



AANPASBARE SYSTEEMSOFTWARE MET BEHULP VAN HET DIPS COMPONENTEN-RAAMWERK

Lieven DESMET

Eindwerk aangeboden tot het behalen
van de graad van burgerlijk ingenieur
in de computerwetenschappen

2001–2002

Promotor: Prof. Dr. ir. P. VERBAETEN

Naam en voornaam student: Desmet Lieven

Titel:

Aanpasbare systeemsoftware met behulp van het DiPS componenten-raamwerk

Engelse vertaling:

Adaptive system software with the DiPS component framework

ACM Classificatie: D.2.9, D.2.11

Korte inhoud:

Applicaties worden meer en meer geconfronteerd met sterk variërende opdrachten of een veranderende omgeving. Vooral plotse piekbelastingen brengen bestaande systemen nogal gemakkelijk op de knieën. Er is dus nood aan ondersteunende systeemsoftware, die zich mee kan aanpassen, om dergelijke variaties zo performant mogelijk door te komen.

Het doel van deze thesis is na te gaan in welke mate het mogelijk is om aanpasbare software te creëren in een component-gebaseerd raamwerk als DiPS.

Er wordt een algemeen model voor adaptief beheer in DiPS ontwikkeld. De doelstellingen hierbij zijn een duidelijke scheiding tussen functionaliteit en beheer, modulariteit en herbruikbaarheid van het beheersysteem en het gemakkelijk toevoegen en verwijderen van de beheerextensie aan de bestaande componentkettingen.

Via twee case studies wordt de bruikbaarheid van het model aangetoond. Enerzijds wordt er een adaptief gelijktijdigheidsmodel uitgewerkt, anderzijds een prototype van een adaptief bestandsysteem, beide in DiPS.

*Eindwerk aangeboden tot het behalen van de
graad van burgerlijk ingenieur in de computerwetenschappen*

Promotor: Prof. Dr. ir. P. Verbaeten

Assessoren: E. Truyen
ir. N. Mazur

Begeleider: S. Michiels

Voorwoord

Een thesis is voor mij meer dan het afgeven van mijn tekst op het secretariaat. Het is veeleer een bundeling van mijn prille stappen in de onderzoekswereld, ontzettend veel verrijkende ervaringen, hopen aanmoedigingswoordjes, een snelcursus schrijven zonder (al te veel) fouten en een tikkeltje zelfvoldoening.

Ik zou graag van deze gelegenheid gebruik maken om een aantal mensen van harte te bedanken. Zonder hen zou deze thesis immers nooit geworden zijn wat hij nu is.

In de eerste plaats gaat mijn dank uit naar mijn begeleider, Sam Michiels. Veel ideeën uit deze thesis kregen vorm tijdens onze wekelijkse ontmoetingen. Sam stond altijd klaar om te helpen, als ik hem nodig had. Zoals een goede begeleider stippelde hij samen met mij een plan uit, waarin hij voldoende ruimte liet voor eigen inbreng. Op cruciale momenten porde hij me aan om niet bij de pakken te blijven zitten.

Verder wil ik ook Greet Desmet en Kathleen Vunckx bedanken. Beiden hebben deze tekst grondig doorgenomen om de talrijke spellings- en grammaticafouten er stuk voor stuk uit te halen. De tekst is er een heel stuk leesbaarder door geworden.

Tenslotte zijn er nog een aantal personen die onrechtstreeks hun steentje bijgedragen hebben tot het verwezenlijken van deze thesis. Mijn ouders en mijn vriendin Kathleen waren, gedurende mijn studies en vooral tijdens mijn thesis, een ware schouder om op te steunen. Ze hebben mij de voorbije maanden maar al te vaak moeten missen, omdat ik aan mijn thesis wilde werken.

Veel leesplezier,

Lieven

Inhoudsopgave

1	Inleiding	1
1.1	Context	1
1.1.1	Component-gebaseerde systemen	1
1.1.2	Behoeftte aan aanpasbare software	1
1.2	Doelstellingen	2
1.3	Overzicht	3
2	Literatuurstudie	4
2.1	Het Distrinet Protocol Stack (DiPS) raamwerk	4
2.1.1	Functionele componenten	5
2.1.2	Connectoren	5
2.1.3	Reflection points	6
2.2	Gelijktijdigheidsmodellen	6
2.2.1	De nood aan alternatieve gelijktijdigheidsmodellen	7
2.2.2	Sequentieel model	7
2.2.3	Thread gebaseerd model	8
2.2.4	Bounded thread pool model	8
2.2.5	SEDA: Staged Event-Driven Architecture	9
2.2.6	ActiveConnector model in DiPS	11
2.3	Adaptieve bestandssystemen	12
2.3.1	Een inleidend voorbeeld : Pathfinder	12
2.3.2	Aanpak van een adaptief bestandssysteem	13
2.3.3	Resultaten van een adaptief bestandssysteem	16
2.4	Synthese van de literatuurstudie	16

3	Adaptief, modulair beheer	18
3.1	Doelstellingen	18
3.1.1	Scheiding van functionaliteit en beheer	18
3.1.2	Modulariteit	19
3.1.3	Herbruikbaarheid	19
3.2	De voorbereidingen	19
3.2.1	Beperkingen binnen DiPS	19
3.2.2	DiPS II	20
3.3	Communicatie tussen functionaliteit en beheer	22
3.3.1	Communicatie van meta-laag naar functionele laag	22
3.3.2	Communicatie van functionele laag naar meta-laag	22
3.3.3	Performantiesensoren	24
3.3.4	De monitor	24
3.4	Het beheersysteem als onafhankelijke extensie	25
3.5	Het volledige model	25
3.6	Toetsen van de doelstellingen	26
3.7	Besluit	26
4	Case study : Adaptieve parallelisatie in DiPS	27
4.1	Een statisch parallelisatie model	27
4.1.1	De vereisten	27
4.1.2	Het conceptueel ontwerp	28
4.2	Het adaptief model	30
4.2.1	De sensoren	30
4.2.2	De monitor met inplugbare strategie	31
4.2.3	De meta-monitor met inplugbare strategie	31
4.3	Toetsen van het systeem	32
4.3.1	Testopstelling	33
4.3.2	De gebruikte strategie	33
4.3.3	De resultaten	35
4.4	Conclusies en verder verloop	35

5	Case study : Een adaptief bestandssysteem in DiPS	38
5.1	De voorstelling van een primitief bestandssysteem	38
5.1.1	De vereisten	38
5.1.2	Het conceptueel ontwerp	39
5.1.3	De cache strategieën	41
5.2	Het adaptief model	42
5.2.1	De sensoren	42
5.2.2	De classificatie van het toegangspatroon	43
5.2.3	De monitor met inplugbare strategie	43
5.3	Toetsen van het systeem	44
5.3.1	De testopstelling	44
5.3.2	De gebruikte monitor strategie	44
5.3.3	De resultaten	45
5.4	Conclusie en het verder verloop	48
6	Besluit	49
6.1	Evaluatie van het adaptief model	49
6.2	Testresultaten	50
6.2.1	Dynamische parallelisatie	50
6.2.2	Adaptief bestandssysteem	50
6.3	Verder verloop	50
	Bibliografie	51

Hoofdstuk 1

Inleiding

In dit inleidende hoofdstuk wordt eerst de context van deze thesis beschreven. Vervolgens worden de verschillende doelstellingen uiteengezet. De inleiding sluit af met de verhaallijn doorheen de resterende hoofdstukken.

1.1 Context

Alvorens de doelstellingen van deze thesis op te sommen, wordt in deze sectie de context van de thesis geschetst. Meerbepaald wordt er eerst kort ingezoomd op wat component-gebaseerde systemen zijn en vervolgens wordt de behoefte aan aanpasbare software toegelicht.

1.1.1 Component-gebaseerde systemen

Component-gebaseerde software is opgebouwd uit een aaneenschakeling van herbruikbare softwarecomponenten. Elke component bevat zijn eigen basisfunctionaliteit. De functionaliteit van de software wordt bekomen door de componenten, als bouwblokken, aan elkaar te rijgen tot de gewenste functionaliteit. Het opsplitsen van de functionaliteit in kleine herbruikbare bouwblokken, zorgt voor een eenvoudig software-ontwerp en een snelle ontwikkeltijd.

DiPS (DistriNet Protocol Stack) is het component-gebaseerde raamwerk van de onderzoeksgroep DistriNet. In deze thesis wordt dit raamwerk gebruikt als referentiekader om aanpasbare systeemsoftware in te ontwikkelen.

1.1.2 Behoeftte aan aanpasbare software

Hedendaagse software kan perfect geoptimaliseerd worden om in bepaalde situaties zo performant mogelijk te zijn. In de praktijk echter worden applicaties meer en meer geconfron-

teerd met sterk variërende taken of met een veranderende omgeving tijdens de uitvoering. Er is dus nood aan software die mee kan variëren met zijn taken tijdens de uitvoering ervan.

Allerhande netwerkdiensten worden tegenwoordig geplaagd met een sterk variërende netwerkbelasting. Vooral grote piekbelastingen brengen bestaande systemen nogal gemakkelijk op de knieën. Het is dan ook belangrijk om systemen te ontwikkelen die zich kunnen aanpassen om dergelijke piekmomenten zo goed mogelijk door te komen, zonder te moeten inboeten op de performantie.

1.2 Doelstellingen

Het doel van deze thesis is na te gaan in welke mate het mogelijk is om aanpasbare software te creëren in een component-gebaseerd raamwerk zoals DiPS. De beheermodules, die dynamisch de onderliggende software aanpassen, moeten voldoen aan de hierop volgende kenmerken.

Scheiding tussen functionaliteit en beheer De functionaliteit van de component-gebaseerde software moet duidelijk gescheiden zijn van het beheer ervan. Het moet mogelijk zijn om op een eenvoudige manier bestaande applicaties van een beheermodule te voorzien.

Modulariteit Het beheer moet modulair opgebouwd zijn. Het beheersysteem moet bestaan uit een kleine kern met verschillende uitbreidingsmodules. Naarmate er meer complexiteit vereist is, kunnen dan extra modules ingeschakeld worden. Op die manier kunnen zowel kleinere als veel complexere problemen op dezelfde manier aangepakt worden.

Herbruikbaarheid Het beheersysteem moet zoveel mogelijk opgebouwd zijn uit herbruikbare componenten. Op die manier kan een snelle ontwikkeltijd verkregen worden bij het bouwen van nieuwe beheersystemen.

Inplugbare extensie De beheermodule moet gemakkelijk als extensie kunnen ingeplugd worden op bestaande software, zonder de werking van de software te verstoren. Op dezelfde manier moet het beheersysteem er ook terug afgehaald kunnen worden. Het beheer is een extensie, die tijdelijk of permanent aan werkende software kan worden toegevoegd.

In deze thesis wordt een algemeen, adaptief model nagestreefd met de voorgaande eigenschappen. Een dergelijk generiek model kan dan gemakkelijk geïmplementeerd worden in verschillende situaties waar een adaptief beheer nodig is.

1.3 Overzicht

In de literatuurstudie wordt in eerste instantie ingezoomd op DiPS. Verder worden twee domeinen van naderbij bekeken waar adaptiviteit van de software een belangrijke impact heeft op de performantie: gelijktijdigheidsmodellen en bestandssystemen.

Vanuit de ervaringen uit deze twee domeinen, worden in hoofdstuk 3 de doelstellingen van een aanpasbaar en modulair beheer in DiPS vastgelegd. Op basis van deze doelstellingen wordt stap voor stap een adaptief beheermodel ontworpen.

In de hoofdstukken 4 en 5 wordt dit adaptief model geëvalueerd aan de hand van twee case studies. Enerzijds wordt een dynamisch parallelisatie model ontworpen als alternatief gelijktijdigheidsmodel. Anderzijds wordt een prototype gemaakt van een adaptief bestandssysteem.

In het besluit volgt de algemene evaluatie van het aanpasbaar, modulair beheer in DiPS. De tekortkomingen in het opgebouwde model worden kort geschetst en mogelijke uitbreidingen worden gesuggereerd om verder te bouwen op deze thesis.

Hoofdstuk 2

Literatuurstudie

Adaptiviteit van systeemsoftware en gelijktijdigheid (*concurrency*) wordt een alsmaar belangrijker element in het creëren van hoog performante applicaties. In dit hoofdstuk worden twee concrete domeinen van naderbij bekeken waar adaptiviteit tot een hogere performantie leidt: gelijktijdigheid en bestandssystemen.

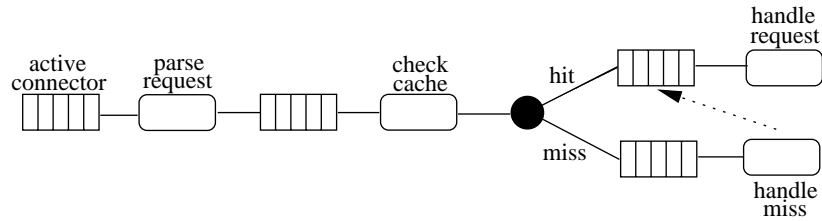
Alvorens met adaptiviteit binnen deze twee domeinen van start te gaan, wordt in het eerste deel van dit hoofdstuk kort ingezoomd op DiPS[1, 2]. DiPS is het component gebaseerde raamwerk waarvoor in deze thesis de mogelijkheden tot adaptieve uitbreiding worden bekeken.

2.1 Het Distrinet Protocol Stack (DiPS) raamwerk

De *Distrinet Protocol Stack*, kortweg DiPS, is het component gebaseerd raamwerk van de onderzoeksgroep Distrinet [1, 2]. DiPS biedt de mogelijkheid om basiscomponenten aan elkaar te rijgen tot de gewenste systeemfunctionaliteit. Een stuk software wordt op die manier herleid tot een ketting van herbruikbare bouwblokken zoals schematisch voorgesteld in figuur 2.1.

Het voordeel van de functionaliteit op te splitsen in kleine herbruikbare basisblokken doet zich vooral voelen in het eenvoudige design en de snelle ontwikkeltijd. Het verkrijgen van nieuwe functionaliteit is meestal niet meer dan het samenrijgen van (reeds bestaande) bouwblokken tot het gewenste geheel.

De basisbouwblokken zelf worden strikt opgedeeld in functionele componenten, connectoren en *reflection points*. In de volgende paragrafen wordt elk van deze categorieën kort besproken.



Figuur 2.1: Voorbeeld van de verschillende bouwblokken in DiPS

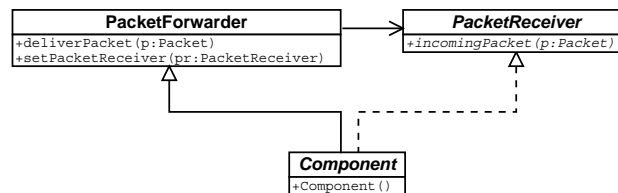
2.1.1 Functionele componenten

Functionele componenten bevatten de pure functionele stukken van de code. Elke component heeft een specifieke basisfunctionaliteit en bevat geen multiprogrammatie. Zodoende kunnen de verschillende componenten gemakkelijk herbruikt worden in een andere context. Functionele componenten hebben allemaal een identieke interface. Ze nemen boodschappen aan in de vorm van een `Packet` object en verwerken deze daarna volgens hun welbepaalde functionaliteit om uiteindelijk de boodschap verder door te geven in de ketting.

Een component in DiPS vervult zowel de functie van een `PacketReceiver`¹ als van een `PacketForwarder`². Ter illustratie hiervan is een beknopt klassenschema voorgesteld in figuur 2.2.

DiPS laat functionele componenten toe om extra informatie toe te voegen aan een boodschap (*Packet*). Deze *meta-informatie* kan daarna door andere componenten anoniem opgevraagd en aangepast worden.

De functionele componenten zijn in figuur 2.1 voorgesteld als witte rechthoekjes.



Figuur 2.2: Beknopt klassenschema van de functionele component

2.1.2 Connectoren

Connectoren schakelen de verschillende functionele componenten aan elkaar. Ze rijgen als het ware de verschillende componenten aaneen als de verschillende parels van een pearsnoer.

¹Een `PacketReceiver` kan een `Packet` ontvangen met de methode `incomingPacket(Packet p)`

²een `PacketForwarder` kan een `Packet` doorgeven aan de volgende `PacketReceiver` via de methode `deliverPacket(Packet p)`

Naast het koppelen zorgen ze tevens voor een vorm van component abstractie. De verschillende componenten hangen niet rechtstreeks aan elkaar, maar via connectoren, en hebben op die manier geen weet van elkaar. Deze componentvervreemding is heel belangrijk met het oog op het dynamisch vervangen en verplaatsen van componenten binnen de ketting, zonder de overige componenten daarvan op de hoogte te hoeven brengen.

Behoudens het koppelen en de componentvervreemding wordt ook het communicatiemodel uitgewerkt binnen de connectoren. Deze duidelijke splitsing tussen functionaliteit (binnen de functionele componenten) en gelijktijdigheid (binnen de connectoren) maakt het mogelijk om gemakkelijk van gelijktijdigheidsmodel te veranderen zonder de functionele componenten te hoeven aan te passen.

De gearceerde rechthoekjes in figuur 2.1 stellen connectoren voor.

2.1.3 Reflection points

Op sommige plaatsen binnen de ketting moet beslist worden welke weg de boodschap op dat moment moet volgen. De beslissingen hiervoor worden bijvoorbeeld genomen op basis van de meta-informatie in de boodschap. De bouwblokken die een dergelijk *kruispunt* binnen de ketting voorzien, worden reflection points genoemd. Ze bevatten verder geen functionaliteit of multiprogrammatie.

In figuur 2.1 is het reflection point aangeduid met een zwarte stip.

2.2 Gelijktijdigheidsmodellen

In paragraaf 2.1.2 werd reeds aangegeven dat we de functionaliteit strak willen loskoppelen van het gelijktijdigheidsmodel. Maar wat wordt nu juist bedoeld met het gelijktijdigheidsmodel en zijn er verschillende alternatieven? De volgende paragrafen geven een eerste antwoord op deze vragen.

Vooraf misschien nog een klein woordje uitleg over multiprogrammatie en threads. In een sterk vereenvoudigde voorstelling kan je een proces opsplitsen in verschillende deelprocessen (*threads*). Deze deelprocessen kunnen dan gelijktijdig uitgevoerd worden.

Als het aantal deelprocessen groter is dan het aantal processoren, dan is er een scheduling algoritme nodig om de verschillende threads elk op hun beurt de processor te laten gebruiken. Deze omwisseling gebeurt op een zodanige wijze dat het voor de eindgebruiker lijkt alsof alle deelprocessen gelijktijdig worden uitgevoerd. Dit fenomeen staat bekend als multiprogrammatie.

In de volgende paragraaf wordt nagegaan waarom er eigenlijk nood is aan alternatieve gelijktijdigheidsmodellen.

2.2.1 De nood aan alternatieve gelijktijdigheidsmodellen

Met de groei van Internet worden aan computersystemen bijkomende eisen opgelegd: de systemen moeten enorm grote gebruikersgroepen snel en betrouwbaar bedienen. De websites van Microsoft moeten bijvoorbeeld meer dan 300 miljoen aanvragen per dag kunnen verwerken. Bovendien worden deze diensten gekenmerkt door grote variaties in belasting. Het zijn vooral piekbelastingen waarop systemen gepast moeten kunnen reageren.

Het *SlashDot*-effect is een voorbeeld van een dergelijke piekbelasting. *SlashDot*³ is een heel druk bezochte portalsite. Wanneer op deze website een nieuw bericht wordt opgenomen met een referentie naar een externe site, dan krijgt deze externe site plots een hele hoop extra bezoekers te verwerken. Vele sites zijn al aan een dergelijke piekbelasting bezweken. Er zijn twee mogelijke aanpakken om deze piekbelastingen beter op te vangen: ofwel wordt de volledige computerinfrastructuur via replicatie overgedimensioneerd, ofwel worden robuuste infrastructures gebouwd, die voorbereid zijn om een dergelijke variërende belasting aan te kunnen.

Overdimensionering via replicatie lijkt onhaalbaar. De extreme variaties in belasting leidt tot een extreem duur computerpark dat slechts in zeldzame piekmomenten zal ingeschakeld worden.

Het aanpassen van het gelijktijdigheidsmodel is een voorbeeld van de tweede aanpak. Door een alternatief model te gebruiken, wordt geprobeerd goed geconditioneerde, scaleerbare Internet diensten[3, 4] te verwezenlijken. Met goed geconditioneerd wordt bedoeld dat de componentenketting zich gedraagt als een lange pijp, waarbij een constante, maximale doorstroom bereikt kan worden. Als het systeem niet goed geconditioneerd is, dan gedraagt het systeem zich eerder als een aaneenschakeling van dikke en dunne buizen. De maximale doorstroom doorheen de volledige pijp is dan gelijk aan de doorstroom doorheen de smalste buis. In dit geval is de smalste buis dus de flessenhals (*bottleneck*) van het hele systeem.

In de volgende paragrafen worden enkele gelijktijdigheidsmodellen bekeken: het sequentieel model, het thread gebaseerd model, het bounded thread pool model en de *SEDA* architectuur. Als laatste komt een eerste poging tot alternatief gelijktijdigheidsmodel in DiPS aan bod: de *ActiveConnector*.

2.2.2 Sequentieel model

In het meest simpele model wachten de boodschappen vooraan de ketting in een wachtrij alvorens verwerkt te worden. Eén enkele thread begeleidt hen, boodschap na boodschap, door de volledige ketting. Dus, nadat *boodschap_i* de verschillende componenten in de ketting doorlopen heeft, behandelt de thread *boodschap_{i+1}*.

Het systeem valt te vergelijken met een aantal wachtende mensen bij de oversteek van een rivier. Een bootje pendelt heen en terug om persoon per persoon naar de overkant te brengen.

³<http://www.slashdot.org>

Boodschappen moeten niet enkel de tijd in rekening brengen om de oversteek te maken, maar tevens de tijd die de boodschappen voor hen in de wachtrij nodig hebben om de oversteek te maken. Vooral als sommige componenten moeten wachten op geblokkeerde bronnen (*locked resources*), kan dat extra vertragingen introduceren. Tijdens die momenten wacht het volledige systeem op het vrijkomen van die bron en wordt er niets anders nuttigs gedaan. De uitdaging is dan ook om alternatieve gelijktijdigheidsmodellen te vinden die een hogere efficiëntie bereiken bij het gebruik van de beschikbare bronnen.

2.2.3 Thread gebaseerd model

Een alternatieve aanpak bestaat erin om voor elke boodschap aan het begin van de ketting een nieuwe thread aan te maken. Deze thread zal dan de boodschap doorheen de ketting begeleiden.

Dit model zorgt ervoor dat elke boodschap direct verwerkt wordt en niet eerst hoeft te wachten op de andere boodschappen. Thread scheduling zorgt dan voor een eerlijke verdeling van de processortijd over de verschillende threads (en dus ook boodschappen). Boodschappen hoeven ook niet langer vertraging op te lopen door andere boodschappen die wachten op een geblokkeerde resource.

Er zijn echter ook nadelen verbonden aan dit model. Zo zijn er aanzienlijke opstartkosten voor het aanmaken van een thread. Er is extra processortijd en geheugenplaats voor nodig. Naast de extra opstartkosten is er ook het probleem van sterke performantiedaling bij een hoog aantal threads. Bij teveel threads wordt er een enorme scheduling overhead geïntroduceerd. Dit heeft nefaste gevolgen voor de performantie. De drempelwaarde van deze daling ligt redelijk hoog (enkele honderden threads volgens [3, 4]), maar voor Internet diensten nog steeds veel te laag om werkbaar te zijn. Het aantal gelijktijdige aanvragen dat deze diensten tegenwoordig te verwerken krijgen, ligt immers stukken hoger. Zelfs met een netwerk van servers kunnen grote pieken niet opgevangen worden, tenzij door immense overdimensionering om op elke machine het aantal threads laag genoeg te houden.

2.2.4 Bounded thread pool model

Het *bounded thread pool* model poogt de nadelen van het vorige model weg te werken, met name de extra opstartkosten van de threads en de sterke performantiedaling.

In plaats van telkens een thread op te starten bij het aankomen van een boodschap, wordt een pool van threads opgestart bij de start van de applicatie. Deze threads worden dan telkens opnieuw gebruikt om boodschappen te begeleiden doorheen de ketting. Dus in plaats van op het einde van de ketting een thread verloren te laten gaan, wordt de thread teruggeplaatst in de pool van beschikbare threads. Bij aankomst van een boodschap aan het begin van de ketting, wordt nagekeken of er een beschikbare thread in de pool aanwezig is. Indien dit het geval is, wordt deze thread direct gebruikt voor de begeleiding; indien

niet, dan wordt de boodschap in een wachtrij geplaatst of genegeerd. Deze aanpak wordt traditioneel gebruikt bij de Apache webserver⁴.

Wanneer tijdens piekmomenten alle threads geblokkeerd zijn, of bezig met de verwerking van andere boodschappen, dan wordt de nieuwe boodschap in de wachtrij geplaatst. Wanneer de boodschappen zich op die manier opstapelen, worden er lange wachttijden geïntroduceerd, analoog aan het sequentieel model. Het was precies de verdienste van het thread model om hiervoor een oplossing te bieden. Dus moet er een middel gevonden worden om deze wachttijden te verkleinen.

Een mogelijk probleem bij dit model is de **unfairness**: eenvoudige aanvragen moeten somsodeloos wachten op moeilijkere aanvragen. Er wordt immers geen onderscheid gemaakt welke aanvragen eerst behandeld worden.

2.2.5 SEDA: Staged Event-Driven Architecture

SEDA is een architectuur die ondersteuning biedt voor het bouwen van goed geconditioneerde, scaleerbare Internet diensten[3, 4]. In de volgende paragrafen worden de verschillende doelstellingen en de gebruikte aanpak van SEDA uit de doeken gedaan.

Doelstellingen

SEDA⁵ heeft 3 doelstellingen. Als eerste poogt het een simpel raamwerk voor goed geconditioneerde, scaleerbare Internet diensten aan te bieden aan ontwikkelaars. Met behulp van de SEDA-architectuur kunnen ontwikkelaars hun diensten implementeren zonder rekening te moeten houden met het gelijktijdigheidsmodel. SEDA zal dan voor deze diensten de gelijktijdigheid regelen.

De tweede doelstelling van SEDA is ervoor te zorgen dat performantiedalingen tengevolge van piekaanvragen en threads vermeden worden. Het gaat hier zowel over het performantieverlies in het thread model als de lange wachttijden in het bounded thread pool model. Tenslotte wil SEDA op een gepaste manier reageren op de veranderende omgeving en het wisselende karakter van de aanvragen. Op basis daarvan zal SEDA adaptief de systeembronnen verdelen over het systeem.

Event Driven

De *event-driven aanpak* bestaat erin een klein aantal threads te gebruiken (meestal één per processor). De applicatie doet beroep op events om naar de volgende stap in de functionele ketting te springen. De functionele stukken code bestaan uit allemaal kleine toestandsmachines die hun stukje functionaliteit implementeren. De events maken dus de overgangen tussen de verschillende toestandsmachines mogelijk. In dit model houdt elke

⁴<http://www.apache.org>

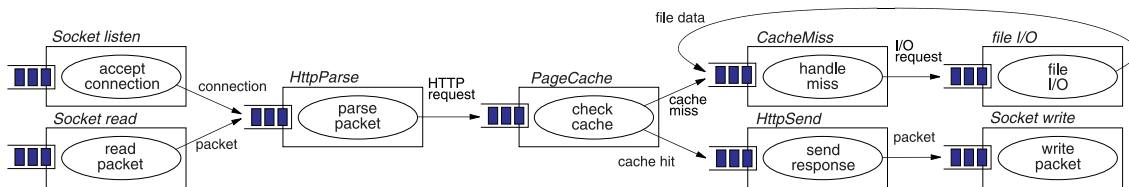
⁵SEDA: Staged Event-Driven Architecture

boodschap zijn eigen status bij, en wordt dit niet gedaan door de thread context. Het voordeel hiervan is dat er geen directe koppeling meer bestaat tussen een boodschap en een thread. Het is nu perfect mogelijk dat verschillende threads één boodschap begeleiden, elk op een ander stukje van de ketting.

Zoals blijkt uit de bestudering van de Jaws webserver[5] is het event-driven model heel robuust onder belasting, met slechts een kleine performantiedaling. Een nadeel echter in dit model is dat er uitgegaan wordt van niet blokkerende in- en uitvoer. Ook moet de applicatie beslissen welke events eerst verwerkt worden.

SEDA architectuur

Bij SEDA wordt de componentenketting in verschillende stukken geknipt. Elk deel bevat een groep functionele componenten. Aan het begin van elk deel wordt een wachtrij geplaatst. Elk stukje ketting, met zijn wachtrij, wordt in SEDA een **stage** genoemd. Het volledige systeem bestaat dan eigenlijk uit een aaneenschakeling van de verschillende stages. Elke stage krijgt zijn **scheduler(s)**: dit zijn threads die ervoor zorgen dat de wachtrijen verwerkt worden. Figuur 2.3 toont zo'n aaneenschakeling van verschillende stages bij de SEDA HTTP-server.



Figuur 2.3: Een aaneenschakeling van stages bij de SEDA HTTP-server

De voordelen van het opsplitsen in verschillende stages zijn een kortere kettinglengte en een beter beheer van de systeembronnen.

Enerzijds wordt er een fysische grens gecreëerd waar een thread niet over kan. De thread kan enkel een boodschap of event begeleiden tot aan de volgende wachtrij. Op die manier is tevens de kettinglengte die een thread moet lopen beperkt, en kan de thread sneller terugkeren naar de volgende boodschap of event.

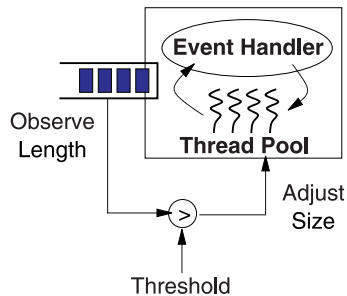
Anderzijds is er, door de opsplitsing in verschillende stages, een beter bronnenbeheer mogelijk. Daar waar het in een lange ketting heel moeilijk is de knelpunten terug te vinden en te verhelpen, is het bij het staged model bijvoorbeeld mogelijk om per stage de nodige systeembronnen ter beschikking te stellen. Heel rudimentair gesteld is het mogelijk om stages, die meer processor tijd nodig hebben dan andere, meer schedulers toe te wijzen.

Hoe dit dynamisch beheer van systeembronnen precies in elkaar steekt wordt in de volgende paragraaf uit de doeken gedaan.

Het dynamische beheer van de systeembronnen

Zoals reeds kort aangehaald in de vorige paragraaf, wil men de bronnen verdelen over de verschillende stages. Als het kan, zou dat ook best nog eens dynamisch gebeuren om te kunnen reageren op veranderingen in de belasting of in de omgeving.

Het bronnenbeheer gebeurt op twee niveaus. Op het eerste niveau worden de bronnen per stage beheerd. Als de stage bijvoorbeeld meer processor tijd nodig heeft om zijn wachtrij te verwerken, wordt een extra scheduler toegevoegd aan de stage. Omgekeerd, bij een overschot aan schedulers (als bijvoorbeeld een scheduler geruime tijd geen werk gehad heeft) wordt een scheduler weggehaald van de stage. De parameters, die hierbij gebruikt worden, zijn een drempelwaarde in wachtrijlengte en de opgeleverde doorstroom van de stage. In figuur 2.4 wordt de terugkoppeling tussen wachtrijlengte en de thread pool gevisualiseerd.



Figuur 2.4: Het dynamische beheer van de systeembronnen in SEDA

Op het tweede niveau worden de resources verdeeld tussen de stages. Het heeft immers geen zin om schedulers bij te produceren in elke stage als we al boven een bepaalde drempelwaarde van aantal schedulers zijn. Dit kan enkel negatieve invloed hebben op de performantie. Er moet rekening gehouden worden met de status van het systeem (zoals beschikbare processor tijd, beschikbaar geheugen, ...).

Over de manier waarop het bronnenbeheer precies wordt aangepakt wordt heel weinig informatie verstrekt in SEDA publicaties.

2.2.6 ActiveConnector model in DiPS

DiPS bevat reeds een eerste poging tot aanpassing van het gelijktijdigheidsmodel. De `ActiveConnector` splitst, in analogie met de stages van SEDA, de componentenketting op in twee delen. Tussen de twee delen wordt een wachtrij geplaatst. Een scheduler zorgt voor het verwerken van de boodschappen in de wachtrij.

Om het geheel generiek te maken, kunnen verschillende soorten `PacketSchedulers` inplugd

in de `ActiveConnector` en kan de gepaste `OverflowStrategy` gebruikt worden. Deze laatste beslist, wat er gebeurt, als de wachtrij volzit (bijvoorbeeld nieuwe pakketjes weggooien, de wachtrij uitbreiden, de minst interessante boodschappen uit de wachtrij weggooien en zo plaatsmaken voor de nieuwe, ...).

Mits wat kleine aanpassingen kan de `ActiveConnector` gebruik maken van meerdere schedulers. Op die manier zou een eerste statische belastingsverdeling geïmplementeerd kunnen worden.

2.3 Adaptieve bestandssystemen

Parallele applicaties hebben soms complexe en onregelmatige toegangspatronen. De huidige bestandssystemen zijn niet goed geoptimaliseerd voor een dergelijke toegang. Ze kunnen slechts geoptimaliseerd worden voor één of ander toegangspatroon, maar kunnen een variatie in toegangspatronen niet de baas. Het is dan ook de bedoeling een bestandssysteem te gebruiken dat wel op deze variaties kan inspelen.

Heel simpel uitgedrukt: adaptieve bestandssystemen zijn bestandssystemen die zich aanpassen aan de omgeving. Afhankelijk van de omgevingsfactoren en de verschillende aanvragen zal het bestandssysteem op een zodanige manier reageren dat het toch een optimale performantie kan bereiken.

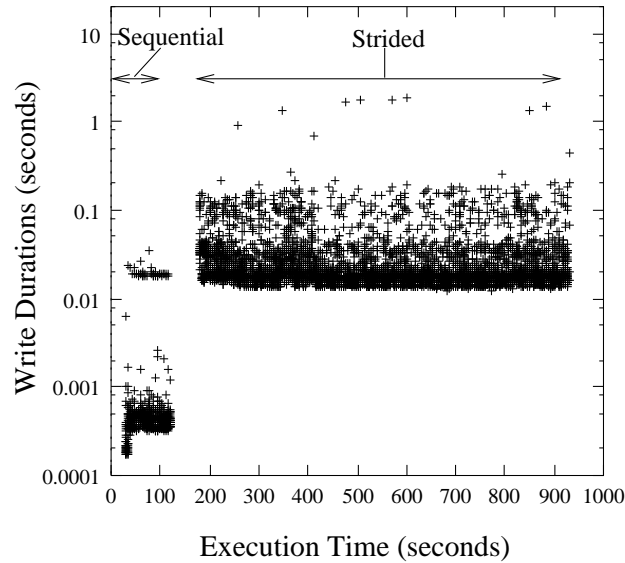
In de volgende paragraaf wordt met een voorbeeldje aangeduid waar de huidige bestandssystemen tekortkomen. Daarna wordt in drie stappen uitgelegd hoe het realiseren van een adaptief bestandssysteem wordt aangepakt. Tot slot wordt het voorbeeldje hernomen, maar dan met een adaptief bestandssysteem.

2.3.1 Een inleidend voorbeeld : Pathfinder

Het Pathfinder project verwerkt bestaande data om daaruit globale klimaatsveranderingen te kunnen bestuderen[6, 7]. Op het einde van de verwerking worden de resultaten weggeschreven. Dit laatste gebeurt in twee fasen. In de eerste fase wordt een `write-only`, sequentieel patroon gebruikt met heel lange toegangen; in de tweede fase een `write-only`, `strided` toegangspatroon met grote aanvragen. Met `strided` wordt een verspreide toegang bedoeld, in tegenstelling tot de sequentiële toegang.

In het statische geval wordt gekozen voor een MRU (=Most Recently Used) cache strategie. In deze strategie worden de meest recent gebruikte datablokken uit de cache vervangen door nieuwe. Zoals blijkt uit figuur 2.5 is deze strategie een goede keuze voor de eerste fase. In de tweede fase echter komen er heel veel aanvragen die zich niet in de cache bevinden. Daardoor kunnen de aangepaste blokken niet in aansluitende, grotere blokken worden weggeschreven naar de data-drager, met langere schrijftijden tot gevolg.

Een adaptieve reactie van het bestandssysteem zou kunnen zijn dat vanaf dat het systeem detecteert dat het zich in de tweede fase bevindt (rond 300 ms) het zijn cache strategie aanpast om deze aanvragen vlotter te kunnen verwerken, bijvoorbeeld door een grotere



Figuur 2.5: Pathfinder: de statische versie

cache bij te houden. Op die manier kan de data die naar het bestandssysteem geschreven moet worden toch in grotere blokken vanuit de cache worden weggeschreven.

2.3.2 Aanpak van een adaptief bestandssysteem

In deze sectie wordt nagegaan hoe een adaptief bestandssysteem wordt aangepakt. De implementatie in het PPFS⁶ en de bijhorende studie[7] dient hierbij als referentie.

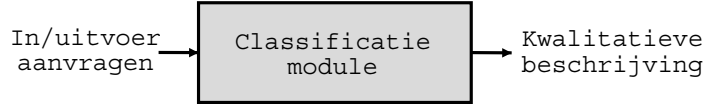
De aanpak kan in grote lijnen opgedeeld worden in drie stukken. In een eerste deel bepaalt het systeem een kwalitatieve beschrijving van het overheersende toegangspatroon van alle aanvragen die binnenkomen in het systeem. In een tweede deel wordt op basis van deze beschrijving een gepaste strategie gekozen voor het bestandssysteem om vervolgens in het laatste deel de parameters van deze strategie verder bij te sturen met performantiesensoren. In de volgende paragrafen wordt elk van deze stappen van naderbij bekeken.

Classificatie van toegangspatronen

De classificatie heeft als doel het overheersende toegangspatroon te herkennen in de stroom van aanvragen die het systeem bereiken. In het meest simpele geval kan de eindgebruiker of de applicatie hints geven aan het bestandssysteem wat het huidige toegangspatroon is. Op basis hiervan kan het systeem dan de nodige stappen ondernemen.

⁶Het Portable Parallell File System (PPFS) is een bibliotheek die verschillende bestandssysteem strategieën kan uittesten.[8]

Het is echter interessanter als het systeem op zich kan bestaan en het zelf het toegangspatroon kan herkennen zonder hulp van buitenaf. Hiervoor hebben we een classificatiesysteem nodig zoals schematisch voorgesteld in figuur 2.6.



Figuur 2.6: Schematische voorstelling van het classificatiesysteem

De gebruikte techniek om patronen te classificeren en te herkennen is het trainen van een neuraal netwerk. Tabel 2.1 toont de herkende eigenschappen van het getrainde neuraal netwerk.

Category	Category eigenschappen			
Read/Write	Read Only	Write Only	Read-Update-Write	Read/Write Nonupdate
Sequentialiteit	Sequentieel	1D-strided	2D-strided	variabel strided
Grootte aanvragen	uniform		variabel	

Tabel 2.1: Herkende eigenschappen van het neuraal netwerk

Lokale en globale classificatie

Bij een parallelle applicatie kunnen toegangspatronen op twee niveaus bestudeerd worden: enerzijds lokaal per thread en anderzijds globaal over de volledige applicatie. Een parallel programma kan bijvoorbeeld over de verschillende threads heen een bestand inlezen op een zodanige manier dat elke thread het bestand **strided** inleest. Wanneer we echter vanop afstand de verschillende threads bekijken, lijkt hun toegangspatroon toch puur sequentieel te zijn, verspreid over de verschillende threads.

Selectie van de strategie

Eenmaal het toegangspatroon bepaald is, moet het systeem op basis daarvan de juiste strategie kiezen voor het bestandssysteem. Als bijvoorbeeld een sequentieel leespatroon herkend wordt door de classificatiemodule, dan kan er een agressieve **prefetch** worden opgestart. Hierbij worden constant een aantal blokken in de cache ingeladen, zonder dat deze expliciet zijn opgevraagd. Op deze manier probeert het systeem te anticiperen op aanvragen die binnenkort zouden kunnen volgen.

Bij een sequentieel schrijfpatroon daarentegen kan bijvoorbeeld een **delayed write** strategie verkozen worden. Hierbij worden aangepaste blokken in de cache niet onmiddellijk weggeschreven naar de data-drager. In plaats daarvan worden op geregelde tijdstippen de

aangepaste blokken uit de cache samen weggeschreven naar de data-drager. Het is immers voordeliger om in één keer een aantal blokken weg te schrijven, dan een aantal keer één blok, omwille van de grote opstarttijd van de data-drager bij het wegschrijven.

De beschrijvingen van de classificatiemodule zijn compleet platform-onafhankelijk. De keuzes van de juiste strategie bij elke classificatie daarentegen zijn gedeeltelijk platformafhankelijk. Voor elk platform moeten drempelwaarden experimenteel bepaald worden voor onder andere de cache-grootte, het te groot of te klein zijn van de aanvragen, ...

De gekozen strategieën moeten op hun beurt generiek genoeg zijn. Ze moeten immers nog verder aan de aanvragen, de omgeving en de performantie van het systeem kunnen worden aangepast. In figuur 2.7 is een heel eenvoudig voorbeeld van een selectie algoritme weergegeven.

```
if (sequential) {
    if(write only) {
        enable caching
        use MRU replacement policy
    } else if (read only && average request size > LARGE_REQUEST) {
        disable caching
    } else {
        enable caching
        use LRU replacement policy
    }
}
if (variably strided || 1-D strided || 2-D strided {
    if (regular request sizes) {
        if (average request size > SMALL_REQUEST) {
            disable caching
        } else {
            enable caching
            increase cache size to MAX_CACHE_SIZE
            use LRU replacement policy
        }
    } else {
        enable caching
        use LRU replacement policy
    }
}
```

Figuur 2.7: Voorbeeld van een eenvoudig selectie algoritme

Verdere bijsturing via performantiesensoren

Het uiteindelijk succes van een strategie wordt bepaald door de performantie die gehaald wordt. Vandaar dat performantiesensoren gebruikt worden om een terugkoppeling tussen performantie en strategie te voorzien.

In eerste instantie worden de sensorgegevens gebruikt om betere parameters te gebruiken voor de gekozen strategie. In tweede instantie wordt via de sensoren de tijd afgeschat die het systeem nodig heeft na een strategie-aanpassing om terug in een stabiele toestand terecht te komen.

Bij wijze van voorbeeld staat een mogelijke koppeling tussen sensorwaarden en strategie-aanpassingen beschreven in tabel 2.2.

Sensor waarden	Strategie-aanpassingen
(<code>poor_read_service_times</code>) & (<code>many_read_requests</code>) & (<code>managable_byte_throughput</code>) & (NOT <code>high_hit_ratio</code>)	Increase Cache Size Increase Prefetch Amount

Tabel 2.2: Koppeling tussen sensorwaarden en strategie-aanpassingen

2.3.3 Resultaten van een adaptief bestandssysteem

Eenmaal de verschillende modules voor het classificeren, het selecteren van de strategie en de bijsturing klaar zijn en in elkaar gepast worden, kan het voorbeeldje uit paragraaf 2.3.1 herhaald worden in een adaptief bestandssysteem.

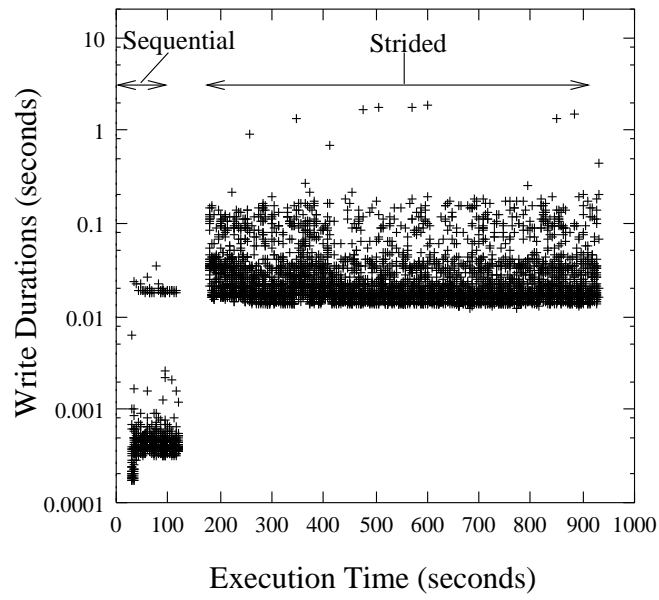
Dezelfde applicatie Pathfinder uit het inleidend voorbeeld wordt losgelaten op het adaptief bestandssysteem. De resultaten van zowel het statische als het dynamische bestandssysteem zijn terug te vinden in figuur 2.8.

In het dynamische geval heeft de classificatiemodule het nieuwe toegangspatroon herkend en heeft het systeem daarop gereageerd door de cache gevoelig te vergroten. Door de grotere cache kunnen de verspreide aanvragen toch sequentieel in grote blokken weggeschreven worden dankzij een `delayed write` strategie.

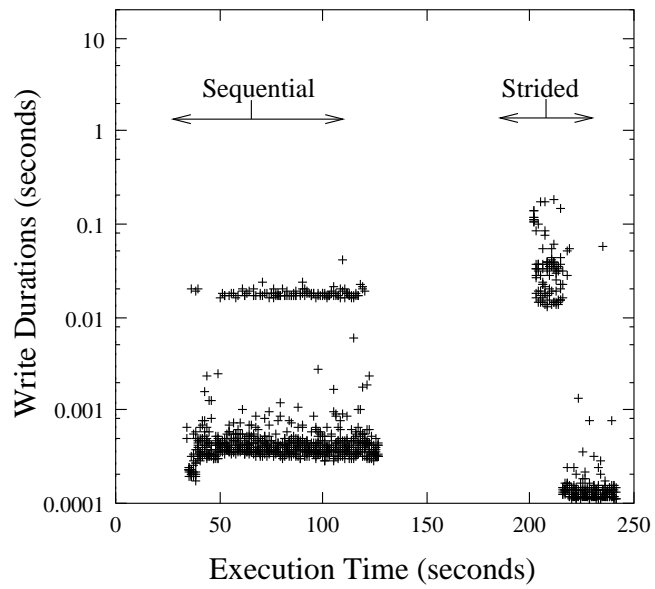
2.4 Synthese van de literatuurstudie

In dit hoofdstuk werd eerst het DiPS raamwerk kort besproken. Daarna kwamen twee domeinen aan bod waar het adaptieve karakter invloed heeft op de performantie: gelijktijdigheid en bestandssystemen.

Met deze twee voorbeelden in het achterhoofd wordt in het volgend hoofdstuk een generiek model voor adaptiviteit binnen DiPS ontwikkeld. Daarna volgen twee *case studies* om dit model te toetsen: een adaptief gelijktijdigheidsmodel en een adaptief bestandssysteem, beiden in DiPS.



(a) Statische versie



(b) Dynamische versie

Figuur 2.8: Pathfinder resultaten

Hoofdstuk 3

Adaptief, modulair beheer

In het vorige hoofdstuk werd het nut aangetoond van applicaties die zich aanpassen aan de omgeving en de opdrachten die ze te verwerken krijgen. In dit hoofdstuk wordt nagegaan of een dergelijk adaptief gedrag in DiPS kan worden gerealiseerd.

In een eerste stap worden de vereisten vastgelegd waaraan een adaptief en modulair beheer moet voldoen. Vervolgens wordt er stap voor stap een dergelijk beheersysteem ontworpen in DiPS. Uiteindelijk wordt in het laatste deel van dit hoofdstuk nagegaan in welke mate het ontwerp voldoet aan de vooropgestelde doelstellingen.

3.1 Doelstellingen

Alvorens te starten met het ontwerp van een adaptief en modulair beheer, worden in deze sectie een aantal vereisten gedefinieerd: de scheiding van functionaliteit en beheer, de modulariteit en de herbruikbaarheid.

Elk van deze doelstellingen wordt in de volgende paragrafen duidelijker toegelicht.

3.1.1 Scheiding van functionaliteit en beheer

Een eerste doelstelling is een duidelijke scheiding van de functionaliteit en het beheer van het systeem. Het moet mogelijk zijn de functionaliteit in te schakelen zonder enige vorm van beheerfunctie. Het functionele deel moet op zich kunnen bestaan en het stukje beheer moet autonoom kunnen ingeplugd en terug weggehaald worden.

Een loskoppeling van functionaliteit en beheer impliceert dat er verschillende variaties van beheer gebruikt kunnen worden met hetzelfde stukje functionele software.

Tevens zorgt net die loskoppeling er voor dat er voor bestaande functionele code gemakkelijk een beheer kan worden ontworpen, zonder de code van de componenten te moeten aanpassen. De beheerfunctionaliteit is een aparte component die als extensie kan samenwerken met de basisfunctionaliteit van de software.

3.1.2 Modulariteit

Een tweede doelstelling is de modulaire opbouw van het beheersysteem. Er wordt gestart met een kleine kern, en naarmate er meer complexiteit nodig is, kan deze kern uitgebreid worden met extra modules. Het beheersysteem kan op die manier vlot de beheernoden volgen van uiteenlopende toepassingen.

3.1.3 Herbruikbaarheid

Er wordt een zo groot mogelijk hergebruik nagestreefd bij de componenten die deel uitmaken van de beheerfunctie. Op die manier wordt een snelle ontwikkeltijd van nieuwe beheersystemen verkregen. Het bouwen van een nieuw beheersysteem is dan, in extremis, niet meer dan het samennemen van enkele herbruikbare componenten, die op de juiste manier geïnitieerd en met elkaar verbonden worden.

3.2 De voorbereidingen

In deze sectie wordt ingezoomd op DiPS. Er wordt nagegaan wat de tekortkomingen en beperkingen zijn binnen DiPS om een beheersysteem op poten te zetten. Er worden enkele aanpassingen en uitbreidingen gesuggereerd die leiden tot een vernieuwde versie van DiPS, met name DiPS II.

3.2.1 Beperkingen binnen DiPS

In sectie 2.1 werd reeds een korte introductie tot DiPS gegeven. In deze paragraaf wordt er iets dieper ingegaan op de structuur en de functie van componenten en connectoren.

In DiPS wordt de gewenste functionaliteit bereikt door componenten via connectoren aan elkaar te schakelen. Hierbij zit het functionele deel van de ketting in de componenten, en dienen de connectoren voor de aaneenschakeling van deze basisfunctionaliteiten. Niet-functionele aspecten (zoals een gelijktijdigheids- of communicatiemodel) worden verwerkt in de connectoren.

Componenten

Figuur 2.2 toont een beknopt klassenschema van een `Component`. Een `Component` is een uitbreiding van een `PacketForwarder` en implementeert de interface `PacketReceiver`.

Een *component* bevat zowel het functioneel deel als de interfaces naar andere componenten (`PacketReceiver` en `PacketForwarder`). Er is echter geen duidelijke loskoppeling tussen deze interfaces en de functionaliteit. Hierdoor kan een component niet veranderen van

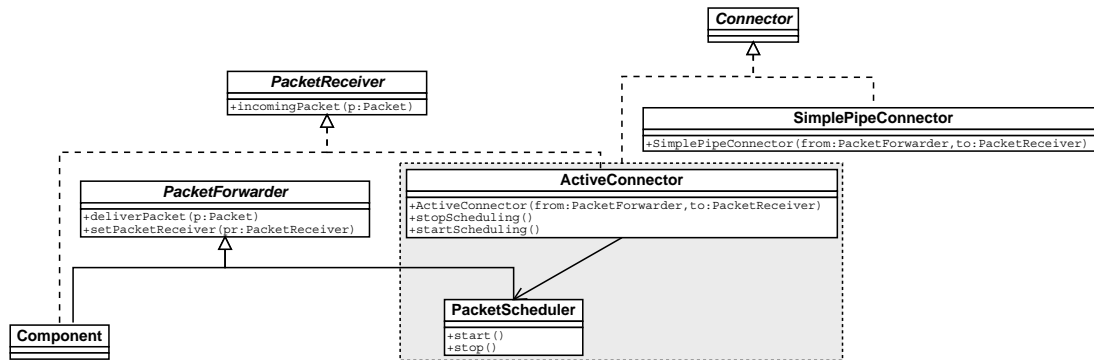
functioneel deel, terwijl de connecties met andere componenten (via zijn *receiver* en *forwarder*) bewaard blijven.

Het algemeen uitbreiden of aanpassen van de receivers of forwarders binnen het DiPS-model is bijna onhaalbaar. Bij een aanpassing van de `PacketReceiver` bijvoorbeeld moet elke component deze aanpassing eveneens doorvoeren in zijn code.

Connectoren

Een connector in DiPS is een conceptueel begrip. Het verzorgt de connectie tussen twee componenten en is voorgesteld door een lege interface `Connector`. Voorbeelden van connectoren zijn de `SimplePipeConnector`, die de forwarder van de ene component koppelt aan de receiver van de andere component, en de `ActiveConnector`, die reeds werd besproken in paragraaf 2.2.6.

In figuur 3.1 is een klassenschema voorgesteld van connectoren en componenten. Het valt op dat de `ActiveConnector` eigenlijk op een verwrongen plaats zit in het klassenschema. De `ActiveConnector`, in combinatie met zijn `PacketScheduler`, hoort veel meer thuis in de structuur van een component. De `ActiveConnector` implementeert de `PacketReceiver` en de `PacketScheduler` een uitbreiding van de `PacketForwarder`. Het is dan ook logisch om een `Component` en een `ActiveConnector` op dezelfde manier te behandelen, zonder de conceptuele verschillen te doorbreken.

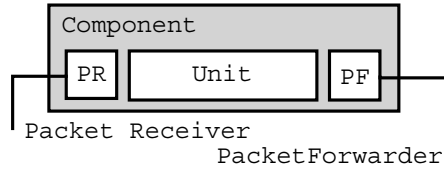


Figuur 3.1: Beknopt klassenschema van een connector in DiPS

3.2.2 DiPS II

In de vorige paragraaf werden enkele beperkingen in DiPS opgesomd. In dit hoofdstuk wordt geprobeerd om deze beperkingen in DiPS weg te werken.

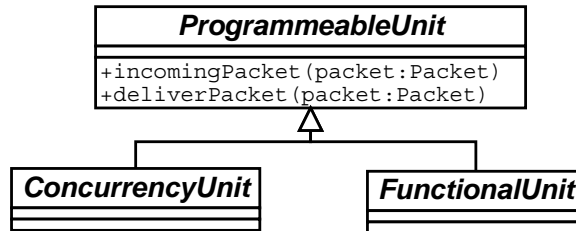
Zoals reeds gesuggereerd, worden in DiPS II enerzijds niet-functionele onderdelen (zoals de `ActiveConnector`) consistent behandeld als de andere componenten. Anderzijds wordt



Figuur 3.2: Een Component in DiPS II

het karakteristieke deel losgekoppeld van de communicatie interfaces naar de buurcomponenten.

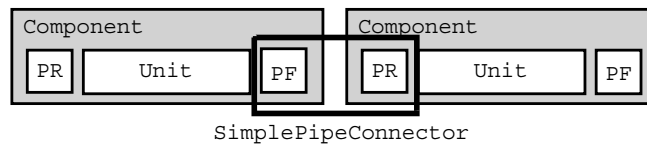
In DiPS II bestaat een **Component** uit een **PacketReceiver**, een **Unit** en een **PacketForwarder** (figuur 3.2). De **Unit** staat in voor de karakteristieke van een component. Om het conceptuele verschil te benadrukken, worden deze karakteristieken opgesplitst in functionele karakteristieken (**FunctionalUnit**) en karakteristieken met betrekking tot de gelijktijdigheid (**ConcurrencyUnit**), zoals voorgesteld in figuur 3.3. De **FunctionalUnit** kan best vergeleken worden met het functionele deel van de component uit DiPS I.



Figuur 3.3: Klassenschema Unit

Daar waar niet-functionele karakteristieken in DiPS I werden weggewerkt als connectoren (bijvoorbeeld de **ActiveConnector**), worden deze in DiPS II consistent behandeld als units. Een duidelijke opsplitsing binnen deze units zorgt voor het nodige conceptuele onderscheid.

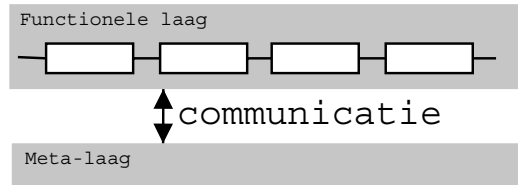
De **PacketReceiver** en **PacketForwarder** zorgen voor de koppeling van componenten. Een **SimplePipeConnector** zal de **PacketReceiver** van één component schakelen aan de **PacketForwarder** van een andere (figuur 3.4). Door de loskoppeling van het karakteristieke deel en deze interfaces kunnen componenten aan elkaar gekoppeld blijven, terwijl hun karakteristieke units vervangen worden door andere units.



Figuur 3.4: De functie van een SimplePipeConnector

3.3 Communicatie tussen functionaliteit en beheer

Eén van de doelstellingen is de scheiding van functionaliteit en beheer (paragraaf 3.1.1). Om deze scheiding te benadrukken, wordt het systeem verdeeld in twee lagen: de functionele en de meta-laag. De functionele laag bevat de aaneengeschakelde componenten, zonder beheermodules. De meta-laag is een tweede laag die echter niets bijbrengt tot het puur functionele aspect van de ketting. De taak van deze laag is het systeem observeren en de nodige aanpassingen maken aan de componentenketting.



Figuur 3.5: Nood aan communicatie tussen de twee lagen

Een scheiding tussen deze twee lagen is mogelijk, maar er zal steeds interactie nodig zijn tussen modules van beide lagen. Enerzijds moet het beheersysteem allerlei informatie kunnen opvragen vanuit de functionele laag naar de meta-laag. Anderzijds moet het beheersysteem zijn beslissingen en aanpassingen kunnen doorvoeren in de functionele laag. De volgende paragrafen gaan dieper in op de beide vormen van communicatie.

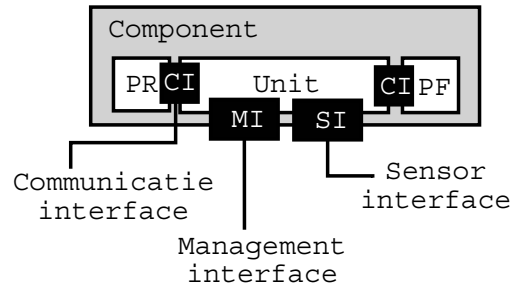
3.3.1 Communicatie van meta-laag naar functionele laag

Om communicatie mogelijk te maken vanuit de meta-laag naar de functionele laag, wordt deze laatste uitgebreid met een *management interface*. Via deze interface kan de meta-laag interageren met de functionele laag. Aangezien de beheermodules ingrijpen op het functionele deel van de component (de unit), wordt deze management interface toegevoegd aan de units. De management interface bevat de mogelijke opdrachten, die vanuit de meta-laag kunnen worden doorgegeven naar de functionele laag.

Een `Unit` bevat uiteindelijk drie types interfaces. Enerzijds zijn er de communicatie interfaces (CI) die de connectie met zijn `PacketReceiver` en `PacketForwarder` regelen, en anderzijds zijn management en sensor interfaces (MI en SI) die communicatie met de meta-laag mogelijk maken. Deze verschillende interfaces zijn symbolisch weergegeven in figuur 3.6.

3.3.2 Communicatie van functionele laag naar meta-laag

In deze paragraaf wordt ingezoomd op de manier waarop informatie in de functionele laag verzameld wordt en voor de meta-laag beschikbaar wordt gesteld.



Figuur 3.6: De verschillende interfaces bij een Component

Sensoren worden gebruikt om informatie te verzamelen over enerzijds de toestand van de componenten in de functionele laag en anderzijds de soorten boodschappen die door het systeem passeren. Deze sensoren zijn bevraagbaar vanuit de meta-laag.

Het bepalen van de interne toestand van het systeem (Toestandssensoren)

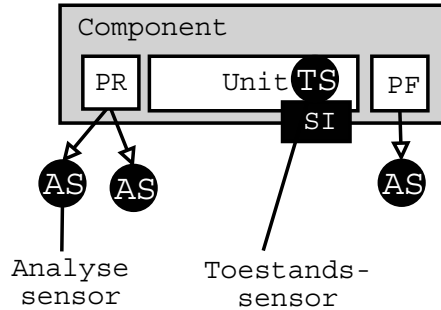
Het bepalen van de interne toestand van de verschillende units, gebeurt via de sensor interface van de unit. Deze bevat de nodige methodes die toelaten de interne toestand van een unit - in de mate dat die gekend mag zijn door de meta-laag - opvraagbaar te maken. In figuur 3.7 is een toestandssensor (TS) en zijn bijhorende sensor interface (SI) weergegeven.

De analyse van de boodschappenstroom (Analysesensoren)

Op de één of andere manier moet de meta-laag een analyse kunnen maken van de boodschappenstroom doorheen het systeem. Een elegante oplossing is de `PacketReceiver` en `PacketForwarder` uitbreiden, zodat boodschappen niet alleen doorgestuurd worden naar de units of de volgende component, maar ook naar geïnteresseerde `PacketListeners`. Boodschappen volgen dan niet alleen de normale stroom doorheen het systeem, maar passeren dan ook de verschillende *listeners*, die ingeschreven staan bij de `PacketReceiver` of `PacketForwarder`. Dit alternatief pad is niet bedoeld om functionele zaken te doen met de boodschappen.

Deze aanpassing aan de `PacketReceiver` en `PacketForwarder` is mogelijk dankzij de opsplitsing in DiPS II van de `Component` in een karakteristiek deel en in de interfaces.

Sensoren die analyses maken van de boodschappenstroom, worden opgevat als *listeners*. Ze worden ingeplugd op de `PacketReceiver` of de `PacketForwarder` en zien de boodschappen passeren. Ze bevinden zich tussen de functionele laag en de meta-laag. De analysesensoren zijn in figuur 3.7 voorgesteld als AS.



Figuur 3.7: De sensoren van het beheersysteem

Herbruikbare sensoren

Er wordt geopteerd om de sensoren zo algemeen en generiek mogelijk te houden. Zo kunnen deze gemakkelijk herbruikt worden in diverse beheersystemen. Een boodschappenteller is bijvoorbeeld een sensor die gemakkelijk in meerdere systemen als sensormodule kan opgenomen worden.

3.3.3 Performantiesensoren

Performantiesensoren kunnen zowel als toestandssensor of als analysesensor worden geïmplementeerd. Ze zorgen voor een terugkoppeling tussen de prestatie van het systeem en de monitor.

3.3.4 De monitor

Na het verzamelen van alle informatie via sensoren, is het nu de beurt aan de verwerking van deze informatie tot zinvolle beslissingen voor het systeem. Een **Monitor** speelt de centrale en beslissende rol in het beheersysteem.

Een *monitor* staat in contact met de verschillende sensoren en haalt op regelmatige basis hun gegevens op. Deze worden door de monitor verwerkt tot mogelijke verbeteringen of aanpassingen aan het functionele systeem. Via de management interface worden deze verbeteringen doorgevoerd in het functionele systeem.

Om niet bij elke wijziging van de beslissingsstrategie een nieuwe monitor te moeten ontwikkelen, worden het algemeen monitor gedeelte en de monitor strategie van elkaar gescheiden. In een monitor kan de gepaste strategie inplugd worden.

Bij het gebruik van performantiesensoren, dient er na elke wijziging voldoende spertijd in acht genomen te worden. Indien dit niet gebeurt, worden performantieresultaten gebruikt die nog afkomstig zijn van vóór de aanpassingen. Dit laatste kan leiden tot onjuiste interpretaties en sterke schommelingen in de prestatie, te vergelijken met de resultaten van een onstabiel systeem.

Bij gebruik van meerdere monitors in het systeem, is het nuttig een extra **meta-monitor** te introduceren. Deze meta-monitor heeft als taak op de meta-meta-laag (= een nog hogere laag dan de meta-laag) de coördinatie te voorzien tussen de verschillende monitors en hun beslissingen. Hierop worden dieper ingezoomd in hoofdstuk 4.

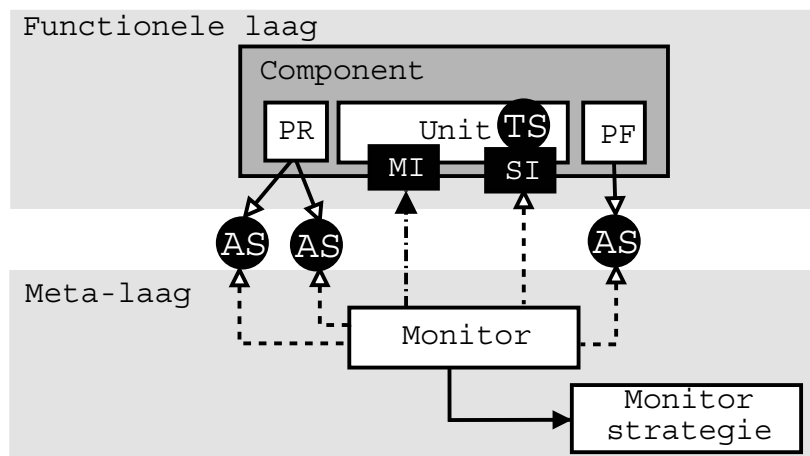
3.4 Het beheersysteem als onafhankelijke extensie

Eén van de doelstellingen was een gemakkelijk inplugbaar beheersysteem. Dit houdt in dat de beheerfunctie gemakkelijk op het systeem gezet en er terug afgehaald kan worden, zonder de algemene werking van het systeem hierdoor te hinderen. Het beheersysteem is een onafhankelijke extensie die bij een bestaand systeem kan worden toegevoegd.

Bij het ontwerpen van het beheersysteem in de vorige paragrafen, is er reeds voldoende rekening mee gehouden dat de functionele laag geen weet heeft van het onderliggend beheer in de meta-laag. Enerzijds is de communicatie tussen de twee lagen (via de management interface) steeds geïnitieerd vanuit de meta-laag. Anderzijds zijn de sensoren inplugbare modules (in de receivers en de forwarders) die zelf tussen de twee lagen inzitten. Ze hebben geen koppelingen naar de beheerfunctie, en kunnen vrij gemakkelijk terug losgekoppeld worden uit de functionele laag.

3.5 Het volledige model

Het volledige model is samengebracht in figuur 3.8. Het toont de twee lagen, de opsplitsing van een component in zijn receiver, unit en forwarder, de management en sensor interface, de sensoren en de monitor met zijn strategie.



Figuur 3.8: Het volledige model

3.6 Toetsen van de doelstellingen

In het begin van dit hoofdstuk werden drie doelstellingen gedefinieerd voor een adaptief en modulair beheer: een scheiding tussen functionaliteit en beheer, modulariteit en herbruikbaarheid. In deze sectie wordt het opgebouwde model kort getoetst aan deze doelstellingen.

Scheiding tussen functionaliteit en beheer

In het ontwerp zijn twee lagen geïntroduceerd: een laag voor het functionele aspect en een meta-laag voor het beheer. De functionele ketting heeft geen directe kennis over de beheercomponenten die op het systeem zijn vastgemaakt. De communicatie tussen de twee lagen gebeurt via de management en sensor interface van de unit. Verder is het beheersysteem inplugbaar op de functionele ketting. Dit alles wijst erop dat aan de scheiding tussen functionaliteit en beheer voldaan is.

Modulariteit

Bij het ontwerpen van een beheersysteem kan, afhankelijk van de complexiteit van het probleem, gekozen worden hoe uitgebreid het beheersysteem is. Er kunnen extra sensoren worden ingeplugd en ook de strategie van de monitor kan desgewenst aangepast worden. Het geheel is modulair opgebouwd.

Herbruikbaarheid

In het voorgestelde ontwerp is er aandacht besteed aan de herbruikbaarheid. Enerzijds zijn er generieke sensoren ter beschikking. Anderzijds kunnen monitors vaak herbruikt worden omdat ze, via de inplugbare strategieën, voldoende kunnen inspelen op variaties, zonder de monitor zelf te moeten veranderen.

Het model voldoet dus aan de vooropgestelde doelstellingen.

3.7 Besluit

In dit hoofdstuk zijn eerst de vereisten vastgelegd voor een adaptief en modulair beheer. Vervolgens werd een ontwerp stap voor stap opgebouwd met deze doelstellingen in het achterhoofd. Dit model is daarna nogmaals kort getoetst aan de doelstellingen en positief bevonden.

In de volgende hoofdstukken wordt via twee *case studies* nagegaan of het ontwerp inderdaad gebruikt kan worden voor een adaptief en modulair beheer. Er wordt gelet op de beperkingen die dit model met zich meebrengt en wat de problemen zijn bij de implementatie.

Hoofdstuk 4

Case study : Adaptieve parallelisatie in DiPS

In dit hoofdstuk wordt ondersteuning voor adaptieve parallelisatie voorzien in DiPS, vergelijkbaar met de SEDA architectuur uit sectie 2.2.5. De case study is opgebouwd uit vier delen. In het eerste deel wordt een statisch parallelisatie model in DiPS ontworpen. Vervolgens wordt stap voor stap het adaptief model opgebouwd om tot een adaptieve parallelisatie te komen. Verder volgt een toetsing van het systeem en worden de gebruikte strategieën toegelicht. Op het einde van het hoofdstuk worden het adaptief model en de gerealiseerde parallelisatie geëvalueerd.

4.1 Een statisch parallelisatie model

In deze sectie wordt een statisch parallelisatie model opgebouwd. Met een statisch parallelisatie model wordt bedoeld dat de verwerking van boodschappen in de verschillende delen van de componentenketting gelijktijdig verloopt, met de mogelijkheid om sommige stukken van de ketting meer processortijd toe te wijzen dan andere.

In de eerste paragraaf worden de vereisten van een dergelijk gelijktijdigheidsmodel vastgelegd. Vervolgens komen de verschillende onderdelen uit het ontwerp aan bod.

4.1.1 De vereisten

Scheiding tussen functionaliteit en gelijktijdigheid

In het ontwerp moet een duidelijke scheiding zijn tussen de functionaliteit van de componentenketting en de aanpassingen aan het gelijktijdigheidsmodel. Op die manier kan het parallelisatie model gemakkelijk herbruikt worden in andere toepassingen, zonder dat de DiPS ontwikkelaar hoeft na te denken over gelijktijdigheid in zijn ontwerp. Het parallelisatie model is een eenvoudige extensie op bestaande applicaties.

Regelbare parallelisatie

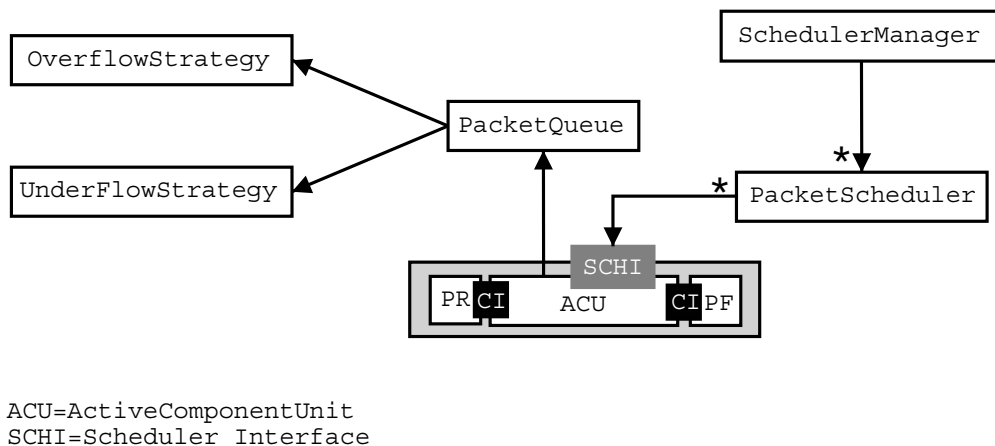
Het parallelisatie model moet de componentenketting opsplitsen in verschillende delen. Het moet mogelijk zijn aan elk van die stukken een gewicht toe te kennen om, op basis van deze gewichten, de procestijd te verdelen over de verschillende stukken ketting. Op die manier wordt een regelbare parallelisatie bekomen.

Aanpasbaar

Om het ontwerp in een verder stadium uit te breiden naar een dynamische versie, moet het statisch model voldoende aanpasbaar zijn. Zo moeten bijvoorbeeld de gewichten van de verschillende stukken ketting gemakkelijk aangepast kunnen worden.

4.1.2 Het conceptueel ontwerp

In deze sectie wordt stap voor stap het ontwerp opgebouwd. Hierbij worden de SEDA architectuur en de bestaande `ActiveConnector` uit DiPS gebruikt als basis. Het volledige model is voorgesteld in figuur 4.1.



Figuur 4.1: Conceptueel ontwerp van een statisch parallelisatie model

In SEDA wordt de ketting opgedeeld in verschillende *stages*. Elke stage is een aaneenschakeling van enkele componenten. Verder bevat elke stage een wachtrij voor de boodschappen die door die stage verwerkt moeten worden, en een aantal schedulers (threads) die instaan voor het verwerken van die boodschappen.

PacketQueue

De `PacketQueue` is de wachtrij voor de boodschappen, horend bij een stage. Deze wachtrij biedt methodes aan om boodschappen op te slaan of terug op te halen. De capaciteit van de `PacketQueue` kan gemakkelijk groter of kleiner worden gemaakt.

De `PacketQueue` bevat een inplugbare `OverflowStrategy` en `UnderflowStrategy`. Deze wachtrij strategieën treden in werking als er te veel of te weinig boodschappen in de wachtrij zitten. Mogelijk reacties kunnen zijn: het vergroten of verkleinen van de capaciteit, het hersorteren van de boodschappen in de wachtrij, het uitfilteren van boodschappen, ...

ActiveComponentUnit

De `ActiveComponentUnit` (ACU) is de centrale schakel in de parallelisatie, maar bevat zelf geen actief gedrag. Vanaf een dergelijke unit begint telkens een nieuwe stage in de ketting, zoals voorgesteld in figuur 4.2.



Figuur 4.2: Opdeling van de ketting in verschillende stages

Wanneer nieuwe boodschappen aankomen in de unit, worden deze in de `PacketQueue` gedeponeerd. De boodschappen blijven daar zitten tot een scheduler de boodschap ophaalt en verder begeleidt doorheen de stage tot de volgende ACU. Om deze bewerkingen mogelijk te maken vanuit de scheduler, implementeert de ACU de *scheduler interface* (SCHI) met de methodes `getPacket` en `deliverPacket`, zoals symbolisch voorgesteld in figuur 4.1.

PacketScheduler en SchedulerManager

De `PacketScheduler` is verantwoordelijk voor het ophalen van nieuwe boodschappen in de `ActiveComponentUnit` en het verder begeleiden van deze boodschappen doorheen de ketting. Elke scheduler is een aparte thread en deze threads bezorgen de stage zijn actief gedrag.

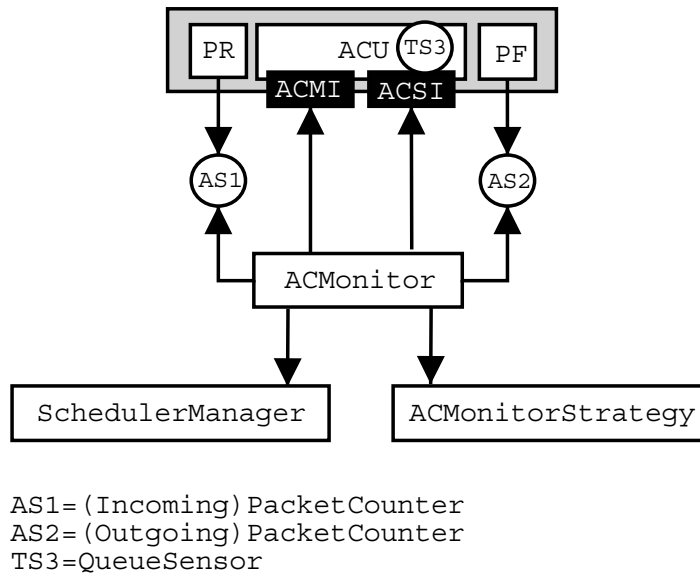
Een nieuwe thread aanmaken vergt een aanzienlijke opstartkost. Om te besparen op deze opstartkosten worden `PacketSchedulers` herbruikbaar gemaakt. Schedulers kunnen gemakkelijk verplaatst worden van de ene ACU naar de andere en daar hun taak verder uitvoeren. Dit is een stuk voordeliger dan eerst de bestaande scheduler af te sluiten om vervolgens een nieuwe aan te maken voor een andere ACU.

De `SchedulerManager` behoudt het overzicht over de verschillende `PacketSchedulers`. De `SchedulerManager` verwerkt de aanvragen om schedulers toe te voegen of te verwijderen van ACU's.

De verschillende componenten van dit statisch parallelisatie model zijn samengebracht in figuur 4.1. In de volgende sectie wordt dit model uitgebreid tot een adaptief parallelisatie model.

4.2 Het adaptief model

In deze sectie wordt het adaptief model uit hoofdstuk 3 gebruikt om te komen tot een dynamisch parallelisatie model. In een eerste stap worden de sensoren gedefinieerd. Daarna volgt een beschrijving over de monitor en de nood aan een meta-monitor.



Figuur 4.3: Het adaptief beheer van het parallelisatie model

4.2.1 De sensoren

PacketCounter

De `PacketCounter` houdt bij hoeveel boodschappen er voorbijgekomen zijn sinds de vorige reset. Het is een herbruikbare analysesensor, die zowel het aantal inkomende als uitgaande boodschappen van een component kan registreren, al naargelang de sensor in de `PacketReceiver` (AS1) of `PacketForwarder` (AS2) wordt ingeplugd (figuur 4.3).

QueueSensor

De `QueueSensor` is een toestandssensor. Deze sensor bevat informatie over de capaciteit van de `PacketQueue` en het aantal boodschappen dat in de wachtrij is opgeslagen. Deze informatie kan opgevraagd worden via de `ACSensorInterface` (ACSI) van de `ActiveComponentUnit`, zoals symbolisch voorgesteld in figuur 4.3.

4.2.2 De monitor met inplugbare strategie

De `ACMonitor` haalt op regelmatige tijdstippen de sensorgegevens op bij zowel zijn inkomende als uitgaande `PacketCounter` en de `QueueSensor`. De monitor verwerkt deze informatie tot zinvolle verbeteringen of aanpassingen (zoals het toevoegen of verwijderen van een scheduler, het vergroten of verkleinen van de wachtrij, het veranderen van de wachtrij strategie, ...).

Het aanpassen van het aantal schedulers wordt geregeld via de `SchedulerManager`, om zo de onnodige opstartkosten bij het aanmaken van schedulers te vermijden. Deze overige aanpassingen worden door de monitor doorgevoerd via de `ACManagementInterface` (ACMI) van de `ActiveComponentUnit`.

De manier waarop de monitor zijn beslissingen neemt, is vastgelegd in een inplugbare `ACMonitorStrategy`. Deze duidelijke scheiding tussen het monitor gedeelte en de monitor strategie zorgen ervoor dat verschillende monitor strategieën gemakkelijk gebruik kunnen maken van dezelfde herbruikbare monitor. De monitor en zijn bijhorende strategie zijn terug te vinden in figuur 4.3.

De `ACMonitor` bevat ook de nodige methodes om het adaptief beheer op een ACU te klikken (`setComponent`) en het er terug af te halen (`unsetComponent`). Het adaptief beheer is hierdoor een gemakkelijk toevoegbare extensie op de bestaande statische parallelisatie.

Wanneer de componentenketting in slechts twee stages wordt opgesplitst, kan een `ACMonitor` zelfstandig de juiste beslissingen nemen. Indien er echter meerdere `ActiveComponentUnits` en bijhorende `ACMonitors` in het systeem aangebracht zijn, dan is er coördinatie nodig tussen de verschillende monitors.

Het is bijvoorbeeld zinloos dat tijdens piekmomenten elke monitor zelfstandig beslist om extra schedulers toe te voegen aan zijn unit, om zo de boodschappenstroom de baas te kunnen. Het eindresultaat zou zijn dat er een veelvoud aan schedulers gebruikt wordt, met de nodige *scheduling overhead* tot gevolg. In vele gevallen zal de te verdelen processor-tijd ook niet optimaal verdeeld zijn over de ketting, of zal het veel langer duren alvorens de optimale verdeling bereikt wordt, dan wanneer er wel coördinatie zou zijn tussen de verschillende monitors.

4.2.3 De meta-monitor met inplugbare strategie

De `ACMetaMonitor` zorgt voor de nodige coördinatie tussen de verschillende `ACMonitors` in het systeem. De meta-monitor staat in voor het beheer van de beheerlaag en bevindt

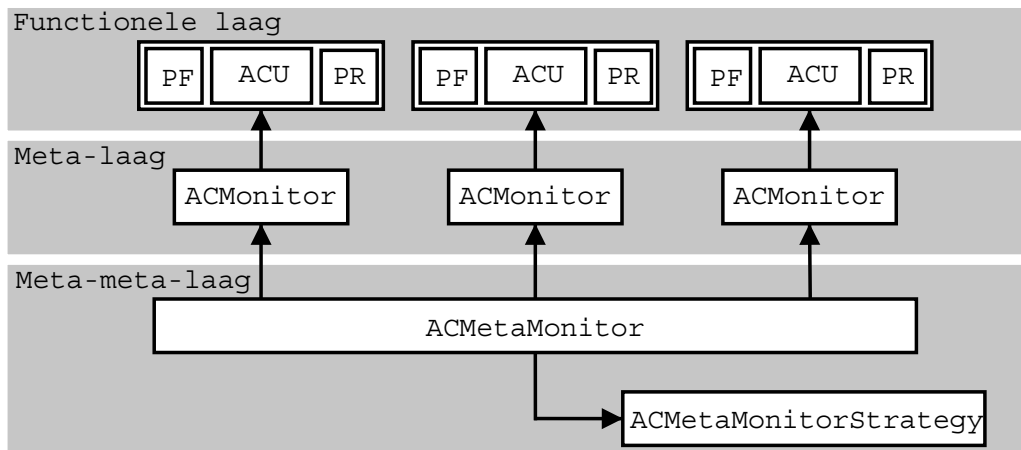
zich in de meta-meta-laag (figuur 4.4).

De meta-monitor haalt op regelmatige tijdstippen informatie op bij de verschillende monitors en verwerkt deze tot een globale beslissing. Deze beslissingen worden dan op hoog niveau doorgegeven aan de verschillende monitors, die ze op hun beurt uitvoeren op de ACU's. De meta-monitor bevat dus zelf ook actief gedrag.

De meta-monitor is, net zoals de monitor, voorzien van een inplugbare strategie, de `ACMetaMonitorStrategy`. Zodoende kunnen verschillende meta-monitor strategieën dezelfde meta-monitor hergebruiken.

Het actief gedrag bevindt zich nu dus op twee plaatsen. Enerzijds in de meta-laag bij de monitor en anderzijds in de meta-meta-laag bij de meta-monitor. Bij gebruik van de meta-monitor kan er worden geopteerd om het actief gedrag van de monitor op zich uit te schakelen. Is het echter gewenst dat de monitor kleinere lokale beslissingen kan nemen, dan is er nog een synchronisatie tussen beide lagen nodig. De verdere uitwerking van deze synchronisatie ligt echter buiten het bereik van deze thesis.

Het gebruik van de `ACMetaMonitor` en zijn `ACMetaMonitorStrategy` is voorgesteld in figuur 4.4.



Figuur 4.4: Het gebruik van de meta-monitor

4.3 Toetsen van het systeem

In deze sectie wordt de dynamische parallelisatie aan een test onderworpen. Eerst wordt de testopstelling beschreven, daarna wordt de gebruikte strategie toegelicht, om te besluiten met de resultaten van de test.

4.3.1 Testopstelling

Er wordt een componentenketting gemaakt met drie stages. In elke stage bevindt zich een `ActiveComponentUnit` en een `BusyUnit`. De `ActiveComponentUnits` krijgen allemaal een `ACMonitor`, gekoppeld aan een gemeenschappelijke `ACMetaMonitor`. De `BusyUnit` is een functionele unit die de processor een heel tijdje bezighoudt met een blok zinloze rekenbewerkingen. Het aantal keer dat deze bewerkingen uitgevoerd moet worden, per boodschap die passeert, kan aangepast worden via de methode `setLoad`. Dit aantal wordt verder in de tekst het gewicht van de `BusyUnit` genoemd. De volledige ketting is symbolisch voorgesteld in figuur 4.5.



Figuur 4.5: De gebruikte testopstelling

De testopstelling ondergaat drie fasen. In elk van deze fasen krijgen de `BusyUnits` andere gewichten. De parameters staan uitgezet in tabel 4.1. In de eerste fase heeft vooral unit B veel processor tijd nodig ten opzichte van unit A en C. In de tweede fase is het zwaartepunt verschoven naar unit C. In fase 3 wordt de verdeling uit fase 1 hersteld. Gedurende de test wordt een zware piekbelasting aangelegd van acht boodschappen per seconde. De beschreven variabele belasting kan ook een voorbeeld zijn van een echt systeem waar bijvoorbeeld gedurende een periode de encryptie wordt afgezet.

	BusyUnit A	BusyUnit B	BusyUnit C
Fase 1	100	700	100
Fase 2	200	200	600
Fase 3	100	700	100

Tabel 4.1: De verschillende fasen van de testopstelling

De volledige testopstelling wordt uitgevoerd met behulp van *green threads* in de java virtuele machine. Dit betekent dat de java virtuele machine zelf de thread scheduling regelt.

4.3.2 De gebruikte strategie

In deze eenvoudige testopstelling is enkel gebruikt gemaakt van actief gedrag van de `ACMetaMonitor`. De `ACMonitor` voert dus passief de opdrachten van de meta-monitor uit. De gebruikte `ACMetaMonitorStrategy` valt uiteen in twee delen, en wordt in de volgende paragrafen kort toegelicht.

Het egaliseren van de doorstroom

In een eerste stap wordt gepoogd om de doorstroom in de verschillende stages te egaliseren. De volledige ketting kan vergeleken worden met een aaneenschakeling van buizen. Het is de bedoeling om de brede en smalle buizen te egaliseren tot één lange buis van normale dikte. Stages met een groter debiet zorgen voor een ophoping van boodschappen bij het begin van de volgende stage en gebruiken hiervoor processortijd, die beter benut kan worden door de stages met een lager debiet.

Het algoritme dat hiervoor gebruikt wordt, vergelijkt de doorstroom per scheduler van de verschillende stages en kent gewichten toe aan elke stage om te komen tot de optimale doorstroom. De verschillende stages samen beschikken over negen schedulers en met de gewichten worden deze negen schedulers verdeeld.

De huidige doorstroom in elke stage wordt voorgesteld als f_i en het huidige aantal schedulers als t_i . De gewenste doorstroom in elke stage als is f'_i na het algoritme en de bijhorende nieuwe schedulerverdeling t'_i . In een eerste stappen wordt de doelstelling uitgedrukt dat de doorstroom geëgaliseerd wordt over alle stages.

$$f = f'_i \quad (4.1)$$

Verder wordt verondersteld dat de doorstroom per scheduler in een stage niet verandert.

$$\frac{f_i}{t_i} = \frac{f'_i}{t'_i} \quad (4.2)$$

Op basis daarvan kan de nieuwe doorstroom f berekend worden.

$$f = \left(\sum \frac{t_i}{f_i} \right)^{-1} * 9 \quad (4.3)$$

En met de nieuwe doorstroom f , kan een nieuwe verdeling van de schedulers uitgedrukt worden.

$$t_i = \frac{f * t_i}{f_i} \quad (4.4)$$

Het opzoeken van knelpunten

De verdeling uit de vorige paragraaf zoekt wel steeds een uitmiddeling van de huidige situatie, maar kan gemakkelijk vastlopen op een niet-optimale, stabiele verdeling. Het doel van dit tweede deel van de strategie is het opsporen van mogelijke knelpunten in de ketting en deze lichtjes proberen te corrigeren door het toevoegen van een extra scheduler. Dit heeft tot gevolg dat we het evenwicht uit de ketting lichtjes verstoren om zo bij de volgende herverdeling een hopelijk nog optimaler resultaat te bekomen. Gebruik maken van excitaties om van de ene stabiele toestand over te gaan naar een andere, is ook in andere disciplines als scheikunde en biologie een gekend fenomeen.

Een knelpunt in de ketting kan beschreven worden als een stage die meer boodschappen

binnenkrijgt, dan het kan verwerken. Er kunnen zich in de ketting meerdere dergelijke knelpunten voordoen. Nu is het de bedoeling van dit deel van de strategie om enkel het knelpunt te verhelpen dat de kleinste doorstroom van het systeem oplevert.

Op eenzelfde manier kan gezocht worden naar de stages die meer boodschappen kunnen verwerken per tijdseenheid dan dat ze binnenkrijgen. Hier kan de stage met de grootste doorstroom geselecteerd worden om een scheduler af te staan.

De bovenstaande twee strategie delen worden beurtelings uitgevoerd, met de nodige tussenpauze om geen besluiten te nemen op achterhaalde gegevens. De bekomen strategie is bijlange nog niet optimaal, maar geeft wel een eerste aanvoelen hoe er met een dynamische verdeling kan gespeeld worden.

4.3.3 De resultaten

De strategie uit de vorige paragraaf wordt gebruikt bij de testopstelling. In figuur 4.6 is de doorstroom in functie van de tijd weergegeven. Dankzij de dynamische parallelisatie wordt doorheen de verschillende fasen een maximale doorstroom behouden. De bijhorende verdeling van de schedulers over de verschillende stages zijn uitgezet in figuur 4.7. De drie fasen zijn duidelijk uit de grafiek af te lezen doordat op die plaatsen de verdeling van de schedulers grondig verandert.

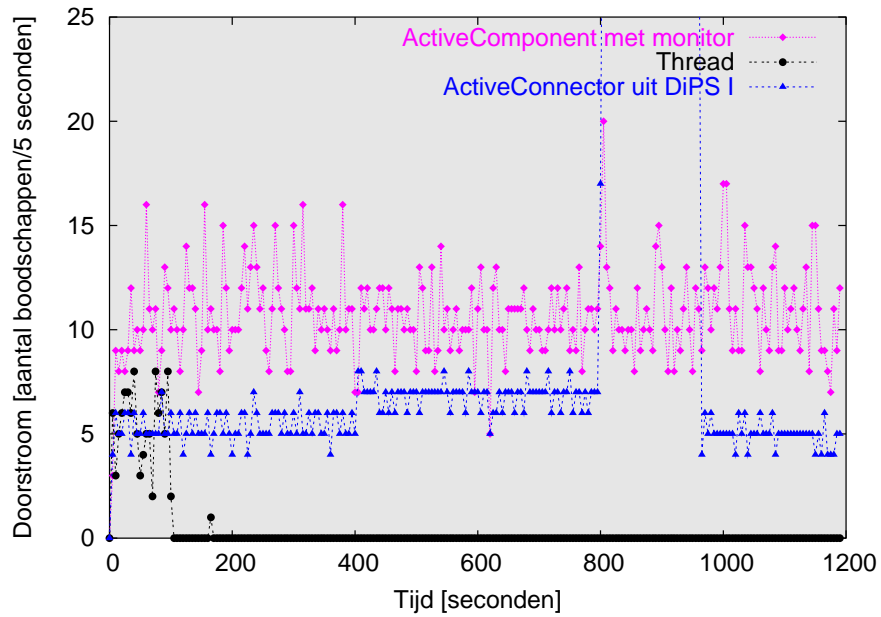
Ter referentie is in figuur 4.6 naast het dynamische model ook het statisch thread model opgenomen en de bestaande `ActiveConnector` uit DiPS I. Bij het statische thread model wordt bij elke inkomende boodschap een nieuwe thread opgestart, die de boodschap begeleidt doorheen de ketting. Aangezien het systeem zich in een piekbelasting bevindt, gaat hier de scheduling overhead al vrij snel primeren op de functie van de ketting.

De piek in doorstroom bij de `ActiveConnector` (vanaf 800 seconden in grafiek 4.6) is als volgt te verklaren. In de tweede fase wordt er in elke stage gepoogd zoveel mogelijk boodschappen te transporteren. Aangezien het gewicht van de laatste stage echter een stuk hoger is dan de andere stages, stapelen boodschappen zich op in de wachtrij. Bij de aanvang van de derde fase, verlaagt het gewicht van de laatste stage, waardoor plots de boodschappen in de wachtrij heel snel verwerkt kunnen worden. Eenmaal de wachtrij verwerkt is, ligt de doorstroom terug een stuk lager.

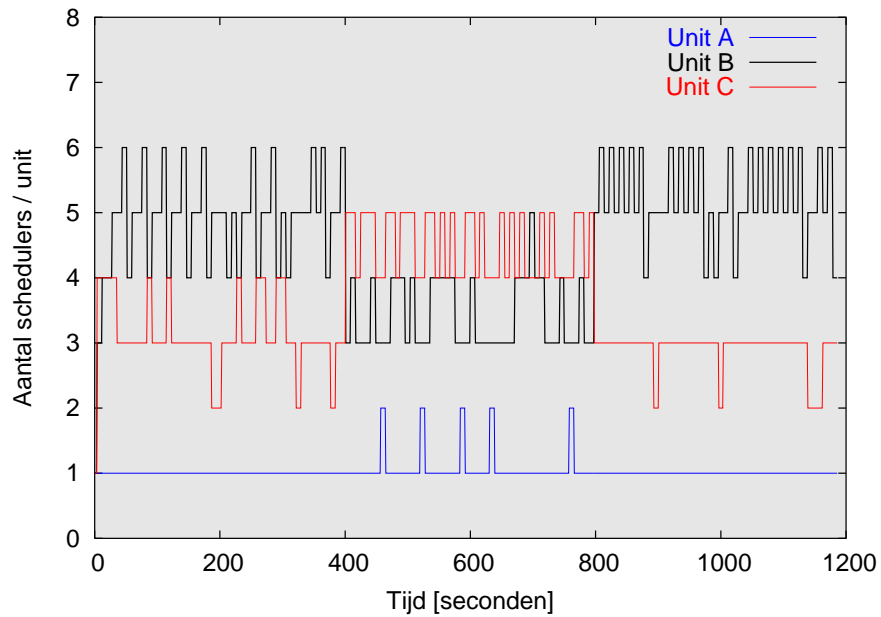
In figuur 4.6 valt ook op dat de doorstroom bij het dynamische model nog niet volledig stabiel is. Het constant op en neer springen van de doorstroom is te wijten aan een meta-monitor strategie die nog niet optimaal is. Het verder uitwerken van een dergelijke strategie ligt echter buiten het bereik van deze thesis.

4.4 Conclusies en verder verloop

In dit hoofdstuk werd een dynamische parallelisatie gebouwd met behulp van het adaptief model uit hoofdstuk 3. In een eerste stap werd een statische parallelisatie ontworpen.



Figuur 4.6: De doorstroom in de testopstelling in functie van de tijd



Figuur 4.7: De schedulerverdeling in de testopstelling in functie van de tijd

Vervolgens werd hiervoor een dynamische extensie gemaakt, door het implementeren van het algemeen adaptief model. Op het einde van dit hoofdstuk werd dan het nieuwe ge-

lijktijdigheidsmodel getest in een dynamische omgeving. Het dynamische model haalde in deze testopstelling een constante doorstroom, terwijl de zwaartepunten van de ketting gedurende de test werden aangepast.

In deze case study is het idee van een meta-monitor verder geconcretiseerd en gebruikt om de verschillende monitors uit de meta-laag te coördineren. In deze case study is het actief karakter van de monitor weggelaten ten voordele van het actief gedrag van de meta-monitor. Actief gedrag op beide niveau's is mogelijk, maar daarvoor moet eerst nog een synchronisatiemechanisme tussen de twee uitgewerkt worden. Deze synchronisatie lag echter buiten het bestek van deze thesis. Verder zijn er geen problemen ontdekt met het adaptief model uit hoofdstuk 3.

Deze case study was de eerste aanzet tot dynamische parallellisatie. Voor concreet gebruik zou de strategie van de meta-monitor nog een stuk intelligenter moeten worden gemaakt om de huidige instabiliteiten weg te werken. Ook een terugkoppeling tussen de `QueueSizeSensor` en de beslissingen behoort tot de mogelijke uitbreidingen.

Hoofdstuk 5

Case study : Een adaptief bestandssysteem in DiPS

In dit hoofdstuk wordt een adaptief bestandssysteem gebouwd, vergelijkbaar met de systemen uit sectie 2.3. De nadruk bij deze case study ligt vooral op het evalueren van het adaptieve model, veeleer dan op het verkrijgen van een volledig uitgewerkte en direct bruikbare implementatie van een dergelijk bestandssysteem.

Dit hoofdstuk is opgebouwd uit drie delen. In het eerste deel wordt een primitief en generiek bestandssysteem ontworpen. Vervolgens wordt stap voor stap het adaptieve model opgebouwd en wordt de gebruikte cache strategie toegelicht. Op het einde van het hoofdstuk volgt de evaluatie van het adaptieve bestandssysteem en het adaptief model in DiPS.

5.1 De voorstelling van een primitief bestandssysteem

In deze sectie wordt een bestandssysteem opgebouwd. In de eerste paragraaf worden de vereisten vastgelegd, waaraan het bestandssysteem moet voldoen. Vervolgens komen de verschillende onderdelen uit het ontwerp aan bod. Als afsluiter worden enkele cache strategieën voorgesteld.

5.1.1 De vereisten

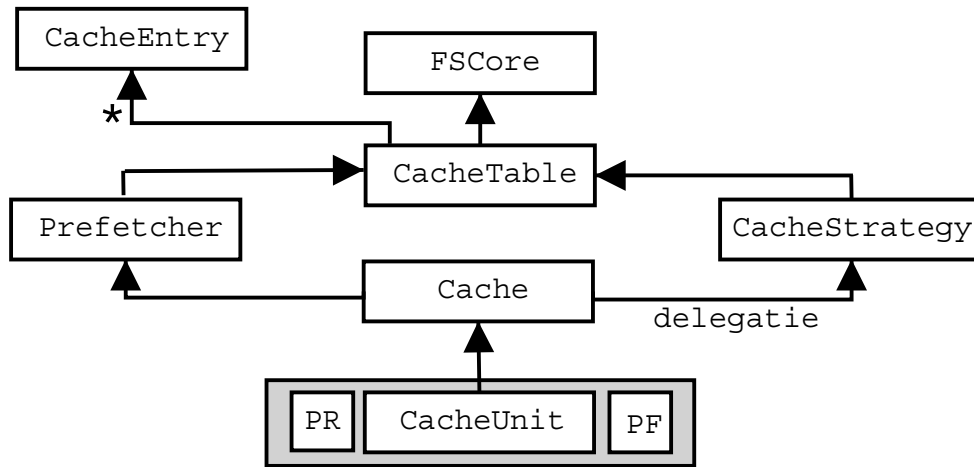
Eenvoudig

Het bestandssysteem moet vrij eenvoudig zijn. Het is niet de bedoeling van deze case study om tot een volledig uitgewerkt bestandssysteem te komen. Er wordt eerder gestreefd naar een conceptuele voorstelling, die een aantal belangrijke basiseigenschappen van een bestandssysteem bezit. Zo moet het mogelijk zijn om data op te vragen en weg te schrijven, en moet het ook een vorm van cache bevatten.

Aanpasbaar

Het bestandssysteem moet voldoende aanpasbaar zijn, anders heeft een beheermodule geen zin. Zo moet het mogelijk zijn een andere cache strategie te kiezen, om de cache grootte aan te passen, ...

5.1.2 Het conceptueel ontwerp



Figuur 5.1: Conceptueel ontwerp van een bestandssysteem

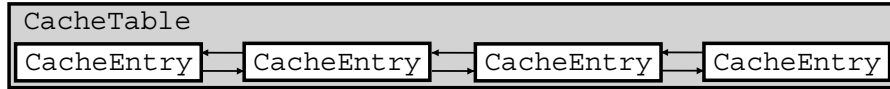
FSCore

De `FSCore` is de conceptuele voorstelling van de data-drager. Woorden kunnen worden opgehaald (`getWords`) en worden weggeschreven (`setWords`). Net zoals bij een harde schijf, wordt er bij elk van deze methodes een opstarttijd en een tijd per woord voorzien. Het is dus voordeliger om drie woorden in één keer in te lezen, dan drie keer apart één woord in te lezen.

CacheTable

De `CacheTable` stelt de cache voor van het bestandssysteem. Het bevat een aaneenschakeling van `CacheEntries`, zoals symbolisch voorgesteld in figuur 5.2. Elke *entry* bevat de cache informatie van één woord.

De `CacheTable` verwerkt twee soorten opdrachten. Enerzijds verwerkt het de expliciete cache opdrachten (`cache` en `uncache`). Anderzijds biedt het toegang tot de data-drager (`getWords` en `setWords`), zij het via een cache systeem.



Figuur 5.2: De CacheTable is opgebouwd uit een aaneenschakeling van CacheEntries

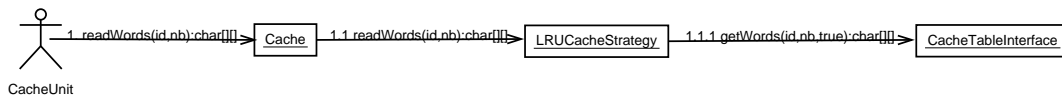
Bij het ontwerp van deze module is rekening gehouden met de genericiteit. De methodes bevatten extra parameters zodat de module in meerdere situaties gebruikt kan worden. Er kan bijvoorbeeld expliciet gekozen worden tussen een **write-through** of een **delayed write** bij het schrijven van woorden. In het eerste geval wordt bij het wegschrijven van een woord in de cache, direct ook de onderliggende data-drager mee aangepast. In het tweede geval wordt de data-drager ofwel aangepast op regelmatige tijdstippen als de aangepaste woorden in de cache in groep worden weggeschreven, ofwel pas als de entry wordt verwijderd uit de cache.

Als de cache het maximaal aantal entries bereikt heeft, dan wordt bij het toevoegen van een extra entry, de achterste uit de lijst verwijderd. Ook hier is een vorm van genericiteit ingebouwd. Bij het ophalen en wegschrijven van een woord, kan het woord zowel vooraan als achteraan in de cache worden teruggestopt. Op deze manier kan dezelfde cache module herbruikt worden voor diverse cache strategieën.

Cache, CacheStrategy en Prefetcher

Een generieke cache is wel mooi qua herbruikbaarheid, maar de interface naar de eindgebruiker is nogal ingewikkeld met al die generieke parameters. Vandaar dat er een eenvoudiger concept wordt geïntroduceerd: de **Cache**.

De **Cache** biedt vereenvoudigde methodes aan, zonder de generieke parameters. Deze module delegeert elke methode naar een **CacheStrategy**, die de strategie-specifieke parameters invult en de generieke **CacheTable** aanspreekt. Deze delegatie is geïllustreerd in figuur 5.3.



Figuur 5.3: Delegatie van de Cache naar de CacheStrategy

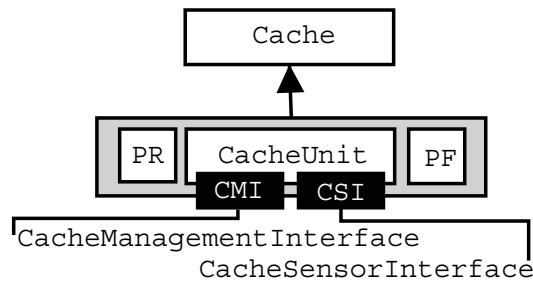
De **Prefetcher** krijgt vanuit de **Cache** geregeld *hints* om een aantal woorden reeds in te laden in de cache, zonder dat deze expliciet zijn opgevraagd. Op deze manier probeert het systeem te anticiperen op aanvragen die binnenkort kunnen worden geplaatst.

CacheUnit

Bij het ontwerp van het bestandssysteem is bewust geen gebruik gemaakt van de componentstructuur binnen DiPS. Het ontwerp van een volledig componentgebaseerd bestandssysteem ligt immers buiten het bestek van deze thesis.

De `CacheUnit` zorgt voor de ontbrekende schakel tussen DiPS en het bestandssysteem. Deze *unit* zet de in- en uitvoer boodschappen om in aanvragen naar de `Cache`.

De `CacheUnit` heeft een `CacheManagementInterface` (CMI) om de beheertaken vanuit de meta-laag te kunnen uitvoeren. Daarnaast bevat het ook een `CacheSensorInterface` (CSI) met de nodige methodes om de interne sensoren te laten bevragen. Dit is geïllustreerd in figuur 5.4.



Figuur 5.4: De CacheUnit

5.1.3 De cache strategieën

Het generieke ontwerp van het bestandssysteem laat toe om verschillende cache strategieën te gebruiken. In de volgende paragrafen worden drie dergelijke strategieën kort besproken: de `MRUCacheStrategy`, de `LRUCacheStrategy` en de `NoCacheStrategy`.

MRUCacheStrategy

De `MRUCacheStrategy` implementeert de *Most Recently Used* cache strategie. Hierbij worden de meest recent gebruikte entries eerst verwijderd. In het generieke model van de `CacheTable` komt het erop neer dat ingelezen of weggeschreven woorden achteraan in de lijst geplaatst worden. Op die manier worden de meeste recent gebruikte entries eerst verwijderd bij het ophalen van een nieuwe entry.

LRUCacheStrategy

De `LRUCacheStrategy` is de implementatie van de *Least Recently Used* cache strategie. In deze strategie worden de entries verwijderd die het langst onberoerd bleven. Concreet

worden in de `CacheTable` alle entries die opgevraagd of weggeschreven worden, vooraan in de lijst geplaatst.

NoCacheStrategy

De `NoCachingStrategy` schakelt de cache volledig uit. Telkens een aanvraag binnenkomt, wordt deze rechtstreeks tot de data-drager gericht. Er wordt ook geen prefetching gedaan.

5.2 Het adaptief model

Nu het bestandssysteem volledig is opgebouwd, wordt in deze sectie het adaptief beheer besproken. In navolging van sectie 2.3 worden achtereenvolgens de sensoren, de classificatie van het toegangspatroon, de selectie van de cache strategie en de toevoeging ervan aan de `CacheUnit` behandeld. Het volledig adaptief model is schematisch voorgesteld in figuur 5.5.

5.2.1 De sensoren

ReadWriteCounter

De `ReadWriteCounter` houdt bij hoeveel lees- en schrijfaanvragen er voorbij gekomen zijn sinds de vorige reset. Deze sensor (AS1 in figuur 5.5) wordt als *listener* ingeplugd in de `PacketReceiver`, horend bij de `CacheUnit`.

RequestSizeAnalyser

De `RequestSizeAnalyser` berekent de gemiddelde grootte van de aanvragen sinds de vorige reset. Net zoals de vorige sensor wordt ook deze analysesensor (AS2 in figuur 5.5) als *listener* ingeplugd in de `PacketReceiver`.

CacheHitRatio

De `CacheHitRatio` bevat informatie over het aantal *cache hits* en *cache misses* van het systeem. Een *cache hit* wordt geregistreerd als de opgevraagde woorden zich reeds in de cache bevinden, en dus niet meer bij de data-drager opgehaald moeten worden. Een *cache miss* wordt geregistreerd als de opgevraagde woorden zich niet in de cache bevinden.

In tegenstelling tot de vorige twee sensoren, bevindt deze toestandssensor zich in het bestandssysteem. Deze sensor (TS3) is toegankelijk via de `CacheSensorInterface` (CSI) van de `CacheUnit`, zoals getoond in figuur 5.5.

5.2.2 De classificatie van het toegangspatroon

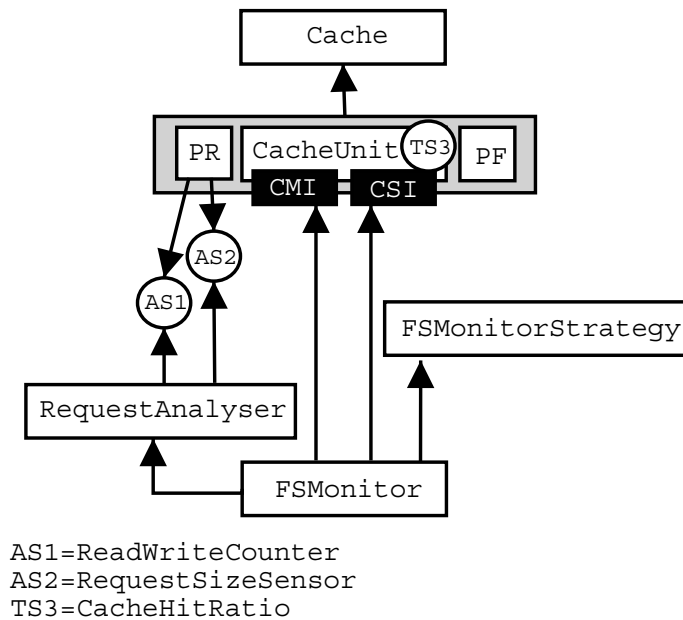
De classificatie heeft als doel het overheersende toegangspatroon herkennen in de boodschappenstroom. De `RequestAnalyser` neemt deze taak op zich. Via de `ReadWriteCounter` en de `RequestSizeAnalyser` verzamelt deze module voldoende gegevens om tot een eenduidige beschrijving te komen van het overheersende toegangspatroon. Deze beschrijving wordt dan verder in het beheersysteem gebruikt.

5.2.3 De monitor met inplugbare strategie

De `FSMonitor` haalt op regelmatige basis informatie op bij zijn `RequestAnalyser` en eventuele andere sensoren. De monitor verwerkt deze informatie tot zinvolle verbeteringen of aanpassingen aan het bestandssysteem (zoals het veranderen van de cache strategie, het aanpassen van de cache grootte, ...). Via de `CacheManagementInterface` van de `CacheUnit` kan de monitor deze wijzigingen doorvoeren in het bestandssysteem.

Zoals voorgesteld in hoofdstuk 3 is er een scheiding tussen het monitor gedeelte en de `FSMonitorStrategy`. Op die manier kunnen verschillende monitor strategieën gebruikmaken van dezelfde herbruikbare monitor.

De monitor bevat tevens de nodige methodes voor de gemakkelijke inschakeling van het beheer op een bestandssysteem. Via `setComponent` wordt het hele beheer vastgeklikt op een component, de methode `unsetComponent` haalt het beheer er terug zorgvuldig vanaf. In de volgende sectie wordt een specifieke monitor strategie in detail besproken.



Figuur 5.5: Het adaptieve beheer van het bestandssysteem

5.3 Toetsen van het systeem

In deze sectie wordt het adaptief bestandssysteem aan een test onderworpen. Eerst wordt de testopstelling beschreven, daarna wordt de gebruikte strategie toegelicht, om te besluiten met de resultaten van de test.

5.3.1 De testopstelling

De testopstelling ziet er als volgt uit. In het begin van de componentenketting staat een `RequestProducer` die aanvragen tot het bestandssysteem genereert. De `CacheUnit` met achterliggend bestandssysteem is iets verder in de ketting geplaatst. Verder zijn in de ketting nog een `TimeStampSetUnit` en `TimeStampDelayPrintUnit` aanwezig, respectievelijk voor en na het cache systeem. Deze laatste twee units berekenen hoe lang het bestandssysteem en de cache doen over de verwerking van elke vraag. Het zijn dan ook deze tijdsverschillen die terug te vinden zijn in de performantiegrafieken.

De testopstelling doorloopt vier fasen:

- fase 1: 30 leesaanvragen, lange aanvragen (gemiddeld 40 woorden elk)
- fase 2: 80 gemengde aanvragen, korte aanvragen (gemiddeld 4 woorden elk)
- fase 3: 20 leesaanvragen, lange aanvragen (gemiddeld 40 woorden elk)
- fase 4: 100 gemengde aanvragen, korte aanvragen (gemiddeld 4 woorden elk)

Voor de volledigheid zijn de verwerkingstijden van de gebruikte data-drager uitgezet in tabel 5.1.

	Leesaanvragen	Schrijfaanvragen
Opstarttijd	1000 ms	1000 ms
Tijd per woord	10 ms	100 ms

Tabel 5.1: Verwerkingstijden van de gebruikte data-drager

5.3.2 De gebruikte monitor strategie

In figuur 2.7 werd een voorbeeld gegeven van een cache selectie algoritme voor een adaptief bestandssysteem. Deze case study beperkt zich tot de implementatie van slechts het sequentiële deel van dit algoritme, zoals voorgesteld in figuur 5.6. Er worden enkel sequentiële aanvragen doorheen het testsysteem gestuurd.

Deze vereenvoudiging zorgt voor een gemakkelijkere classificatie van patronen. Er is immers geen sensor nodig die nagaat of de aanvragen al dan niet sequentieel zijn. Alle aanvragen zijn per definitie sequentieel.

Een voorbeeld van een eenvoudige monitor strategie wordt getoond in figuur 5.6.

```
if (read only && average request size > LARGE_REQUEST) {
    disable caching
} else {
    enable caching (LRU) + prefetch
}
```

Figuur 5.6: Voorbeeld van een eenvoudige monitor strategie

Read only, grote aanvragen

Figuur 5.7(a) stelt het eerste deel van de monitor strategie voor, het *read only* patroon met grote aanvragen. De verwerkingstijden voor veertig dergelijke aanvragen zijn er uitgezet voor de verschillende cache strategieën. Aangezien de aanvragen groter zijn dan de grootte van de cache (=32 woorden), kunnen de aanvragen best rechtstreeks aan de data-drager worden opgevraagd. De `NoCacheStrategy` ligt dan ook voor de hand. In de andere strategieën is de extra opstarttijd, nodig omdat de grote aanvraag in twee keer behandeld moet worden, duidelijk zichtbaar.

Overige aanvragen

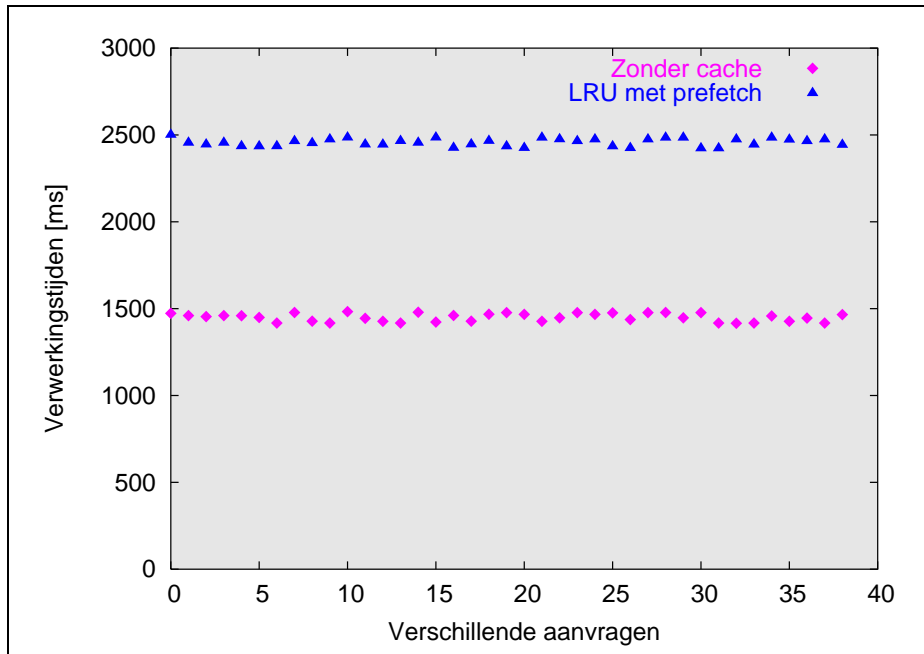
De overige patronen zijn voorgesteld in figuur 5.7(b). Hierbij wordt het meeste voordeel gehaald door gebruik te maken van de `LRUCacheStrategy`, in combinatie met `prefetch`.

5.3.3 De resultaten

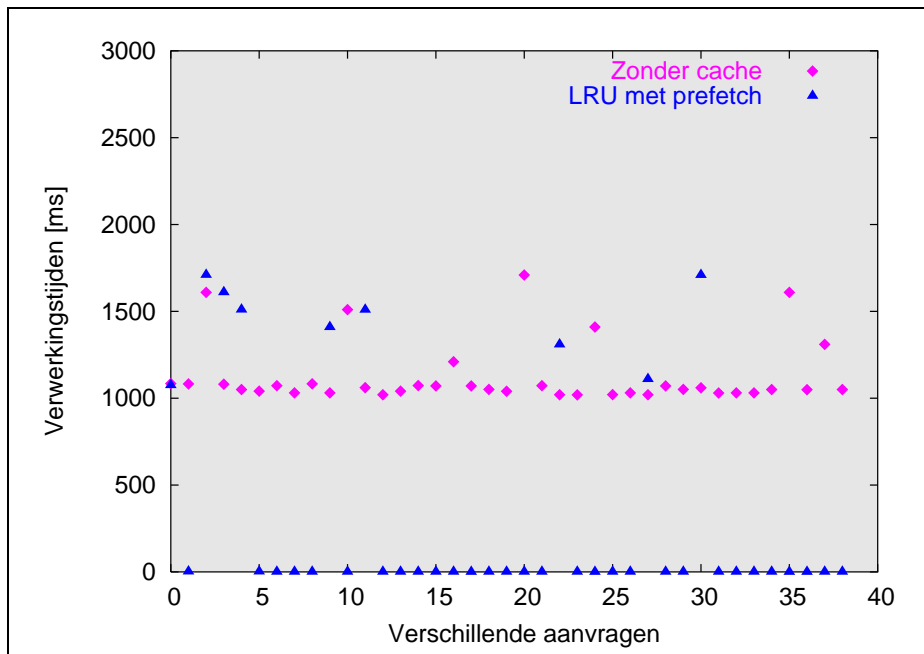
Het resultaat van deze test in het adaptief bestandssysteem staat uitgezet in figuur 5.8(a). In de eerste en derde fase zet het adaptief systeem automatisch zijn cache af, in de tweede en vierde fase wordt er een LRU cache strategie gebruikt. Op die manier wordt in elke fase de optimale strategie gebruikt en de laagst mogelijke verwerkingstijd verkregen (figuur 5.8(a)). Bij aanvang van elke fase is er een korte periode dat het systeem nog met een oude strategie werkt. Deze overgangsfase is duidelijk uit de grafiek af te lezen.

De verwerkingstijden tussen 1000 en 1500 ms voor de aanvragen in fase 1 en 3 in figuur 5.8(a) zijn het gevolg van een aantal schrijfaanvragen tussen de leesaanvragen.

Ter referentie is grafiek van dezelfde test (zonder monitor) met een statische LRU cache strategie en zonder cache uitgezet in figuur 5.8(b).

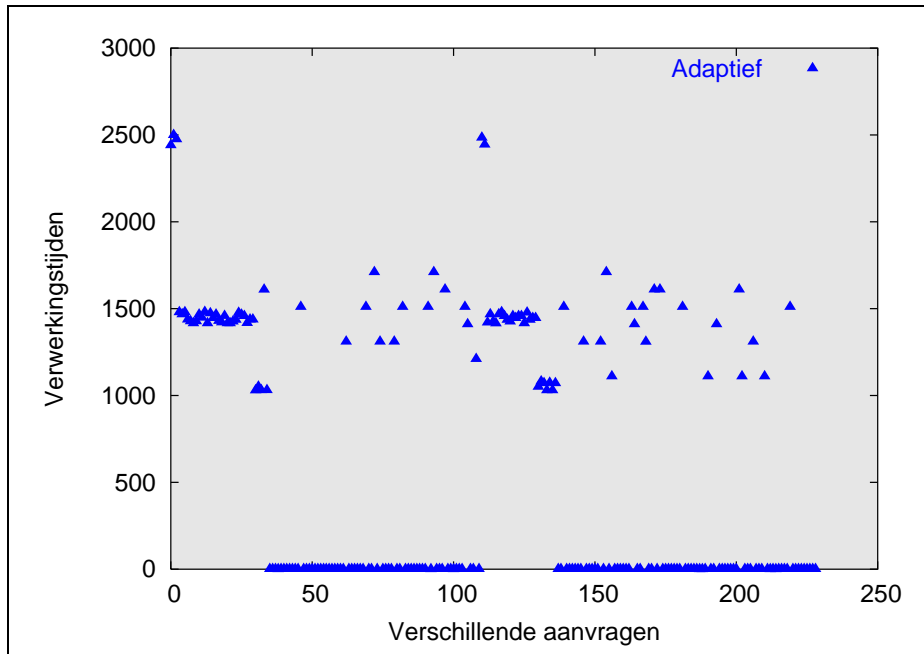


(a) Read only, grote aanvragen

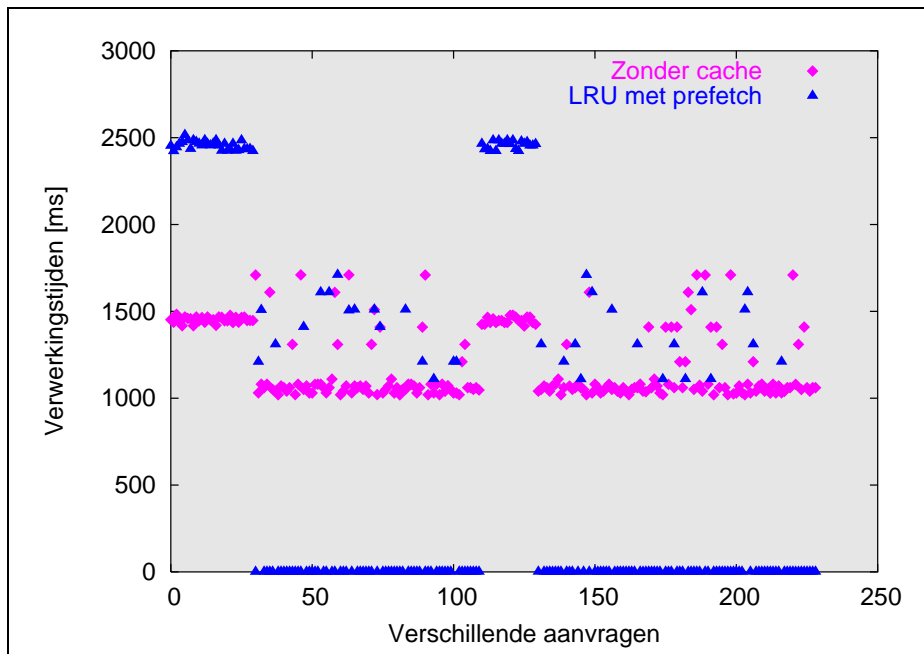


(b) Overige patronen

Figuur 5.7: Verschillende cache strategieën



(a) adaptief veranderende cache strategieën



(b) LRU cache strategie en zonder cache

Figuur 5.8: De testopstelling met verschillende de strategieën

5.4 Conclusie en het verder verloop

In dit hoofdstuk is het adaptief model uit hoofdstuk 3 getoetst aan de hand van een adaptief bestandssysteem. In een eerste stap werd een simpel en generiek bestandssysteem ontwikkeld. Daarna werd het adaptief model geconcretiseerd voor dit bestandssysteem en werd er een eenvoudige monitor strategie gebruikt. In het laatste deel van dit hoofdstuk werd een kleine testopstelling gemaakt met een veranderend toegangspatroon doorheen de tijd. Het adaptief bestandssysteem heeft de test goed doorstaan en veranderde voor elke fase naar zijn optimale cache strategie, met minimale verwerkingstijden. Dit toont duidelijk de voordelen aan van het gebruik van het adaptief model.

Verder zijn doorheen deze case study geen tekortkomingen opgemerkt aan het adaptief model. Het model is dus goed bevonden.

Deze case study is echter nog maar de eerste stap naar een adaptief bestandssysteem. Het aantal sensoren is beperkt, de strategie is sterk vereenvoudigd, de toegangspatronen zijn gemakkelijk te onderscheiden en de implementatie van het bestandssysteem op zich is zeker nog niet optimaal.

Een volgende stap op basis van deze case study kan bestaan uit het ontwikkelen van een intelligentere `RequestAnalyser` die bijvoorbeeld op basis van een neuraal netwerk ook in moeilijkere boodschappenstromen een juiste classificatie kan maken. Ook de monitor strategie kan een heel stuk intelligenter gemaakt worden.

Hoofdstuk 6

Besluit

In dit laatste hoofdstuk wordt eerst het algemeen adaptief model uit hoofdstuk 3 geëvalueerd. Daarna worden de testresultaten van de twee case studies nog kort nabesproken, om uiteindelijk te komen tot hoe het nu verder moet na deze thesis.

6.1 Evaluatie van het adaptief model

In de twee case studies werd het adaptief, modulair model uit hoofdstuk 3 geïmplementeerd. Dankzij de uitbreidingen die gedaan zijn aan DiPS, om te komen tot DiPS II, voldoet het opgebouwde model aan de doelstellingen van deze thesis.

Door gebruik te maken van een twee-lagen systeem, is er een duidelijk onderscheid tussen de functionele componentenketting en het beheersysteem. In de case studies werd ook aangetoond dat het beheer apart gebouwd wordt van de functionaliteit van de software. De beheermodules zijn extensies van het bestaande systeem, die gemakkelijk op de componentenketting kunnen worden ingeplugd, om er nadien even gemakkelijk terug afgehaald te worden.

Het beheersysteem is modulair opgebouwd. Naast een centrale monitor kunnen zoveel sensoren als nodig worden opgenomen in het beheer. Ook de complexiteit van de monitor is regelbaar via inplugbare monitor strategieën.

Inplugbare monitor strategieën zorgen ervoor dat de monitors gemakkelijk herbruikt kunnen worden voor de verschillende, mogelijke oplossingen. Generieke sensoren dragen eveneens bij tot de herbruikbaarheid van de beheercomponenten en tot de snelle ontwikkeltijd van nieuwe beheersystemen.

Wat wel nog ontbreekt in het adaptief model en de uitbreiding ervan met de meta-monitor, is de nodige synchronisatie tussen de meta-laag en de meta-meta-laag. Zonder een dergelijke synchronisatie kunnen de monitor en de meta-monitor gelijktijdig aanpassingen doorvoeren in het systeem, die elkaar tegenwerken, of elkaar teveel versterken. Ook de interpretatie van performantiegegevens is in een dergelijk geval gevaarlijk.

6.2 Testresultaten

In deze sectie worden heel kort de testresultaten van de twee case studies doorgenomen.

6.2.1 Dynamische parallellisatie

De resultaten van de testen uit sectie 4.3 tonen aan dat het adaptief systeem gepast reageert op veranderingen in de componentenketting. Het verkrijgt een ongeveer constante doorstroom die een stuk hoger ligt dan bij alternatieve gelijktijdigheidsmodellen, zelfs bij een piekbelasting.

Het adaptief systeem heeft wel nog last van schommelingen in de doorstroom. Aanpassingen aan de meta-monitor strategie zijn nodig om deze instabiliteiten verder weg te werken, en eventueel een nog iets hogere performantie te bereiken.

6.2.2 Adaptief bestandssysteem

In sectie 5.3 werd een prototype van een adaptief bestandssysteem getest. Uit de resultaten blijkt dat het beheersysteem in staat is de gepaste cache strategieën te activeren op basis van de boodschappenstroom.

De variatie in de gebruikte toegangspatronen is met opzet heel eenvoudig gehouden. Op deze manier was de classificatie van het overheersende patroon heel eenvoudig. Ook de bijhorende cache strategie selectie in de monitor was eerder aan de sobere kant. In een meer uitgewerkte versie zou voor de classificatie een neurale netwerk gebruikt kunnen worden en zou een veel uitgebreidere monitor strategie voorzien kunnen worden.

6.3 Verder verloop

Aan het adaptief model hoeft enkel nog de synchronisatie tussen de monitor en de meta-monitor te worden uitgewerkt. De rest van het model heeft de testen goed doorstaan.

Zoals in de vorige sectie gesuggereerd, moet er nog wat gesleuteld worden aan de monitor strategieën. De momenteel gebruikte strategieën zijn eerder prototypes van welke beslissingen het adaptief beheersysteem zoal kan nemen.

Wat ook belangrijk is in het verdere verloop, is het testen van het adaptieve model op reeds bestaande applicaties. De evaluaties, die in deze thesis gebeurd zijn, waren steeds op basis van zelfontworpen prototypes. De eenvoud en voorspelbaarheid van deze prototypes gaven de evaluaties misschien soms een iets vertekend beeld.

Bibliografie

- [1] Frank Matthijs. *Component framework technology for protocol stacks*. PhD thesis, Department of Computer Science, K.U.Leuven, Leuven, Belgium, December 1999. 221+vi pages.
- [2] Sam Michiels, Frank Matthijs, Dirk Walravens, and Pierre Verbaeten. DiPS: A Unifying Approach for Developing System Software. In A. D. Williams, editor, *Proceedings - The Eighth Workshop on Hot Topics in Operating Systems*, page 175. University of Karlsruhe, IEEE Computer Society, 2001.
- [3] Matt Welsh, David Culler, and Eric Brewer. Seda: An architecture for well-conditioned, scalable internet services.
- [4] M. Welsh. The staged event-driven architecture for highly concurrent server applications, 2000.
- [5] James C. Hu, Sumedh Mungee, and Douglas C. Schmidt. Techniques for developing and measuring high performance web servers over high speed ATM networks. In *INFOCOM (3)*, pages 1222–1231, 1998.
- [6] A. JAMES, N. NASA, A. Land, S. User, G. Distributed, A. Center, G. Space, and C. Greenbelt, 1994.
- [7] Tara M. Madhyastha, Christopher L. Elford, and Daniel A. Reed. Optimizing input/output using adaptive file system policies. In *Proceedings of the Fifth NASA Goddard Conference on Mass Storage Systems*, pages II:493–514, 1996.
- [8] Jay Huber, Christopher L. Elford, Daniel A. Reed, Andrew A. Chien, and David S. Blumenthal. PPFs: A high performance portable parallel file system. In *Proceedings of the 9th ACM International Conference on Supercomputing*, pages 385–394, Barcelona, 1995. ACM Press.