaynetwerken, omdat ze een netwerk vormen bovenop het Internet.

[2] Een ad-hoc netwerk is een netwerk zonder structuur of hiërarchie dat spontaan ontstaat

[3] [1] beschrijft een recente techniek die het mogelijk maakt om met meerdere IEEE 802.11 netwerken verbinding te maken, gebruik makend van een enkele draadloze netwerkkaart. Deze techniek, MultiNet gedoopt, is een ontwikkeling van Cornell University in samenwerking met Microsoft Research. Er wordt gebruik gemaakt van een tussenlaag onder IP, die voortdurend wisselt tussen meerdere netwerken. Deze techniek moet transparant zijn voor de gebruiker en bovenliggende netwerklagen in de protocolstapel.

4.1.2 Offline metingen

Door de aanpak met configuratiebestanden te hanteren, kunnen we het systeem ontzien van het uitvoeren van online-metingen van gebruikte en beschikbare resources. Het nadeel is wel dat we vooraf de waarden nauwkeurig moeten bepalen. Deze taak laten we aan de applicatieprogrammeur over. De informatie in de configuratiebestanden is ook niet zo dynamisch. Binnen deze thesis is er echter geen plaats om hier verder op in te gaan.

[13] toont een methode om het geheugengebruik van componenten te bepalen in een embedded componentensysteem. [4] biedt een methode om nauwkeurig de bandbreedte van een netwerkverbinding te bepalen door metingen uit te voeren.

De veranderingen in werklading voor elke knoop bestaan uit het inladen, uitladen en verplaatsen van componenten. Elke actie van die soort lokt een reactie van het systeem uit. Er wordt berekend of het opportuun is om een component te verplaatsen, om de werklast te herverdelen.

De systeembronnen die we in deze thesis in rekening brengen zijn geheugen en bandbreedte. Er is een spanningsveld tussen beide systeembronnen. We willen het geheugengebruik optimaal spreiden, maar wanneer we dit doen zonder rekening te houden met de bandbreedtevereisten van de componenten, kan het gebeuren dat we een componentenuitrol bekomen waarin de commmunicatie tussen componenten steeds over het netwerk moet gebeuren. Het is beter om componenten met een hoge interactiefrequentie samen op dezelfde hardwareknopen te plaatsen, om het netwerkgebruik minimaal te houden. We krijgen op die manier een componentenuitrol die binnen de beperkingen van het netwerk, een maximale spreiding van de geheugenlast bekomt in afweging met de belasting van het netwerk.

4.2 Diffusie door zenderinitiatief

We hanteren een aanpak die gebruik maakt van een lokale herverdelingsomgeving om de geheugenlast van een hardwareknoop te spreiden. De inspiratie voor deze aanpak komt uit [17] waar een vijftal strategieën besproken worden om aan werklastbalancering te doen op massief-parallelle computers. In [17] wordt de werklast verdeeld op het niveau van processen. Het besturingssysteem verplaatst identieke processen, die mogelijk onderling communiceren. Deze processen berekenen bijvoorbeeld een drukverdeling in een eindige-elementen modellering van een vliegtuigvleugel.

De strategie die in de context van een mobiele, gedistribueerde omgeving toegepast kan worden is die van diffusie door zenderinitiatief [Sender Initiated Diffusion (SID)]. *Diffusie door zenderinitiatief* is een gedistribueerde lokale aanpak. Er wordt gebruik gemaakt van de werklastinformatie van de nabije buren om een surplus aan werklading van zwaar beladen rekenknopen toe te wijzen aan onderbeladen buur-rekenknopen. Globaal evenwicht wordt bereikt doordat de werklast van zwaar beladen omgevingen verspreid wordt over de licht beladen omgevingen.

4.2.1 Beschikbare informatie

We houden bij de werklastverdeling rekening met de systeembronnen geheugen en bandbreedte. Telkens er een verandering optreedt in de werklast, verstuurt de knoop in kwestie een *LoadUpdate* boodschap naar knopen binnen zijn balanceringsdomein. Na elke verandering binnen het domein wordt binnen dit balanceringsdomein dus elk lid op de hoogte gebracht van de veranderingen. Nadat de boodschappen verstuurd zijn kan elke knoop op basis van alle beschikbare informatie zijn beslissingsmechanisme activeren.

4.2.2 Harde voorwaarden

We geven formeel aan welke invarianten steeds voldaan moeten zijn binnen het systeem.

Bandbreedte

De vereiste bandbreedte van componenten die hun boodschappen over het netwerk sturen blijft binnen de beperkingen van de bandbreedte van de netwerkverbinding voor elke knoop.

$$\forall K : \sum_{c \in C_N} BW(c) < BW(K)$$

waarbij C_N de verzameling van componenten is die het netwerk gebruiken voor de knoop K. $BW(c)$ is de bandbreedtevereiste voor component c. $BW(K)$ is de bandbreedte beschikbaar op knoop K.

$$C_N = C_{IM} \bigcup C_M$$

waarbij C_{IM} de verzameling van immigrante componenten is, componenten afkomstig van externe knopen. En C_M de verzameling van migrante componenten, componenten die oorspronkelijk op deze knoop ingeladen werden, maar op het moment van de berekening uitbesteed zijn aan externe componenten.

Er kan ook transitverkeer zijn, waarbij de componenten die voor deze communicatie verantwoordelijk zijn zelf niet lokaal op de knoop uitgevoerd worden. Van deze componenten zijn enkel proxies lokaal gekend. Dit transitverkeer moet ook in rekening gebracht worden.

Omdat de distributiemodule van Draco de proxy van een component die terug naar zijn oorspronkelijke knoop verplaatst wordt niet verwijdert, moet goed opgelet worden dat enkel verbindingen tussen proxies met verschillende connecties in rekening gebracht worden als transitverkeer.

Bij [10] (zie 2.1.4) wordt gebruik gemaakt van de interactiefrequentie tussen componenten gebruikt om de communicatie tussen componenten te kwantificeren. In onze aanpak specificeren we het bandbreedtegebruik per connector. Impliciet zit hierin de interactiefrequentie ook vervat. De grootte van de boodschappen wordt bovendien op deze manier ook in rekening gebracht. Bij [10] wordt niet gesproken over de grootte van boodschappen. Om beperkingen op te leggen aan het (samen-)plaatsen van componenten, wordt in [10] gebruik gemaakt van twee functies: *loc* en *colloc*. Wij gaan ervan uit dat elke component waarvoor geheugenvereisten

opgegeven zijn, verplaatst mag worden. We leggen geen beperkingen op aan het samenplaatsen van componenten. Bij [10] worden deze functie een beetje artificieel toegevoegd, om zelf manueel invloed te kunnen uitoefenen op de componentenuitrol van de applicatie.

Geheugen

De geheugenlimiet wordt op geen enkele knoop overschreden. Met andere woorden: de som van het geheugen van alle componenten die zich op het moment van de berekening op knoop K bevinden, is kleiner dan het beschikbare geheugen van knoop K.

$$\forall K : \sum_{c \in C_L} MEM(c) < MEM(K)$$

waarbij C_L de verzameling is van componenten lokaal aan knoop K op het moment van de berekening. $MEM(c)$ is de geheugenvereiste voor component c. $MEM(K)$ is het geheugen toegewezen aan knoop K.

$$C_L = (C_O \backslash C_M) \bigcup C_{IM}$$

waarbij C_O de verzameling is van componenten die oorspronkelijk ingeladen werden op knoop K.

Bij [10] (zie 2.1.4) wordt op een gelijkaardige manier het geheugengebruik en aangeboden geheugen gedefinieerd. Om een nieuw deploymentplan uit te werken, begint de methode van [10] steeds van nul. Ze houden geen rekening met de huidige toestand van de applicatie wanneer tot een nieuwe componentenuitrol overgegaan wordt. Wij hanteren een incrementele techniek, die steeds verder bouwt op de bestaande situatie.

4.2.3 Werklastherverdelingsstrategie

Wanneer er een verandering in de werklast optreedt, lokaal of in een van de buren, wordt er bepaald of het opportuun is om de werklast te balanceren. Dit wordt gedaan door de lokale werklast te vergelijken met de gemiddelde werklast binnen het lokale balanceringsdomein. Wanneer deze sterk afwijkt (hoger dan een bepaalde threshold) van het gemiddelde wordt overgegaan tot actie.

De gemiddelde werklast wordt berekend binnen het lokale domein.

$$M_L = \frac{\sum_{k \in K_B} MEM(k) + MEM(K)}{\#K_B + 1}$$

waarbij K_B de verzameling van buren voor knoop K is. Vervolgens wordt de werklastonbalansfactor [load imbalance factor] bepaald door het verschil te maken tussen de eigen werklast en het gemiddelde.

$$\Phi_L = MEM(K) - M_L$$

Bij voldoende grote afwijking van het gemiddelde wordt de beslissing genomen om te trachten, binnen de grenzen van de harde beperkingen, de werklast te herverdelen.

$$\Phi_L > Threshold$$

Migratie van componenten

Wanneer we een component-buurpaar selecteren dat in aanmerking komt om te migreren, berekenen we voor dat component-buurpaar de systeembronvereisten voor de potentieel nieuwe situatie, en controleren we of ze binnen de harde voorwaarden blijven die steeds voldaan moeten zijn.

Voor de component-buurparen die aan deze voorwaarden voldoen, wordt een getal berekend dat het voordeel van de verplaatsing kwantificeert. Hoe groter dit getal, hoe beter het is om die component te verkiezen om te verplaatsen naar de gekozen buur. Er wordt dus telkens zo gulzig mogelijk gekozen.

Bij de berekening van dat getal worden twee factoren in rekening gebracht. De eerste factor stelt de algemene balans voor die binnen het balanceringsdomein bereikt wordt. Informeel kunnen we stellen dat de maximale verbetering in balans het beste is. Deze kunnen we bekomen door voor de nieuwe situatie de werklastonbalansfactor te berekenen en die van de huidige af te trekken. Hoe groter de absolute waarde van dit getal, hoe beter.

Verbetering van de geheugenspreiding: (groter is meer verbetering)

$$Vm_c = |\Phi_{L\text{now}} - \Phi_{L\text{after move}}|$$

Dit getal ligt tussen 0 en 1.

De tweede factor die in rekening gebracht wordt, is het bandbreedtegebruik. We willen voor de nieuwe configuratie een zo klein mogelijk bandbreedteverbruik verkrijgen. We berekenen dus het verschil tussen het huidige en het potentieel toekomstige bandbreedtegebruik.

Verandering toekomstig geheugengebruik: (kleiner is beter)

$$Vbw_c = \frac{BW_{\text{after move}}}{BW_{n\text{tot}}}$$

Dit getal ligt tussen -1 en 1.

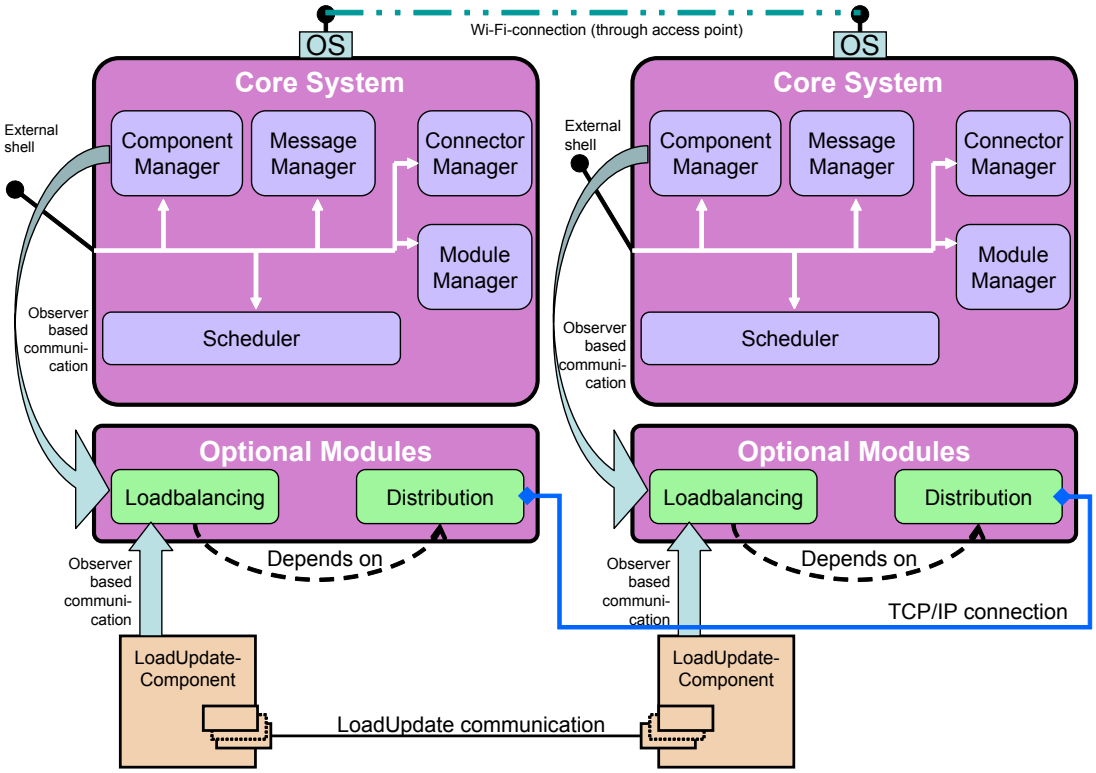
Van deze twee getallen maken we een gewogen convexe[4] combinatie. Voor elke factor kunnen we dus het relatieve gewicht in de totale beslissing voor het uiteindelijke getal beïnvloeden. Afhankelijk van de verhouding in de kost van de bandbreedte en het geheugen kunnen we dit getal bijstellen. We hebben gekozen om evenveel belang te hechten aan geheugen als aan bandbreedte.

$$V_c = M_{\text{frac}} * Vm_c - BW_{\text{frac}} * Vbw_c$$

4.3 Implementatie

We hebben Draco uitgebreid met een module (de `werklastverdelingsmodule` [loadbalancing]) om de werklastverdeling af te handelen. Zie figuur 3.6 en 4.1. Deze module wordt door de Component-manager op de hoogte gebracht van het in- en uitladen van componenten via

[4] $M_{\text{frac}} + BW_{\text{frac}} = 1$



Figuur 4.1: Plaats van de werklastverdelingsmodule binnen de Draco-architectuur. De figuur toont twee verbonden Draco-instanties.

het observer-patroon. De module wordt één keer ingeladen per Draco-instantie. De module maakt een Draco-component (`LoadUpdateComponent`) aan die instaat voor het verzenden en ontvangen van *LoadUpdate* berichten binnen het balanceringsdomein. De module communiceert met deze component door middel van het observer-patroon. Bij elke ontvangst van een *LoadUpdate* bericht wordt de informatie over de buren binnen de module up-to-date gebracht, en wordt gecontroleerd of het opportuun is om tot herbalancering over te gaan.

Wanneer de beslissing genomen wordt om een component te verplaatsen, zal de werklastverdelingsmodule de `Move`-opdracht doorgeven aan de distributiemodule, waarna de component verplaatst wordt, en vervangen door een proxy. De distributiemodule moet daarom ingeladen worden voordat de werklastverdelingsmodule ingeladen wordt. Het opzetten van de communicatie tussen de werklastverdelingscomponenten op de verschillende hardwareknopen moet handmatig gebeuren. Dit proces kan geautomatiseerd worden door het gebruik van een script. In hoofdstuk 5 werken we een scenario uit waarin we tonen hoe een systeem opgestart moet worden.

4.4 Evaluatie van de gekozen methode

4.4.1 Afweging informatie en overhead

We moeten een afweging maken tussen de beschikbare informatie en de overhead die het verspreiden van deze informatie met zich meebrengt. Wanneer we in een systeem volledige kennis hebben over de werklast van elke knoop op elk moment, kunnen we een gecentraliseerd algoritme gebruiken om een optimaal deploymentplan voor de applicatie te bepalen. (Zie sectie 2.1.4 als voorbeeld.) Hiertegenover staat dat het niet altijd gemakkelijk is om deze informatie centraal te verzamelen. Bovendien kan het onhaalbaar zijn om indien deze informatie beschikbaar is een optimale spreidingsarchitectuur te bepalen, omwille van complexiteitsproblemen. (zie [10].) Dan zal er toch overgegaan moeten worden op benaderingsmethodes.

De techniek die we in deze thesis gebruiken, baseert de spreiding van de werklast (softwarecomponenten) op lokale informatie. De nabije buren worden in overweging genomen, op basis van de gekende hoeveelheid werklast, om componenten aan uit te besteden. De werklastinformatie die we moeten uitwisselen blijft beperkt tot het lokale balanceringsdomein. Hierdoor kan het zijn dat er elders in het netwerk knopen bestaan die beter geschikt zijn om de werklast over te nemen. We kunnen een sub-optimale spreiding van werklast bekomen.

4.4.2 Overlappende balanceringsdomeinen

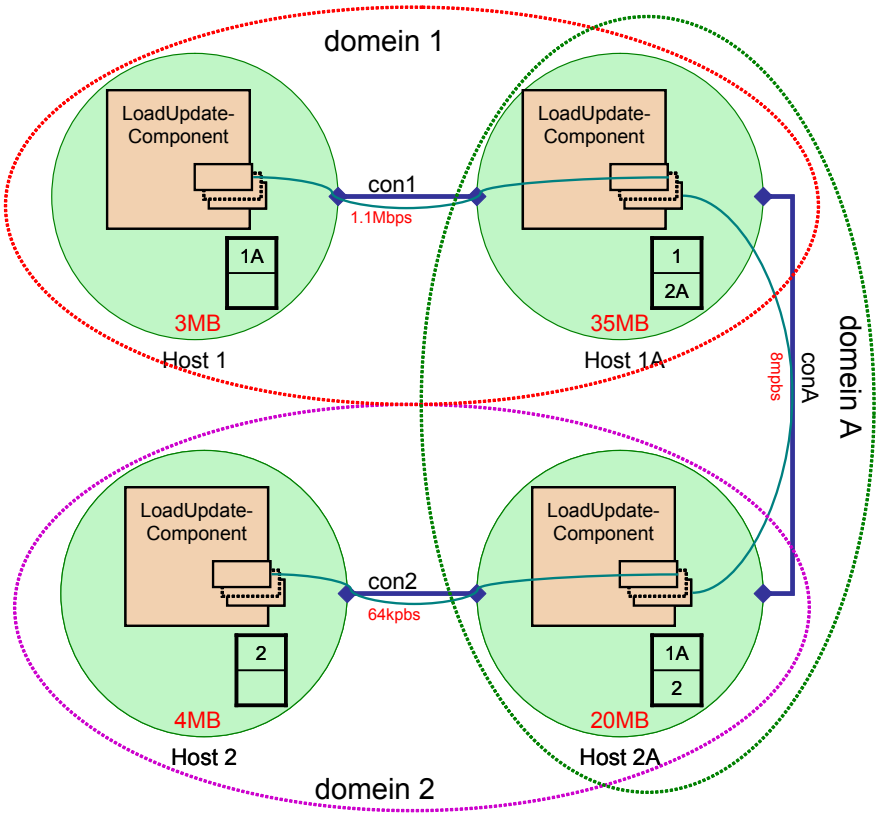
Wanneer we ervoor zorgen dat de balanceringsdomeinen binnen het systeem overlappen, kan de werklast verder verspreid worden. De randknopen, die tot meer dan één balanceringsdomein behoren, kunnen werklast afkomstig van het ene balanceringsdomein overdragen aan een ander balanceringsdomein. Op termijn zal de werklast dus over het hele systeem verspreid kunnen worden. Hoge werklast vindt door middel van diffusie zijn weg naar gebieden met lagere werklast binnen het systeem, totdat er een evenwicht bereikt is. Zie figuur 4.2.

Het moet echter mogelijk zijn om binnen de beperkingen van de aangeboden en vereiste systeembronnen, componenten door te sturen. Het kan gebeuren dat een knoop niet optimaal belast wordt, omdat deze afgescheiden is van de rest van het systeem door een minder sterke knoop, waar de werklast niet over kan passeren.

4.4.3 Andere systeembroninformatie gebruiken

Wanneer we in de methode andere systeembroninformatie willen gebruiken om de verdeling van de werklast te beïnvloeden, kunnen we deze meesturen in de *LoadUpdate*-berichten die worden uitgewisseld tussen buren in een balanceringsdomein. Deze informatie kunnen we in de beslissingsprocedure opnemen als extra term in de convexe combinatie die componentknoopparen selecteert.

Als we bijvoorbeeld voor elke knoop in het netwerk de resterende batterijcapaciteit meedelen aan de buren in het balanceringsdomein, en we kunnen voor elke component de afname van batterijcapaciteit kwantificeren, dan kan het algoritme de batterijwerklast binnen het systeem spreiden. De totale werktijd zal toenemen, omdat systemen met relatief weinig resterende batterijcapaciteit minder zwaar belast zullen worden. Het is echter niet eenvoudig om



Figuur 4.2: Overlappende balanceringsdomeinen. Domein 1 is volledig gescheiden van domein 2. Wanneer we domein A introduceren, dan kan werklast tussen de twee domeinen doorgegeven worden. In de tabellen bij elke knoop geven we aan voor welke knoop informatie bijgehouden wordt. Een applicatie die op knoop 1 gestart werd, kan componenten hebben die zich op knoop 2 bevinden. Hierbij zal wel steeds aan de opgelegde beperkingen voldaan moeten zijn.

de invloed – als deze al meetbaar is – van een component op de resterende batterijcapaciteit te bepalen.

4.4.4 Veranderingen in de fysieke omgeving

In een mobiele omgeving kunnen hardwareknopen in het netwerk eenvoudig vertrekken. Iemand schakelt bijvoorbeeld zijn apparaat uit, of verliest zijn verbinding met het netwerk. In de huidige toestand van het syteem zullen er componenten verloren gaan, en zal de applicatie waarschijnlijk blokkeren, omdat bepaalde componenten niet meer bereikbaar zijn.

Wanneer we online informatie uit de omgeving zouden hebben, kunnen we wel zorgen dat componenten die afgesloten worden alle migrante componenten wegzenden. Het kan zijn dat een verbinding wegvalt, maar dat er nog andere paden in het netwerk bestaan. De verbindingen zullen dan veranderd moeten worden, maaar de applicatie zal kunnen blijven werken.

De informatie die we gebruiken, komt echter voort uit configuratiebestanden die tijdens het opstarten van het systeem eenmalig uitgelezen worden. We kunnen dus niet op impulsen

van buiten het systeem reageren. Binnen het bestek van deze thesis kunnen we voor dit probleem geen oplossing voorstellen.

Hoofdstuk 5

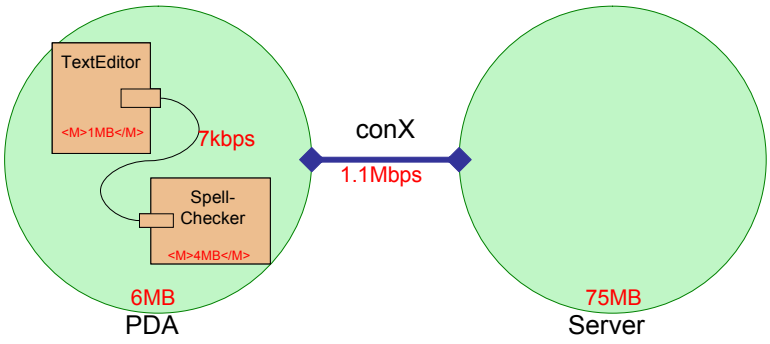
Empirische evaluatie

We testen de effectiviteit van de herverdelingsstrategie door enkele scenario's uit te proberen en te analyseren hoe de strategie op een aantal uitdagingen reageert.

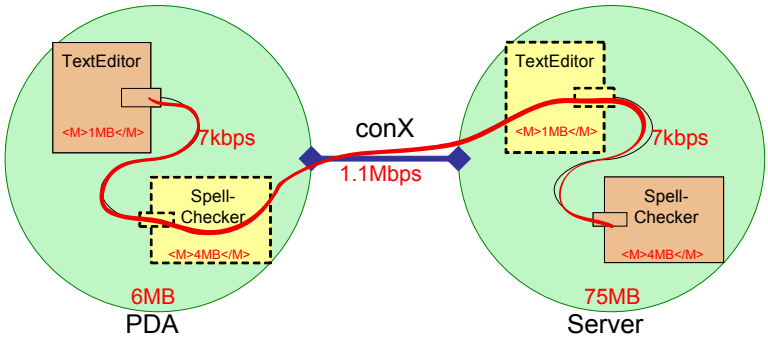
5.1 Voorbeeld

In deze sectie geven we een eenvoudig voorbeeld hoe de methode die we ontwikkeld hebben nuttig kan gebruikt worden. Hiervoor ontwikkelden we een kleine applicatie. Een teksteditor bestaat uit twee componenten: de gebruikersinterface en een spellingscontrolecomponent. Wanneer deze component een stuk tekst ontvangt, vergelijkt het de woorden met een woordenboek en geeft voor niet gekende woorden een lijst van alternatieven terug.

De applicatie kan op de PDA gebruikt worden. Wanneer we de werklastverdelingsmodule gebruiken, zal wanneer mogelijk de spellingscontrolecomponent naar een van de buren gestuurd worden. Op figuur 5.1 tonen we de opbouw van de applicatie. Op figuur 5.2 tonen we de situatie nadat de spellingcontrolecomponent verplaatst is, om het geheugengebruik te spreiden.



Figuur 5.1: Beginsituatie. Op beide knopen wordt de loadbalancing module ingeladen. Voor elke knoop wordt de geheugencapaciteit ingelezen. Voor elke component moeten geheugenvereisten opgegeven worden.



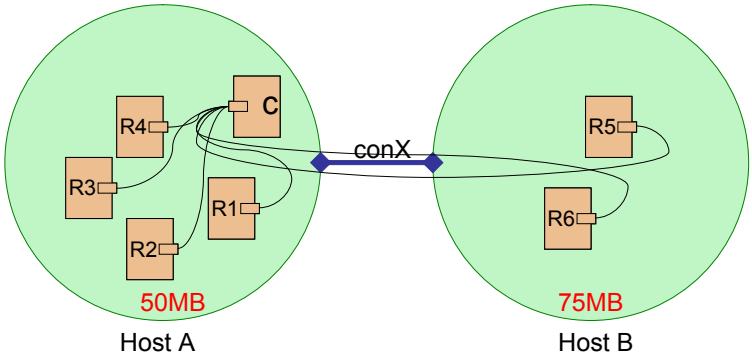
Figuur 5.2: Situatie na balancering. De componenten in streepjeslijn zijn proxies.

5.2 Threshold – Geheugen

We gaan na wat de invloed van verschillende waarden voor de threshold op de werklastverdeling is. De threshold is de waarde die uitdrukt hoeveel afwijking er minimum moet zijn van de gemiddelde werklast in het lokale balanceringsdomein om tot herverdeling over te gaan.

5.2.1 Testmethode

Om dit te testen gebruiken we een eenvoudige setup. Twee knopen worden verbonden en op elke knoop laden we de werklastverdelingsmodule. Vervolgens laden we op een knoop 70 componenten van 1MB. Zie figuur 5.3. We kijken vervolgens hoe deze componenten verdeeld worden over de knopen.

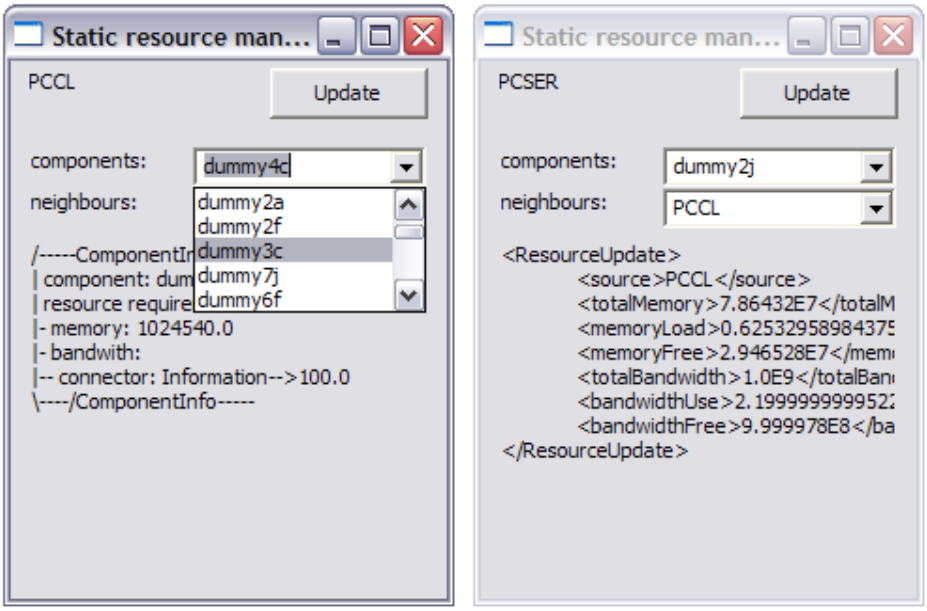


Figuur 5.3: De “applicatie” bestaat uit 1 vaste component (C) die verbonden wordt met een willekeurig aantal potentieel mobiele componenten (R) met een geheugengebruik van elk 1MB. Deze vaste knoop is nodig omdat niet-verbonden componenten niet verplaatst kunnen worden door de huidige distributiemodule.

5.2.2 Resultaten

We voeren zes experimenten uit. Drie verschillende waarden voor de threshold, en voor elke van die waardes laden we telkens de componenten in op de twee knopen. Het opzetten van

het systeem gebeurt manueel. Wanneer we beginnen met het inladen van de 70 componenten, moeten we ongeveer een halve minuut wachten tot ze allemaal ingeladen zijn en de herverdeling voltooid is. Tussen het inladen van elke component zit een korte wachtperiode om het systeem tijd te geven de nodige connectors te verbinden. De resultaten kunnen eenvoudig op de console afgelezen worden. Elke component geeft bij ontvangst van de "Information"-boodschap, die via de vaste component C gestuurd wordt, aan waar deze zich op het moment van ontvangst bevindt. We ontwikkelden een kleine grafische gebruikersinterface om de toestand van de knoop, zijn buren en lokale componenten in de gaten te kunnen houden (zie figuur 5.4). Hierbij hielden we er rekening mee dat de interface ook op de PDA bruikbaar moet zijn.



Figuur 5.4: Gebruikersinterface van de werklastverdelingsmodule. Wanneer op de knop update geklikt wordt, zal de informatie die op dat moment beschikbaar is in de interface beschikbaar zijn. De linkerfiguur toont de informatie die over een component gekend is. De rechterfiguur toont de ontvangen informatie van een buur.

We beginnen met een threshold waarde van $0,1$. Wanneer we de werklast laden op knoop A, wordt deze als volgt verdeeld: 36 componenten worden naar knoop B gestuurd en 34 componenten worden niet verplaatst. De werklast voor knoop A is $0,68$, die voor knoop B $0,48$. De gemiddelde werklast binnen het balanceringsdomein is $0,58$. De afwijking van het gemiddelde voor knoop A is $0,10$. Deze waarde is niet groter dan de threshold, dus wordt op dat moment de werklastverdeling stopgezet. De afwijking voor knoop B is dan $-0,10$ wat erop duidt dat deze knoop minder werklast heeft dan het gemiddelde binnen het balanceringsdomein.

Wanneer we de werklast laden op knoop B, wordt deze als volgt verdeeld: 22 componenten worden naar knoop A verplaatst, de overige 48 knopen worden niet verplaatst. De werklast voor knoop A is $0,44$, die voor knoop B $0,64$. De gemiddelde werklast binnen het balanceringsdomein is $0,54$. De afwijking van het gemiddelde voor knoop A is $-0,10$, die voor knoop B is $0,06$. Deze laatste waarde is niet groter dan de threshold, dus wordt op dat moment de

		Knoop A		50	Knoop B		75	
threshold		#comp	werklast	afwijking	#comp	werklast	afwijking	gemiddelde
0,1	A	34	0,68	0,10	36	0,48	-0,10	0,58
	B	22	0,44	-0,10	48	0,64	0,06	0,54
0,05	A	31	0,62	0,05	39	0,52	-0,06	0,57
	B	25	0,50	-0,05	45	0,60	0,02	0,55
0,01	A	28	0,56	0,00	42	0,56	-0,02	0,56
	B	28	0,56	0,00	42	0,56	-0,02	0,56

Tabel 5.1: Resultaten experiment 1: invloed van de waarde van threshold op werklastspreiding.

werklastverdeling stop gezet.

We geven in tabel 5.1 de resultaten van dit en de andere experimenten. We zien dat hoe lager we de threshold nemen, hoe beter de spreiding van de werklast gebeurt. De afwijking van de gemiddelde werklast neemt af.

Het is echter niet aan te raden om de threshold te klein te kiezen. Wanneer twee knopen verbonden zijn, en er geen exact gelijke werklastverdeling bereikt kan worden, zal bij een te kleine threshold constant geprobeerd worden om een component te verplaatsen teneinde een betere verdeling te krijgen. De afwijking van het gemiddelde (Φ_L) daalt dan nooit onder de threshold en het systeem komt in een oneindige hysteresislus terecht.

5.3 Bandbreedte

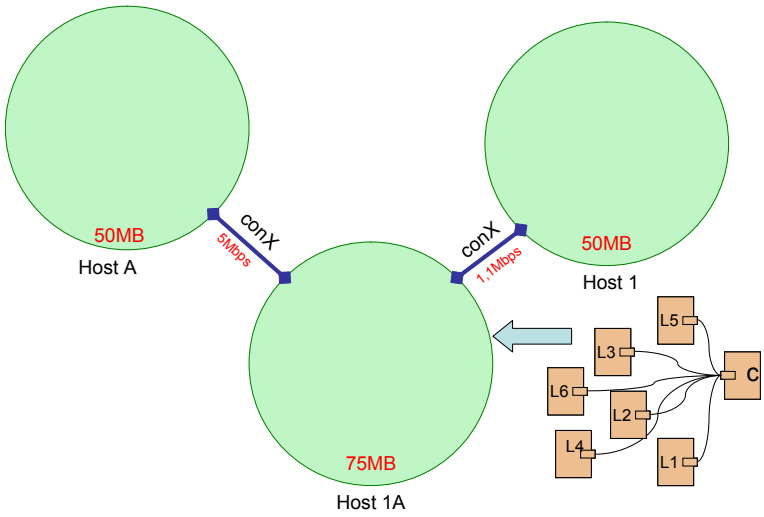
We gaan na hoe de methode rekening houdt met de beschikbare bandbreedte in de beslissing om componenten te verplaatsen naar een bepaalde knoop.

5.3.1 Testmethode

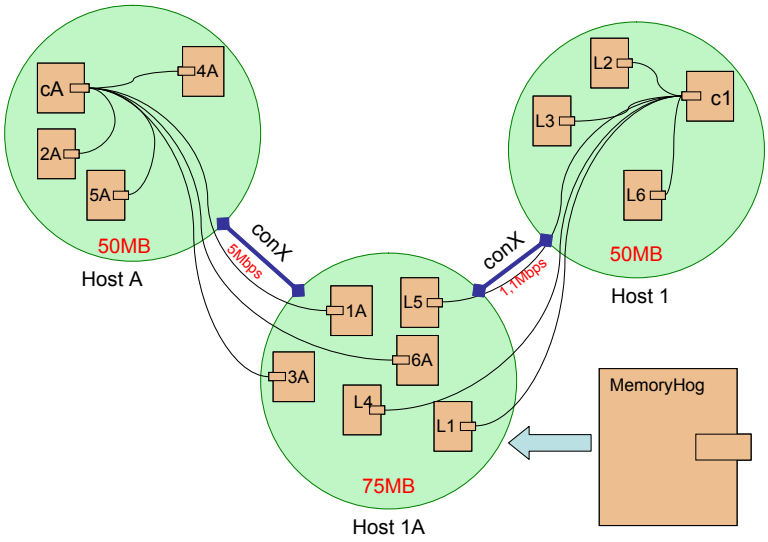
We gebruiken weer een eenvoudige, maar doeltreffende methode om weer te geven wat de beslissing van de methode is bij het verplaatsen van componenten. Dit keer zorgen we ervoor dat het effect van de bandbreedte duidelijk wordt.

We bouwen een netwerk op van 3 knopen. Er zijn twee balanceringsdomeinen (1 en A) die overlappen. De bandbreedte naar het ene domein (A) is groter dan de bandbreedte naar het andere domein (1) (zie figuur 5.5). In een eerste situatie laden we op de overlapknoop (1A) een aantal componenten in. We kijken in welke volgorde ze verspreid worden over de knopen. We zorgen er hierbij voor dat de invloed van het geheugen genegeerd kan worden door componenten met gelijke geheugenvereisten en buren met gelijk aangeboden geheugen te gebruiken.

In een tweede experiment laden we identieke componenten op de eindknopen in, die zich zullen verspreiden over het netwerk (zie figuur 5.6). Vervolgens laden we een zware component in op de tussenliggende knoop, waardoor componenten die zich daar bevinden mogelijk opnieuw toegewezen worden. We gaan na of de componenten teruggestuurd worden vanwaar ze komen, om geen onnodige transferbandbreedte te gebruiken.



Figuur 5.5: Testopstelling voor situatie 1.

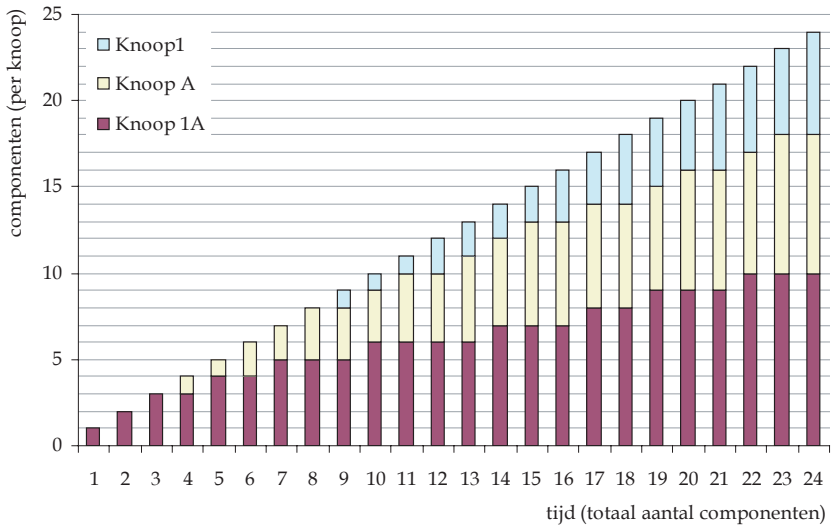


Figuur 5.6: Testopstelling voor situatie 2. Dit is de uitgangssituatie waarvan we vertrekken. We laden dan op knoop 1A een component met grote geheugenvereisten in (MemoryHog), die dwingt dat de migrante componenten weggestuurd worden.

5.3.2 Resultaten

Experiment 1

We observeren in welke volgorde de componenten verspreid worden over de twee beschikbare knopen. In figuur 5.7 tonen we grafisch de resultaten. Op de grafiek zien we in welke volgorde de componenten over het syteem verspreid worden. Aanvankelijk worden er geen componenten verplaatst. De afwijking van de gemiddelde geheugenlast is op dat moment niet



Figuur 5.7: Resultaat van bandbreedte experiment 1.

hoog genoeg. Wanneer dit verschil wel groot genoeg wordt, wordt er een component geselecteerd om verplaatst te worden (tijdstip 4). We zien dat de methode de buur kiest met de meeste beschikbare bandbreedte (knoop A). Nadat er nog een aantal componenten ingeladen zijn, zien we dat ook knoop 1 componenten toegewezen krijgt (tijdstip 9). De beschikbare bandbreedte is op dat moment groter dan die voor knoop A, bovendien is de geheugenlast voor knoop A op dat moment al groter geworden dan die voor knoop 1. De methode werkt zoals we het verwacht hadden. Ze selecteert telkens op een gulzige manier de beste knoop die op dat moment beschikbaar is. We zien duidelijk dat de afweging tussen bandbreedte- en geheugenlastspreiding goed gebeurt.

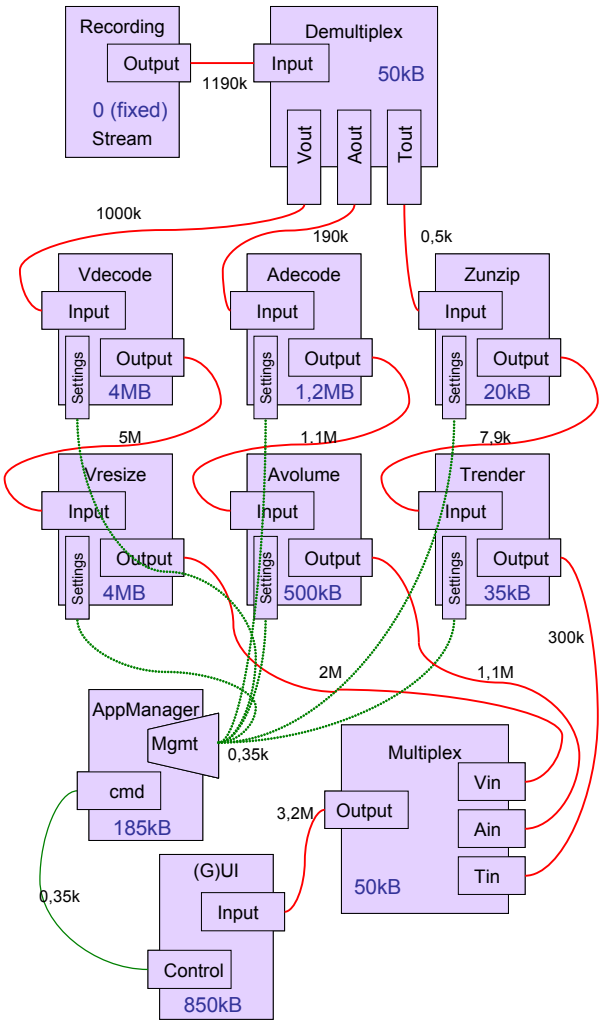
Experiment 2

We observeren wat er gebeurt: de componenten worden stuk voor stuk teruggestuurd vanwaar ze afkomstig zijn. Dit is op zich een minder interessant experiment, maar toch is het fijn om te zien dat de methode de componenten gewoon terugstuurt vanwaar ze afkomstig zijn. Op die manier wordt er geen onnodige transferbandbreedtegebruik geïntroduceerd. Dit bevestigt dat de bandbreedte op een correcte manier in rekening gebracht wordt.

5.4 Integratietest

Om te verifiëren dat onze module goed functioneert, ook in minder triviale situaties, ontwikkelden we een dummyapplicatie met een grotere omvang. Zie figuur 5.8 voor een beschrijving van deze testapplicatie.

We zetten ook een systeem op dat iets complexer is dan enkel maar twee knopen. We gebruiken 3 fysieke machines waar we 5 Draco-instanties op draaien. Op figuur 5.9 tonen we

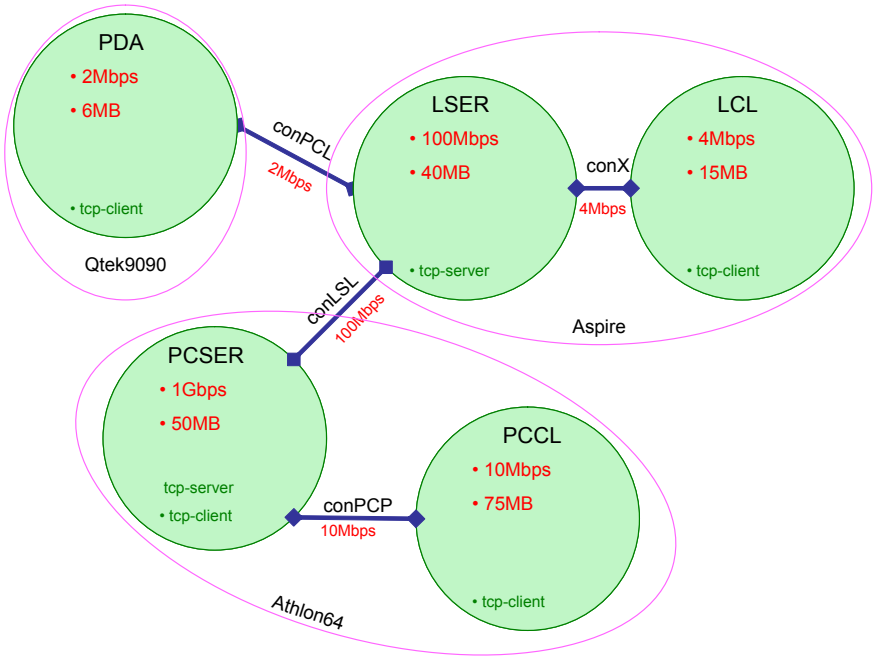


Figuur 5.8: Dummyapplicatie voor de behandeling van een multi-mediastream. De applicatie is opgebouwd uit een tiental componenten. De idee is dat er een gecodeerde stroom gegevens binnenkomt. De verschillende stromen data worden gescheiden door de Demultiplexer. Vervolgens worden de stromen behandeld en gemanipuleerd om uiteindelijk terug samengevoegd te worden. We voorzien ook een component die de instellingen van de manipulatie componenten kan sturen.

de opbouw van het systeem, met de hardwarebeschrijving.

5.4.1 Resultaten

We oberveren hoe de applicatie verspreid wordt in het netwerk als we alle componenten inladen op knoop LCL. De enige verbinding die deze knoop heeft, legt een bandbreedtebeperking op van 4Mpbs. De componenten Vdecode en Vresize kunnen hierdoor niet verplaatst worden. Componenten Adecode en Avolume vinden we terug op knoop PCSER. Component AppManager en Trender op knoop LSER, en Tunzip op knoop PDA. We merken dat alle harde voorwaarden voldaan zijn bij de bereikte componentenuitrol. De werklastspreiding is niet op-



Figuur 5.9: 5 DRACO-instanties worden weergegeven in een complexe configuratie. We geven voor elke knoop weer hoeveel geheugen beschikbaar is en wat de aangeboden bandbreedte is. De bandbreedte tussen de links tussen de knopen is gelijk aan het minimum van de twee buren. Voor de volledigheid geven we aan welke functie de knopen vervullen bij het opzetten van de netwerkverbindingen in de distributiemodule. We duiden aan op welke machines de DRACO-instanties in werkelijkheid draaien.

timaal, omdat de componenten met de zwaarste geheugenvereisten niet verplaatst kunnen worden vanwege de bandbreedtebeperkingen.

Vervolgens observeren we wat er gebeurt wanneer we de applicatie inladen op knoop PCCL. Dit is de knoop met het grootste beschikbare geheugen. We merken dat na het laden van de applicatie enkel componenten Vdecode en Vresize verplaatst zijn. Deze maken de grootste geheugenlast voor de applicatie uit. We vinden ze terug op knoop PCSER. De andere componenten werden niet verplaatst. Ook nu werden geen harde voorwaarden geschonden. We speelden voor dit experiment met de factoren die de componentselectie beïnvloeden, om ervoor te zorgen dat er minder belang gehecht werd aan de kost van bandbreedtegebruik. Op deze manier werden de componenten met het grootste geheugengebruik sneller geselecteerd, ook al gebruiken ze de meeste bandbreedte.

Hoofdstuk 6

Besluit

6.1 Oorspronkelijke doelstellingen en problemen

De doelstellingen waren in het begin van de thesis vrij ruim omschreven. Er was een grote vrijheid welke richting we zouden uitgaan. De oorspronkelijke titel van het thesisvoorstel was *Resource aware computing spaces*. Oorspronkelijk hebben we vrij intensief naar oplossingen gezocht om de configuratie van de verbindingen te automatiseren. We wilden gebruik maken van UPnP [14] om automatisch knopen in de omgeving te vinden en verbindingen op te zetten en informatie uit te wisselen. We hebben hiervoor een initiële oplossing ontwikkeld. Wanneer we deze echter uitprobeerden bleek snel dat de JAVA-implementatie die we gebruikten niet performant genoeg was om gebruikt te kunnen worden op de PDA. Het verbinden met het draadloos netwerk zelf liep met de originele firmware van het toestel[1] zelfs niet altijd van een leien dakje. Al snel werd duidelijk dat we te veel hooi op de vork genomen hadden, door de doelstellingen te ruim te interpreteren. Ondertussen hadden we ons echter met minder relevante, maar toch nuttige zaken bezig gehouden.

We vernauwden voor onszelf de doelstellingen, en besloten de nadruk in deze thesis te leggen op het *computing spaces*-aspect van de titel, en niet zozeer op de *resource awareness*. Deze komt binnen de geboden oplossing niet automatisch. We werken met statische configuratiebestanden en offline metingen voor het bepalen van systeembroneisen.

6.2 Bijgestelde doelstellingen en oplossing

Doelstelling

In deze thesis hebben we een methode ontwikkeld die het mogelijk maakt om een applicatie, opgebouwd uit softwarecomponenten, vlot te gebruiken op een toestel met beperkte systeembronnen in een mobiele omgeving. De methode die we ontwikkeld hebben verdeelt sofware-

[1] We maken gebruik van een Qtek 9090. Dit is een multifunctioneel apparaat met zeer veel mogelijkheden. Het gebruiksgemak van het toestel is echter niet altijd optimaal. Vaak valt de schermverlichting uit. Als je het een tijd links laat liggen, kan het gebeuren dat het volledig naar de oorspronkelijke toestand gereset is. Veel informatie over het toestel kan je vinden op http://wiki.xda-developers.com/?pagename=HTC_Blueangel.

componenten van de applicatie over systeembronnen die beschikbaar zijn in de omgeving. Deze doelstellingen hebben we gehaald.

Oplossing

We hebben het componentensysteem Draco uitgebreid met een werklastbalanceringsmodule, zodat Draco-applicaties op een PDA automatisch verspreid worden via het draadloze netwerk over hardwareknopen in de omgeving. De componentenuitrol van de applicatie wordt dynamisch aan veranderingen van werklast in de omgeving aangepast. De beslissingsmethode houdt hiervoor rekening met de systeembronnen die aangeboden worden door de hardware, en de vereisten die de software aan systeembronnen stelt.

We hebben de methode *Diffusie door zenderinitiatief* toegepast. Dit is een gedistribueerde lokale aanpak. Er wordt gebruik gemaakt van de werklastinformatie van de nabije buren om een surplus aan werklading van zwaar beladen rekenknopen toe te wijzen aan onderbeladen buur-rekenknopen. Deze incrementele, gulzige methode hebben we toegepast op het niveau van softwarecomponenten, welke met hun geheugenvereisten de werklast vormen. We hebben ook rekening gehouden met de bandbreedtevereisten die de communicatie tussen componenten met zich meebrengt. Globaal evenwicht wordt bereikt doordat de werklast van zwaar beladen omgevingen verspreid wordt over de licht beladen omgevingen, dit is mogelijk als we de omgevingen laten overlappen.

6.3 Kritische noot

Het inwerken in de code van Draco heeft enige tijd in beslag genomen. Het zou praktisch zijn als er een document zou bestaan met de best-practises voor het ontwikkelen van Draco-modules. Omdat we werken op een gedistribueerd, parallel systeem moesten we de methode ook wapenen tegen deadlocksituaties en race conditities. Het is niet altijd eenvoudig om dit te debuggen en met alle mogelijkheden rekening te houden. De distributiemodule kan soms lang werken aan het verplaatsen van componenten, en hierdoor in de problemen komen.

Een nadeel van het gebruik van Draco op mobiele toestellen is de vereiste dat er een volledige JAVA-omgeving beschikbaar moet zijn. Voor vele van de huidige mobieledtoestellen is enkel een afgeslankte versie beschikbaar. Applicaties voor deze toestellen kunnen in een programmeertaal geschreven worden waarvoor een compiler bestaat die efficiënte code genereert voor de gebruikte processorarchitectuur. Wanneer we specifiek kijken naar het toestel dat we gebruikt hebben, merken we dat hier *Windows Mobile™ 2003* als besturingssysteem op draait. Microsoft biedt aan ontwikkelaars een uitgebreide en beproefde ontwikkelomgeving (Visual Studio) met handige tools en emulators. Er zou kunnen samengewerkt worden met andere universiteiten die hiervoor een specifieke oplossing ontwikkeld hebben. Goede samenwerking over lange afstand is uiteraard niet evident. *Prism-MW*[2] [9] zou een alternatief voor Draco kunnen zijn, voor gebruik op toestellen met beperkte systeembronnen.

Misschien kan de mogelijkheid onderzocht worden om een dienstengeoriënteerde architectuur [Service-Oriented Architecture, SOA] te gebruiken in deze mobiele omgeving, dit eventueel in combinatie met Draco. Hierbij zou het toestel met beperkte systeembronnen ontlast

[2] http://sunset.usc.edu/~softarch/Prism/middleware.html

worden van berekeningen en enkel een venster zijn op de aangeboden diensten, die elders op een server draaien. Eigenlijk kan het componentenmodel van Draco, en onze methode van het uitbesteden van werk, gezien worden als een dienstengeoriënteerde aanpak. Bepaalde componenten bieden diensten aan andere componenten aan. Het verschil zit in de schaal waarop deze diensten aangeboden worden. Bovendien wordt in SOA gebruik gemaakt van standaarden om de communicatie met een breed gamma van gebruikers mogelijk te maken. Het grote probleem bij deze aanpak is de kost van de bandbreedte, maar we vermoeden dat deze kost in de toekomst zal blijven dalen[3].

Het werk dat we in deze thesis geleverd hebben, heeft echter wel onderzoekswaarde. De herontdekking van deze methode van werklastbalancering en toepassing in de mobiele context is innovatief.

6.4 Toekomstig werk: uitbreidingen en aanpassingen

We hebben aangegeven hoe andere systeembroninformatie geïntegreerd kan worden in de werklastverdelingsmethode. We sommen enkele uitbreidingen op die nuttig zijn om het systeem bruikbaarder te maken:

- **Online metingen**: Wanneer online metingen uitgevoerd kunnen worden, (hiervoor is ondersteuning van de virtuele machine nodig) kan de methode aangepast worden om gebruik te maken van deze informatie. Wanneer bijvoorbeeld een voldoende grote wijziging opgetreden is in de beschikbaarheid van een systeembron, kan de beslissingsprocedure gestart worden[4].
- **Automatische configuratie**: Het opzetten van de netwerkverbindingen, het inladen van de modules en het opzetten van de communicatie tussen deze modules moet momenteel handmatig gebeuren. Om dit probleem op te lossen kan bijvoorbeeld gebruik gemaakt worden van UPnP-discovery [14].
- **Naamgeving**: Er is een probleem wanneer identieke toepassingen gebruikt worden binnen eenzelfde systeem. De namen van de componenten zijn dan gelijk, en wanneer die componenten verplaatst worden, zullen overlappingen kunnen optreden. In het huidige systeem wordt dit niet toegelaten. De interne werking van Draco is fel gebaseerd op de naamgeving van de componenten. Als oplossing kan er bijvoorbeeld een methode ontwikkeld worden die automatisch de componenten op een voor de applicatie transparante manier hernoemt.

[3] iCity project. http://www.i-city.be/icity/

[4] Met behulp van de resultaten van de thesis [Draco in Control] van Tom Distelmans, kan een module gebouwd worden die online metingen uitvoert, gebruik makend van ondersteuning van de virtuele machine. Vervolgens kunnen de gegevens beschikbaar gemaakt worden voor de werklastverdelingsmodule.

Nabeschouwing

Ervaringen door het geleverde werk

We zijn begonnen met het doorspitten van de DRACO-handleiding[5] [16] en hebben enkele eenvoudige applicaties proberen te bouwen om de concepten van DRACO onder de knie te krijgen. Het compileren van de broncode van DRACO was vrij omslachtig. Het importeren van de code in eclipse[6] was eenvoudig, maar de scripts die het compilatieproces automatiseren werden niet meegeleverd met de broncode. Toen we op de hoogte gebracht werden van de gebruikte tool ant [11], en het compilatiescript gegeven werd, kregen we de compilatie eindelijk in gang.

Wanneer we de distributiemodule gecompileerd hadden, konden we beginnen experimenteren met (zwak-)mobiele componenten. Er werd ook al nagedacht over een mogelijke toepassing. Het is niet evident een eenvoudige toepassing te bedenken die weinig bandbreedte gebruikt, maar wel een groot geheugengebruik heeft. We kwamen op het idee een bestaande spellingscontrole-component te gebruiken en deze te koppelen aan een tekstverwerkingsprogrammaatje[7]. (Zie figuur 6.1.)

Al snel werd beslist om niet te proberen een profiler[8] te maken die op DRACO werkt. Het is niet eenvoudig om zonder hulp van de virtuele machine de nodige informatie uit het systeem te krijgen, zonder daarbij het systeem zelf zwaar te belasten. De prestaties van de PDA vormen daarvoor vaak problemen. Bovendien was het niet het doel van deze thesis om hier mee bezig te zijn. We zouden gaan werken met offline metingen en configuratiebestanden om dit probleem te omzeilen.

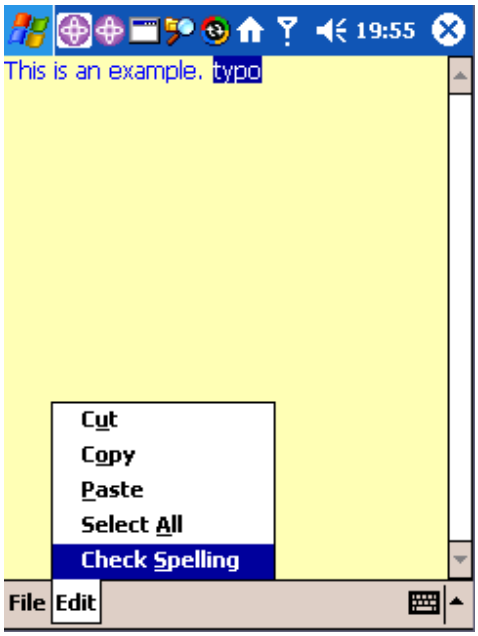
We hebben heel wat literatuur doorworsteld die verband houdt met automatische configuratie en autonomic computing [5]. Ook hebben we bekeken of het mogelijk was gebruik te maken van UPnP [14] om automatisch netwerkverbindingen op te zetten en de juiste modules te laden. Dit bleek niet haalbaar op het toestel dat we voorhanden hadden. De javaimplementaties [7] [6] en benodigde xml-parser [12] die we gebruikten waren te zwaar. Om

[5] Deze handleiding is eerder gericht op applicatieontwikkelaars, en niet zozeer op ontwikkelaars die DRACO zelf willen uitbreiden.

[6] Meer informatie over eclipse op `http://www.eclipse.org`. We moesten werken met een Subversion-repository van het departement computerwetenschappen. Om toegang te krijgen tot dit systeem moest gebruik gemaakt worden van een publieke en private sleutel. Het gebruik van deze authenticatiemethode werkt voor dit versiecontrolesysteem in eclipse vrij slecht. Er moest een tijd gezocht worden naar een werkende configuratie.

[7] Voor het ontwikkelen van de gebruikersinterface van deze tekstverwerker gebruikten we [2].

[8] Een profiler is een programma voor het analyseren van de uitvoersnelheid van een ander programma. Een systeemprofiler voert online metingen uit om de systeembronbelasting in realtime na te gaan.



Figuur 6.1: Spellchecker gebruikersinterface

deze reden beslisten we de configuratie van het netwerk binnen deze thesis manueel uit te voeren.

Oorspronkelijk wilden we de Crumb-module [3][16] uitbreiden om op basis van contracten de herverdelingsstrategie vorm te geven. We zouden gebruik maken van jess [8] als rule-engine. Het gebruik van deze engine is echter niet eenvoudig. De documentatie is beperkt en het was niet dadelijk duidelijk hoe deze aanpak zou bijdragen aan de gekozen herverdelingsstrategie.

We beslisten een eigen module te implementeren die Draco uitbreidt. We implementeerden de methodes om configuratiebestanden te parsen en stelden een interne voorstelling van de beschikbare informatie op. We implementeerden de nodige observerpatronen om de informatie van componenten van de Draco-kern te verkijgen.

De inspiratie om de gekozen methode gebaseerd op diffusie van de werklast door zenderinitiatief kwam tijdens een les van het vak *Algoritmen voor Parallelle Computers* van Dirk Roose. Dat was een echt eurekamoment. We pasten de methode toe in de context van softwarecomponenten in een mobiele omgeving. We implementeerden de algoritmen om de werklastverdeling te berekenen en de beslissingsprocedure kreeg vorm.

Om de ontwikkelde methode te testen maakten we gebruik van een eenvoudige dummy-applicatie die eigenlijk geen functionaliteit verzorgt. We modelleerden de componenten en hun verbindingen echter wel naar iets wat een nuttige applicatie-architectuur zou kunnen zijn. We hebben in het labo een hardwareconfiguratie met enkele pc's en de pda opgezet om te kijken hoe de methode op de aangeboden systeembronnen reageert.

Bibliografie

[1] Ranveer Chandra, Paramvir Bahl, and Pradeep Bahl. Multinet: Connecting to multiple ieee 802.11 networks using a single wireless card. *Proceedings of IEEE Infocom 2004, Hong Kong*, 2004.

[2] IBM OTI Labs Christophe Cornu. http://www.eclipse.org/articles/Article-small-cup-of-swt/pocket-PC.html. *Eclipse.org*, 2003.

[3] Jens Gerlach and Stefan Van Baelen. Run-time evolution and dynamic (re)configuration of components: Model, notation, process and system support. *Empress*, pages 17–21, 2003.

[4] Ningning Hu and Peter Steenkiste. Estimating available bandwidth using packet pair probing, 2002.

[5] IBM. An architecturual blueprint for autonomic computing. *Autonomic Computing, White Paper*, 2005.

[6] SuperBonBon Industries. http://www.sbbi.net/site/upnp. *UPnPLib*.

[7] Satoshi Konno. http://www.cybergarage.org/net/upnp/java. *CyberGarage Cyberlink: Development Package for UPnP Devices*.

[8] Sandia National Laboratories. http://www.jessrules.com.

[9] Sam Malek, Marija Mikic-Rakic, and Nenad Medvidovic. A style-aware architectural middleware for resource-constrained, distributed systems. *IEEE Transactions on Software Engineering*, 31(3):256–272, 2005.

[10] Marija Mikic-Rakic and Nenad Medvidovic. Support for disconnected operation via architectural self-reconfiguration. *First International Conference on Autonomic Computing (ICAC'04)*, pages 114–121, 2004.

[11] Apache ANT Project. http://ant.apache.org. *Ant*.

[12] Apache XML Project. http://xerces.apache.org/xerces-j. *Xerces*.

[13] Wim De Swert. Memory usage in an embedded component system. *Thesis distrinet, KULeuven*.

[14] UPnP. http://www.upnp.org.

[15] Yves Vandewoude and Peter Rigole. Draco: An adaptive run-time environment for components. *Empress*, 2003.

[16] Yves Vandewoude, Peter Rigole, Davy Preuveneers, and Andrew Wils. http://www.cs.kuleuven.ac.be/~yvesv/Draco/Manual.pdf. *Draco Reference Manual, Version 0.2*, 2005.

[17] M. H. Willebeek-LeMair and A. P. Reeves. Strategies for dynamic load balancing on highly parallel computers. *IEEE Trans. Parallel Distrib. Syst.*, 4(9):979–993, 1993.