Universiteit Antwerpen

De evolutie naar Desktop Grids, problemen en oplossingen

Proefschrift voorgelegd op 03 december 2008 tot het behalen van de graad van Doctor in de Wetenschappen, richting Informatica, bij de faculteit wetenschappen, aan de Universiteit Antwerpen.

Promotoren:

prof. dr. Frans Arickx
prof. dr. Jan Broeckhove

Peter Hellinckx

CoMP Onderzoeksgroep Computationeel Modelleren en Programmeren

Copyright © 2008 Peter Hellinckx
Alle rechten voorbehouden.

Inhoudsopgave

Lijst van figuren **x**

Lijst van tabellen **xi**

1 Belangrijke afkortingen **1**

2 Voorwoord **3**

3 Inleiding **5**

I CoBRA **7**

4 Inleiding **9**

5 The Cobra grid: A highly configurable lightweight grid **13**

5.1 Inleiding 13
5.2 Structuur 15
5.3 Technische ruggengraat 19
5.4 Low Level Layer 24
5.5 CoBRA Middleware Layer 30
5.6 User Layer 35
5.7 Klasse Diagramma 42
5.8 Toepassingen, features en testen 43
5.9 Toekomst perspectief 63

6 Conclusion **65**

II GIPSy **67**

7 Inleiding **69**
7.1 Problematiek . . . 69
7.2 Doel . . . 70
7.3 Gerelateerd onderzoek . . . 71
7.4 Verloop hoofdstuk . . . 73

8 Predictie Techniek **75**
8.1 Inleiding . . . 75
8.2 Startgrid Selector . . . 76
8.3 Model builder . . . 77
8.4 Sample Selector . . . 78

9 Predictie Framework **81**
9.1 Inleiding . . . 81
9.2 Opzet . . . 81
9.3 Design . . . 88
9.4 Implementatie . . . 99
9.5 Extra Features . . . 135

10 Predictie Modellen **139**
10.1 Inleiding . . . 139
10.2 Gerelateerd onderzoek . . . 139
10.3 Iverson techniek . . . 148
10.4 Polynomiale Modellen . . . 150
10.5 Neurale Netwerken . . . 156
10.6 Support Vector Machines . . . 163
10.7 Radiale basis functies Modellen . . . 167
10.8 Kriging Modellen . . . 169
10.9 Naaste Buren Model . . . 170
10.10 Gecombineerde Modellen . . . 175
10.11 Gesplitte Modellen . . . 176

11 Startgrid Selector **179**
11.1 Inleiding . . . 179
11.2 Random Startgrid Selector . . . 179
11.3 Basic Startgrid Selector . . . 180

12 De Sampleselector **181**
12.1 Inleiding . . . 181
12.2 RandomSampleSelector . . . 182
12.3 ErrorSampleSelector . . . 183
12.4 MultiPredictorErrorSampleSelector . . . 183
12.5 DistanceSampleSelector . . . 184
12.6 DistanceAndErrorSampleSelector . . . 185

12.7 FluctuationSampleSelector . . . 185
12.8 SelectionNeighbourPenalisationSelector . . . 186
12.9 Universele ClusterPrevention . . . 187

13 De Kwaliteitscontrole **189**
13.1 Inleiding . . . 189
13.2 Foutmaten . . . 189
13.3 Implementatie . . . 193
13.4 Inputwaarden . . . 195
13.5 Besluit . . . 195

14 Testen **197**
14.1 Introductie . . . 197
14.2 Testtechniek . . . 197
14.3 Kwaliteitscontrole . . . 198
14.4 Testen en resultaten . . . 199
14.5 Conclusie . . . 212

15 Conclusie **213**

III Prediction Based Scheduling **215**

16 Inleiding **217**

17 PGS (Prediction based Grid Scheduling) **219**
17.1 Introductie . . . 219
17.2 Principe . . . 219
17.3 Implementatie . . . 224
17.4 Integratie van GIPSy . . . 226

18 Testen en resultaten **229**
18.1 Inleiding . . . 229
18.2 Testtechniek . . . 229
18.3 Testen en resultaten . . . 230
18.4 Relevantie van GIPSy . . . 238
18.5 Conclusie . . . 240

19 Conclusie **243**

IV Conclusie en toekomst perspectief **245**

20 Conclusie **247**

21 Toekomst perspectief **249**

Appendices **251**

Bijlage A H2O **253**
A.1 Inleiding . . . 253
A.2 Harness . . . 255
A.3 H2O . . . 260
A.4 Ondersteunende Concepten . . . 264
A.5 Operationeel Model . . . 279

Bijlage B Jini **283**
B.1 Inleiding . . . 283
B.2 Basis Principes . . . 287
B.3 Ondersteunende Concepten . . . 291
B.4 Operationeel Model . . . 325

Bijlage C CoBRA Voorbeelddistributie Output **327**

Bijlage D GIPSy Configfiles **329**
D.1 Het Gobale configuration object . . . 329
D.2 Het ErrorCalculator configuration object . . . 329
D.3 Het SampleSelector configuration object . . . 330
D.4 Het Model configuration object . . . 330
D.5 Het StartDataSelector configuration object . . . 330

Bijlage E Neurale Netwerk Software Libraries **331**
E.1 Yawn 2 . . . 331
E.2 JAS . . . 332
E.3 NeuralJ . . . 332
E.4 NNTK . . . 333
E.5 Boone . . . 334
E.6 SNARLI . . . 334
E.7 JCortex . . . 335
E.8 Joone . . . 336
E.9 JavaNNS . . . 337
E.10 Conclusie . . . 337

Lijst van figuren

4.1 Abstractielagen in distributed computing 9

5.1 De restricties binnen het CoBRA lagen model per gebruiker. Wit betekent alle rechten, licht grijs betekent dat deze code enkel uitgebreid mag worden, donkergrijs beperkt de gebruiker tot toegang via interfaces 17
5.2 Het globale operationele model van de CoBRA grid. 19
5.3 Het Jini Concept 22
5.4 CoBRA filesystem abstractie lagen 26
5.5 CoBRA filesystem klassen. 27
5.6 CoBRA socket filesystem klassen. 28
5.7 Uiteenzetting van de broker infrastructuur 31
5.8 Het Klassediagramma van de CoBRA grid. 42
5.9 Stuctuur van de economic scheduling techniek op de CoBRA grid. . 47
5.10 Een typisch distributie schema: x taken van lengte t_a worden verdeeld over y computers 50
5.11 Simulatie van de takenlijst uit figuur 5.10. 50
5.12 Uitvoering van de laatste taak wordt gespreid voor de berekening van de OH. 55
5.13 Mogelijke keuze van Jacobi coördinaten voor de ${}^3H{+}n{+}n$ configuratie in ${}^5H$. 57

7.1 Voorbeeld van finite state machine predictie. 73

8.1 Flowchart van de iteratieve opbouw van opeenvolgende modellen. . . 76
8.2 Vergelijking tussen de modellen gebouwd met twee verschillende startgrids (X-as = taak nummers, Y-as = tijd) 77
8.3 Vergelijking tussen 3 types modellen: Polynomiaal, Lineair en Exponentiëel (X-as = taak nummers, Y-as = tijd) 78
8.4 Vergelijking tussen een domme en een slimme sampleselector (X-as = taaknummers, Y-as = tijd). 80

9.1 De figuur toont dat de predictiecontainer zowel in het testframework als in een runtime prediction based scheduler moet kunnen geplugd worden. 83
9.2 Vergelijking tussen de tijdsafhankelijke en niet-tijdsafhankelijke versie (X-as = taak nummers, Y-as = tijd) 87
9.3 Toepassing van de plugable kwaliteitsbepaling op de sampleselection en de kwaliteitsbepaling. 89
9.4 Gebruiker georienteerde usecase van de ModelBuilder 91
9.5 Prediction user sequence diagram: Geeft de opeenvolgende stappen weer die een prediction user doorloopt bij het gebruik van de Modelbuilder. 92
9.6 Developer sequence diagram: Geeft de opeenvolgende stappen weer die een developer doorloopt bij het gebruik van de Modelbuilder. . . 93
9.7 Use case die de Prediction configuration in meer detail beschrijft. . . 98
9.8 Algemeen Klassediagram 99
9.9 Overzicht van de configuratiestructuur 101
9.10 Opeenvolgende stappen tijdens de configuratie van een component . 105
9.11 Een (Kriging) model opgebouwd aan de hand van de blauwe samplepunten 121
9.12 Opeenvolgende stappen tijdens de configuratie van een predictor. (Let op de verschillen met figuur 9.10). 121
9.13 Een afbeelding van de Manuele Interface. 136
9.14 Een afbeelding van de Automatische Interface. 137

10.1 Een voorbeeld van een transitiediagram uit paper [28]. 141
10.2 Artificieel neuron 157
10.3 Plots van verschillende activatiefuncties. 158
10.4 Voorstelling van de XOR ruimte 159
10.5 Verschillende soorten netwerken 159
10.6 Een lager dimensionale ruimte waarin lineaire classificatie niet mogelijk is wordt afgebeeld op een feature-space (hoger dimensionale ruimte) waarin lineaire classificatie wel mogelijk is. 164
10.7 Een classificatie aan de hand van een hypervlak(een rechte) in een 2-dimensionale ruimte 165
10.8 Een hypervlak wordt bepaald aan de hand van de support vectors aangeduid met de vette letters. 166
10.9 Voorbeeld van de lokale minima en maxima in een interpolatie tussen ruisdata. 176
10.10Een gesplit model van een inputruimte met ruisdimensie Y. 177

12.1 Gewichten bepaling door verschil tussen twee predictor curves. . . . 183

13.1 Deze figuur toont de plaats van de globale en lokale kwaliteitscontroles binnen het plugable framework. 191

14.1 Vergelijking van modelleringstechnieken aan de hand van een selectie *eup* plots voor de quantum fysica testset. 202
14.2 Vergelijking van modelleringstechnieken aan de hand van een selectie *eup* plots voor testset 1. 203
14.3 Vergelijking van modelleringstechnieken aan de hand van een selectie *eup* plots voor testset 2. 204
14.4 Vergelijking van sampleselectors aan de hand van een selectie *eup* plots voor de quantum fysica testset. 206
14.5 Vergelijking van sampleselectors aan de hand van een selectie *eup* plots voor testset 1. 207
14.6 Vergelijking van sampleselectors aan de hand van een selectie *eup* plots voor testset 2. 208
14.7 De minimale, de gemiddelde en de maximale tijd nodig om een model te construeren en de samples te selecteren voor de quantum fysica testset. 209
14.8 De minimale, de gemiddelde en de maximale tijd nodig om een model te construeren en de samples te selecteren voor de testset 1. 210
14.9 De minimale, de gemiddelde en de maximale tijd nodig om een model te construeren en de samples te selecteren voor de testset 2. 210

17.1 De opstelling van de verschillende queues binnen de PGS scheduler. 220
17.2 De availability windows worden gesorteerd volgens grootte. 221
17.3 Er wordt een Job geplaatst op elke QpR in de Scheduler. 222
17.4 Er wordt sequentieel voor elke job een plaats gezocht in een availability window volgens de gesorteerde window lijst. 223
17.5 Jobs in de QpRs worden verplaatst naar lege queues om aldus de load te balancen. 223
17.6 Vereenvoudigd model van het PGS klassediagram. 225
17.7 Integratie van CoBRA, PGS en GIPSy. 227

18.1 Testresultaten met betrekking tot de distributie van het quantum fysica probleem. 232
18.2 Testresultaten met betrekking tot testset 1. 233
18.3 Testresultaten met betrekking tot testset 2. 234
18.4 De verhouding standaard afwijking / gemiddelde runtime. 237
18.5 Verschil in eup spreiding tussen smallest distance sampleselector (SDSS) en largest distance sampleselector (LDSS). 240

A.1 Het Harness lagen model [64] 256
A.2 De services, Resource providers en clients binnen het H2O framework [112]. 262
A.3 Verschil tussen het traditionele container model en het H2O container model [112]. 263
A.4 DVM enabeling components [112] 264
A.5 Interoperability in RMIX [65] 265

A.6 Verschillende aanpasbare endpoints [65] 268
A.7 De RMIX oproep stack [66] 268
A.8 Client - Kernel interactie [112] 271
A.9 Pluglet Interfaces [112] 272
A.10 Global en pluglet-related policies 275
A.11 H2O gebruik scenario's 282

B.1 Het Jini Concept 285
B.2 Java vs. Jini 287
B.3 Jini Concepten 288
B.4 Het Jeri Lagen Model 299
B.5 Local proxy design 300
B.6 Forwarding proxy design 301
B.7 Smart proxy design 301
B.8 Het landlord leasing principe 306
B.9 Proxy betrouwbaarheidsverificatie 314
B.10 Het Operationeel Model 326

Lijst van tabellen

5.1 Code access rights of user types 18
5.2 De overhead bij de distributie van T taken van lengte l over R resources. 52
5.3 De speedup bij de distributie van T taken van lengte l over R resources. 53
5.4 Het percentage overhead POH bij de distributie van T taken van lengte l over R resources. 54
5.5 Eigenschappen van de drie gebruikte testsets. 54
5.6 Het percentage overhead POH bij de distributie van T taken van lengte l over R resources. 55
5.7 Het percentage overhead POH bij de distributie van T taken van lengte l over R resources. 55
5.8 Specificaties van de quantum fysische distributie volgens parameter K. 61
5.9 De Specificaties van de verschillende gebruikte workers. 61
5.10 Resultaten van de quantum fysica test. 62

10.1 Transitietabel 140
10.2 Informatie omtrent verschillende activatiefuncties. 158

12.1 Tabel van de model optimalisatie sampleselectors en hun inverse sampleselctors die de runtime predictie van de geselecteerde punten optimaliseert. 182

14.1 Eigenschappen van de drie gebruikte testsets. 199

18.1 Eigenschappen van de drie gebruikte tests. 230

HOOFDSTUK 1

Belangrijke afkortingen

ANN	Artificiëel Neuraal Netwerk
API	Application Programming Interface
BOINC	Berkeley Open Infrastructure for Network Computing
BoT	Bag of Task
CoBRA grid	Computational Basic Reprogrammable Adaptive grid
Exp	Exponentiëel
FlSS	Fluctuation SampleSelector
GIPSy	Grid Information Prediction System
HL-LWG	High Level Light Weight Grid
HTTP	Hypertext Transfer Protocol
IIOP	Internet Inter-ORB Protocol
IMQ	Inverse MultiQuadric
JAMA	Java Matrix Package
JAR	Java ARchive
JERI	Jini Extensible Remote Invocation
k-NN	k Nearest Neighbors
LDSS	Largest Distance SampleSelector
LL-LWG	Low Level Light Weight Grid
LWG	Light Weight Grid
MPI	Message Passing Interface
MQ	MultiQuadric
NN	Neuraal Netwerk
NWS	Network Weather Service
PGS	Prediction based Grid Schedudeler
PBS	Prediction based Scheduler
QF test	QuantumFysica test
QpR	Queue per Resource
RaSS	Random SampleSelector

RBF	Radial basis function
RMI	Remote Method Invocation
RPC	Remote Procedure Call
SDSS	Smallest Distance SampleSelector
SETI	Search for Extraterrestrial Intelligence
SOAP	Simple Object Access Protocol
SoSS	Sorted SampleSelector
SSH	Secure Shell
SVM	Support Vector Machines
UML	Unified Modeling Language
URI	Uniform Resource Identifier
URL	Uniform Resource Locator
XML	eXtensible Markup Language
ZIP	Zone Information Protocol

HOOFDSTUK 2

Voorwoord

Het behalen van een doctoraat is een veelzijdige opdracht, die op verschillende vlakken, tal van uitdagingen en problemen met zich meebrengt. Het volbrengen van deze uitdagingen en het oplossen van deze problemen zijn zelden een “one man effort”. Bij deze zou ik dan ook graag de verschillende personen, die aan mijn zijde hebben gestaan om dit werk te verwezenlijken, willen danken.

Op professioneel gebied denk ik hier aan mijn Promotoren Prof. Dr. Frans Arickx en Prof. Dr. Jan Broeckhove en aan Prof Dr. Tom Dhaene, lid van mijn doctoraatscommissie. Naast de kans die ze mij hebben gegeven dit doctoraat aan te vatten, hebben ze mij ook steeds bijgestaan met goede raad en met suggesties om mijn onderzoek in de goede richting te sturen. Verder wil ik ook de collega’s van de CoMP onderzoeksgroep danken. Ik denk hier in de eerste plaats aan Kurt Van Mechelen die steeds klaar stond met deskundig advies. Hiernaast mag ik zeker Sam Verboven en Wouter Hendrickx niet vergeten die, naast hun research inspanningen, ook al mijn professionele tegenslagen en frustraties te verwerken kregen.

Naast mijn collega’s, werd ik eveneens bijgestaan door vrienden en familie. Ik denk hier aan mijn vriendin, Delphine Draelants, mijn vader, Daniel Hellinckx en zijn vrouw, Berna Van Meensel. Ik wil hen danken voor de steun die ik van hen kreeg op momenten van tegenslagen en voor de moed die ze mij steeds gaven om verder te gaan. Tevens wil ik speciaal de aandacht vestigen op mijn nonkel, Jan Vogels, die met een enorm engelengeduld de literaire kwaliteit heeft bijgeschaafd.

Tot slot, wil ik heel graag mijn overleden moeder, Maria Vogels, danken. Haar participatie in mijn opvoeding heeft zeker een groot stuk bijgedragen tot de resultaten die ik vandaag bereikt heb.

Bedankt Iedereen!

HOOFDSTUK 3

Inleiding

Reeds sinds de ingebruikname van de eerste computers leefde bij wetenschappers de wens voor meer computationele performantie. In de eerste decennia zorgde deze drang voor een exponentiële toename van het geheugen en van de rekenkracht. Dit leidde tot de hedendaagse supercomputers. Deze evolutie kon de vraag naar resources echter niet invullen. Verder onderzoek toonde aan dat het bundelen van performante resources binnen clusters een mogelijke oplossing kon bieden. Meer recent werd dit principe uitgebreid door ook deze clusters te bundelen in wereldwijd verspreide dedicated grid systemen. Nu de resources van deze laatste systemen ook niet aan de vraag kunnen voldoen, proberen de zogenaamde *desktop grid* systemen een andere bron van resources aan te spreken. De doelstelling van deze systemen is ongebruikte resources van een klassieke desktop gebruiker te bundelen. Het gaat hier dan voornamelijk over personal computers, die voor privé doeleinden gebuikt worden, en over computers in openbare instellingen, scholen en bedrijven. Deze evolutie zorgt ervoor dat steeds meer en goedkopere rekenkracht beschikbaar kan worden gemaakt. Zoals elke evolutie gaat ook deze gepaard met specifieke uitdagingen en problemen. Deze thesis heeft tot doel drie dergelijke specifieke uitdagingen aan te gaan.

De eerste uitdaging kan best omschreven worden als een zoektocht naar een voedingsbodem voor desktop grid research. Concreet dient een methode ontwikkeld te worden die het voor grid researchers eenvoudig maakt nieuwe technieken te ontwikkelen en uit te testen op een realistische desktop grid. Om dit te verwezenlijken, wordt in deze thesis de CoBRA (Computational Basic Reprogrammable Adaptive) grid geïntroduceerd. Dit framework tracht, aan de hand van zijn pluggability eigenschappen, een synergie te creëren tussen de grid gebruiker en de grid researcher om aldus zowel de research als het gebruik te optimaliseren. In essentie komt het erop neer dat een grid systeem ontwikkeld dient te worden dat, op universitair niveau, eenvoudig voor distributiedoeleinden gebruikt kan worden en dat tezelfdertijd door

grid researchers als testbed aangewend kan worden. Op deze manier kunnen de gebruikers nauw samen werken met de onderzoekers om zo vitale informatie uit te wisselen. De ware toedracht van deze uitdaging wordt uitgebreid beschreven in deel I van deze thesis.

Vele technieken die aangewend worden binnen zowel desktop grid systemen als binnen andere grid systemen, berusten op de kennis van de Job runtimes. Ze gaan er aldus vanuit dat de uitvoertijd van een taak vooraf gekend is. Dit is echter een valse veronderstelling. Deze informatie kan enkel verschaft worden door de gebruiker, en de betrouwbaarheid ervan is niet steeds gegarandeerd. De tweede uitdaging bestaat erin aan te tonen dat dergelijke runtimes voorspeld kunnen worden voor parameter sweep distributies[1]. Om deze doelstelling te bereiken, wordt het plugable GIPSy(Grid Information Prediction System) framework geïntroduceerd. Het opzet van het framework is, aan de hand van de pluggability, een gebruiker de mogelijkheid te bieden elk onderdeel van de runtime predicties te vervangen door een alternatieve implementatie. Dit om op deze manier het perfecte predictiemechanisme te ontwikkelen. Dit framework, de gebruikte technieken en een concrete predictieopstelling worden in detail besproken in deel II van deze thesis.

Het grote verschil tussen desktop grids en dedicated grids is de eigenaar van de resources. In tegenstelling tot de situatie bij dedicated grids, zal de eigenaar van een resource bij de desktop grid deze resource niet ten allen tijde ter beschikking stellen. De resource zal immers niet langer beschikbaar zijn wanneer de eigenaar deze machine zelf wenst te gebruiken. Dit verschil heeft enorme consequenties [62]. Er moet in dit geval ten allen tijde vermeden worden dat een Job op een resource uitgevoerd wordt wanneer de eigenaar deze machine zelf wil gebruiken. De Job moet dan immers gestopt worden en het reeds uitgevoerde werk zal bijgevolg verloren gaan. Het verdelen van de Jobs over de resources rekening houdend met zowel de job runtimes als met het availability window van de rersource vormt de derde uitdaging van dit doctoraat. Om deze doelstelling te verwezenlijken, wordt in deze thesis het Prediction Based Scheduling (PBS) framework voorgesteld. Gebruik makend van de CoBRA grid, wordt aangetoond dat deze techniek op basis van de GIPSy voorspellingen het falen van jobs kan vermijden. De concrete uitwerking van dit concept is terug te vinden in deel III van deze thesis.

[1]Een parameter sweep distributie is een bag of task distributie waarin elke taak dezelfde opdracht moeten uitvoeren maar met een verschillende inputwaarde voor elk van de inputparameters.

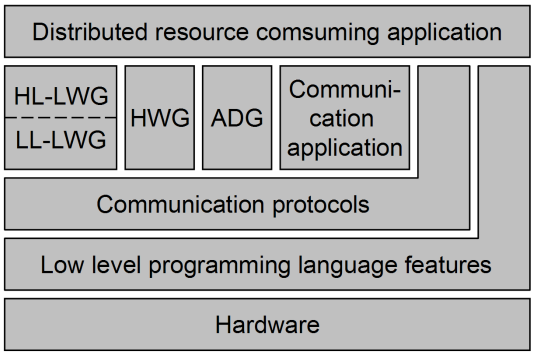
Deel I

CoBRA

HOOFDSTUK 4

Inleiding

De verschillende technieken die aangewend worden in distributed computing, kunnen onderverdeeld worden in verschillende abstractie niveaus bovenop de heterogene hardware. Dit alles wordt duidelijk aan de hand van figuur 4.1.



Figuur 4.1: Abstractielagen in distributed computing

De onderste abstractielaag krijgt de naam *low level programming language features*. De laag bestaat uit technieken, zoals sockets, die raw data tussen de verschillende resources transfereren.

De *Communication protocols* laag, die zich bovenop deze eerste laag bevindt, heeft tot doel betekenis en structuur in de getransfereerde ruwe data te brengen. De protocols zoals SOAP [123], HTTP [7], IIOP [8], MPI [110] en ssh [10], die in deze laag verzameld zitten, ondersteunen het transport van getypeerde data. Ze bieden, met andere woorden, de mogelijkheid basistypes, zoals Integers en Reals, tot zelfs uitgebreide datatypes, zonder verlies van betekenis, over het netwerk te transfereren.

Bovenop deze twee low level communicatie mechanismen werden tal van gedistribueerde abstracties ontwikkeld. De belangrijkste vier voor dit onderzoek zijn:

- De *communication applications* zoals ssh clients. Deze applicaties worden voornamelijk gebruikt door script georiënteerde gedistribueerde applicaties. De communicatie applicaties worden door de scripts gebruikt voor zowel de communicatie als de data transfer tussen de verschillende participerende resources.

- De gedistribueerde applicaties worden de *application dependent grids* of *ADG* genoemd. Deze verzameling bestaat uit applicaties die gespecialiseerd zijn in de uitvoer van één specifiek type van gedistribueerde problemen. Zo zijn 'APST' [24][2] en 'Nimrod' [9] voorbeelden van gedistribueerde applicaties die speciaal ontwikkeld zijn met het doel parameter sweeps te distribueren.

- De *Heavyweight Grids* of *HWGs*. De kracht van deze heavyweight grids gaat uit van vier belangrijke eigenschappen:

  - Robustness: Bestand tegen onnatuurlijk gebruik door derden (foute input, slecht werkende jobs, ...).
  - Reliability: Garandeert de afhandeling van taken (Er gaat niets verloren).
  - Scalability: Elke hoeveelheid van resources kan ondersteund worden zonder dat het systeem hier nadelige gevolgen van ondervindt.
  - Monitoring: Het is mogelijk de verschillende entiteiten in de grid (Resources, Jobs, ...) te monitoren.

  Deze vier eigenschappen bieden de HWGs de mogelijkheid grote hoeveelheden resources te groeperen. De HWG's hebben dan ook tot doel enorme hoeveelheden computationele kracht, opslagcapaciteit, ... aan te bieden. Projecten zoals Globus [34, 49, 6] (gebruikt door EGEE [4], ShanghaiGrid [68], ...) of BOINC [20] (gebruikt door Seti@Home [15, 102], Folding@home [5]) bewijzen dit. Het grote nadeel van de HWG's zijn de deployment[1] en de onderhoudskost. Het installeren of het updaten van dergelijke systemen vergt veel manuren en system down time.

- De *LightWeight Grids* of *LWGs* trachten een lage deployment kost te combineren met de krachtpunten van HWG's. Er kunnen twee types LWGs onderscheiden worden:

  - De *Low Level LWGs* of *LL-LWGs* focussen op het scalability krachtpunt van de HWGs.
  - De *High Level LWGs* (HL-LWG) trachten de robustness, de reliability en de monitoring krachtpunten van de HWG's te ondersteunen.

  LL-LWGs, zoals H2O [42] en Jini [30, 57], kunnen enorme hoeveelheden resources verbinden. Ze zijn echter niet in de mogelijkheid deze resources te monitoren en robustness/reliability te garanderen. HL-LWGs zoals Condor

[1] Het ter beschikking stellen van de component voor het gebruik door derden.

[27] zijn robust, reliable en eenvoudig te monitoren maar moeten trucs, zoals flocking[2] gebruiken om aanvaardbare hoeveelheden resources te verzamelen.

De verschillende gedistribueerde applicaties worden gegroepeerd in de *Distributed resource consuming layer*. Aan de hand van een command line interface of van programmeertaal libraries worden deze programma's opgesplitst in een grote groep van subtaken, die verdeeld worden over de verschillende resources. Dit gebruik makend van een techniek uit de onderliggende lagen die voldoet aan de vereisten van de gebruiker.

Het doel van dit hoofdstuk is de ontwikkeling van een grid systeem dat rekening houdt met de noden van zowel een grid researcher als van een standaard grid gebruiker. Een standaard grid gebruiker streeft vooral naar eenvoud in de implementatie van het probleem en naar snelheid in het uitvoeren ervan. Een grid developer wil dan weer eenvoudig elke component van het systeem kunnen uitbreiden en uittesten en dit dan nog liefst wanneer een grid gebruiker een reëel probleem aan het uitvoeren is. Deze vereisten kunnen niet ingevuld worden met de grid systemen die momenteel voorhanden zijn. De monitoring en de robuustheidsvereisten die essentieel zijn om dit doel te bereiken, worden enkel ondersteund door de HWGs en de HL-LWG's. De flexibiliteitsvereisten en de aanpasbaarheidvereisten, die even belangrijk zijn in deze doelstelling, zijn enkel terug te vinden in de LL-LWG categorie. Om deze doelstelling alsnog te bewerkstelligen, wordt in dit hoofdstuk het CoBRA (Computational Basic Reprogrammable Adaptive) grid framework voorgesteld. Dit framework ontwikkelt een robuust gridsysteem bovenop een bestaande LL-LWG, zonder de flexibiliteit en de aanpasbaarheid van deze LL-LWG op te offeren.

[2]De resourcemanager van een pool kan, aan de hand van flocking, jobs versturen naar de resourcemanager van een andere pool om daar de jobs te laten uitvoeren.

HOOFDSTUK 5

The Cobra grid: A highly configurable lightweight grid

5.1 Inleiding

In dit hoofdstuk wordt een volledige uiteenzetting gegeven omtrent de doelstellingen, de principes, de opbouw en het uiteindelijk gebruik van de CoBRA (Computational Basic Reprogrammable Adaptive) grid. Het idee omtrent de CoBRA grid is ontstaan uit de afwezigheid van een gedistribueerd framework waarbinnen gebruikers[1] alsook developers van gedistribueerde componenten[2] op eenzelfde framework kunnen werken en ervaringen kunnen uitwisselen. Dit om enerzijds de developers de kans te geven de grid aan te passen aan de wensen/noden en de opmerkingen van de gebruikers, alsook om anderzijds de mogelijkheid te bieden nieuwe technieken te testen op een gedistribueerd systeem met een realistische load. Om dit te verwezenlijken, is de CoBRA grid gebaseerd op drie pijlers:

1. *Distributie*: De eerste pijler, de distributie pijler, staaft dat de kwalitatieve waarde van het product naar de gebruiker toe gegarandeerd moet zijn. Dit houdt in dat het voor elk type gebruiker eenvoudig moet zijn, een distribueerbaar probleem te deployen op de CoBRA grid. Deze eis omvat drie belangrijke onderdelen:

    - Het moet voor de gebruiker eenvoudig zijn, zijn gedistribueerd probleem te specificeren zodat het gedeployed kan worden binnen het CoBRA framework. Concreet komt dit erop neer dat de helling van de leercurve aangaande CoBRA distributie, in de mate van het mogelijke, afgezwakt dient te worden en dat het geheel als een transparante werkomgeving moet ervaren worden.

[1] De individuen die problemen willen distribueren.
[2] De individuen die onderzoek doen naar de ontwikkeling van gedistribueerde technieken.

- Het deployen van het probleem moet eenvoudig en intuïtief zijn.
- De afhandeling van dit gedistribueerd probleem moet op een performante en een correcte wijze gebeuren.

2. *Onderzoek*: Naast de distributievereisten, die in principe voor elk gedistribueerd systeem van toepassing zijn, definieert deze tweede pijler de nood aan onderzoeksmogelijkheden. De toegevoegde waarde van de CoBRA grid moet erin bestaan een gridonderzoeker de mogelijkheid te bieden, op een eenvoudige wijze, zijn eigen implementatie van een van de reeds bestaande componenten, of van nieuwe componenten, in te pluggen in een reeds actieve grid. Dit kan het best geschetst worden aan de hand van een voorbeeld. Voor een developer, die bijvoorbeeld een `Resource` component[3] uitbreidt, zodat deze in de mogelijkheid is jobs af te handelen op BEGrid, moet het mogelijk zijn deze resource onmiddellijk te gebruiken in een reeds actieve CoBRA grid.

3. *Synergie*: Naast de voorgaande twee pijlers, die in principe de vereisten van de CoBRA grid beschrijven, definieert deze derde pijler het uiteindelijke opzet van het gehele systeem. *Synergie* betekent volgens Van Dale:

   > *Een situatie waarin het effect van twee of meer samenwerkende of gecombineerde organen of functies groter is dan de som van de effecten die elk van de organen of functies alleen zou kunnen opwekken.*

   Concreet betekent dit, dat het design van CoBRA de samenwerking tussen de verschillende participerende entiteiten van de CoBRA grid (de gebruikers en de developers) in dergelijke mate moet versterken, dat er een rechtstreekse wisselwerking ontstaat tussen de bevindingen van de gebruikers en de optimalisaties/aanpassingen van de developers. Dit heeft voor de gebruiker tot gevolg dat de implementatie procedure zal vereenvoudigen omdat:

   - De drempel om vragen te stellen aan de developers verkleint.
   - De transparantie van het framework geoptimaliseerd is volgens de noden van de gebruiker.

   Dit heeft voor de developer tot gevolg dat de verschillende componenten sneller en beter geïmplementeerd kunnen worden omdat:

   - Er rechtstreekse feedback is van de gebruiker.
   - Nieuwe componenten onmiddellijk getest kunnen worden in een live systeem.

Om dit alles te verwezenlijken, is de CoBRA grid ontwikkeld. De CoBRA grid kan het best omschreven worden aan de hand van volgende definitie die, gedurende de ontwikkeling van het framework, stapsgewijs, tot stand is gekomen:

[3] Een component die instaat voor de afhandeling van een job.

> *A portable generalized plug-in middleware for Desktop computing that allows developers to dynamically adapt and extend the distributed computing components without compromising the functionality, transparancy, consistency and robustness of the deployed framework.*

De essentie van deze definitie weerspiegelt de eerste twee pijlers: *distributie* en *onderzoek* en resulteert onvoorwaardelijk in de gewenste synergie tussen developer en gebruiker.

Omwille van de grote omvang van dit project en van de verwevenheid van de verschillende topics in dit hoofdstuk, zal dit hoofdstuk ietwat speciaal doch doordacht opgebouwd zijn. In de eerste sectie (sectie 5.2) wordt een globaal oppervlakkig overzicht gegeven van het volledige project. Elk van de verschillende basiscomponenten en van de technieken hierin zullen geplaatst worden in de globale structuur van het framework. De volgende vier hoofdstukken, respectievelijk *De Technische ruggengraat, De Low Level Layer, De Middleware Layer* en *De Userlayer*, bespreken elk van deze componenten en technieken in meer detail. Er is bewust gekozen deze uiteenzetting vanuit de laagste laag naar de hoogste laag op te bouwen. Op deze manier wordt het mogelijk gemaakt elk van deze componenten te plaatsten binnen het algemene operationele model uit sectie 5.2.3. Deze uiteenzetting wordt afgesloten met de bespreking van het globale klassendiagramma in sectie 5.7. In sectie 5.8 wordt het framework grondig getest door zowel toepassingen als extra features te ontwikkelen en uit te testen. Tot slot bespreekt sectie 5.9 enkele zwakke punten van het CoBRA framework en de manier waarop deze naar de toekomst toe verholpen kunnen worden.

5.2 Structuur

5.2.1 Introductie

Dit hoofdstuk geeft een globaal overzicht van de structuur van de CoBRA grid. Dit met als doelstelling de meer specifieke details te kunnen benaderen vanuit een reeds verworven globaal overzicht en deze zo beter te begrijpen. Dit overzicht wordt geschetst vanuit twee invalshoeken. De eerste invalshoek belicht de diversiteit tussen de verschillende gebruikers en de repercussies hiervan op de structuur. De tweede invalshoek geeft een globaal overzicht aan de hand van een oppervlakkig operationeel model.

5.2.2 Usertype gespecificeerde transparantie lagen

Zoals reeds werd aangeduid in de inleiding van hoofdstuk 5, is de voornaamste doelstelling van de CoBRA grid een synergie creëren tussen de grid developer en de grid gebruiker, en zo de leercurve voor de gebruiker afvlakt. Dit wordt bewerkstelligd door developer en gebruiker dichter bij elkaar te brengen en ze te laten samenwerken op dezelfde grid opstelling. Dit nauwe contact brengt met zich mee dat:

- de gebruiksvriendelijkheid van het systeem verbetert door directe reflectie van de gebruiker naar de developer.

- de impact van gebruikersproblemen verkleint door eenvoudige en snellere remediëring door de developers.

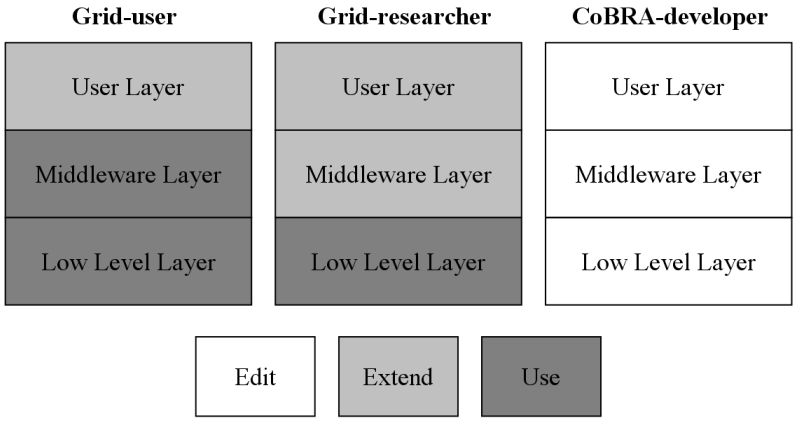
Wanneer de participerende groepen van nabij bekeken worden, is het duidelijk dat naast de standaard gebruikersgroep, de griddevelopergroep opgesplitst kan worden in twee deelgroepen: de Grid-researchers enerzijds en de CoBRA-developers anderzijds. De CoBRA-grid community bestaat dus, in essentie, uit drie types participanten met elk een specifieke doelstelling:

- *Grid-gebruiker*: Dit type groepeert alle individuen die, gebruik makend van de CoBRA grid, gedistribueerd resources willen benutten (bv. ontwikkelaar van het gedistribueerde quantum fysica probleem, beschreven in sectie 5.8.8).

- *Grid-researcher*: Deze groep van participanten richt zich op grid/distributie gerelateerd onderzoek. De CoBRA grid dient als solide basis ter realisatie en ter verificatie van "proof of concept" implementaties. Voorbeelden hiervan zijn onder andere de developers van projecten zoals de BEGrid-portal (zie sectie 5.8.6) of de Runtime Prediction Scheduler (zie sectie 5.8.5).

- *CoBRA-developer*: De CoBRA developers ontwikkelen, verbeteren en onderhouden de essentiële componenten van de CoBRA implementatie, zoals de onderliggende backbone laag, de data transfer module en de notification infrastructuur alsook de componenten van de basisimplementatie van de CoBRA grid. Deze basisimplementatie verzorgt een werkende grid infrastructuur die distributie mogelijk maakt, maar dient essentieel om door de developers uitgebreid te worden om aldus extra functionaliteit te verwezenlijken.

De introductie van verschillende types gebruikers heeft een administratieve meerkost als logische repercussie. In traditionele systemen uit dit zich in authenticatie en in authorisatie mechanismen. Dit gaat in de meeste gevallen ten koste van performantie en van gebruiksvriendelijkheid. Aangezien deze repercussies niet compatibel zijn met de vooropgestelde streefdoelen, wordt deze problematiek op een meer efficiënte manier opgelost. Het gebruik van de Java programmeertaal maakt het mogelijk, aan de hand van inheritance en overloading, elk type gebruiker slechts beperkte rechten te geven door hem enkel te voorzien van een beperkte subset van classes en interfaces. Concreet betekent dit dat dit lagenmodel gebruikt zal worden om verschillende types gebruikers af te schermen van de voor hen ontoegankelijke onderdelen van het framework. De verschillende lagen worden aldus de *Transparantielagen* genoemd. Om dit mechanisme te laten werken, dient de code opgesplitst te worden in verschillende lagen. Elk type gebruiker zal dan een beperkte subset van deze lagen ter beschikking hebben, de onderliggende lagen zijn bijgevolg onzichtbaar voor hem. Concreet wordt de code opgesplitst in de drie lagen die terug te vinden zijn in figuur 5.1:



Figuur 5.1: De restricties binnen het CoBRA lagen model per gebruiker. Wit betekent alle rechten, licht grijs betekent dat deze code enkel uitgebreid mag worden, donkergrijs beperkt de gebruiker tot toegang via interfaces

- *User Layer*: De user layer bevat, zoals zijn naam al laat vermoeden, de Java code (classes en interfaces) die iedereen mag gebruiken en uitbreiden. In essentie komt het erop neer dat deze code ertoe dient twee taken te bewerkstelligen:

  1. Een basisframework aanbieden waarbinnen, op een eenvoudige wijze, distribueerbare code kan geschreven worden.
  2. Een eenvoudige methode aanbieden om de distribueerbare code te deployen op de CoBRA grid

  Deze twee taken worden meer in detail besproken in sectie 5.6.

- *Middleware Layer*: De middleware layer is de layer die het originele doel van de CoBRA grid moet waarmaken. Hij moet ervoor zorgen dat grid-researchers op een eenvoudige manier, de verschillende standaard gridcomponenten, zoals een Worker, een Scheduler, een Broker, ... kunnen aanpassen en verbeteren om aldus onderzoeksmogelijkheden te bieden die geen repercuties hebben op de grid-gebuiker. De afscherming naar de grid-gebruiker toe gebeurt door deze enkel interfaces naar componenten uit de Middleware layer aan te bieden, zodat de componenten achter deze interfaces eenvoudig aangepast kunnen worden. De ware toedracht van deze layer wordt in meer detail besproken in sectie 5.5.

- *Low Level Layer*: Om *"portability"* te ondersteunen, is het essentieel dat de volledige software van de CoBRA grid gebaseerd is op portable code. Deze noodzaak wordt gerealiseerd door een basislaag aan te bieden die elk type gebruiker kan gebruiken, aan de hand van de beschikbare interface, maar die enkel aanpasbaar is door de CoBRA-developer. Deze developer is er dan ook verantwoordelijk voor dat de volledige code, die in deze Low Level layer omvat zit, volledig portable is. In essentie komt het erop neer dat deze layer de volledige backbone en de interfacing naar deze backbone omvat, samen met

enkele noodzakelijke standaard functionaliteiten zoals filetransfer, codetransfer en notificatie. Deze low level layer wordt in meer detail besproken in sectie 5.4.

Wanneer dit nu vanuit het oogpunt van de verschillende gebruikers bekeken wordt, dan krijgt elk type gebruiker een verschillende JAR (Java Archive) file die de verschillende onderdelen bevat, zoals aangegeven in tabel 5.1. De Grid-gebruiker beschikt over een JAR file met de class files van de User layer en met de benodigde interfaces van de Middleware Layer en van de Low Level Layer. De Grid-researcher beschikt over een JAR file met de class files van de UserLayer en van de Middleware layer en de noodzakelijke interfaces van de Low Level Layer. De CoBRA-developer beschikt over de volledige code en is verantwoordelijk voor het aanmaken en het ter beschikking stellen van de volledige set JAR files (noidg voor de andere types gebruikers en om het systeem te starten). De methoden die ervoor zorgen dat de interfaces binnen een laag aan de implementaties van een onderliggende laag gekoppeld worden, zijn "final"(in Java terminologie) om aldus elke vorm van misbruik te voorkomen.

Contains	Grid gebruiker JAR	Grid researcher JAR	CoBRA developer CODE
User Layer Class Files	X	X	X
Middleware Layer Interfaces	X	X	X
Low Level Layer Interfaces	X	X	X
Middleware Layer Class Files	-	X	X
Low Level Layer Class Files	-	-	X
Source Code	-	-	X

Tabel 5.1: Code access rights of user types

5.2.3 Operationeel Model

Deze sectie beschrijft, aan de hand van figuur 5.2, de opeenvolgende algemene stappen die worden doorlopen bij de distributie van een probleemstelling.

1. Als eerste stap ontwikkelt de Grid-gebruiker code, gebruik makend van de abstracte `ProblemSolver`[4] en `Job`[5] klassen uit de User-layer. Deze twee klassen zijn het ruwe skelet van de distributie van een probleemstelling. Concreet worden in deze stap de `ProblemSolver` en `Job` klassen uitgebreid en zorgt de implementatie van de aanwezige abstracte methoden voor de toevoeging van de probleemspecifieke code.

[4]De Problemsolver implementatie beschrijft het globale te distribueren probleem en splitst dit probleem op in subtaken.

[5]De Job implementatie beschrijft te effectieve opdracht die een subtaak moet uitvoeren.

2. In de tweede stap wordt deze code via de `Deployer` (eveneens uit de User-layer) op de `Broker` geplaatst.

3. In de derde stap handelt de broker de code, gespecificeerd in de `ProblemSolver`, af. Concreet houdt dit in dat er lokaal instructies uitgevoerd worden en Jobs gecreëerd worden.

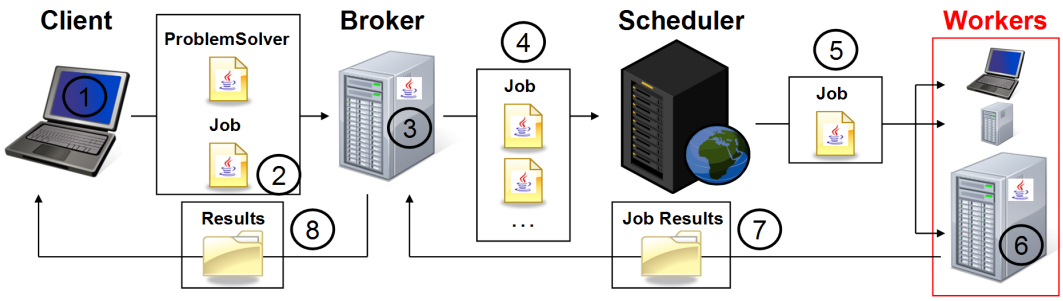
4. In de vierde stap worden deze jobs naar de `Scheduler` verstuurd.

5. In de vijfde stap kiest de `Scheduler` voor elke `Job` een `Resource` uit en verstuurt hij deze `Job` naar de `Resource`.

6. In de zesde stap zorgt de `Resource` ervoor dat de code uit de `Job` klasse wordt uitgevoerd.

7. De resultaten worden in de zevende stap naar de `Broker` terug gestuurd om met deze data een verdere afhandeling van de `ProblemSolver` mogelijk te maken.

8. Tot slot worden de data, die het resultaat zijn van de `ProblemSolver`, verstuurd naar een vooraf bepaalde machine, meestal de client machine.



Figuur 5.2: Het globale operationele model van de CoBRA grid.

5.3 Technische ruggengraat

5.3.1 Introductie

Bij de constructie van een gedistribueerd systeem is het belangrijk dat het systeem kan terugvallen op een degelijke technische ruggengraat. Die bestaat essentieel uit twee belangrijke aspecten. Het eerste aspect omvat de programmeeromgeving. Er dient een programmeertaal gekozen te worden die enerzijds degelijk is en anderzijds binnen de filosofie van het project past. De keuze voor JAVA is in dit geval vanzelfsprekend aangezien zowel interoperabiliteit als eenvoud in gebruik doelstellingen

zijn van het CoBRA framework. Het tweede aspect omvat het onderliggend communicatieframework. Ook hier dient het wiel niet heruitgevonden te worden. De keuze voor JAVA als progammeeromgeving beperkte de keuze tot vier mogelijkheden:

- Standaard java sockets uit het `java.net` package.
- RMI (Remote Method Invocation)[41] of alternatieven zoals RMIX [65][66], JERI(Jini Extensible Remote Invocation)[30], ...
- De H2O middleware [42][83].
- De Jini middleware [57][30].

Aangezien zowel Jini als H2O extra functionaliteit (leasing, dynamic classloading, ...) bevatten, die essentieel is voor de constructie van het CoBRA framework, kunnen de opties gereduceerd worden tot deze twee LL-LWG systemen.

Bij aanvang van het CoBRA project primeerde zowel de stabiliteit als de functionaliteit en bijgevolg ook de globale kwaliteit van H2O op Jini. Dit maakt dat, tijdens de ontwikkeling, in eerste instantie voor het H2O framework gekozen is. Performantie issues en de inhaalbeweging van Jini op H2O hebben er later voor gezorgd dat er tevens een implementatie op basis van Jini ontwikkeld is. Om de onderlinge incompatibiliteit aan te pakken en de plugabiliteitseigenschappen van CoBRA door te trekken tot op het niveau van de communicatie layer, is er tot slot voor gekozen een Universele communicatie laag te ontwikkelen. Met deze laag kunnen de verschillende componenten van de grid zowel op basis van H2O als op basis van Jini functioneren.

In de volgende secties wordt een kort overzicht gegeven van de drie verschillende communicatie backbones.

5.3.2 H2O Laag

Het H2O project werd in 2002 opgestart door het Distributed Computing Laboratory aan de Emory University te Atlanta. Het opzet was de realisatie van een component gebaseerd, service georiënteerd, gedistribueerd computing framework. De originele 'mission statement' werd als volgt samengevat:

> The development of a generalized plug-in paradigm for distributed virtual machines that allows users or applications to dynamically customize, adapt and extend the distributed computing environment's features to match their needs without compromising security or the consistency of the programming environment (afgeleid uit [83] en [39]).

Vrij vertaald wordt het H2O project gedefinieerd als een implementatie van het plug-in paradigma dat ertoe dient een gedistribueerde virtuele machine te ondersteunen.

Om de werking van het H2O principe beter te begrijpen, kan dit het best vergeleken worden met het servlet principe. Servlets zijn kleine Java programma's die functionaliteit aan een webserver toevoegen. Om dit te verwezenlijken, dienen ze gedeployed[6] te worden in een servlet container. Deze servlet container voorziet de functionaliteit om de dienst, die de servlet aanbiedt, te koppelen aan de webserver en aan de client. Populaire voorbeelden van dergelijke servlet containers zijn de Apache Tomcat container[1], de Winston Servlet Container[11],... .
Op dezelfde manier, zoals bijvoorbeeld Tomcat een container is voor servlets, zal de H2O kernel een container zijn voor de H2O pluglets (vergelijkbaar met servlets). Het grote verschil tussen de H2O kernel en elk ander traditioneel model zit hem echter in het feit dat, naast de eigenaar van een container (H2O-kernel), ook gebruikers, die voldoen aan de lokaal gedefinieerde constraints, services (pluglets) mogen deployen op de H2O kernel. In tegenstelling tot de traditionele statische containers maakt dit principe het mogelijk op een dynamische manier gedistribueerde resources te delen.

Een volledige uiteenzetting van het H2O framework is terug te vinden in appendix A

Omwille van het feit dat H2O de eerste backbone was waarop de CoBRA grid ontwikkeld werd, zal het framework recht toe recht aan gebruikt worden. Hiermee wordt bedoeld dat elke component, zoals bijvoorbeeld een worker of de scheduler, implementaties zullen zijn van een H2O pluglet die rechtstreeks gedeployed worden op een H2O kernel.

5.3.3 Jini Laag

Hoewel de eerste versie van Jini op 25 januari 1999 aan het grote publiek werd voorgesteld, start het Jini verhaal veel vroeger. Om dit terdege te begrijpen, is het belangrijk het initiële opzet van Java te kennen. De developers, van wat toen nog de naam Oak droeg, hadden als doelstelling een taal te creëren die een geheel van elektronische componenten, door middel van het uitwisselen van data, kon laten samenwerken. Java boekte zijn eerste grote succes in 1994 bij de release van WebRunner. Deze webbrowser maakte het mogelijk code onder de vorm van applets van een webserver te downloaden en uit te voeren. De aard van dit en de daaropvolgende successen dreef Java af van zijn vooropgestelde doelstellingen naar de taal zoals ze nu gekend is.

Onder begeleiding van een groot gedeelte van het originele Java team werd in alle discretie het Jini project op poten gezet. Het opzet was simpel: de originele doelstellingen van Java realiseren. Dit project bleef geheim tot het in 1998 in een artikel op de voorpagina van de New York Times verscheen. Verdere publicaties in o.a. Wired magazine, zorgden ervoor dat, zoals reeds vermeld, de eerste versie op 25 januari 1999 aan het grote publiek werd voorgesteld.

De doelstellingen van Jini waren eenvoudig:

- *Plug and play:* Een toestel dat een service levert, moet enkel voorzien worden

[6] Code ter beschikking stellen zodat deze kan uitgevoerd worden.

van stroom en van een netwerk toegang om zijn service ter beschikking te stellen.

- *Driver onafhankelijk*: Gebruikers hebben geen drivers nodig om een service te kunnen gebruiken.
- *Administratie vrij*: Elke service wordt onderhouden door diegene die hem ter beschikking stelt. Zo is het systeem vrij van enige globale administratie.
- *Combinatiemogelijkheden*: Services kunnen gebruik maken van andere services om zo een meer uitgebreide service aan te bieden.
- *Failure onafhankelijkheid*: Elke service werkt onafhankelijk en blijft dus ter beschikking bij het eventuele falen van delen van het netwerk of van andere machines.
- *Automatisch leren*: Software en toestellen leren automatisch van elkaar.

Figuur 5.3: Het Jini Concept

Het principe werkt, net zoals H2O, aan de hand van een Kernel, in dit geval Container genoemd, waarin een Pluglet, in dit geval Service genoemd, gedeployed kan worden. Aangezien de functionaliteit van een H2O kernel en een Jini container en van een H2O pluglet en een Jini service vergelijkbaar zijn, zullen deze termen in het verdere verloop van de tekst door elkaar gebruikt worden.

Een volledige uiteenzetting van het Jini framework is terug te vinden in appendix B.

Hoewel de overeenkomst tussen de twee frameworks laat uitschijnen dat het porten van CoBRA naar het Jini framework een eenvoudige procedure is, blijkt dit niet zo te zijn. De hoofdreden hiervoor is dat H2O iets uitgebreider is dan Jini en een aantal extra eigenschappen bezit die essentieel zijn voor het CoBRA framework. De belangrijkste drie zijn:

- *Deployment Descriptors*: H2O bezit een mechanisme dat, aan de hand van een XML file, beschrijft op welke manier en met welke parameters een pluglet op een kernel gedeployed moet worden. Dit mechanisme bestaat niet in Jini en moet dus toegevoegd worden.

- *AutoDeployment*: H2O bezit een mechanisme dat, aan de hand van een dergelijke deployment descriptor, tijdens het starten van een kernel verschillende pluglets automatisch deployed op deze kernel. Dit is noodzakelijk in het CoBRA framework omdat de gebruiker wil dat, bij het starten van een CoBRA component, naast de kernel, tevens de gewenste componenten (worker, scheduler, ...) worden geladen. Tevens kan op deze manier het lookup en discovery mechanisme, dat later wordt beschreven, vlot in de verschillende participerende kernels worden ingeladen. Ook dit principe is niet aanwezig in Jini en dient toegevoegd te worden.

- *Export van remote Objects*: H2O bezit de eigenschap dat al de objecten, waarvan een remote interface wordt doorgegeven aan een externe virtuele machine, automatisch remote ter beschikking worden gesteld. Bij Jini dienen deze interfaces geëxporteerd te worden alvorens ze remote beschikbaar zijn. Dit brengt met zich mee dat doorheen heel de CoBRA-Jini code de verschillende interfaces respectievelijk geëxporteerd en na gebruik geünexporteerd dienen te worden.

De volledige uitwerking van deze omschakeling is vooral een technische en programmatorische kwestie en valt buiten dit werk. Een volledige uiteenzetting is terug te vinden in een licentiaatsthesis [84] die tot stand kwam uit een samenwerking tussen Kurt Vanmechelen, Svetozar Misljencevic en Peter Hellinckx.

5.3.4 Universele Laag

De universele laag werkt naar een evenwicht tussen de voorgaande twee implementaties en tracht dit te bereiken aan de hand van een tussenlaag. Deze tussenlaag bestaat uit drie belangrijke componenten:

1. *CoBRAPluglet*: De belangrijkste component in de tussenlaag is waarschijnlijk deze `CoBRAPluglet`. De `CoBRAPluglet` implementeert zowel de `Pluglet` als de `Service` interface. Eventuele uitbreidingen van deze klasse zullen bijgevolg zowel in een Kernel als in een Container gedeployed kunnen worden. Naast deze implementatie wordt deze pluglet tevens uitgerust met een export-unexport mechanisme. Aan de hand van deze standaard methoden kunnen developers, interfaces die remote gebruikt worden, exporteren en unexporteren. Het spreekt voor zich dat deze methoden niets zullen ondernemen wanneer H2O de onderliggende laag is.

2. *CoBRAConnection*: De communicatie tussen de verschillende componenten in CoBRA gebeurt via connecties tussen de verschillende componenten. Deze connecties worden opgezet door vanuit een component, via H2O of Jini, te connecteren met de Pluglet of Service die de andere component implementeert. Nu de onderliggende laag onbekend is, dient dit connectie object ertoe de backbone van de remote machine na te gaan en via het correcte connectiemechanisme te connecteren. Dit gebeurt eenvoudig door bij de initiële connectie de verschillende connectiemechanismen te proberen tot er een connectie tot stand komt. Het type backbone voor deze remote machine wordt vervolgens bijgehouden en gebruikt voor de daaropvolgende connecties.

3. *CoBRA Discoverer*: Zoals later zal besproken worden, is het ontdekken van de verschillende componenten een belangrijk onderdeel van het CoBRA framework. Zowel in de H2O als in de Jini implementatie gebeurt dit aan de hand van het Jini Discovery mechanisme. De manier waarop de verschillende componenten worden bijgehouden en ter beschikking gesteld worden, verschilt echter wel. In deze CoBRA Discoverer worden aan de hand van beide mechanismen al de aanwezige pluglets en services opgevraagd en gegroepeerd. Zo lijkt het, voor de gebruiker, slechts één mechanisme te zijn dat de verschillende componenten ontdekt.

In praktijk dient door deze implementatie enkel een AutoDeployFile te worden meegegeven aan de Jini of H2O startup scripts om aldus de gewenste CoBRA component te starten. Het spreekt voor zich dat, door het gebruik van deze techniek, componenten met een Jini backbone kunnen samenwerken met de componenten met een H2O backbone en vice versa.

5.3.5 Besluit

Tijdens de implementatie van de CoBRA grid is de backbone geëvolueerd van een H2O versie en een Jini versie tot een samensmelting van deze twee backbones in een universele backbone laag. Voor de volledigheid dient hierbij opgemerkt te worden dat, omwile van startup performantie redenen, er meestal voor gekozen wordt de Jini backbone te gebruiken onder deze universele laag.

5.4 Low Level Layer

5.4.1 Introductie

De *low level layer* heeft tot doel de tekortkomingen van de onderliggende middlewares (H2O, Jini) te overbruggen om aldus de constructie van het CoBRA framework mogelijk te maken. De layer omvat, zoals reeds werd aangehaald, de volledige backbone en de interfacing naar de backbone samen met de noodzakelijke standaard functionaliteit die nodig is om het CoBRA framework te ondersteunen. Deze layer

bestaat, naast de technische ruggengraat die besproken werd in sectie 5.3, uit vier onderdelen die essentieel zijn in de werking van het CoBRA framework:

- Lookup and Discovery (zie sectie 5.4.2).
- CoBRA File System (zie sectie 5.4.3).
- Code Transfer (zie sectie 5.4.4).
- Notification (zie sectie 5.4.5).

Het verdere verloop van deze sectie zal elk van deze punten in meer detail bespreken.

5.4.2 Discovery

Zoals reeds werd aangehaald in sectie 5.3.4 is het ontdekken van de verschillende componenten binnen het CoBRA framework een essentieel onderdeel van dit framework. Het is immers belangrijk dat een scheduler weet over welke workers hij taken kan verdelen. Zowel de Jini als de H2O backbone gebruiken het lookup principe uit het Jini framework. De werking van dit principe draait rond een LUS (Lookup Service). Elke service die gedeployed is, heeft de mogelijkheid zich te registreren bij deze LUS. Alle mogelijke Jini gebruikers kunnen dan van deze LUS gebruik maken om na te gaan welke services er op welke remote machine beschikbaar zijn. Een diepere uitwerking van dit concept is terug te vinden in appendix B.2.1.

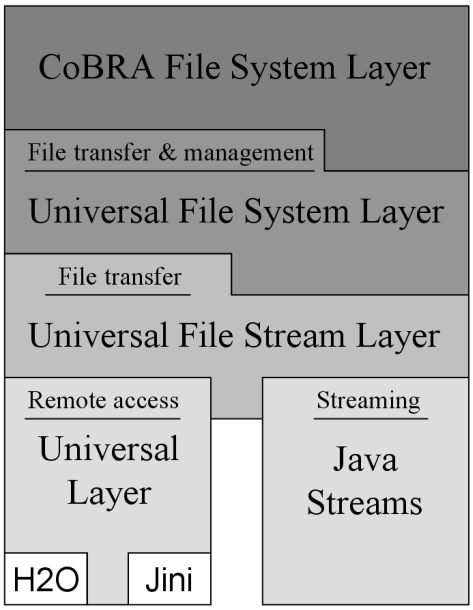
Het spreekt voor zich dat het, omwille van compatibiliteitsproblemen, standaard niet mogelijk is H2O pluglets te registreren bij een dergelijke LUS. Om dit probleem uit de weg te gaan, is de techniek van [39] verder uitgewerkt en toegepast binnen de H2O lookup implementatie. De techniek die hier gebruikt wordt, registreert een Jini service die als ambassadeur de H2O pluglet vertegenwoordigt in de LUS. Een H2O lookup pluglet kan dan, aan de hand van de specifieke vorm van deze ambassadeur services, achterhalen welke H2O pluglets er geregistreerd zijn. De Universele lookupservice combineert de informatie uit de LUS met deze uit de H2O lookupservice om aldus uniforme lookupmogelijkheden aan te bieden.

5.4.3 CoBRA File System

Het spreekt voor zich dat een robuust file systeem essentieel is om het input en output management van een gedistribueerd systeem te ondersteunen. De platform onafhankelijkheidsvereisten impliceren dat er tijdens het development uitsluitend Java libraries gebruikt worden. Omwille van de scalability constraints kan er voor het development van de CoBRA grid enkel voor een gedecentraliseerd filesysteem gekozen worden. Dit bepert de keuze tot:

- *Decentralised shared filesystem*: Deze abstractielaag geeft de gebruiker het beeld dat alle files op de lokale machine aanwezig zijn. De files zijn echter verspreid over al de participerende machines.
- *Decentralised individual filesystem*: Elke machine beheert een eigen verzameling files. Indien er files van een andere machine noodzakelijk zijn, dienen deze expliciet overgebracht te worden naar de lokale machine.

In het CoBRA framework is gekozen voor het individual filesystem om eventuele locking problemen te vermijden.



Figuur 5.4: CoBRA filesystem abstractie lagen

De afwezigheid van een remote filemanagement mechanisme, in zowel H2O als in Jini, maakt dat de rechtstreekse development van de CoBRA grid op deze frameworks moeilijk realiseerbaar is. Om deze procedure te vereenvoudigen, is het CoBRA file systeem ontwikkeld. Om de transparantie naar de developer toe te vergroten, is dit file systeem opgebouwd uit drie lagen die de Universele laag[7] en de Java Streams API verbinden tot het CoBRA file systeem (zie figuur 5.4):

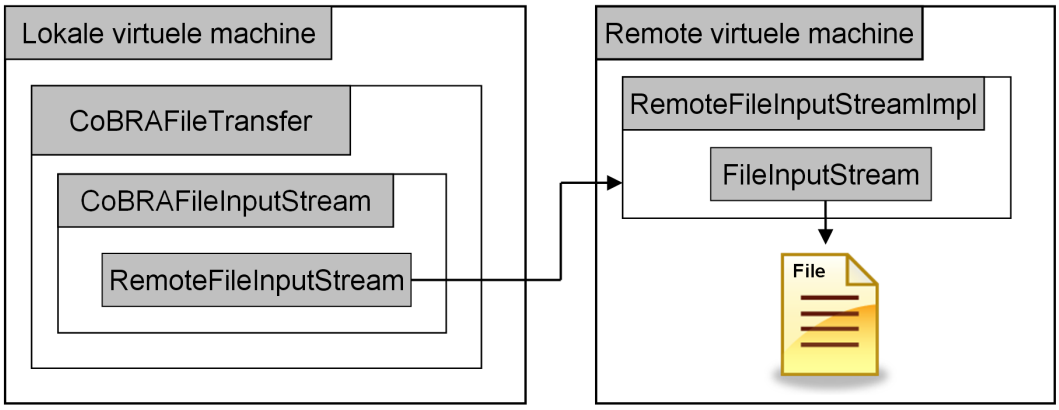
- *CoBRA File System Layer*: Deze laag staat in voor het filemanagement binnen de CoBRA grid. Op basis van de `CoBRAFileTransfer` klasse zorgt ze ervoor dat elke component in het framework de mogelijkheid heeft zowel directories en files te transfereren als links en files te manipuleren. De implementatie van deze layer berust op de functionaliteit uit de *universal file system layer*.

- *Universal File System Layer*: Aan de hand van deze layer worden de individuele filemanipulaties van de *CoBRA File System Layer* binnen de gewenste constraints afgehandeld. Deze layer werkt volgens het `deployment identification` principe. Hiermee wordt bedoeld dat elke deployment van een problemsolver het idee krijgt dat het op elke component over een eigen filesysteem beschikt. Op deze manier worden de files van andere deployments verborgen en bestaat er geen gevaar voor interferentie of voor security problemen. De

[7] De laag bovenop Jini en H2O besproken in sectie 5.3.4.

effectieve file manipulaties worden uitgevoerd op basis van de functionaliteit uit de *Universal File Stream Layer*.

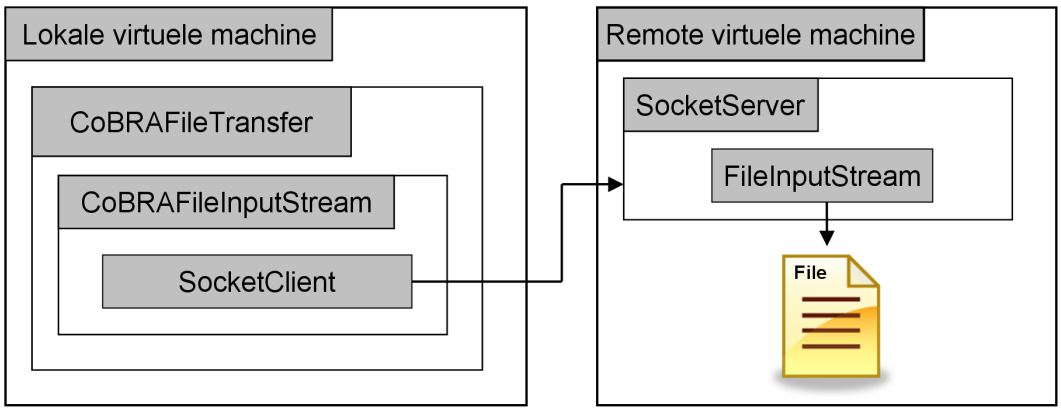
- *Universal File Stream Layer*: Deze laag transfereert data op basis van zowel de communicatieinfrastructuur uit de *universal layer* als op basis van standaard Java streams. Aangezien de implementatie voor deze techniek analoog zal zijn voor `InputFileStreams` en voor `OutputFileStreams` zal enkel de eerste van dit duo besproken worden. In essentie komt het erop neer dat binnen de `CoBRAFileTransfer` klassen een `CoBRAFileInputStream` klasse geïnstantieerd wordt die afgeleid is van de standaard Java `FileInputStream` klasse. De implementatie van elk van de `FileInputstream` onderdelen gebeurt aan de hand van een remote interface naar een `RemoteFileInputStreamImpl`. Deze remote klasse gebruikt de functionaliteit van een lokale `FileInputStream` om de functionaliteit, die `CoBRAFileInputStream` vereist, te ondersteunen. Dit alles wordt visueel voorgesteld in figuur 5.5.



Figuur 5.5: CoBRA filesystem klassen.

Hoewel deze techniek terdege werkt is er in de testfase van CoBRA gebleken dat zowel het Jini framework als het H2O framework niet in staat zijn connecties voldoende snel vrij te geven. Hiermee wordt bedoeld dat een socket, naar een machine die niet langer bestaat, slechts na lange tijd opnieuw gebruikt kan worden om met een andere machine te connecteren. Indien machines vaak up en down gaan, en er veel filetransfer gebeurt, zullen er veel dergelijke dode connecties ontstaan. Het aantal beschikbare connecties kan in dit geval uitgeput raken. Hiertegen is het framework niet bestand. Om dit te verhelpen is de eerste versie van de filetransfer methode in beperkte mate aangepast. De nieuwe versie werkt niet langer via de remote interfaces maar via slechts één aparte socket, die expliciet kan afgesloten worden wanneer een machine niet langer beschikbaar is. Dit impliceert enerzijds dat het aantal gebruikte sockets drastisch vermindert. (Niet langer één per file, maar één voor alle files.) Anderzijds wordt de socket ook steeds expliciet afgesloten wat maakt dat er geen dode sockets ontstaan. De implementatie gebeurt door in de `CoBRAFileTransfer` klasse een socketverbinding op te zetten met de remote virtuele

machine en met de `CoBRAFileInputStream` via sockets met de `FileInputStream` op de remote machine te verbinden (zie figuur 5.6).



Figuur 5.6: CoBRA socket filesystem klassen.

Naast deze standaard implementatie dient er nog vermeld te worden dat de data optioneel ook gecomprimeerd getransfereerd kunnen worden. Om dit te verwezenlijken, worden de verschillende te transfereren files gecomprimeerd volgens de ZIP standaarden. Hierbij kan de gebruiker zelf het level van compressie bepalen. Vervolgens wordt de gezipte file, via de standaard filetransfermechanismen, naar de remote machine getransporteerd. Deze techniek blijkt echter computationeel intensief te zijn en wordt enkel aangeraden bij gridopstellingen waar de transfersnelheid ondermaats is aan de computationele kracht van de verschillende grid componenten. Dergelijke situaties kunnen zich voordoen op netwerken waar datatransfer primeert op datamanipulatie zoals bijvoorbeeld in gedistribueerde filesystemen.

5.4.4 Code Transfer

Aangezien er binnen een gedistribueerd systeem gewerkt wordt, moet de code, die remote dient uitgevoerd te worden, getransfereerd worden naar deze remote machines. Concreet houdt dit in dat een klasse, die nodig is op een remote machine, volledig getransfereerd dient te worden naar deze machine. Dit brengt twee problemen met zich mee:

1. Er moet een klasse aangemaakt worden op de remote machine die identiek is aan die op de lokale machine.

2. De definitie van de klasse dient gekend te zijn op deze remote machine.

De methode waarmee dit verwezenlijkt wordt, werkt op basis van remote[8] transferklassen (`JobCarrier`, `ProblemSolverCarrier`). In deze remote klassen worden al de data verzameld die nodig zijn om een identieke kopie van de te transfereren

[8]Dit wil zeggen dat deze klassen op de lokale virtuele machine blijven maar via een remote interface aanspreekbaar zijn op de remote machine.

klasse te maken. Deze informatie bestaat uit al de gegevens van de huidige instantie van de klasse en een link naar de codebase (JAR file) waarin de definitie van de klasse vervat zit. De gegevens van de huidige klasse worden verzameld aan de hand van het marshalling - unmarshalling principe. De klassen en zijn datavelden worden door dit principe geserialiseerd in een zogenaamd Marshalled object. Op de remote machine zal dit Marshalled object vervolgens unmarshalled of gedeserialiseerd worden om aldus een kopie van de desbetreffende klasse te instantiëren. Alvorens dit deserialisatieproces van start kan gaan, dient de remote machine de definitie van de desbetreffende klasse te kennen. Zoniet zouden de constructie van de klasse, en bijgevolg ook de deserialisatie, onmogelijk zijn. Gebruik makend van de link naar de codebase wordt op de remote machine een nieuwe URIClassloader[9] geïnitialiseerd. Dit object kan, gebruik makend van zijn referentie naar de codebase, de desbetreffende klasse construeren en bijgevolg de deserialisatie mogelijk maken. Deze techniek wordt gebruikt om zowel `Problemsolvers` als `Jobs` naar de gewenste bestemming te transporteren.

5.4.5 Notification

Een belangrijk concept in een middleware is de communicatie van gebeurtenissen. Het is immers belangrijk dat betrokken componenten op de hoogte worden gebracht van bepaalde gebeurtenissen. Een scheduler moet bijvoorbeeld op de hoogte gebracht worden wanneer een job faalt, zodat hij deze job opnieuw kan submitten of er op een andere gepaste manier op kan reageren. Deze functionaliteit wordt binnen de CoBRA grid gerealiseerd aan de hand van het *event listener model*[40]. Dit principe is gekozen voor zijn degelijkheid en om de uniformiteit in het systeem te bewaren. Het principe werd immers reeds aangewend om het discovery mechanisme, beschreven in sectie 5.4.2, te ondersteunen.

In praktijk bestaat een taak uit vier fases: datatransfer naar de worker; uitvoeren van de taak; datatransfer van de resulterende data; afronden van de volledige taak. Elk van deze taken kan lukken (succes) of falen (fail). Componenten die in aanraking komen met een taak kunnen listeners verbinden met elk van deze vier fases. Een listener is een implementatie van de `CoBRAListener` klasse. Deze abstracte klasse bevat een abstracte `succes` en een abstracte `fail` methode. Wanneer een listener geïmplementeerd wordt voor een bepaalde component en vervolgens met een fase van een taak verbonden wordt, zal de listener, bij het beëindigen van de desbetreffende fase, de code in de `succes` of `fail` uitvoeren op de betrokken component. Zo kan een listener van de scheduler die verbonden is met een datatransfer fase, bij het falen van deze datatransfer, er aan de hand van de `fail` methode voor zorgen dat de data opnieuw gestuurd worden.

[9] De classloaders zijn de componenten van Java die, gebruik makend van het classpath (codebase), de verschillende gebruikte klassen construeren en in het geheugen laden.

5.5 CoBRA Middleware Layer

5.5.1 Introductie

De CoBRA MiddlewareLayer is de abstractielaag die zich boven de Low Level Layer bevindt. Deze laag bevat de componenten die noodzakelijk zijn om de computationele problemen uit de *User Layer* gedistribueerd uit te voeren. De basis CoBRA implementatie bestaat uit vier componenten: de Broker, de Scheduler, de Worker en de Resource manager. In het verdere verloop van deze sectie wordt elk van deze componenten in meer detail besproken. Tevens wordt de uitbreidbaarheid en de plugability van deze componenten besproken.

5.5.2 Resource Manager

Alvorens componenten kunnen samenwerken, dienen ze weet te hebben van elkaars bestaan. De resource manager zorgt ervoor dat elk van de CoBRA componenten de mogelijkheid heeft de verschillende instanties van een bepaald type component op te vragen.

De `ResourceManagerImpl` klasse is een concrete implementatie van de abstracte `CoBRAPluglet` klasse. Het is, met andere woorden, een service die zowel op H2O als op Jini gedeployed kan worden. De klasse implementeert tevens de `ResourceManager` interface. Deze interface bevat slechts één methode die de functionaliteit van de component beschrijft:

```
CompResourceCarrier[] getResources(ResourceDescriptor desc)
```

De methode gaat, op basis van een `ResourceDescriptor`, op zoek naar de verschillende CoBRA componenten die voldoen aan deze beschrijving en geeft een array `CompResourceCarrier` objecten terug.

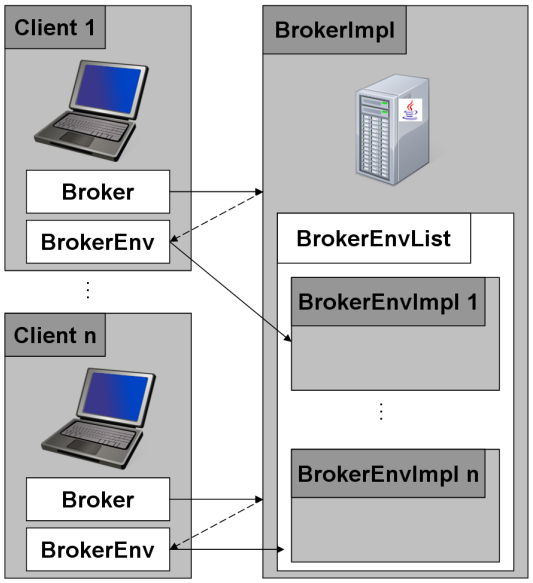
De `CompResourceCarrier` objecten bevatten, naast een remote interface naar de desbetreffende componenten, extra informatie. Deze informatie varieert van de lijst jobs, die op dat tijdstip op de desbetreffende resource behandeld worden, tot gegevens die noodzakelijk zijn bij meer geavanceerde technieken, zoals prediction based scheduling (bijvoorbeeld de verwachte uptime van een worker).

De selectie van gewenste componenten gebeurt op basis van het `ResourceDescriptor` object. Dit object wordt geïnitialiseerd met de naam van een component en met eventuele data die de gewenste component specifieker kunnen beschrijven (eventueel minimale CPU of memory vereisten). De basisimplementatie van de `ResourceManager` gebruikt het *discovery* mechanisme uit de low level layer (zie sectie 5.4.2) om al de geregistreerde Jini en H2O componenten op te vragen. Aan de hand van de descriptor wordt een connectie gemaakt met de gewenste componenten. Deze remote interface connecties worden in nieuwe `CompResourceCarriers` geplaatst en gegroepeerd, onder de vorm van een array, teruggegeven.

5.5.3 Broker

De Broker zal, zoals in sectie 5.2.3 reeds werd besproken, instaan voor de afhandeling van het globale te distribueren probleem. Tijdens deze afhandeling worden er normalerwijze verschillende taken gecreëerd die verdeeld moeten worden over de verschillende beschikbare resources. De broker zorgt ervoor dat deze taken bij een scheduler terecht komen om zo verder verdeeld te worden.

Concreet bestaat de broker uit een `BrokerImpl` klasse die een effectieve implementatie is van de abstracte `CoBRAPluglet` klasse. De klasse implementeert tevens de `Broker` interface. Deze interface bevat slechts één methode die een `BrokerEnvironment` opvraagt. Op deze manier kan elke gebruiker inloggen op een broker en aan de hand van een eigen broker environment zijn probleemstelling distribueren. Dit alles wordt duidelijk aan de hand van figuur 5.7.



Figuur 5.7: Uiteenzetting van de broker infrastructuur

De `BrokerEnvironment` is een remote interface naar de `BrokerEnvironmentImpl` klasse die de volledige broker functionaliteit implementeert. De methoden uit deze klasse kunnen onderverdeeld worden in drie groepen:

- De eerste groep methoden staat in voor het deployen[10] en het uitvoeren van het globale gedistribueerde probleem. Dit vertaalt zich in twee methoden:
  - `DeployProblemsolver`: Deze methode plaatst de code en de nodige data op de broker machine.

[10]Het plaatsen van alle gegevens die nodig zijn om een taak uit te voeren op de desbetreffende machine.

  - `ExecuteProblemSolver`: Deze methode voert de code van de `ProblemSolver`[11] uit.

- De tweede groep bestaat enkel uit de `submit(JobCarrier job)` methode. Deze methode staat in voor het submitten van een job naar een scheduler. Om dit te verwezenlijken, dient er tijdens de initialisatie van de broker een connectie gemaakt te worden met de Resource Manager om daar de beschikbare schedulers op te vragen. Uit deze lijst wordt één scheduler geselecteerd. Deze scheduler wordt vervolgens door elke aanroep van deze `submit` methode gebruikt om de Jobs te verdelen. Alvorens de Jobs naar een scheduler worden gestuurd, worden ze geserialiseerd en in een `JobCarrier` object geplaatst (zie sectie 5.4.4). Naast deze serialisatie worden tevens de noodzakelijke BrokerJobListeners aan de `JobCarrier` toegevoegd om het notificatie mechanisme te ondersteunen (zie sectie 5.4.5).

- De derde groep bestaat uit een lijst van filemanagement methoden. Deze methoden worden door de `ProblemSolver` aangewend om zowel lokale als remote file manipulaties uit te voeren. Enkele voorbeelden hiervan zijn:

  - `GetRemoteDir`: Transfereert een remote directory naar het lokale filesysteem.
  - `GetFileOutputStream`: Opent een outputstream naar een lokale file.
  - `ExecuteBinary`: Deze methode maakt het mogelijk een binary file uit te voeren.

  Hier dient wel bij opgemerkt te worden dat deze abstractielaag er, via het CoBRA file systeem, voor zorgt dat elke sessie zijn individueel lokaal systeem lijkt te hebben (meer informatie is terug te vinden in sectie 5.4.3).

5.5.4 Scheduler

De Scheduler heeft als taak jobs, die hij ontvangt, te verdelen over de verschillende beschikbare resources.

Concreet bestaat de scheduler uit een `Scheduler` klasse die een effectieve implementatie is van de abstracte `CoBRAPluglet` klasse. De klasse implementeert tevens de `Resource` interface. De `Resource` interface bestaat uit twee groepen methoden:

- De eerste groep bestaat slechts uit één methode namelijk de `execute( JobCarrier job)`. Deze methode wordt gebruikt door de Broker, of eventuele andere resources, om Jobs via deze scheduler te verdelen. De methode voegt de noodzakelijke SchedulerJobListeners aan de JobCarrier toe ter ondersteuning van het notificatie mechanisme (zie sectie 5.4.5) en plaatst de Job in de JobQueue. De JobQueue zal, zoals verder wordt besproken, aangewend worden om de Jobs te verdelen over de verschillende beschikbare resources.

[11] De problemsolver beheert de uitvoering van het globale gedistribueerde probleem en wordt in meer detail besproken in sectie 5.6.2.

- De tweede groep bestaat uit een lijst van filemanagement methoden. Deze methoden kunnen in een schedulerimplementatie aangewend worden om zowel remote als lokale files te manipuleren. In deze basis implementatie wordt hier echter geen gebruik van gemaakt. Enkele voorbeelden hiervan zijn:
  - `GetRemoteDir`: Transfereert een remote directory naar het lokale filesysteem.
  - `GetFileOutputStream`: Opent een outputstream naar een lokale file.
  - `ExecuteBinary`: Deze methode maakt het mogelijk een binary file uit te voeren.

Naast de implementatie van deze interface worden, tijdens de initialisatie van het `Scheduler` object, nog twee threads aangemaakt ter ondersteuning van de Job verdeling:

- `SchedulerResourceMonitor`: Deze thread staat in voor het verzamelen en het monitoren van de beschikbare resources. Gebruik makend van de `Resource Manager`, beschreven in sectie 5.5.2, wordt, aan de hand van een lijst gewenste resource types, een lijst van de beschikbare resources opgesteld. Deze resourcetypes kunnen sterk variëren. Zowel de standaard worker en scheduler implementaties als alternatieve implementaties van deze scheduler en worker tot nieuwe implementaties van de `Resource` interface, kunnen als resource geselecteerd worden.

  Nieuwe resources worden automatisch aan de resource lijst in de scheduler toegevoegd. Wanneer een resource plots blijkt te verdwijnen, zal deze thread ervoor zorgen dat de jobs, die door de desbetreffende resource behandeld worden, op een andere resource gescheduled kunnen worden. Dit gebeurt aan de hand van de joblist die in elk van de gebruikte `Resource Carriers` vervat zit. De verschillende jobs in deze lijst worden vooraan op de `jobQueue` geplaatst zodat ze met de hoogste prioriteit gerescheduled worden.

- `SchedulerJobHandler`: Deze thread zorgt ervoor dat de verschillende jobs in de `jobQueue` verdeeld worden over de aanwezige resources. Om dit te bewerkstelligen, wordt het *first come first served* principe gebruikt. Job per job worden vooraan van de queue genomen en op een random beschikbare resource geplaatst. De resources die hiervoor in aanmerking komen, zijn diegene uit de resourcelist waarvan het aantal jobs hun maximum niet overschrijdt.

Tot slot dient nog vermeld te worden dat de SchedulerJobListener ervoor zal zorgen dat, bij succes, de job uit de jobList van de desbetreffende `ResourceCarrier` verwijderd wordt en dat, bij falen van een job, deze job terug vooraan op de `jobQueue` geplaatst wordt (na drie pogingen wordt de job echter verwijderd).

5.5.5 Worker

De Worker heeft als taak de jobs, die hij ontvangt, af te handelen. Dit houdt in dat de code die in de job vervat zit, uitgevoerd wordt en de resulterende data terug

gestuurd worden naar de `Broker` machine.

Concreet bestaat de worker uit een `Worker` klasse die een effectieve implementatie is van de abstracte `CoBRAPluglet` klasse. De klasse implementeert tevens de `Resource` interface. De `Resource` interface bestaat uit twee groepen methoden:

- De eerste groep bestaat slechts uit één methode namelijk de `execute( JobCarrier job)`. Deze methode wordt gebruikt door de scheduler om taken te laten uitvoeren door deze worker. De jobs, die via deze methode in de worker terecht komen, worden achteraan op de JobQueue geplaatst. Deze JobQueue zal, zoals verder wordt besproken, een buffer zijn waar een job wacht tot ze uitgevoerd kan worden door de worker. Dit wordt verder besproken in het topic omtrent de `WorkerJobHandler` thread.

- De tweede groep bestaat uit een lijst van filemanagement methoden. Deze methoden worden in de worker aangewend om zowel remote als lokale files te manipuleren. Concreet komt het erop neer dat de filemanipulatie methoden in een Job worden doorgelinkt naar deze methoden zodat de job zonder problemen op het lokale filesysteem afgehandeld kan worden. Enkele voorbeelden hiervan zijn:

  - `GetRemoteDir`: Transfereert een remote directory naar het lokale filesysteem.
  - `GetFileOutputStream`: Opent een outputstream naar een lokale file.
  - `ExecuteBinary`: Deze methode maakt het mogelijk een binary file uit te voeren.

Naast de implementatie van deze interface wordt, tijdens de initialisatie van het `Worker` object, een `WorkerJobHandler` thread gestart die de uitvoering van de verschillende jobs in de `jobQueue` bewerkstelligt. Concreet komt het erop neer dat deze methode ervoor zorgt dat de `JobCarrier` de desbetreffende Job deserialiseert. Dit houdt in dat in de `JobCarrier` een nieuwe `URIClassloader` wordt geïnitialiseerd met het gewenste classpath en dat het Marshalled textttJob object geunmarshalled wordt. Nadat de verschillende joblisteners (uit Broker en Scheduler) met de job verbonden zijn, wordt het `Job` object aan de `WorkerJobHandler` thread teruggegeven. In deze thread worden de opeenvolgende fases van een job doorlopen (datatransfer naar de worker, uitvoeren van de taak, datatransfer van de resulterende data, afronden van de volledige taak) en de desbetreffende joblistener methoden aangeroepen. Wanneer dit proces volledig is afgerond, gaat de thread op zoek naar de volgende Job in de `jobQueue`.

5.5.6 Plugability

Zoals reeds werd aangehaald, is *plugability* het key concept van de CoBRA grid. In de middleware layer kan dit bewerkstelligd worden op drie niveaus:

1. Op het hoogste niveau is het mogelijk elk van de hierboven besproken componenten uit te breiden. Elk van de methoden, die los staan van het filemanagement, kunnen uitgebreid worden (de overige zijn voor security redenen final gemaakt en bijgevolg niet uitbreidbaar). Volgens deze techniek zijn er al verschillende uitbreidingen gemaakt:

   - *Economic scheduling*: Deze scheduling methodiek herimplementeert zowel de Broker, de Scheduler als de Worker om aldus op basis van marktmodellen de verschillende taken te verdelen. Meer informatie is terug te vinden in sectie 5.8.4.
   - *Prediction scheduling*: Deze scheduler herimplementeert eveneens zowel de Broker, de Scheduler als de Worker om aldus op basis van de runtimepredicties van de verschillende jobs de taken optimaal over de verschillende aanwezige resources te verdelen. Meer informatie is terug te vinden in sectie 5.8.5.

2. Op het middelste niveau is het mogelijk een nieuwe implementatie te maken van de Resource interface. Deze techniek werd bijvoorbeeld aangewend om een Globus[34] enabled resource te ontwikkelen. Deze resource verdeelt de jobs over de resources aangebonden door een Globus grid zoals BEGrid. Meer informatie hieromtrent is terug te vinden in sectie 5.8.6

3. Op het laagste niveau is het tevens mogelijk een volledig nieuwe CoBRA pluglet te ontwikkelen om aldus nieuwe onbestaande functionaliteit aan te bieden. Omdat deze nieuwe functionaliteit buiten de scope van deze thesis valt, is dit tot dusver niet uitgetest. De mogelijkheden zijn hier echter nagenoeg onbeperkt. Een mogelijke implementatie kan bijvoorbeeld, onder de vorm van Authenticatie en Authorisatie, een security layer rond de CoBRA grid ontwikkelen. Tevens zou het mogelijk zijn op deze manier Monitoring tools op te zetten om de load van het gehele systeem te visualiseren.

5.6 User Layer

5.6.1 Introductie

Zoals reeds werd aangehaald in sectie 5.2, bevat de user layer de Java code die iedere gebruiker mag gebruiken en uitbreiden. De code heeft twee fundamentele doelstellingen:

- Een basisframework aanbieden waarbinnen, op een eenvoudige wijze, distribueerbare code kan geschreven worden.
- Een eenvoudige methode aanbieden om de distribueerbare code te deployen op de CoBRA grid.

Om dit te bewerkstelligen, kan deze layer opgesplitst worden in drie onderdelen:

1. *ProblemSolver*: De problemsolver implementeert het globale te distribueren probleem. Aan de hand van deze implementatie worden de verschillende jobs aangemaakt en verspreid over de CoBRA grid.

2. *Job*: Een job implementeert een subtaak van het globale probleem die op een individuele resource verwerkt kan worden.

3. *Deployer*: Deze component staat in voor de deployment van de Problemsolver op de Broker.

Elk van deze drie onderdelen wordt in meer detail besproken in het verdere verloop van deze sectie.

5.6.2 Implementatie van het gedistribueerde probleem

De gedistribueerde implementatie van een probleemstelling gebeurt in CoBRA aan de hand van twee componenten: de `ProblemSolver` en de `Job`. De `ProblemSolver` implementeert de volledige probleemstelling en splitst het geheel op in subtaken: de jobs. Deze jobs worden dan geïmplementeerd aan de hand van de `Job` component.

De abstracte `ProblemSolver` klasse implementeert zowel de Serializable als de Runnable interface. De serializability zorgt ervoor dat de klasse eenvoudig gemarshalled naar de broker verstuurd kan worden. De runnable interface impliceert dat deze klasse een afgelijnd stuk code bevat dat onafhankelijk uitgevoerd kan worden. De `ProblemSolver` klasse bestaat uit vier groepen methoden:

- De eerste groep bevat enkel de `SetBroker` methode. Aan de hand van deze methode wordt een interface naar de huidige brokerenvironment in de problemsolverimplementatie geplaatst. Dit is essentieel om de nodige functionaliteit voor het filemanagement en het submitten van taken uit de huidige broker te kunnen gebruiken.

- De tweede groep methoden bestaat uit een lijst van job management methoden. Deze methoden zorgen ervoor dat de verschillende job manipulaties correct afgehandeld worden. Dit door gebruik te maken van de functionaliteit die beschikbaar is in de huidige brokerenvironment. Voorbeelden van deze methoden zijn:

  - `Submit(Job)`: Deze methode dient gebruikt te worden in de implementatie van de probleemstelling en roept de `submit(JobCarrier)` methode uit de broker aan (zie sectie 5.5.3) om aldus de jobs te verspreiden op de CoBRA grid.
  - `JobDone(Job)`: Deze methode gebruikt de brokerenvironment om te detecteren of een gesubmitte job reeds is uitgevoerd. Dit is belangrijk bij implementaties waar jobs onderlinge dependencies hebben.

- De derde groep bestaat uit een lijst van filemanagement methoden. Deze methoden worden in de problemsolver implementatie aangewend om zowel remote

als lokale files te manipuleren. Concreet komt het erop neer dat de filemanipulatie methoden in de `ProblemSolver` worden doorgelinkt naar de overeenkomstige methodes uit de brokerenvironment waarin deze problemsolver wordt uitgevoerd. Dit wordt bewerkstelligd door de brokerinstantie die, aan de hand van de reeds eerder besproken `SetBroker` methode, gezet wordt. Deze methodiek zorgt ervoor dat een problemsolverimplementatie zonder problemen op het lokale filesysteem van de `Broker` afgehandeld kan worden. Enkele voorbeelden van deze methoden zijn:

  - `GetRemoteDir`: Transfereert een remote directory naar het lokale filesysteem.
  - `GetFileOutputStream`: Opent een outputstream naar een lokale file.
  - `ExecuteBinary`: Deze methode maakt het mogelijk een binary file uit te voeren.

- De laatste groep methoden zijn de abstracte methoden uit de `ProblemSolver` klasse. Deze methoden dienen door de gebruiker ingevuld te worden om aldus de functionaliteit van de probleemstelling te implementeren. Deze implementatie dient gebruik te maken van de methoden uit groepen twee en drie om de file en job manipulaties te verwezenlijken. Concreet dienen er drie methoden geïmplementeerd te worden:

  - `GetData()`: Deze methode dient de inputfiles van de remote machines te vergaren, gebruik makend van de datamanipulatie methoden uit de derde groep.
  - `PutData()`: Deze methode dient de gewenste resulterende data te verplaatsen naar een machine van de gebruiker, aan de hand van de datamanipulatie methoden uit de derde groep.
  - `Run()`: Deze methode dient de concrete uitwerking van het probleem te implementeren. Deze implementatie bestaat uit een combinatie van lokale berekeningen en van job manipulaties, die aan de hand van de methoden uit de tweede groep dienen uitgevoerd te worden.

De abstracte `Job` klasse implementeert de Serializable interface. De serializability zorgt ervoor dat de klasse eenvoudig gemarshalled naar een resource (in de meeste gevallen een worker) verstuurd kan worden. De `Job` klasse bestaat uit vier groepen methoden:

- De eerste groep bevat enkel de `SetResource` methode. Aan de hand van deze methode wordt een interface naar de resource, waarop de job wordt uitgevoerd, in de jobimplementatie geplaatst. Dit is essentieel om de nodige functionaliteit voor het filemanagement uit de huidige resource te kunnen gebruiken.

- De tweede groep methoden dient ter ondersteuning van de *notification* techniek (zie sectie 5.4.5). In essentie bestaat deze groep uit twee soorten methoden:

  - De methoden waarmee, voor elke fase in de afhandeling van een job, joblisteners kunnen toegevoegd worden:
    * `AddGetDataListener(JobListener listener)`.
    * `AddPutDataListener(JobListener listener)`.
    * `AddExecutionListener(JobListener listener)`.
    * `AddGLobalListener(JobListener listener)`.
  - De methoden die ervoor zorgen dat de juiste methode uit deze joblisteners aangeroepen wordt wanneer een bepaalde jobfase is afgelopen. Deze methoden worden door de Resource aangeroepen wanneer een jobfase faalt of succesvol wordt beëindigd. In essentie komt het erop neer dat er, voor elk type van de bovenstaande listeners, zowel een `Succes` als een `Fail` methode bestaat die de `Succes` en `Fail` methoden van elk van de listeners van dat bepaald type uitvoeren.

- De derde groep bestaat uit een lijst van filemanagement methoden. Deze methoden worden in de jobimplementatie aangewend om zowel remote als lokale files te manipuleren. Concreet komt het erop neer dat de filemanipulatie methoden in de `Job` worden geïmplementeerd aan de hand van de overeenkomstige methoden uit de resource waarop deze job wordt uitgevoerd. Dit wordt bewerkstelligd door de resource die, aan de hand van de reeds eerder besproken `SetResource` methode, gezet wordt. Deze methodiek zorgt ervoor dat een jobimplementatie zonder problemen op het lokale filesysteem van de `Broker` afgehandeld kan worden. Enkele voorbeelden van deze methoden zijn:
  - `GetRemoteDir`: Transfereert een remote directory naar het lokale filesysteem.
  - `GetFileOutputStream`: Opent een outputstream naar een lokale file.
  - `ExecuteBinary`: Deze methode maakt het mogelijk een binary file uit te voeren.
  - `GetRemoteProblemSolverDir`: Transfereert een remote directory vanop de broker machine, waarop de problemsolverimplementatie wordt uitgevoerd, naar het lokale filesysteem.

- De laatste groep methoden zijn de abstracte methoden uit de `Job` klasse. Deze methoden dienen door de gebruiker ingevuld te worden om aldus de functionaliteit van een individuele job te implementeren. Deze implementatie dient gebruik te maken van de methoden uit groep drie om eventuele file manipulaties te verwezenlijken. Concreet dienen er vijf methoden geïmplementeerd te worden:
  - `GetGlobalData()`: Deze methode dient de globale inputfiles van de remote machines te vergaren, gebruik makend van de datamanipulatie methoden uit de derde groep. Het opzet van deze methode is de data, die door elke job gebruikt worden voor het uitvoeren van de eerste job, naar

de resource te verplaatsen zodat dit niet bij elke job opnieuw moet gebeuren.

- `PutGlobalData()`: Deze methode dient de gewenste resulterende data te verplaatsen naar de broker machine aan de hand van de datamanipulatie methoden uit de derde groep. Het opzet van deze methode is al de inputfiles, die door de verschillende jobs gecreëerd zijn, gezamenlijk te verplaatsen naar de broker machine.
- `GetData()`: Deze methode dient de inputfiles, die voor elke job verschillen, van de remote machines te vergaren, gebruik makend van de datamanipulatie methoden uit de derde groep.
- `PutData()`: Deze methode dient de gewenste resulterende data per job naar de broker te verplaatsen aan de hand van de datamanipulatie methodes uit de derde groep.
- `Execute()`: Deze methode dient de concrete uitwerking van een individuele taak te implementeren. Deze implementatie bestaat uit Java code die eventeel gecombineerd wordt met het uitvoeren van extra binaries of scripts.

5.6.3 Problemsolver Deployment

Aangezien het deployen van een problemsolver de verantwoordelijkheid is van de CoBRA grid gebruiker gebeurt dit op een eenvoudige manier. In essentie komt het erop neer dat er acht eenvoudige stappen doorlopen dienen te worden. Deze stappen worden verder besproken aan de hand van het onderstaande codefragment:

```
1 URI brokeruri =new URI(brokerurl);
2 CoBRAConnection connection = new CoBRAConnection(brokeruri);
3 Broker broker = (Broker)connection.getCoBRAPluglet(Broker.plugletname)
    ;
4 BrokerEnv env = broker.getBrokerEnv();
5 ProblemSolver solver = new AProblemSolver(classpath,...);
6 ProblemSolverCarrier carrier = new ProblemSolverCarrier(solver,solver.
    classpath);
7 env.deploy(carrier);
8 env.executeProblemSolver();
```

1. In de eerste stap wordt een URI aangemaakt naar de gewenste Broker machine.
2. In de tweede stap wordt een connectie opgezet naar de desbetreffende Broker machine.
3. In de derde stap wordt een Remote interface opgevraagd naar deze Broker component.
4. In de vierde stap wordt een Remote interface naar een BrokerEnvironment via deze Broker opgevraagd.
5. In de vijfde stap wordt een instantie aangemaakt van de Problemsolver die uitgevoerd dient te worden.

6. In de zesde stap wordt deze Problemsolver in een ProblemSolverCarrier geplaatst.

7. In de zevende stap wordt de Problemsolver op de Broker gedeployed.

8. In de achtste stap krijgt de Broker de opdracht de desbetreffende ProblemSolver uit te voeren.

5.6.4 Voorbeeld Implementatie

In deze sectie wordt, aan de hand van een voorbeeld, de eenvoud van een CoBRA distributie besproken. De taak die in dit voorbeeld uitgewerkt wordt, heeft tot doel het aantal cpu's van elke bruikbare machine in de CoBRA grid op te vragen. De implementatie bestaat, zoals reeds is aangehaald, uit twee delen.

In het eerste deel wordt een `Job` klasse geïmplementeerd. Deze implementatie heeft twee taken:

1. De naam van een machine, waarop de job uitgevoerd wordt, met het aantal cpu's wegschrijven in een file. Dit gebeurt door een FileOutputStream naar een file op het CoBRA file systeem aan te maken en er de gewenste data in te schrijven.

2. De file terugsturen naar de Broker.

Concreet gebeurt dit door de Job klasse uit te breiden en de `PutData()` en de `Execute()` methoden te implementeren. In het volgende stukje code worden de verschillende taken in de concrete code aangeduid[12]:

```
1  // implementeert taak 1
2  protected Object Execute(){
3    FileOutputStream test= this.getFileOutputStream("/" + file);
4    String resultdata = InetAddress.getLocalHost().getHostName() + "
         heeft " + Runtime.getRuntime().availableProcessors() + "
         processors";
5    test.write(resultdata.getBytes());
6    test.close();
7    return null;
8  }
9
10 // implementeert taak 2
11 protected boolean PutData(){
12   this.putRemotePSFile(filename, filename);
13   return true;
14 }
```

In het tweede deel wordt een `Problemsolver` klasse geïmplementeerd. Deze implementatie heeft vijf taken:

1. Het aanmaken van voldoende verschillende jobs zodat elke resource bereikt wordt (in dit geval 50 jobs).

[12] Voor de duidelijkheid zijn de try-catch blokken van de exceptionhandling weggelaten.

2. Het plaatsten van deze jobs in een JobCarrier.

3. Het versturen van de jobs.

4. Het samenvoegen van de verschillende files die door de jobs gecreëerd werden tot een result.txt file.

5. Het versturen van deze result.txt file naar de gebruiker.

Concreet gebeurt dit door de ProblemSolver klasse uit te breiden en de `PutData()` en de `run()` methoden te implementeren. In het volgende stukje code worden de verschillende taken in de concrete code aangeduid[13]:

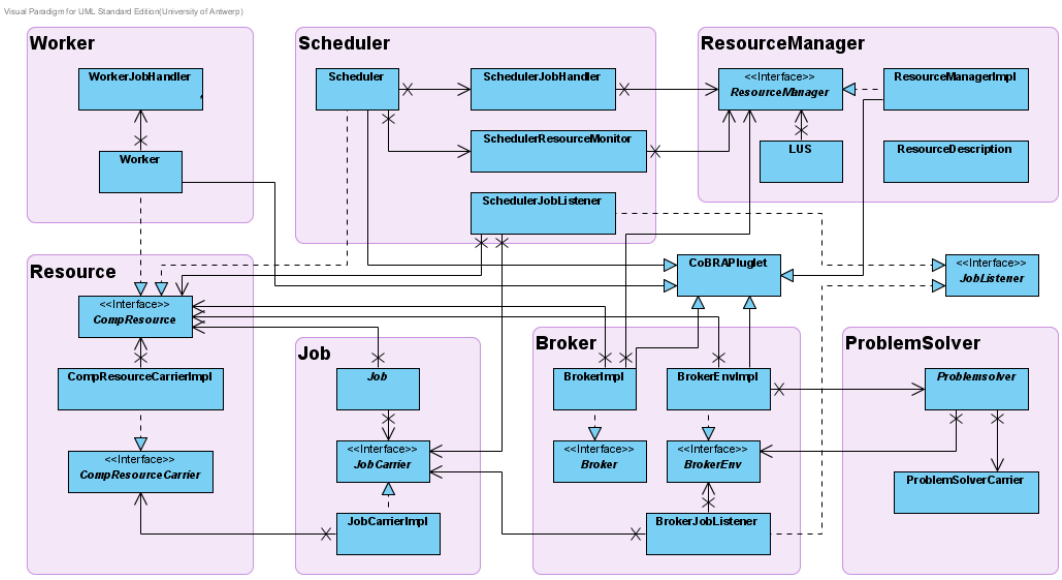
```
1  public void run() {
2    for(int i=0;i<50;i++){
3      // implementeert taak 1
4      VoorbeeldJob job = new VoorbeeldJob(brokerurl,i);
5      // implementeert taak 2
6      JobCarrier carrier = new JobCarrierImpl(job, classpath);
7      // implementeert taak 3
8      this.submit(carrier);
9    }
10
11   // Er wordt gewacht tot alle taken uitgevoerd zijn.
12   this.waitTillEmpty();
13
14   // de overige code van deze methode implementeert taak 4
15   File root = this.getFile("/");
16   System.out.println(root);
17   String[] list = root.list();
18   FileWriter writer = this.getFileWriter("result.txt");
19   for(int i=0;i<list.length;i++){
20     FileReader reader = this.getFileReader(list[i]);
21     int value = reader.read();
22     while(value!=-1){
23       writer.write(value);
24       value = reader.read();
25     }
26     writer.write(10);
27   }
28   writer.close();
29 }
30
31 // implementeert taak 5
32 public boolean putData() {
33   this.putRemoteFile("http://maori.cmi.ua.ac.be", "/result.txt", "/
       resultfiles/result.txt");
34   return true
35 }
```

Het concrete output resultaat is terug te vinden in appendix C.

[13]Voor de duidelijkheid zijn de try-catch blokken van de exceptionhandling weggelaten.

5.7 Klasse Diagramma

Na de sequentiële benadering van elk van de componenten in de CoBRA grid tracht deze sectie, aan de hand van het klassediagramma uit figuur 5.8, een kort globaal overzicht te creëren van het gehele framework.



Figuur 5.8: Het Klassediagramma van de CoBRA grid.

Het is meteen duidelijk dat elk van de besproken abstractielagen in het klassediagramma aanwezig zijn. In het geval van de User en de Middleware layer manifesteert zich dit in de aanwezigheid van de klassen die de verschillende componenten uit deze layers implementeren. Voor de duidelijkheid worden deze klassen per component gegroepeerd aan de hand van een roze afbakening. Naast deze twee top layers is de Low Level Layer ook duidelijk terug te vinden in elk van deze componenten. Dit onder de vorm van carriers (code transfer), listeners (notification) en LUS (discovery). Het filemanagement is hier bewust weggelaten omdat deze implementatie uitgegroeid is tot een individuele API die zich in een andere package bevindt. Voor de volledigheid dient er aldus nog vermeld te worden dat deze API zowel in de `BrokerEnvImpl` als in de `Worker` klasse gebruikt wordt.

Aan de hand van de verschillende pijlen in het diagramma wordt het verhaal van het framework duidelijk. Dit wordt kort samengevat in een aantal aandanchtspunten:

- Elke component van de middleware layer is afgeleid van de `CoBRAPluglet` klasse.
- Een ProblemSolver wordt via een ProblemSolverCarrier op een BrokerEnv op

de Broker gedeployed en uitgevoerd.

- Een Job wordt via een JobCarrier op een Resource uitgevoerd.
- De Broker en de Scheduler gebruiken de ResourceManager om de verschillende beschikbare resources op te vragen.
- Notification wordt ondersteund door de Broker en de SchedulerJobListener die van de JobListener klasse zijn afgeleid.

5.8 Toepassingen, features en testen

5.8.1 Introductie

In deze sectie wordt het gehele framework getest op drie verschillende niveaus:

- *CoBRA-developer niveau*: Er wordt nagegaan hoe eenvoudig het is voor de CoBRA developer extra functionaliteit toe te voegen aan een bestaande component.
- *Grid-researcher niveau*: Er wordt nagegaan hoe eenvoudig het is voor een grid-researcher componenten uit te breiden en aldus nieuwe functionaliteit te voorzien.
- *Grid-user niveau*: Er wordt nagegaan hoe eenvoudig het is voor een grid-user een probleemstelling te distribueren op de CoBRA grid.

5.8.2 Multi-Core/Hyperthreading support

In deze sectie wordt Multi-Core/Hyperthreading support aan de CoBRA functionaliteit toegevoegd. Deze toevoeging is, naast een test op het CoBRA-developer niveau, tevens functionaliteit die essentieel is in de huidige Multi-Core desktop opstellingen. De techniek maakt het mogelijk verschillende taken simultaan op een machine uit te voeren om aldus de verschillende cores optimaal te benutten.

Integratie

Om deze techniek te verwezenlijken, dient enkel de Worker aangepast te worden. Tot op heden werd er slechts één `WorkerJobHandler` thread gebruikt. Wanneer dit aantal verhoogd wordt naar het aantal cores kan op elke core simultaan een job uitgevoerd worden. Het is echter wel belangrijk dat erop gelet wordt dat elke `WorkerJobHandler` het idee krijgt een eigen filesysteem voorhanden te hebben. Dit werd opgelost aan de hand van een `WorkerJobHandler` specifieke directory structuur.

Concreet wordt het volgend stukje code uit de `init` methode van de `Worker`:

```
1 handler = new WorkerJobHandler(queue, datalist, this);
2 handler.start();
```

vervangen door het volgend stuk code:

```
1 int amount = Runtime.getRuntime().availableProcessors();
2 handlers=new WorkerJobHandler[amount];
3 for(int i=0;i<amount;i++){
4   handlers[i] = new WorkerJobHandler(queue,datalist,i,this);
5   handlers[i].start();
6 }
```

Tevens wordt aan de id in de `WorkerJobHandler`, die gebruikt wordt voor het filemanagement, het nummer van de desbetreffende `WorkerJobHandler` toegevoegd.

Conclusie

Na uitgebreide testen met zowel Hyperthreading machines als Dual core machines als een Quad core machine blijkt deze techniek uitstekend te werken. De eenvoud van de toegevoegde code geeft een indicatie van de eenvoud waarmee de CoBRA grid kan aangepast worden. Er dient hier echter wel opgemerkt te worden dat het niet bewezen is dat meer complexe veranderingen even vlot toegevoegd kunnen worden. De eerste indicaties zijn echter wel positief.

5.8.3 Checkpointing

Introductie

Checkpointing wordt het best omschreven als een techniek die *fault tolerance* in een framework tracht te ondersteunen. De techniek tracht, bij de afhandeling van een Job, op welbepaalde tijdstippen een snapshot te nemen van de huidige status van deze Job. Indien de resource die de job afhandelt, faalt, kan de Job op een andere resource hervat worden vanaf het laatste snapshot zonder dat de voorafgaande berekeningen verloren gaan. Het spreekt voor zich dat deze techniek vooral belangrijk is bij het uitvoeren van lange taken (taken die de uptime van een resource durven te overschrijden).

Deze sectie introduceert enerzijds een techniek die onontbeerlijk is in een hedendaagse grid infrastructuur en bewijst anderzijds dat de `Job` basisklasse eenvoudig uit te breiden is tot een alternatieve implementatie.

Integratie

De techniek wordt geïmplementeerd door de `Job` klasse uit te breiden met de gewenste functionaliteit. Voor deze implementatie is er vanuit gegaan dat elk van de problemen zal bestaan uit een introductiefase gevolgd door een lus waarin de gewenste functionaliteit wordt uitgevoerd en tot slot een afrondingsfase. Na elke iteratie van de loop wordt het tijdsinterval tot de vorige checkpoint berekend. Indien dit de checkpointintervalwaarde overschrijdt, wordt er een snapshot van de taak genomen. Om dit te verwezenlijken, wordt het marshalling principe van het CoBRA systeem overgenomen. De gemarshalde versie van de job in de jobcarrier wordt vervangen door de huidige versie van de job. Indien een Job geresheduled wordt, zal de nieuwe structuur van de Job ervoor zorgen dat ze herstart wordt vanaf de laatste checkpoint.

Concreet worden er vier nieuwe abstracte methoden bijgemaakt in de CheckPointingJob:

- `StartupImpl(JobData data)`: In deze methode moet een eventuele gebruiker de initialisatiefase van de desbetreffende Job implementeren.
- `LoopProcImpl(JobData data)`: In deze methode moet de code binnen de loop van de desbetreffende Job geïmplementeerd worden.
- `FinalizeImpl(JobData data)`: In deze methode wordt de afrondingsfase van de Job geïmplementeerd.
- `CheckLoopImpl(JobData data)`: In deze methode wordt de check geïmplementeerd die aangeeft wanneer het uitvoeren van de loop mag stoppen.

De `JobData` die aan elke methode wordt meegegeven, bevat de lokale data die in elk van de methoden noodzakelijk zijn. Deze klasse dient door de gebruiker zelf uitgebreid te worden met de noodzakelijke datavelden.

De functionaliteit die het geheel mogelijk maakt, zit vervat in de implementatie van de abstracte `Execute()` methode uit de standaard `Job` klasse. De code ziet er als volgt uit:

```
1  protected final Object Execute(){
2    if(state==0){
3      Startup();
4      state++;
5    }
6    if(state == 1){
7      while(CheckLoop()){
8        if((new Date().getTime()-data.getPreviousTime())>data.
             getCheckPointInterval()){
9          CheckPoint();
10         data.setPreviousTime(new Date().getTime());
11       }
12       LoopProc();
13     }
14     state++;
15   }
16   return Finalize();
17 }
```

Eerst wordt er gecheckt of de job reeds gedeeltelijk is uitgevoerd. Dit gebeurt aan de hand van de `state` variabele. Indien die op 0 staat, is er nooit een checkpoint genomen en kan er bij de initialisatiefase (`Startup`) gestart worden. Indien deze status 1 is, werd er reeds een snapshot genomen en dient er herstart te worden binnen de lus. Aangezien al de data die tijdens de laatste checkpoint geserialiseerd werden nu worden gedeserialiseerd (marshalled, unmarshalled) kan de uitvoering eenvoudigweg bij het begin van de lus hervat worden. De Job zal immers een identieke kopie zijn van de job toen het laatste snapshot werd genomen. Telkens er een nieuwe lusiteratie start, wordt het tijdsinterval tot de vorige checkpoint berekend. Indien dit groter is dan het checkpointinginterval, plaatst de `CheckPoint` methode de huidige job geserialiseerd in de JobCarrier en wordt het tijdstip, waarop de laatste snapshot werd genomen, vervangen door het huidige tijdstip.

Conclusie

Deze techniek werd aangewend om de gedistribueerde versie van het GIPSy framework uit het volgende hoofdstuk te implementeren. De implementatie bleek degelijk te werken. Het introduceerde enkel overhead bij het serialiseren en deserialiseren van Jobs indien de omvang van de `JobData` aanzienlijk was. Dit gebeurde vooral wanneer met grote matrices gewerkt werd. De oorzaak van deze overhead ligt in de combinatie van intensieve marshalling procedures en van het transporteren van het geserialiseerde object over het netwerk naar de JobCarrier. De JobCarrier is immers een remote object dat op de Broker machine aanwezig is. Deze overhead was echter verwaarloosbaar aangezien het nemen van een checkpoint enkele minuten duurde en het uitvoeren van de taak soms tot 48 uur duurde.

Naast deze vaststellingen dient tevens opgemerkt te worden dat deze checkpointing techniek enkel werkt voor Java implementaties en geen ondersteuning biedt bij het uitvoeren van binary code.

Tot slot kan gesteld worden dat deze test aantoont dat een Grid-researcher op een eenvoudige manier een `Job` kan uitbreiden om aldus extra functionaliteit te voorzien.

5.8.4 Economic based scheduler

Inleiding

Economic based scheduling bestaat erin resources aan te bieden tegen een bepaalde prijs. Het opzet van deze mechanismen is cpu time te verkopen tegen de huidige marktwaarde. Dit wil zeggen dat de prijs wordt bepaald aan de hand van vraag en aanbod. Onderzoek naar verschillende marktmodellen is terug te vinden in: [116][111][117]

Deze sectie toont aan dat dergelijke systemen eenvoudig in CoBRA te integreren zijn door een Grid-researcher.

Integratie

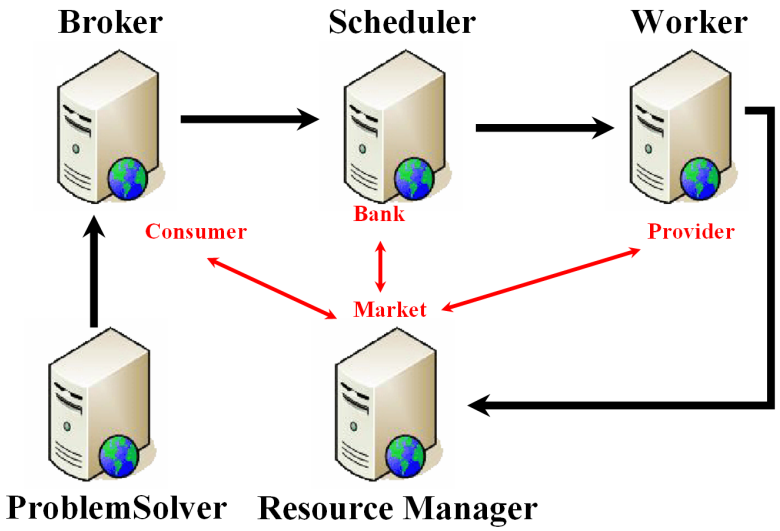
De integratie die uitgewerkt wordt in [121][14] had tot doel een commodity markt model op te zetten binnen de CoBRA grid. In een commodity markt model wordt de prijs bepaald aan de hand van het prijsevenwicht. Dit evenwicht wordt gedefinieerd door de plaats waar de vraagcurve en de aanbodcurve elkaar snijden. De techniek waarop de implementatie van het marktmodel berust, werkt aan de hand van *tokens*. Deze tokens worden door de scheduler aan een problemsolver toegekend om hem aldus de toelating te geven een bepaald aantal jobs door een bepaald type resource te laten uitvoeren. Het onderscheid dat hier gecreëerd is, werkt op basis van de verschillende CPU's maar kan ook uitgebreid worden naar andere karakteristieken. Dit concept is uitgewerkt op basis van een uitbreiding van zes componenten:

1. `Worker`: Deze component wordt zo uitgebreid dat de CPU informatie kan opgevraagd worden.

[14] De Licentiaatsthesis van Sam Verboven in samenwerking met Kurt Vanmechelen en Peter Hellinckx.

2. `ResourceManager`: Deze component wordt zo uitgebreid dat het twee extra taken kan uitvoeren:
   - Het ter beschikking stellen van de resources in combinatie met hun CPU informatie.
   - Het toekennen van tokens aan de verschillende ProblemSolvers op basis van een commodity markt.
3. `Scheduler`: Deze component wordt zo uitgebreid dat het twee extra taken kan uitvoeren:
   - Op basis van het token, jobs schedulen op het type worker waarop de job recht heeft (een worker met een bepaald type CPU).
   - Nagaan of jobs over een valabel token beschikken. Het token moet toegekend zijn door de scheduler. Het aantal resources waarop een token recht geeft, mag niet overschreden worden.
4. `Broker`: Deze component wordt zo uitgebreid dat het tokens kan aanvragen bij de ResourceManager.
5. `ProblemSolver`: De ProblemSolver wordt zo uitgebreid dat hij, op basis van de beschikbare tokens, jobs kan versturen.
6. `JobCarrier`: Een JobCarrier wordt zo uitgebreid dat het over een token kan beschikken waarmee het recht op een resource kan aangetoond worden op de Scheduler.

Dit alles wordt duidelijk aan de hand van figuur 5.9.



Figuur 5.9: Stuctuur van de economic scheduling techniek op de CoBRA grid.

De ProblemSolver wordt op de Broker gedeployed. Onder de vorm van deze Broker wordt hij een resource consumer in het globale markt model. Aan de hand van de huidige markt (de aanwezige resources (Providers) en de vraag naar deze resources) en van het budget van de Problemsolver koopt de Consumer tokens van de ResourceManager. Deze tokens worden door de Jobs bij de Scheduler (de Bank) ingewisseld voor een resource. De Job wordt vervolgens naar de overeenkomstige Resource (provider) gestuurd.

Meer details over de implementatie zijn terug te vinden in [121].

Conclusie

Het werk in [121] concludeert met:

> Na de initiële kennismaking, kon de verdere implementatie zonder noemenswaardige problemen doorgevoerd worden. De uitbreiding maakt optimaal gebruik van de reeds bestaande CoBRA infrastructuur door enkel een laag boven CoBRA te definiëren en verder alle reeds bestaande functionaliteit te hergebruiken.

Hieruit kan geconcludeerd worden dat de implementatie van een Economic based scheduler op een eenvoudige manier op de CoBRA grid gedeployed kan worden en bijgevolg de doelstelling van CoBRA, in dit geval, bereikt is.

5.8.5 Prediction based scheduler

Het gebruik van desktop grids brengt met zich mee dat resources niet ten allen tijde ter beschikking zijn. Prediction based scheduling tracht, aan de hand van voorspelde informatie, Jobs op een zo optimaal mogelijke manier te verdelen over de verschillende resources. Concreet houdt dit in dat, aan de hand van voorspellingen van zowel de uptime van de resources als de runtimes van de Jobs, getracht wordt deze zo te mappen dat elke resource tijdens zijn volledig availability window in gebruik is en dat de jobs op een zo snel mogelijke manier afgehandeld worden. De belangrijkste doelstellingen waarnaar gestreefd moet worden, zijn:

- Voorkomen dat een job gerescheduled moet worden. Dit gebeurt indien een job nog niet is afgehandeld wanneer een resource niet langer beschikbaar is.
- Resources mogen niet idle zijn. Dit gebeurt wanneer er geen jobs op uitgevoerd worden.

Deze complexe materie is een belangrijk onderdeel van deze thesis en wordt in meer detail besproken in Deel III, “Prediction Based Scheduling”.

5.8.6 Globus port

Naast het gebruik van desktop machines in de CoBRA grid blijft het interessant ook dedicated machines als resource aan het framework te kunnen toevoegen. Deze dedicated resources worden, in de meeste situaties, gegroepeerd aan de hand van HWG of HL-LWG systemen zoals Globus en Condor. Deze sectie toont aan dat het, aan de hand van een uitbreiding van de abstracte Resource klasse, mogelijk is de resources uit dergelijke gridsystemen te gebruiken binnen het CoBRA framework.

In de implementatie, die uitgewerkt werd voor het CoBRA framework, wordt gebruik gemaakt van *Cogkit 4.1.3* (Commodity Grid Kit)[3]. Deze toolkit biedt een api die het mogelijk maakt jobs, vanuit een Java programma, op Globus [34] of op Condor [113] te laten uitvoeren. Deze techniek wordt gebruikt om, vanuit een machine met de nodige rechten, Jobs op Globus of Condor te submitten. In essentie komt het erop neer dat de abstracte Resource klasse wordt uitgebreid en dat de `Execute` methode, de Jobs via Cogkit op het gewenste framework distribueert.

Het systeem is concreet getest op een lokale Condor pool en op BEGrid. De resultaten wijzen uit dat deze techniek effectief werkt maar een enorme vertraging met zich mee brengt. Het duurt immers veel langer tot een Job werkelijk wordt uitgevoerd op een dergelijk systeem. Het aantal resources dat op dergelijke manier kan toegevoegd worden, is echter enorm en dit kan grote voordelen bieden bij het uitvoeren van vele langdurige taken. In tegenstelling tot de roundtriptime, kan dit de throughput van de CoBRA grid in sterke mate verbeteren.

Algemeen kan er geconcludeerd worden dat een dergelijke integratie mogelijk is maar dat er rekening moet gehouden worden met de impact die de specificaties van het gridsysteem dat wordt gebruikt heeft op de CoBRA grid (vb. lange roundtrip time).

5.8.7 Sleeptesten

Introductie

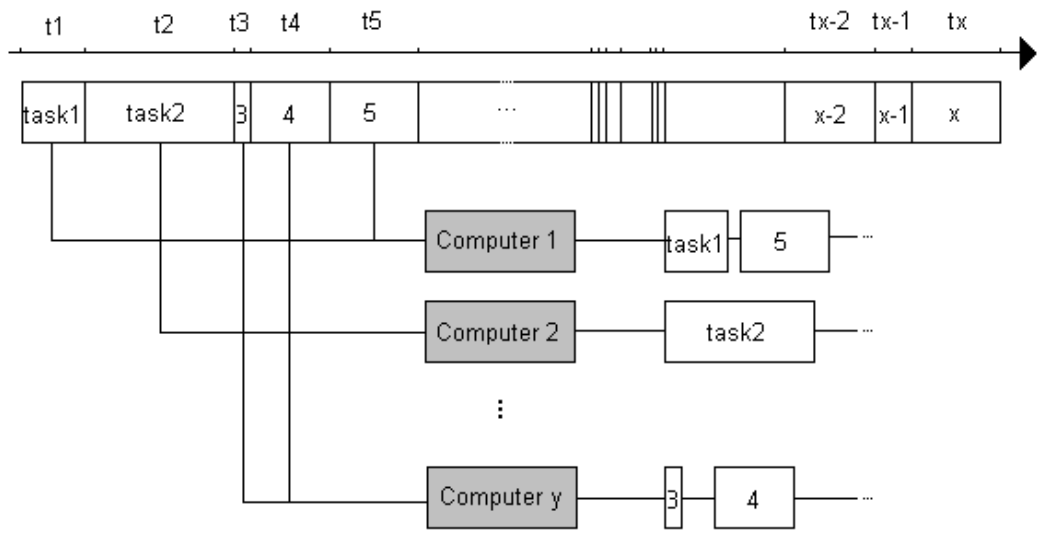
Naast de uitbreidbaarheid van het CoBRA framework is het ook belangrijk performantietesten uit te voeren. In deze sectie wordt, op basis van het sleeptest mechanisme, enerzijds het gebruiksgemak voor de Grid-user getest en anderzijds een performantieanalyse van de CoBRA grid uitgevoerd.

Sleeptesting

De sleeptesting techniek werd geïntroduceerd en voor het eerst gebruikt in de performantieanalyse van het Condor[113] framework. Meer informatie omtrent deze analyse is terug te vinden in [48]. De basiseigenschappen van deze testtechniek kunnen samengevat worden als: *total problem range coverage, simulation correctness en easy definability.* Hiermee wordt bedoeld dat deze techniek, aan de hand van een beperkte set parameters, elk gedistribueerd probleem kan simuleren op een manier die niet te onderscheiden is van de originele distributie. Dit wordt verwezenlijkt door de jobs, de bouwstenen van een gedistribueerd probleem, rechtstreeks te manipuleren. Figuur 5.10 geeft een typisch distributieschema weer.

Het schema bestaat uit x verschillende taken die elk onafhankelijk van elkaar uitgevoerd kunnen worden. Er bestaat geen relatie tussen de taken en/of de computers die de taken uitvoeren.

Verschillende distributieschemas kunnen van elkaar onderscheiden worden aan de hand van hun takenlijst. Het ondersteunen van *total problem range coverage* impliceert dat de techniek in de mogelijkheid moet zijn elke mogelijke takenlijst te simuleren. Dit wordt gerealiseerd door elke taak in een takenlijst te vervangen door een dummy functie die een resource gedurende dezelfde periode als de taak belast.



Figuur 5.10: Een typisch distributie schema: x taken van lengte t_a worden verdeeld over y computers

.

De structuur van de CoBRA grid laat toe hiervoor standaard sleep methoden te gebruiken. De belasting van een resource wordt immers bepaald aan de hand van het aantal taken dat er op de resource verwerkt wordt en niet aan de hand van de cpu belasting. Figuur 5.11 toont de takenlijst die ontstaat wanneer de lijst uit figuur 5.10 gesimuleerd wordt.

Figuur 5.11: Simulatie van de takenlijst uit figuur 5.10.

Aangezien het aantal taken in een distributie vrij hoog kan liggen, is het vrijwel onmogelijk een takenlijst manueel te definiëren. Om dit probleem te verhelpen, is een *testgenerator* ontwikkeld. De *testgenerator* construeert de gewenste takenlijsten aan de hand van een grafische user interface of aan de hand van een XML-file.

Het gebruik van de sleep testing techniek heeft twee grote voordelen:

- Aangezien de taken geen cpu load genereren, kunnen de testen uitgevoerd worden op machines zonder dat de gebruiker dit merkt.

- Het feit dat de taken geen load genereren, impliceert tevens dat de overhead die gegenereerd wordt enkel van de middleware zelf kan komen en aldus eenvoudig opgemeten kan worden.

Implementatie

De implementatie gebeurt, net zoals bij elke CoBRA distributie van een probleemstelling, door de `ProblemSolver` en de `Job` klasse uit te breiden. Het uitbreiden van de `Job` klasse gebeurt eenvoudig door de `Thread.sleep(sleeptime)` methode in de `Execute()` methode te plaatsen. Het uitbreiden van de `ProblemSolver` gebeurt in twee lagen. In de eerste abstractie laag, de `AbstractSleepProblemSolver`, wordt de Problemsolver als volgt uitgebreid:

```
1 public void run() {
2   LinkedList thelist = GetJobList();
3   for(int i=0; i<thelist.size();i++){
4     JobCarrier tmpcarrier = new JobCarrierImpl((Job)thelist.get(i),
          this.classpath);
5     this.Submit(tmpcarrier);
6   }
7 }
```

De implementatie vraagt een JobList op aan de hand van de abstracte `GetJobList()` methode. Deze methode moet dan uitgebreid worden om verschillende soorten sleeptests te implementeren.

Concreet wordt in de testen voor dit werk gebruik gemaakt van twee verschillende `GetJobList()` implementaties. De eerste implementatie maakt gebruik van een instelbare lijst van sleepjobs met aanpasbare runtime. De tweede groep leest tasklist descripties in van een file. Er worden in dit geval Jobs gecreëerd die overeenkomen met de beschrijving uit de ingelezen takenlijst. Op deze manier kunnen simulaties van echte problemen en simulaties gecreëerd door de TestGenerator uitgevoerd worden.

Testen en resultaten

In deze sectie wordt een profiel opgesteld voor de overhead van het CoBRA grid systeem, net zoals dit in [48] gebeurde voor Condor. Hiertoe definiëren we volgende kwantiteitswaarden:

1. *Performantie* P: De tijd die nodig is om x taken met runtime t_x te verdelen over y resources.

2. *Overhead* OH: Het verschil tussen de performantie P en de sequentiële runtime gedeeld door het aantal resources.

$$OH = P - \left\lceil \frac{T}{R} \right\rceil \times t_m \tag{5.1}$$

   met T het aantal taken, R het aantal resources en t_m de gemiddelde runtime van een job

3. *Percentage overhead* POH: Het percentage overhead binnen de performantietijd.

$$POH = \frac{OH}{P} \times 100\% \tag{5.2}$$

4. *Speedup SU*: Het aantal keer dat een gedistribueerde run sneller is dan een sequentiële run.

$$SU = \frac{T \times t_m}{P} \tag{5.3}$$

Elk van de uitgevoerde testen gebeurde op een verzameling computers waarvan de specificaties variëren van PIV 2.4GHZ processoren tot PIV 3GHZ processoren met 512 of 1024 MB RAM geheugen.

De testen in deze sectie worden opgesplitst in twee groepen. In een eerste groep wordt, aan de hand van specifiek gekozen testen, getracht een profiel op te stellen van de overhead van de CoBRA grid. In de tweede groep wordt dit profiel aan de hand van een aantal relevante testsets geverifieerd.

De eerste groep testsets bestaat uit verzamelingen van 100 en 1000 taken (T) met een lengte (l) van 1, 2, 5, 10, 20, 50, 100, 200, 500 en 1000 seconden die worden uitgevoerd op 10, 25 en 50 resources (R). Elk van deze sets wordt 10 keer of meer uitgevoerd om statistisch verantwoord te zijn. Na het verwijderen van de outliers worden de volgende resultaten bekomen:

$l(s)$	Overhead(OH) (ms)					
	$T = 100$			$T = 1000$		
	$R = 10$	$R = 25$	$R = 50$	$R = 10$	$R = 25$	$R = 50$
1	23844	15703	13047	129484	60266	35062
2	20109	15078	11032	128907	57547	35063
5	21172	15078	10015	130453	58656	35094
10	23297	14141	10063	134297	65172	34172
20	24500	15234	10031	132140	59016	35188
50	22094	13469	10047	133032	58124	35390
100	21125	13859	15078	129656	59384	32625
200	22078	14983	10156	136156	60187	35969
500	20969	15154	15266	132812	58921	35390
1000	22812	15926	15500	134104	58400	34781
Gemiddeld	22200	14862,5	12023,5	132104,1	59567,3	34873,4

Tabel 5.2: De overhead bij de distributie van T taken van lengte l over R resources.

Tabel 5.2 toont duidelijk dat de overhead OH onafhankelijk is van de lengte van de taken, enkel het aantal taken en het aantal workers blijken belangrijk bij het bepalen van het profiel van de CoBRA overhead. Gebruik makend van multilineaire regressie wordt het profiel van de CoBRA overhead (in ms) benaderd met de functie:

$$CoBRA_{OH} = \left\lceil \frac{T}{R} \right\rceil \times 1223.3 + 10054.9 \tag{5.4}$$

Hieruit kan geconcludeerd worden dat CoBRA een globale startup kost heeft van om en bij de 10 seconden. Deze startup kost bestaat uit:

- De tijd nodig om de problemsolver te deployen.
- De tijd nodig om de eerste job naar de scheduler te submitten.
- De tijd nodig om de eerste job te schedulen.
- De tijd nodig om de eerste job op de resource te deployen.
- De notification delays.

Tevens ondervindt de CoBRA grid een kost van 1.2 seconden per Job die verdeeld wordt over de verschillende resources. Dit wil zeggen dat elke job, in het tijdsinterval dat hij zich op de resource bevindt, een delay van 1.2 seconden ondervindt. Deze delay wordt toegeschreven aan:

- Het ophalen van de codebase bij de webserver.
- Notification delays.

Bovenop deze analyse is het belangrijk vast te stellen dat de OH omgekeerd evenredig is met het aantal workers. Dit brengt met zich mee dat het systeem optimaal schaalbaar is, en gebruikt kan worden in large scale distributies.

Om een beter visueel beeld te krijgen van deze resultaten geven tabel 5.3 en tabel 5.4 respectievelijk de speedup en het percentage overhead weer van de verkregen resultaten.

$l(s)$	Speedup(SU)					
	$T = 100$			$T = 1000$		
	$R = 10$	$R = 25$	$R = 50$	$R = 10$	$R = 25$	$R = 50$
1	2,95	5,08	6,65	4,36	9,97	18,16
2	4,99	8,67	13,30	6,08	14,54	26,64
5	7,03	14,25	24,98	7,93	19,33	37,01
10	8,11	18,47	33,26	8,82	21,50	42,70
20	8,91	21,00	39,98	9,38	23,28	45,96
50	9,58	23,42	45,44	9,74	24,29	48,29
100	9,79	24,16	46,49	9,87	24,63	49,20
200	9,89	24,54	48,76	9,93	24,81	49,55
500	9,96	24,81	49,25	9,97	24,93	49,82
1000	9,98	24,90	49,62	9,99	24,96	49,91

Tabel 5.3: De speedup bij de distributie van T taken van lengte l over R resources.

Beide tabellen tonen aan dat het verdelen van taken optimaler verloopt bij een grotere taaklengte. Dit is een logische vaststelling aangezien de invloed van de OH, die onafhankelijk is van de taaklengte, wordt weggedeeld door deze taaklengte die wel invloed heeft op de performantie P. Er kan besloten worden dat, in de context van het gebruikte distributeiplatform de verdeling optimaal gebeurt voor taken met

$l(s)$	Percentage overhead(POH)					
	$T = 100$			$T = 1000$		
	$R = 10$	$R = 25$	$R = 50$	$R = 10$	$R = 25$	$R = 50$
1	70,45%	79,70%	86,71%	56,42%	60,11%	63,68%
2	50,14%	65,33%	73,39%	39,19%	41,84%	46,71%
5	29,75%	42,98%	50,04%	20,69%	22,68%	25,98%
10	18,90%	26,12%	33,47%	11,84%	14,01%	14,59%
20	10,91%	16,00%	20,05%	6,20%	6,87%	8,09%
50	4,23%	6,31%	9,13%	2,59%	2,82%	3,42%
100	2,07%	3,35%	7,01%	1,28%	1,46%	1,61%
200	1,09%	1,84%	2,48%	0,68%	0,75%	0,89%
500	0,42%	0,75%	1,50%	0,26%	0,29%	0,35%
1000	0,23%	0,40%	0,77%	0,13%	0,15%	0,17%

Tabel 5.4: Het percentage overhead POH bij de distributie van T taken van lengte l over R resources.

een runtime die een minuut overschrijdt.

In wat nu volgt, wordt, aan de hand van de tweede groep testsets, de relevantie van de overheadfunctie $CoBRA_{OH}$ voor echte gedistribueerde problemen aangetoond. Deze tweede groep bestaat uit drie testsets waarvan elke taak een verschillende runtime heeft. De eerste bestaat uit timings, afkomstig van de uitvoer van een computationeel quantum fysisch probleem dat in sectie 5.8.8 in meer detail besproken wordt. De takenlijst wordt gedurende deze test niet effectief uitgevoerd, maar vervangen door een lijst sleepjobs die dezelfde runtime hebben als de echte taken. De overige twee testsets bestaan uit runtime data gegenereerd aan de hand van de eerder beschreven testsetgenerator. De eigenschappen van de testsets worden weergegeven in tabel 5.5.

	Aantal taken (T)	Sequentiele runtime ($T \times t_m$)	Beschrijving
QF Test	5670	18271,706 s	Veel kleine taken + enkele grote
Test 1	4000	23907,2033 s	Evenveel kleine als grote taken
Test 2	2457	100769,1452 s	Veel grote taken + enkele kleine

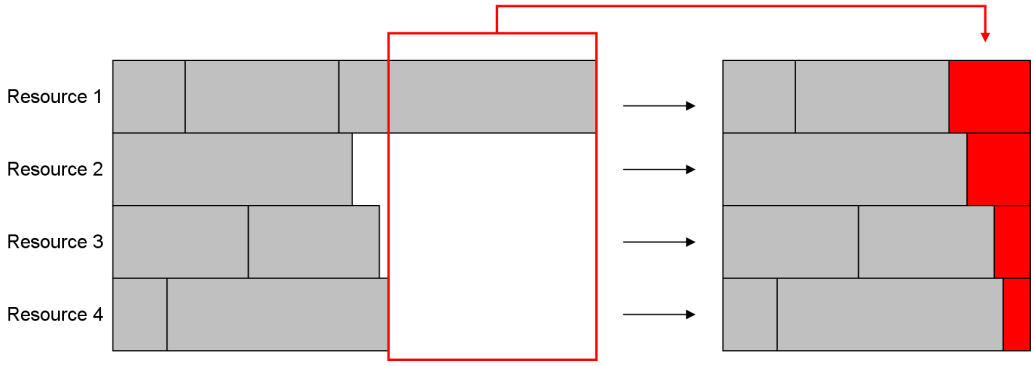
Tabel 5.5: Eigenschappen van de drie gebruikte testsets.

Tabel 5.6 toont de resultaten na uitvoer op de CoBRA grid. Op het eerste gezicht blijken deze resultaten niet bevredigend. De oorzaak van deze afwijking ligt in de structuur van de tweede groep testsets. De runtime van de verschillende taken is niet langer gelijk en het aantal taken is niet langer deelbaar door het aantal resources. Dit heeft tot gevolg dat het, aan het eind van de run, mogelijk is dat slechts

	$R = 10$	prediction	$R = 25$	prediction	$R = 100$	prediction
QF	714829,40	703666.00	302287,76	287499.34	175706,88	148777,12
Test 1	496685,72	499374,90	214117,87	205782,90	119964,94	107918,90
Test 2	360569,56	310619,71	191859,19	130280,82	98024,10	70167,86

Tabel 5.6: Het percentage overhead POH bij de distributie van T taken van lengte l over R resources.

een beperkt aantal workers nog in gebruik zijn. Dit wordt geschetst in figuur 5.12.



Figuur 5.12: Uitvoering van de laatste taak wordt gespreid voor de berekening van de OH.

Het is duidelijk dat de afwijking meer uitgesproken is in de QF test en in Test 2. Dit wordt verklaard door het feit dat de taken, die als laatste gescheduled worden in beide tests, grote taken zijn. Deze taken blijven aldus op een beperkt aantal nodes doorlopen, terwijl de andere nodes idle worden. In Test 1 daarentegen worden aan het einde enkel kleine taken gescheduled. Wanneer wordt gesimuleerd dat de laatste taken evenredig verdeeld worden over de verschillende resources geeft dit de resultaten in tabel 5.7

	$R = 10$	prediction	$R = 25$	prediction	$R = 100$	prediction
QF	705888,32	703666,00	283147,27	287499,34	149284,88	148777,12
Test 1	496685,70	499374,90	204856,87	205782,90	108531,94	107918,90
Test 2	311281,41	310619,71	129851,09	130280,82	70753,84	70167,86

Tabel 5.7: Het percentage overhead POH bij de distributie van T taken van lengte l over R resources.

Deze resultaten blijken overeen te komen met hetgeen het CoBRA profiel voorspelt. Dit bewijst dat het profiel kwalitatief voldoet. Het is belangrijk hier tot slot bij op te merken dat de testen uit deze sectie geen gebruik maakten van datatransfer. De impact van dit onderdeel wordt besproken in sectie 5.8.8.

Conclusie

In deze sectie werd een performantieprofiel voor het CoBRA framework opgezet aan de hand van sleeptest distributies. In eerste instantie bevestigt deze test dat het eenvoudig en efficient is een probleem te distribueren op de CoBRA grid. Bovendien toont het profiel aan dat de overhead voor file onafhankelijke distributies beperkt blijft tot een vaste overhead van 10 seconden per distributie en een kost van 1,2 seconden per job. Het feit dat deze waarden laag zijn en dat het toevoegen van resources de overhead verlaagt, geeft aan dat het systeem schaalbaar is en bruikbaar voor large scale distributies.

5.8.8 Quantum Physics Testen

Introductie

In deze sectie wordt de distributie van een quantum fysisch probleem toegelicht. Deze introductie geeft enkel een korte beschrijving van dit probleem en voor meer probleem specifieke details verwijzen we naar [22]. De ^5H kern heeft een groot neutronenoverschot en ligt bijgevolg boven de neutron drip line. Omwille van deze reden trachten tal van onderzoeken het bestaan van resonantiestructuren in ^5H te bewijzen.

Verschillende theoretische modellen en methoden werden reeds aangewend om de energie en de breedte van de resonantie te berekenen. In dit werk ligt de focus op het *three-cluster* model [120] dat werd voorgesteld in de context van de *Modified J-Matrix* methode [54]. De ^5H kern wordt gemodelleerd met een $^3H+n+n$ drie-cluster configuratie.

De drie-cluster golffunctie kan geschreven worden als:

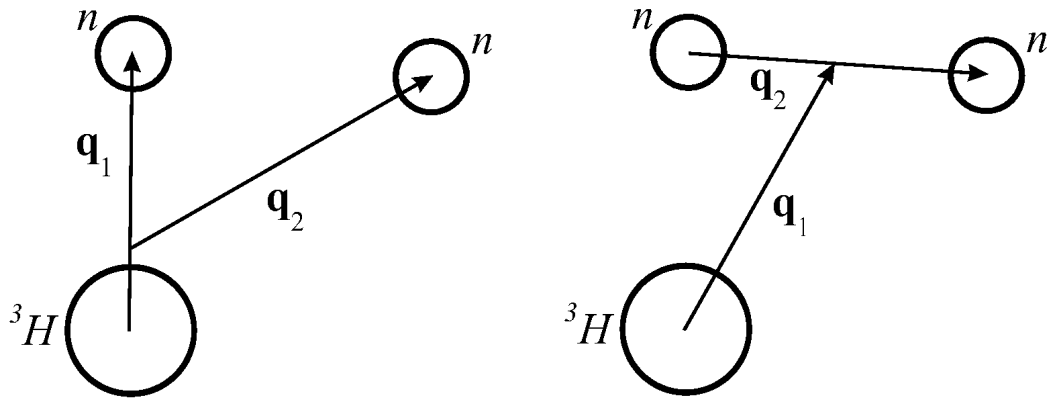
$$\Psi_{SL;JM} = \widehat{\mathcal{A}} \left\{ \left[\Phi_1 (t) \, \Phi_2 (n) \, \Phi_2 (n) \right]_S \phi_L (\mathbf{q}_1, \mathbf{q}_2) \right\}_{JM} \tag{5.5}$$

met $\Phi_1 (t)$ een antisymmetrische shell-model golffunctie die de interne structuur van ^3H met drie nucleonen in de s-shell beschrijft. De neutronen golffunctie $\Phi_2 (n)$ includeert enkel de spin en de isospin variabelen van het neutron. $\widehat{\mathcal{A}}$ definieert de algemene antisymmetrisatie operator. Het relatieve gedrag van de clusters kan beschreven worden aan de hand van twee sets Jacobi coördinaten, zoals gevisualiseerd in figuur 5.13.

De inter-cluster golffunctie $\phi_L (\mathbf{q}_1, \mathbf{q}_2)$, die de relatieve drie-cluster beweging beschrijft, wordt bekomen door de Schrödinger vergelijking met de geschikte randvoorwaarden op te lossen. De golffunctie $\phi_L (\mathbf{q}_1, \mathbf{q}_2)$ wordt hiertoe ontwikkeld in een Hypersferische Harmonische basis:

$$\begin{aligned} \phi_L (\mathbf{q}_1, \mathbf{q}_2) &= \sum_{l_1, l_2} \phi_{l_1, l_2; L} (\rho, \theta) \left\{ Y_{l_1} (\mathbf{q}_1) \, Y_{l_2} (\mathbf{q}_2) \right\}_{LM} \\ \phi_L (\mathbf{q}_1, \mathbf{q}_2) &= \sum_{K, l_1, l_2} \phi_{K; l_1, l_2; L} (\rho) \, \chi_K^{(l_1, l_2)} (\theta) \left\{ Y_{l_1} (\mathbf{q}_1) \, Y_{l_2} (\mathbf{q}_2) \right\}_{LM} \end{aligned} \tag{5.6}$$

Het hypermomentum K en de partiële draaimomenten l_1 (volgens $\mathbf{q}_1$) en l_2 (volgens $\mathbf{q}_2$) bepalen de drie-cluster geometrie en karakteriseren de verschillende



Figuur 5.13: Mogelijke keuze van Jacobi coördinaten voor de ${}^3H+n+n$ configuratie in ${}^5H$.

verstrooiings kanalen. Deze drie quantum getallen worden samen gevat als $c = \{K; l_1, l_2\}$. De hyperradiale golffunctie wordt ontwikkeld in de 6-dimensionale radiale oscillatorbasis:

$$\phi_{c;L}(\rho) = \sum_{n_\rho} C_{n_\rho}^{(c;L)} \Phi_{n_\rho}^{(K)}(\rho) \tag{5.7}$$

Hier dient bij opgemerkt te worden dat alle spectroscopische berekeningen - gebonden toestanden, faseverschuivingen, werkzame doorsnedes, ... onafhankelijk zijn van de specifieke keuze van het Jacobi coördinaten systeem (Zie figuur 5.13) indien een complete set Hypersferische Harmonics voor een gegeven momentum K gebruikt wordt. Hiermee worden alle mogelijke waarden voor partiële draaimomenten bedoeld die compatibel zijn met K.

De Schrödinger vergelijking wordt opgelost door (5.5) te substitueren met de ansatz (5.6) en (5.7). Dit resulteert in een matrix vergelijking met expansie coefficiënten $C_{n_\rho}^{(c;L)}$ (Meer info in [120]). Als gevolg van de oscillator expansie kan een oplossing verkregen worden via de Modified J-Matrix methode [54]. Dit is een implementatie van de Resonating Group methode in de functie ruimte. Net zoals in de coördinaatruimte worden er een interactie en een asymptotische regio onderscheiden, nu echter in de expansie coëfficiënten ruimte. Deze coëfficiënten worden opgesplitst in twee deelverzamelingen. De eerste deelverzameling stelt de drie-cluster golffunctie in het interne gebied voor, en wordt bepaald door de Schrödinger matrixvergelijking op te lossen. De tweede deelverzamelin is gerelateerd met de asymptotische vorm van de golffunctie en stelt de gewenste randvoorwaarden voor. De asymptotische coefficienten kunnen bepaald worden door de referentie hamiltonianen, die de drie niet interagerende clusters beschrijven, op te lossen. Deze hamiltonianen worden gedefinieerd in [119]. Dit resulteert in N_c (het aantal kanalen) onafhankelijke oplossingen, met een asymptotisch gedrag voor grote hyperradius ρ.

$$\Psi_c^{(c_0)} \to \Phi_1(A_1)\,\Phi_2(A_2)\,\Phi_3(A_3) \sum_c f_c^{(c_0)}(\rho)\,\chi_c(\Omega_5) \tag{5.8}$$

met c_0 het inkomende kanaal en f_c de asymptotische expansiecoëfficiënt. Deze wordt als volgt gedefinieerd:

$$f_c^{(c_0)}(\rho) \to \delta_{c_0,c}\psi_c^{(-)}(k\rho) - S_{c_0,c}\psi_c^{(+)}(k\rho) \tag{5.9}$$

met $\psi_c^{(-)}(k\rho)$ ($\psi_c^{(+)}(k\rho)$) de golffunctie voor het inkomende (uitgaande) kanaal en $S_{c_0,c}$ de S-matrix die de transitie van het initiële kanaal c_0 naar het eindkanaal c beschrijft.

De resonantieparameters kunnen afgeleid worden van de eigenverschuivingen bekomen uit de diagonalisatie van de S-matrix. Uit deze eigenkanaal representatie volgt dat:

$$S_\alpha = \exp\{2i\delta_\alpha\}, \qquad \alpha = 1, 2, \ldots N_c$$

waarbij α de ongekoppelde eigenkanalen weergeeft.

De relatie tussen de originele vorm $\left\|S_{c,c'}\right\|$ en de diagonale vorm $\|S_\alpha\|$ is

$$S_{c,c'} = \sum_\alpha U_\alpha^c S_\alpha U_\alpha^{c'}$$

met $\|U_\alpha^c\|$ een orthogonale matrix. De resonantie positie en breedte worden hier op de traditionele manier geëxtraheerd door:

$$\left.\frac{d^2\delta_\alpha}{dE^2}\right|_{E=E_\alpha} = 0, \quad \Gamma = 2\left(\frac{d\delta_\alpha}{dE}\right)^{-1}\Bigg|_{E=E_\alpha} \tag{5.10}$$

De probleemoplossing bestaat dus, in essentie, uit twee delen:

- Het opzetten van de matrixvergelijking door (5.5), (5.6) en (5.7) te substitueren voor de oplossing van de Schrödinger vergelijking (het berekenen van de energie matrix).
- Het oplossen van deze vergelijking, onderworpen aan de geschikte randvoorwaarden.

Een groot voordeel van de beschreven methode is dat er een expliciete representatie van de golffunctie bekomen wordt. Dit biedt de mogelijkheid de resonatiegolffunctie gedetailleerd te analyseren en bijkomende fysische grootheden te berekenen.

In dit werk wordt enkel het eerste probleem (berekening van de energiematrix) voor een uitgebreide basis gedistribueerd op te lossen.

De energiematrix die bepaald moet worden, kan als volgt beschreven worden:

$$\left\langle K_i, (l_1, l_2)_i\, L, n_i|\widehat{H}|K_j, (l_1, l_2)_j\, L, n_j\right\rangle = \left\langle K_i, l_i, n_i|\widehat{H}|K_j, l_j, n_j\right\rangle \tag{5.11}$$

waarbij $\widehat{H}$ de Hamiltoniaan of energieoperator en i en j de basis indices. Het rechterlid van vergelijking 5.11 vereenvoudigt de notatie door aan te nemen dat L een globale constante is, en door de combinatie $(l1l2)_i$ te vervangen door l_i. De manier

waarop 5.11 bepaald wordt, valt buiten de scope van dit werk. Meer informatie kan teruggevonden worden in [119]. De vergelijking kan als volgt opgesplitst worden:

$$\begin{array}{c} \left\langle\left\langle K_i, l_i|\widehat{H}|K_j, l_j\right\rangle\right\rangle \\ = \\ \sum\sum\sum R\left(K_i, l_i, l_r, t\right) \quad \left\langle\left\langle K_i, l_r|\widehat{H}|K_j, l_s\right\rangle\right\rangle_t \quad R\left(K_j, l_s, l_j, t\right) \end{array} \tag{5.12}$$

met $\langle\langle\rangle\rangle$ de matrix over alle n_i, n_j indices. De Raynal-Revai coëfficiënten R, besproken in [54][119], en het aantal t waarden worden bepaald door de kern $\left({}^5H\right)$ en zijn decompositie $(t+n+n)$. De granulariteit van het probleem volgt duidelijk uit 5.12. Het probleem is herleid tot een *fork and join* algoritme dat elke onafhankelijke $\left\langle\left\langle K_i, l_r|\widehat{H}|K_j, l_s\right\rangle\right\rangle$ matrix berekent voor vaste K_i, K_j en elke toegelaten combinatie l_r, l_s, t (de fork), gevolgd door de sommatie om tot vergelijking 5.12 te komen (de join).

In dit werk worden berekeningen gedaan voor $L = 0$ en $K = 0, 2, 4, 6, 8, 10$ en een range $l1 = l2 = 0, 1, , K/2$ waarden. Elk stuk computationele code is geïmplementeerd in Fortran90 gebruik makend van een Intel v7 compiler. Elke taak is file-gebaseerd en onafhankelijk van de andere taken. Hiermee wordt bedoeld dat elke taak enkel files als input en als output mechanisme gebruikt en dat de outputfiles niet door overige taken als input gebruikt worden. Alle input files, met uitzondering van één configuratie file die de specifieke indexen voor die welbepaalde taak bevat, zijn dezelfde voor elke taak.

Implementatie

Ook de ontwikkeling van een dergelijk complex probleem op de CoBRA grid is eenvoudig, zelfs wanneer het gebruik maakt van binaries ontwikkeld op basis van een andere programmeertaal. Net zoals bij de vorige probleemoplossingen dienen enkel de `Job` en de `ProblemSolver` klasse uitgebreid te worden. De implementatie van de `Job` implementatie heeft drie taken:

- Het transfereren van de essentiële configuratie files en van de uit te voeren binary naar de resource die de Job uitvoert.
- Het uitvoeren van de binary.
- Het transfereren van de resulterende datafiles naar de Broker machine.

Om dit te bewerkstelligen dienen vier methoden uitgebreid te worden:

- `GetGlobalData()`: Om de configuratiebestanden en de executable, die voor elke Job gelijk zijn, te transfereren naar de resource (gnrl.cfg, Kl1l2Combs.dat, P_me_Kl1l2e_sngl, ...); dit om te vermijden dat al deze data voor elke Job telkens opnieuw getransfereerd dienen te worden.
- `GetData()`: Om het configuratiebestand, dat voor elke Job verschillend is, naar de resource te transfereren (P_me_Kl1l2e_sngl.cfg).

- `Execute()`: Om de binary te laten uitvoeren (P_me_Kl1l2e_sngl).
- `PutData()`: Om de resulterende data terug naar de broker te sturen.

In de code ziet dit er als volgt uit:

```
1  protected boolean GetGlobalData() {
2    return getRemotePSDir(sourceURL+"/input", "/workerinput");
3  }
4
5  protected boolean GetData() {
6    return getRemotePSFile(sourceURL+ datadir + "/" + datafile, "/
         workerinput/"+datafile);
7  }
8
9  protected Object Execute() {
10   return ExecuteBinary("/workerinput", "/workerinput/" + binary);
11 }
12
13 protected boolean PutData() {
14   return putRemotePSFile("/workerinput/"+ resultfile ,destinationURL+
         "/" +resultfile);
15 }
```

De `Problemsolver` implementatie heeft vier taken:

- Het transfereren van de essentiële configuratie files en van de uit te voeren binary van een remote machine naar de Broker.
- Het aanmaken van de verschillende Jobs. Dit wil zeggen dat er enkele geneste lussen worden aangemaakt zodat een job gecreëerd wordt voor de berekening van elke matrix voor vaste K_i, K_j en elke toegelaten combinatie l_r, l_s, t(de fork).
- De sommatie van de resulterende matrixen uit vergelijking 5.12(de join).
- Het transfereren van de resulterende datafiles naar de plaats die aangeduid werd door de gebruiker.

Om dit te realiseren worden drie methoden uitgebreid:

- `GetData()`: Om de configuratiebestanden en de executable van de remote machine naar de Broker te transfereren.
- `run()`: Om de fork en de join te implementeren.
- `PutData()`: Om de resultaten naar een remote machine naar keuze terug te sturen.

K	aantal taken	gemiddelde taaklengte (ms)
0	135	61
2	360	93
4	900	492
6	1890	2011
8	3465	7091
10	5670	22300
12	8910	63691
14	13050	167382

Tabel 5.8: Specificaties van de quantum fysische distributie volgens parameter K.

CPU Type	Memory (kb)	aantal
Intel(R) Pentium(R) 4 1.7 GHz	505500	4
Intel(R) Pentium(R) 4 2.4 GHz	505500	15
Intel(R) Pentium(R) 4 2.66 GHz	505500	6
Intel(R) Pentium(R) 4 2.66 GHz	1024984	3
Intel(R) Pentium(R) 4 2.8 GHz	1024984	14
Intel(R) Pentium(R) 4 3.0 GHz	1025196	8

Tabel 5.9: De Specificaties van de verschillende gebruikte workers.

Testen en Resultaten

De werking van deze distributie wordt uitgetest door de berekeningen voor $K = 0$, 2, 4, 6, 8 en 10 uit te voeren op 10, 25 en 50 heterogene Workers. Het aantal taken en de gemiddelde taaklengte stijgen bij een toenemende K waarde. Dit wordt getoond in tabel 5.8

Het type van de verschillende gebruikte resources wordt getoond in tabel 5.9

Voor elk van deze testen werden de overhead OH (zie formule 5.1), het percentage overhead POH (zie formule 5.2) en de speedup SU (zie formule 5.3) berekend. Om de diversiteit in de workers te schalen naar een referentieresource wordt, in deze berekeningen, de som van de effectieve uitvoertijden op de workers als sequentiële runtime genomen. Tot slot wordt de test ook uitgevoerd voor $K = 12$ en voor $K = 14$. De computationele vereisten maken echter dat deze tests enkel uitgevoerd werden op 50 workers. De resultaten van de testen worden getoond in tabel 5.10.

De resultaten tonen aan dat de overhead stijgt bij grotere K. Een grotere K zorgt immers voor meer Jobs, wat meer datatransfers tot gevolg heeft. Het percentage overhead daalt dan weer. Dit is het gevolg van de stijging van de gemiddelde taaklengte. De verhouding $uitvoeringstijd/datatransfertijd$ verhoogt, wat een gunstig effect heeft op de POH. De speedups stijgen om dezelfde reden en convergeren naar het aantal gebruikte workers. Dit wil zeggen dat de overhead verwaarloosbaar wordt wanneer de taken een afdoende grote taaklengte hebben. Verder kan er uit de speedups geconcludeerd worden dat het distribueren van het quantum fysica probleem enkel nuttig is voor $K \geq 4$. Bij een kleinere K zal de gedistribueerde uitvoertijd

K	OH 10 (s)	OH 25 (s)	OH 50 (s)	POH 10 (%)	POH 25 (%)	POH 50 (%)	SU 10	SU 25	SU 50
0	26,37	18,47	20,59	99,37	99,66	99,83	0,06	0,08	0, 09
2	56,58	34,33	33,95	94,77	96,45	98,10	0,52	0,88	0,95
4	124,62	59,19	58,20	75,67	78,19	86,79	2,43	5,45	6,61
6	297,11	111,77	79,93	46,14	43,76	51,29	5,39	14,05	24,35
8	502,56	230,90	168,95	18,29	19,80	25,59	8,17	20,05	37,21
10	767,21	508,02	447,25	6,15	9,34	14,83	9,39	22,66	42,58
12	-	-	1269,11	-	-	10,06	-	-	44,97
14	-	-	2941,63	-	-	63,10	-	-	46,85

Tabel 5.10: Resultaten van de quantum fysica test.

groter zijn dan de sequentiële uitvoertijd.

Naast de probleem specifieke vaststellingen, is het belangrijk op te merken dat de datatransfer zijn beperkingen heeft. Er moet ten allen tijde voor gezorgd worden dat de data nodig op de resources, opgevraagd kan worden bij machines die krachtig genoeg zijn om deze informatie aan te kunnen leveren (CPU, bandbreedte, snelheid HD, ...). In dit geval werd al de data opgevraagd aan de Broker machine. Wanneer 25 of 50 resources simultaan data opvragen, kan de Broker dit niet telkens op een optimale snelheid aanleveren. Deze kleine vertragingen zijn duidelijk te merken in de SU of POH waarden. De SU voor $K = 10$ is bijvoorbeeld bijna optimaal voor 10 workers (bijna 10). Bij 25 en 50 workers wijkt deze nog ver af van de optimale waarden (25 en 50). Om dit te vermijden, is het raadzaam dat de CoBRA grid wordt uitgebreid met een gedistribueerd filesysteem dat de files uniform en optimaal over de grid verdeelt, of met een storage element dat op een machine met de noodzakelijke specificaties wordt gedeployed.

Wanneer de resultaten worden vergeleken met de testen uit [16], blijkt dat de performantie van de CoBRA grid in deze opstelling beter presteert dan een distributie op BEGrid. Een MPI implementatie, die geïntegreerd is in de gebruikte Fortran code, zal logischerwijze wel beter presteren. De schaalbaarheid van BEGrid is beter dan deze van MPI en van CoBRA, welke vergelijkbaar zijn. De implementatie in CoBRA is echter veel eenvoudiger dan deze in MPI of op BEGrid. Dit bewijst dat de CoBRA grid een optimaal framework is om lokaal distributies, of proof of concept distributies, uit te voeren.

Conclusie

Deze implementatie heeft aangetoond dat het eenvoudig is een parameter sweep te distribueren op de CoBRA grid. Het gebruik van Java vreemde binaries vormt hier geen struikelblok. Tijdens de implementatie moet er wel rekening gehouden worden met de beperkingen van het netwerk en van de resources inzake het aanleveren van data. Deze tekortkoming kan eventueel in de toekomst opgevangen worden door een gedistribueerd filesysteem of door een krachtig storage element. De performantie van de CoBRA grid is aanvaardbaar en maakt het framework uiterst geschikt voor

lokale distributies en proof of concept distributies.

5.9 Toekomst perspectief

Het spreekt voor zich dat bij de constructie van een omvangrijk framework, zoals de CoBRA grid, er altijd ruimte blijft voor verbeteringen. Hoewel deze mogelijkheden vrijwel onbeperkt zijn, kunnen er drie aandachtspunten onderscheiden worden die de kwaliteit van het framework nog aanzienlijk kunnen verhogen:

- *Security*: De huidige implementatie biedt geen security ondersteuning. Dit wil zeggen dat alles zonder authenticatie en autorisatie, en zonder data encryptie verloopt. Tevens bezit het framework geen technieken om communicatie door firewalls vlot te ondersteunen. Het integreren van dergelijke security mechanismen zou de schaalbaarheid en de toepasbaarheid aanzienlijk verhogen.

- *File System*: Zoals reeds werd aangehaald in sectie 5.8.8, ondervindt het framework bottleneck problemen bij data-intensieve berekeningen. Dit brengt met zich mee dat de gebruiker van het framework rekening moet houden met de specifieke eigenschappen van de infrastructuur. Dit kan vermeden worden door een gedistribueerd file systeem te introduceren, dat files uniform over de aanwezige resources verdeelt, of door een storage element te ontwikkelen dat als extra component in het framework geplugd kan worden. Het spreekt voor zich dat de systeemvereisten van de resource, waarop deze laatste component zich bevindt, toereikend moeten zijn voor deze functie.

- *Backbone*: Het hoge aantal connecties en de trage automatische afbraak van deze connecties maken dat het gebruik van H2O en van Jini, bij de ontwikkeling van nieuwe componenten, vereist dat er rekening wordt gehouden met het feit dat deze verbindingen expliciet afgesloten moeten worden. In dit opzicht zou het nuttig zijn op zoek te gaan naar een alternatieve backbone waarvoor dit niet vereist is en de ontwikkelaar dus niet langer moet rekening houden met dit backbone issue.

HOOFDSTUK 6

Conclusion

In deel I van deze thesis werd een nieuw distributieframework geïntroduceerd. Het framework heeft tot doel een medium aan te bieden waarbinnen gebruikers, alsook developers van gedistribueerde componenten, op eenzelfde framework kunnen werken en ervaringen uitwisselen. Dit om een synergie te creëren die de implementatieprocedure van een gebruiker versnelt en de ontwikkeling van nieuwe technieken door de developer, met feedback van de gebruikers, ondersteunt.

Er werd aangetoond dat de karakteristieken van zowel de LWG (light weight grids) als de HWG (heavy weight grids) niet voldeden om dit opzet te verwezenlijken. De monitoring en de robuustheidsvereisten, die essentieel zijn om dit doel te bereiken, worden enkel ondersteund door de HWGs en door de HL-LWGs (High Level Light Weight Grids) . De flexibiliteitsvereisten en de aanpasbaarheidsvereisten, die even belangrijk zijn in deze doelstelling, zijn enkel terug te vinden in de LL-LWG categorie (Low Level Light Weight Grids).

De CoBRA grid combineert deze essentiële eigenschappen door, gebruik makend van plugability, de robuustheid en de monitoring vereisten aan reeds gekende LL-LWGs toe te voegen. Op basis van een universele laag, die zowel bovenop Jini als H2O functioneert, werd de noodzakelijke functionaliteit verwezenlijkt.

Testen, die zowel de ervaring van de gebruiker als van de developer weerspiegelen, bewezen de gewenste synergie. Performantietesten reflecteerden zowel de bruikbaarheid in concrete distributie als de bruikbaarheid voor proof of concept implementaties. Met het oog op classificatie werd een performantieprofiel opgesteld dat de overhead definieert aan de hand van de invloedsfactoren (aantal taken, lengte van de taken, aantal workers).

Er werd echter wel vastgesteld dat de scalability op het niveau van filetransfer

in extreme gevallen ontoereikend was en er aldus, naar de toekomst toe, een gedistribueerd filesysteem of een high performance storage element dient geïntroduceerd te worden.

Globaal kan dus besloten worden dat het CoBRA framework zijn doel bereikt heeft en nuttig gebruikt kan worden in wetenschappelijke rekenomgeving.

Deel II

GIPSy

HOOFDSTUK 7

Inleiding

7.1 Problematiek

De toegenomen vraag naar resources, in combinatie met de verbetering van technologien, zoals het internet en netwerken in het algemeen, hebben ervoor gezorgd dat de populariteit van *desktop grids* drastisch is toegenomen. Een *desktop grid* kan gedefinieerd worden als een verzameling van heterogene resources die wordt opgebouwd uit desktop computers die arbitreerbaar verspreid zijn en verbonden worden door een netwerk technologie zoals bijvoorbeeld het internet. Enkele voorbeelden van dergelijke grids zijn: BOINC[20] gebaseerde grids zoals Seti@Home [102], Folding@home [5], CoreGrid [59], ... , PrimeGrid [95], CoBRA [46][47], Condor [27][48], ...

Ondanks het succes van dergelijke desktop grid systemen hebben zij te kampen met een ernstig probleem [62]. Het gebruik van resources van derden (de desktopcomputers binnen de grid) brengt met zich mee dat de beschikbaarheid van de resources niet bepaald wordt door één persoon, zoals dit in traditionele grid/cluster systemen het geval was. Elke resource heeft nu immers een eigenaar die ten allen tijde kan beslissen zijn resources niet langer ter beschikking te stellen. Om kwalitatieve scheduling[1] in een dergelijk systeem mogelijk te maken, is het belangrijk het availability window[2] van de resources en de runtime[3] van de taken te vergaren.

De onzekerheden in zowel het availability window als in de runtimes brengen met zich mee dat deze factoren op een zo exact mogelijke wijze moeten voorspeld

[1] Een goede mapping van de runtimes op de availability windows.

[2] De tijd waarbinnen een resource ter beschikking zal staan van de grid ook wel *"uptime"* genoemd.

[3] De tijd die een resource nodig heeft om een taak volledig af te handelen ook wel *"runtime"* genoemd.

worden. Omwille van het feit dat er reeds zeer adequate availability window predictie mechanismen ontwikkeld zijn, in projecten zoals *"The Network Weather Service(NWS)"*[124] of [76] , wordt dit topic in deze doctoraatsthesis niet verder uitgewerkt. In het verdere verloop van de tekst zal er dan ook steeds vanuit gegaan worden dat deze factor gekend is.

In het onderzoek naar runtime predicties liggen de kaarten echter enigszins anders. Over het algemeen lag de focus tot op heden op predicties gebaseerd op het repetitief uitvoeren van een gedistribueerde probleemoplossing. Predicties werden dan op een eenvoudige wijze opgesteld op basis van de vaststellingen uit het verleden. Als job X de vorige n keren gemiddeld runtime t had dan wordt de huidige runtime naar alle waarschijnlijkheid ook t. Wanneer er echter gesproken wordt over desktop grids, die van nature veel verschillende gebruikers hebben, zal het vaak voorkomen dat een gedistribueerd probleem slechts één maal zal uitgevoerd worden. De bedoeling van dit onderzoek is om binnen één run (die verschillende jobs omvat) aan de hand van de reeds uitgevoerde jobs van die run de uitvoeringstijd van de resterende jobs te voorspellen.

7.2 Doel

In dit onderzoek zal het beschreven probleem gereduceerd worden tot de meest populaire groep van gedistribueerde applicaties namelijk de parameter sweep applicaties[4]. Deze reductie biedt de mogelijkheid, op basis van de parameters, reeds bestaande modelleringstechnieken te gebruiken om tot meer accurate voorspellingen te komen.

Dit onderzoek wil binnen deze afbakening twee essentiële dingen realiseren:

- Een medium bieden om op eenvoudige en op eenduidige wijze nieuwe technieken (modellering, ...) te testen en in de praktijk toe te passen. Om dit alles te bewerkstelligen wordt het GIPSy (Grid Information Prediction System) framework voorgesteld. Dit framework maakt het mogelijk elke geïmplementeerde techniek zowel op het testframework als rechtstreeks op een bestaande scheduler te pluggen.

- Een proof of concept bieden. Dit deel en het volgende deel van de thesis zal, gebruik makend van enkele standaard modellen, standaard technieken en nieuwe technieken, bewijzen dat het voorspellen van runtimes een meerwaarde biedt aan een scheduler en dat het gebruik van het GIPSy framework een eenvoudig en eenduidig medium is om onderzoek in dit domein te verrichten.

[4]Een parameter sweep applicatie is een applicatie waarbinnen eenzelfde stuk code verschillende malen zal uitgevoerd worden telkens met andere waarden voor de pre-gedefinieerde parameters. Zo kan bijvoorbeeld het gewicht van een auto een parameter zijn om zijn verbruik te berekenen.

7.3 Gerelateerd onderzoek

Alvorens dit doel verder uit te werken, is het belangrijk eerst een beeld te krijgen van de reeds bestaande job predictie methoden. Deze sectie zal dan ook, aan de hand van voorbeelden, een algemeen overzicht geven van de stromingen binnen de reeds bestaande runtime predictie technieken en de vooropgestelde doelstelling hierin plaatsen.

De verschillende runtime predictie mechanismen die terug te vinden zijn in de literatuur kunnen onderverdeeld worden in drie groepen:

- *Code analyse*: Code analyse is, zoals zijn naam al laat uitschijnen, gebaseerd op de analyse van de source code van een taak. Deze techniek bepaalt het aantal instructies die uitgevoerd moeten worden voor een arbitraire verzameling inputparameters. Aangezien deze techniek een low level approach is, is ze zeer sterk afhankelijk van zowel het type code als van de architectuur waarop de code wordt uitgevoerd. Het spreekt voor zich dat deze limiteringen niet te verankeren zijn in een heterogeen plugable grid systeem waarin een brede toepasbaarheid van essentieel belang is. Voorbeelden van dergelijke technieken zijn terug te vinden in [97] en [33].

- *Analytical benchmarking/code profiling*: De predictie in dit puntje gebeurt door combinatie van twee technieken. De eerste techniek, de code profiling techniek, deelt elk stuk code op in combinaties van wel gedefinieerde primitieve routines[5]. De tweede techniek, de Analytical benchmarking, benchmarkt[6] de runtime van deze primitieve routines op elk van de mogelijke opstellingen (hardware/software combinatie). Het spreekt voor zich dat, mits intensieve profiling en benchmarking aan de hand van deze gegevens, de executietijden van elk van de uit te voeren taken tot in detail berekend kunnen worden. Meer informatie omtrent deze technieken is terug te vinden in [35] [60] [61] [93] [125] [126] [53].

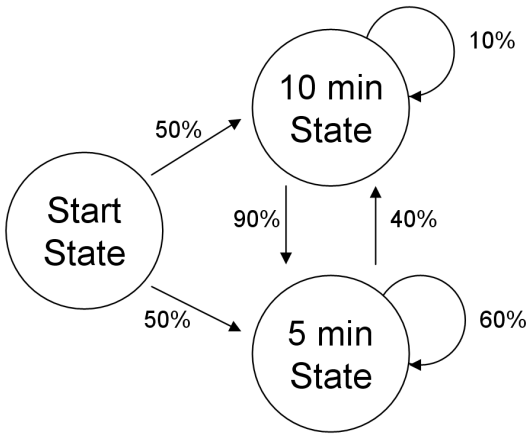
- *Statistical prediction*: Deze methodiek baseert zich op de statistische eigenschappen van de runtimes van de groep reeds uitgevoerde taken. De grote diversiteit in technieken maakt dat deze groep eenvoudig kan opgedeeld worden in deelgroepen. De categorieën die in dit werk gehanteerd worden zijn gebaseerd op twee karakteristieken:

    1. *Op basis van de gebruikte inputdata:* Tot op heden zijn er twee stromingen die predicties doen op basis van een verschillende soort inputdata. Het verschil zit hem in de oorsprong van de data. De eerste groep zal gebruik maken van data die vergaard zijn door middel van het loggen van een eerdere uitvoering van het volledige gedistribueerd probleem. Voor elke parameter sweep job zijn er dus al runtime data uit het verleden

[5] Deze primitieve routines bestaan in de meeste gevallen uit een combinatie van instructies.
[6] Aan de hand van praktijktesten de performantie opmeten in verschillende omgevingen.

bekend. Zo kan de voorspelling van een job met parameter $X = 7$ bijvoorbeeld berekend worden door een gemiddelde te nemen van alle jobs uit het verleden met parameter $X = 7$. Enkele voorbeelden van deze techniek zijn [107][88]. Bij de tweede groep worden de voorspellingen enkel gebaseerd op de uitvoertijden van jobs uit de huidige uitvoering van het gedistribueerd probleem. Zo kan bijvoorbeeld een job met parameter $X = 7$ berekend worden door het gemiddelde te nemen van de twee reeds uitgevoerde jobs met parameter $X = 6$ en $X = 8$. Een mooi voorbeeld hiervan is [53].

2. *Op basis van het predictie mechanisme:* De tweede basis waarop een onderscheid gemaakt wordt, is de manier waarop de voorspelling van de runtimes gebeurt. Omwille van het feit dat een opsomming van alle mogelijke verschillende groepen van predictietechnieken te ver zou leiden, wordt dit punt gestaafd aan de hand van enkele voorbeelden uit de literatuur.
   - De eerste, en voor dit onderzoek, de belangrijkste groep is de predictie gebaseerd op modelleringstechnieken. De verschillende technieken die in deze groep omvat zitten, zullen elk, aan de hand van de verworven informatie, een model(of functie) opstellen. Dit model heeft de parameters van de parametersweep als inputvariabelen en geeft als resultaat de voorspelde runtime terug. Een mooi voorbeeld is terug te vinden in [52].
   - Een tweede groep baseert zich op de *"gelijkaardigheid"* (similarity) van jobs. De groep van parametersweeps zal onderverdeeld worden in groepen van jobs die dezelfde eigenschappen (in dit geval de runtime) vertonen. Concreet zal dan bij de predictie van een job bepaald worden tot welke groep deze job behoort en de runtime zal dan voorspeld worden door de gemiddelde runtime te berekenen van al de reeds uitgevoerde jobs die tot deze groep behoren. Het is belangrijk hierbij op te merken dat de kwaliteit van deze systemen volledig zal afhangen van de kwaliteit van de *"gelijkaardigheids"* definitie. Meer informatie hieromtrent is terug te vinden in [107][14].
   - Nog een ander groep baseert zich dan weer op het gebruik van finite state machines om de runtime te voorspellen. Deze methodiek gaat ervanuit dat er een volgorde zit in de voorspellingen die gemaakt zullen worden. Als taak n bijvoorbeeld 10 minuten duurt dan zegt deze techniek dat er 90% kans is dat de volgende taak maar 5 minuten duurt en 10% kans dat deze taak terug 10 minuten duurt. Uitgedrukt in de finite state machine komt het er dan op neer dat, wanneer er voorspeld wordt dat de huidige job 10 minuten zal duren (we zitten in de 10 min state), de machine dan met 90% kans zal overgaan naar de 5 minuten state en met 10% kans in de huidige state zal blijven. Zie figuur 7.1. Meer informatie over een dergelijke techniek kan teruggevonden worden in [28].



Figuur 7.1: Voorbeeld van finite state machine predictie.

– ...

In sectie 7.2 werd reeds vermeld dat het opzet van dit werk erin bestaat huidige bestaande modelleringstechnieken aan te wenden om tot zo accuraat mogelijke runtime predicties te komen. Dit plaatst de ontwikkelde predictietechnieken binnen dit onderzoek onmiddellijk in de *Statistical prediction* groep. Verdere categorisering van de techniek binnen deze groep gebeurt in de loop van dit hoofdstuk gebruik makend van de verworven kennis aldaar.

7.4 Verloop hoofdstuk

Het vervolg van dit deel van de thesis zal als volgt opgebouwd zijn. In hoofdstuk 8 wordt een theoretische uiteenzetting gegeven van de predictiemethodologie die gebruikt zal worden. Concreet bestaat dit hoofdsuk uit de bespreking van de drie basisonderdelen namelijk de *Startgrid Selector*(8.2), *de Model Builder*(8.3) en *de Sample Selector*(8.4). Hoofdstuk 9 bespreekt de implementatie van deze theoretische benadering die ervoor moet zorgen dat het framework zowel voor effectieve predictie als voor het uittesten van predictietechnieken gebruikt kan worden. Dit gebeurt traditioneel door eerst de ideeën te bundelen in sectie 9.2, vervolgens wordt er een design opgesteld in sectie 9.3 waarna de concrete implementatie is uitgewerkt in sectie 9.4. Tot slot van dit implementatieonderdeel worden in sectie 9.5 nog enkele extra features besproken. Hoofdsukt 10 bespreekt de verschillende reeds bestaande predictietechnieken die in de loop van het onderzoek bruikbaar bleken. Naast deze beschrijving wordt elke techniek, die effectief in het framework vervat zit (geïmplementeerd is) zowel theoretisch als praktisch (implementatie) besproken. Vervolgens werkt hoofdstuk 11 elk van de gebruikte startgridselectie technieken uit. Hoofdstuk12 beschrijft de verschillende gebruikte sample selection technieken. Tot slot geeft hoofdstuk 14 een globaal beeld van de kwaliteit van de verschillende tech-

nieken. Dit zowel op het gebied van de Sample Selectors als op het gebied van de gebruikte modellen.

HOOFDSTUK 8

Predictie Techniek

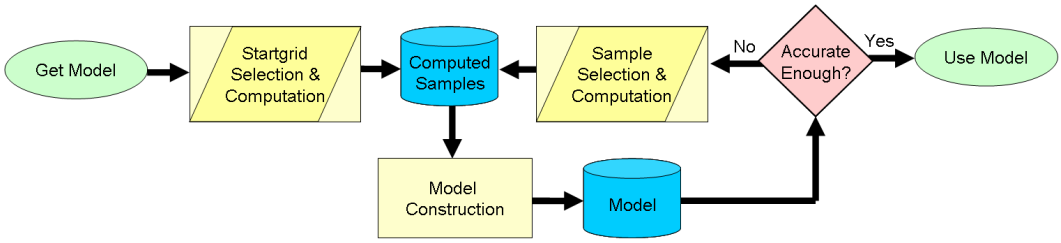
8.1 Inleiding

In dit onderdeel van de thesis wordt besproken op welke manier de traditionele modelleringstechnieken gebruikt worden om tot de gewenste predicties te komen. Het beperken van de gedistribueerde applicaties tot de parametersweep applicaties maakt het mogelijk bestaande modelleringstechnieken te gebruiken om de runtimes te voorspellen. In essentie komt het erop neer dat er een model gebouwd wordt waarin de runtimes uitgezet worden op basis van de parametersweep. Er wordt met andere woorden een functie ontwikkeld met als inputwaarden de parameterwaarden en als resultaat de voorspelde runtimes. In het verloop van deze tekst wordt naar deze functies gerefereerd met de volgende termen, die in essentie allemaal dezelfde betekenis hebben: *Predictie functie, Model, Predictor.* Al naargelang de concrete situatie in de tekst zal er naar de combinatie van inputparameters en naar hun uitvoertijd ook verwezen worden met verschillende termen die dezelfde betekenis hebben: *datapunten, runtimes, samples, sample punten, jobs, taken.*

Het opstellen van dergelijke modellen is een continu systeem waarin, aan de hand van de nieuwe vergaarde data, steeds nieuwe, meer accurate, modellen ontwikkeld worden. Dit kan het best gedemonstreerd worden aan de hand van figuur 8.1.

Op de flowchart[1] in figuur 8.1 zijn duidelijk drie acties te onderscheiden. De eerste actie die ondernomen wordt, is de *Startgrid Selection & Computation.* Deze actie zorgt ervoor dat uit de volledige verzameling van parametersweep jobs een selectie van jobs wordt gemaakt die eerst worden uitgevoerd. Deze selectie zal zo opgesteld worden dat de resulterende data zo snel mogelijk tot een kwalitatief model zullen

[1]Meer informatie omtrent flowcharts is terug te vinden op http://www.smartdraw.com/tutorials/flowcharts/tutorial_01.htm



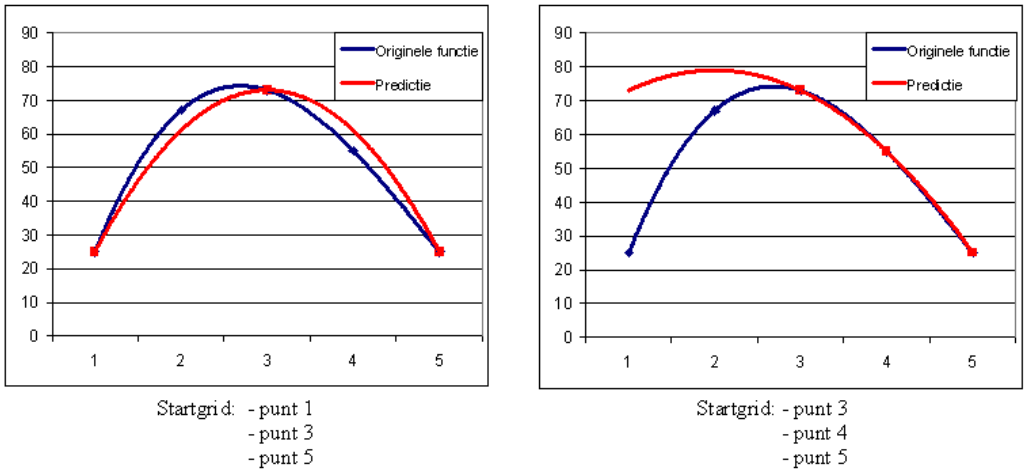
Figuur 8.1: Flowchart van de iteratieve opbouw van opeenvolgende modellen.

leiden. De tweede actie die terug te vinden is op de flowchart is de *Model Construction*. Het doel van deze actie is, aan de hand van de reeds uitgevoerde jobs, op basis van hun parameters, een model op te stellen van hun runtimes. Wanneer deze actie voltooid is, moet er een keuze gemaakt worden. In het optimale geval bereikt het verkregen model het gewenste kwaliteitsniveau en is het model dus acceptabel. In het andere geval wordt de derde actie uitgevoerd: *Sample selection & Computation*. Deze actie zal, op basis van het huidige model, bepalen welke jobs best als eerste worden uitgevoerd om met hun runtimes de kwaliteit van het nieuwe model zo veel mogelijk te optimaliseren. Na deze actie wordt opnieuw de tweede actie uitgevoerd tot uiteindelijk een acceptabel model gevonden is. De volgende drie topics zullen elk van deze acties verder uitwerken.

8.2 Startgrid Selector

De kwaliteit van een model is onlosmakelijk verbonden met de samples waarmee het model is opgebouwd. De juiste keuze van deze punten zal dan ook leiden tot betere modellen. Het gebrek aan voorkennis bij aanvang van de gedistribueerde applicatie brengt met zich mee dat het moeilijk te beslissen is welke samples eerst beschikbaar zouden moeten zijn (welke jobs eerst uitgevoerd moeten worden). Toch blijkt uit onderzoek dat het gebruik van welbepaalde patronen in deze selectie van samples [85] tot betere resultaten leidt dan random sample selectie. De Startgrid Selector is de component die verantwoordelijk is voor de selectie van deze samples. Meer bepaald wordt er een selectie gemaakt uit de aanwezige hoeveelheid taken en worden deze zo snel mogelijk uitgevoerd. De impact die deze selectiemechanismen hebben wordt geïllustreerd in figuur 8.2

Deze figuur wordt als volgt geïnterpreteerd. Op de X-as staan de nummers van de verschillende taken waarvan de duurtijd voorspeld moet worden en op de Y-as staat de respectievelijke duurtijd. De blauwe punten zijn de taken die nog moeten berekend worden en de blauwe lijn is bijgevolg het model dat verkregen zou moeten worden. De rode punten zijn de taken waarvan de duurtijd reeds gekend is. De rode lijn stelt dan ook het model voor dat, gebruik makend van de rode punten, geconstrueerd kan worden. Het is hier zonder meer duidelijk dat bij het kiezen van een betere spreiding van punten in de initiële dataset, het model veel nauwer aansluit



Figuur 8.2: Vergelijking tussen de modellen gebouwd met twee verschillende startgrids (X-as = taak nummers, Y-as = tijd)

bij het gewenste model dan bij een minder goede spreiding. De keuze van de punten 1, 3 en 5 in de linkse plot van figuur 8.2 zorgt er dus duidelijk voor dat de rode lijn veel dichter bij de blauwe ligt dan bij de keuze van de punten 3, 4 en 5 in de rechts plot. De manier waarop, het waarom ervan en concrete tests hierop worden besproken in hoofdstuk 11.

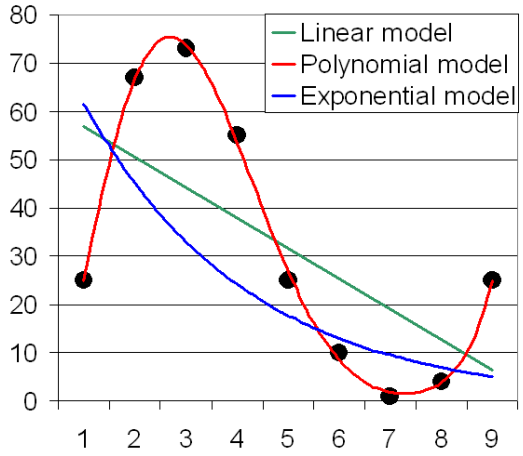
8.3 Model builder

De tweede actie houdt in dat er een runtime model wordt opgebouwd op basis van de reeds beschikbare samplepunten. Het belang van de keuze van het type model blijkt duidelijk uit figuur 8.3. Hier wordt een set van data aan de hand van verschillende modellen benaderd.

Het is hier duidelijk te merken dat er een groot verschil in accuraatheid zal zijn tussen de verschillende modellen. Het polynomiale model zal deze dataset nagenoeg perfect benaderen. Dit in tegenstelling tot het lineaire model dat enorme afwijkingen vertoont. Dit alles om te schetsen dat de keuze van één of enkele competente modelleringstechnieken essentieel zal zijn in het verdere verloop van deze tekst.

Het spreekt voor zich dat er in de literatuur al verschillende technieken bestaan die aan de hand van een bepaald model runtime predicties verwezenlijken. Hoofdstuk 10.2 geeft een theoretisch overzicht van enkele reeds bestaande technieken:

1. Finite state machines (Sectie 10.2.1)
2. Gelijkheids testen (Sectie 10.2.2)
3. Nearest Neighbour prediction (Sectie 10.2.3)



Figuur 8.3: Vergelijking tussen 3 types modellen: Polynomiaal, Lineair en Exponentiëel (X-as = taak nummers, Y-as = tijd)

Naast de reeds bestaande technieken in runtime prediction worden in hoofdstuk 10 tal van technieken uit de modelleringswereld geadopteerd en geïntegreerd. Elk van deze technieken wordt besproken en naar waarde geschat aan de hand van de kwaliteitscontrol beschreven in sectie 13.

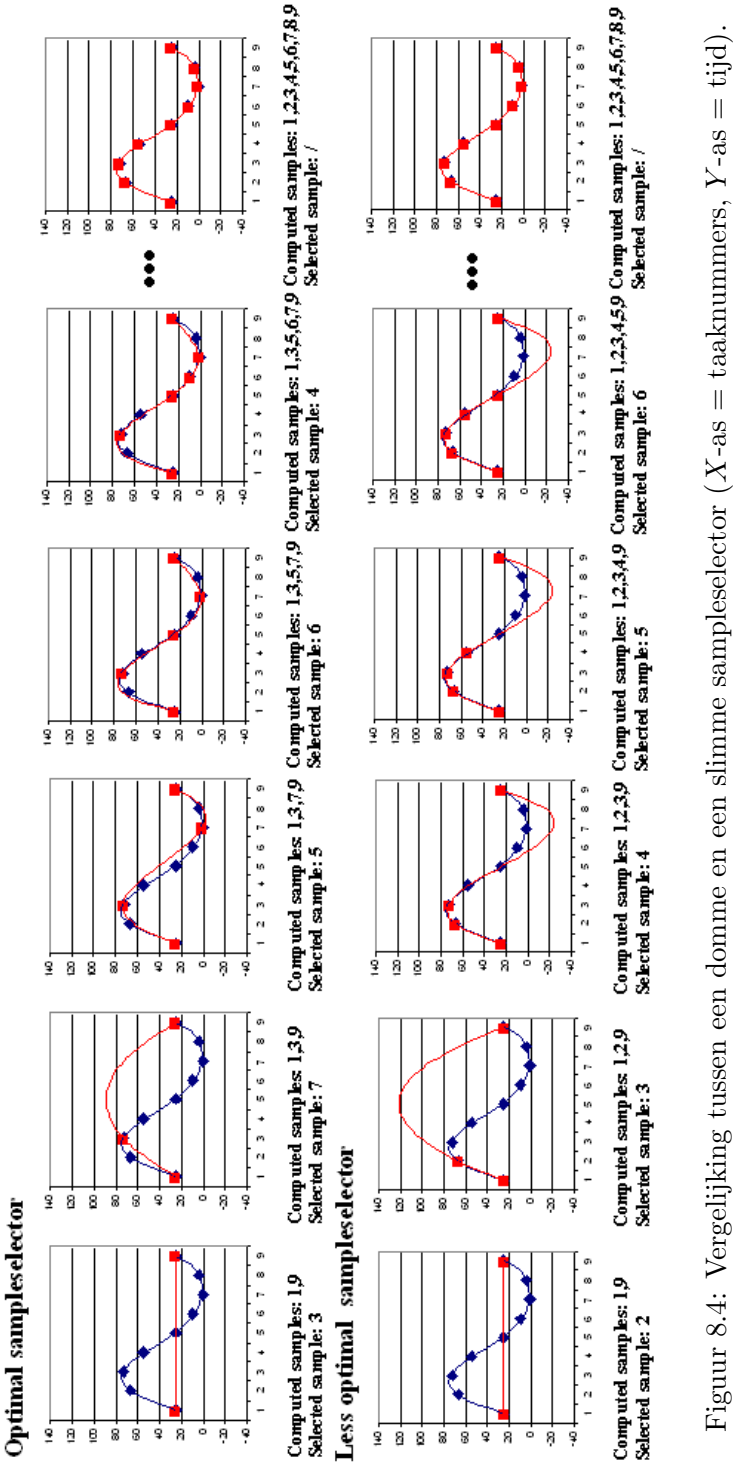
1. Iverson techniek (Sectie 10.3)
2. Polynomiale Modellen (Sectie 10.4)
3. Neurale Netwerken (Sectie 10.5)
4. Support Vector Machines (Sectie 10.6)
5. Radiale basisfuncties modellen Modellen (Sectie 10.7)
6. Kriging Modellen (Sectie 10.8)
7. Naaste Buren Model (Sectie 10.9)
8. Gecombineerde Modellen (Sectie 10.10)

8.4 Sample Selector

Het is reeds enkele malen aangehaald maar gezien het belang ervan wordt hier het doel van dit hoofdstuk nogmaals herhaald. Het is de bedoeling zo snel mogelijk een zo goed mogelijk beeld te krijgen over de runtime van elke nog uit te voeren taak. Aangezien dit zo snel mogelijk moet gebeuren, wil dit dus zeggen dat zo weinig mogelijk taken, en als het kan, dan nog liefst de korte taken, echt uitgevoerd moeten worden om tot een voorspelling te komen die aan de vooropgestelde eisen

voldoet. Dit systeem staat of valt dus met andere woorden met de manier waarop de uitvoervolgorde van de opeenvolgende taken wordt gekozen. Dit was reeds af te leiden uit figuur 8.2 maar wordt nog duidelijker geïllustreerd in figuur 8.4

Ook deze figuur wordt op de zelfde manier geïnterpreteerd als figuur 8.2. Op de X-as staan de nummers van de verschillende taken waarvan de runtime voorspeld moet worden en op de Y-as staat de respectievelijke runtime. De blauwe punten zijn de taken die nog moeten berekend worden en de blauwe lijn is bijgevolg het model dat verkregen zou moeten worden. De rode punten zijn de taken waarvan de runtime reeds gekend is. De rode lijn stelt dan ook het model voor dat, gebruik makend van de rode punten, geconstrueeerd kan worden. De verschillende plotjes naast elkaar stellen van links naar rechts de verschillende modellen voor die verkregen worden wanneer telkens informatie omtrent één taak wordt toegevoegd. Op de bovenste rij plotjes wordt een slimme sampleselector gebruikt die zijn punten daar gaat kiezen waar de fout tussen het voorspelde model en de gewenste data maximaal is. Op de tweede rij worden de punten sequentieel toegevoegd startend bij punt 2 tot en met punt 8. Het is zonder meer duidelijk dat de eerste methode sneller over de informatie van de kritische punten beschikt en dat de geconstrueerde modellen dus sneller hun fout weten te beperken. Verschillende manieren om deze punten zo optimaal mogelijk te kiezen en uitgebreide tests hierop zijn terug te vinden in hoofdstuk 12.



Figuur 8.4: Vergelijking tussen een domme en een slimme sampleselector (X-as = taaknummers, Y-as = tijd).

HOOFDSTUK 9

Predictie Framework

9.1 Inleiding

Zoals in de inleiding reeds is aangehaald, bestaat het opzet van het GIPSy framework erin een mechanisme te bieden om enerzijds, als proof of concept, aan te tonen dat scheduling, aan de hand van runtime predicties, een toegevoegde waarde is aan de groep van reeds bestaande scheduling mechanismen. Anderzijds wordt getracht men een mechanisme aan te bieden dat de verschillende vitale predictieonderdelen van een dergelijke scheduler kan evalueren en optimaliseren. Concreet wordt dit doel nagestreefd door een pluggable predictieframework op te zetten dat en in het testframework en in elke GIPSy enabled scheduler kan geplugd worden.

In dit gedeelte van de thesis worden het opzet, het design en de implementatie van het predictiemechanisme en het evaluatieframework waarin dit mechanisme geplugd wordt uitgebreid besproken. In deel III wordt, aan de hand van een concrete GIPSy enabled scheduler, de samenhang met het evaluatieframework en de werking van runtime prediction based scheduling besproken.

Het vervolg van dit hoofdstuk ziet er als volgt uit. Eerst wordt het doel van de predictie binnen het testframework geschetst, vervolgens wordt op basis van de vastgelegde doelstellingen, aan de hand van UML, een design opgesteld om dan tot slot tot de eigenlijke implementatie van het systeem te komen.

9.2 Opzet

Zoals de inleiding van deze sectie al liet uitschijnen, is het opzet van dit framework tweeledig. Enerzijds wordt getracht een framework te creëren waarin het ontwik-

kelen van predictiemechanismen vereenvoudigd wordt, anderzijds wordt geprobeerd aan te tonen dat deze ontwikkelde predictietechnieken een toegevoegde waarde zijn binnen de scheduling wereld. Hoewel de concrete doelstellingen drastisch van elkaar verschillen, is het toch duidelijk dat ze in de ontwikkeling nauw met elkaar verbonden zijn. Omwille van deze reden zal deze sectie trachten een overzicht te geven van het volledige opzet van dit onderdeel van de thesis.

Het ontwikkelen van een proof of concept betekent in dit geval dat er een implementatie gemaakt wordt van een scheduler die, op basis van voorspelde runtimes, erin slaagt de verzameling van te verdelen jobs beter over de verschillende participerende machines te verdelen. Concreet houdt dit in dat er een predictiemechanisme gekozen wordt en een implementatie van een scheduler gemaakt wordt die de jobs schedulet. De resultaten van deze scheduler zullen dan vergeleken worden met standaard scheduling mechanismen zoals first come first serve. De plugability eigenschappen van het CoBRA framework, zoals die zijn besproken in deel I van deze thesis zorgen ervoor dat deze inspanning herleid wordt tot het uitvoeren van testen met het CoBRA framework waarin, bij de eerste uitvoering van de verzameling testen, de standaard scheduler zal ingeplugd worden en bij de tweede uitvoering de prediction based scheduler zal ingeplugd worden.

Deze vorm van plugability gaat echter niet ver genoeg om op een eenvoudige wijze een gefundeerde proof of concept te verwezenlijken. Deze manier van werken brengt twee problemen met zich mee:

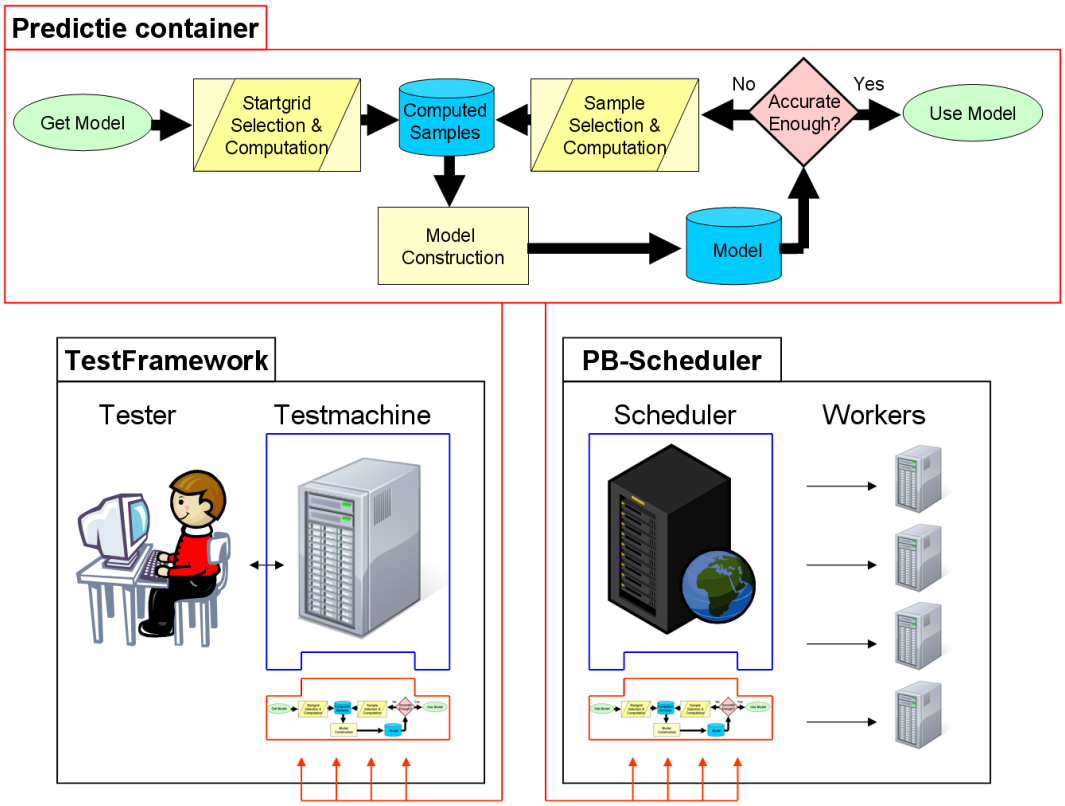
- Telkens er een nieuwe predictiemethodiek getest wordt, moet er een nieuwe implementatie gemaakt worden van de scheduler die deze predictie toepast binnen zijn scheduling methodiek.
- Het predictiegebeuren kan alleen met een nieuwe scheduler in real life getest worden. Dit brengt enerzijds met zich mee dat de scheduler voor elke predictiemethodiek volledig geïmplementeerd moet zijn alvorens de techniek getest kan worden. Anderzijds heeft dit ook tot gevolg dat volledige gedistribueerde problemen moeten uitgevoerd worden om een techniek te testen. Deze punten zorgen er al snel voor dat zowel de vereiste tijd als de vereiste effort te groot zullen worden om de techniek in werkelijkheid te gebruiken.

Deze problemen kunnen volledig opgelost worden door het opzet van dit concept op te splitsen in drie plugable onderdelen:

- Een container die de taak uit figuur 8.1 vervult en waarin op een eenvoudige wijze verschillende versies van de gebruikte technieken geplugd kunnen worden.
- Een testframework waarin deze container kan geplugd worden en hierop de gewenste testen kunnen uitgevoerd worden.
- Een prediction based scheduler (PB-scheduler) waarin de container geplugd kan worden om aldus de gewenste prediction based scheduling te voorzien.

Figuur 9.1 toont deze drie verschillende componenten en de plaats die ze innemen in het plugable geheel. Deze manier van werken heeft als resultaat dat:

- Elke nieuwe predictietechniek zonder herprogrammatie in zowel het testframework als in de schedulers geplugd kan worden.
- Elke predictietechniek uitvoerig getest kan worden op het testframework alvorens ze in een echte scheduler gebruikt wordt.
- De implementatie van een predictietechniek, die in het testframework gebruikt is, zonder herprogrammatie onmiddellijk in een scheduler gebruikt kan worden.



Figuur 9.1: De figuur toont dat de predictiecontainer zowel in het testframework als in een runtime prediction based scheduler moet kunnen geplugd worden.

In wat nu volgt, wordt het opzet van elk van deze componenten kort besproken.

9.2.1 Predictie Container

Zoals de naam al laat vermoeden, zal deze container instaan voor de predictie. In eerste instantie zou er overwogen kunnen worden deze component voor elk type predictie te herdefiniëren en te herprogrammeren. Sectie 9.3 toont echter dat het principe binnen deze container voor elk type predictie gelijk is. In essentie komt het erop neer dat het predictiemechanisme kan gespecificeerd worden aan de hand van de drie basiscomponenten waaruit de container bestaat (startgrid selection, sample selection, model construction). Om de plugability eigenschappen op elk niveau te

kunnen bewaren, wordt deze plugability ook tot op dit niveau doorgetrokken. Elk van deze drie componenten kan ten allen tijde vervangen worden door een andere implementatie van de desbetreffende component.

9.2.2 Prediction based scheduler

Een prediction based scheduler is een scheduler die gebruik maakt van de voorspelde runtimes om aldus de verschillende taken zo goed mogelijk te verdelen over de verschillende resources. Een uitgebreide bespreking van dit concept is terug te vinden in deel III van deze thesis.

9.2.3 TestFramework

Het testframework heeft, zoals zijn naam al laat vermoeden, als doel de verschillende predictietechnieken te testen alvorens ze in een scheduler gehanteerd worden. Het is immers de bedoeling dat de kwaliteit van een predictietechniek reeds bewezen is alvorens haar toepasbaarheid wordt getest op tijdsintensieve applicaties. De manier waarop deze testen uitgevoerd moeten worden, wordt beschreven in wat nu volgt.

Voor de testmethodiek verder besproken wordt, is het belangrijk nogmaals op te merken dat de runtime van taken bij parametersweeps, in principe, kan opgevat worden als het opstellen van een functie met als input de waardes van de verschillende parameters en als output de verwachte runtime. In het verloop van deze tekst zal naar deze functies gerefereerd worden met de volgende termen die in essentie allemaal dezelfde betekenis dragen: *Predictie Functie*, *Model*, *Predictor*. Al naargelang de concrete situatie in de tekst zal er naar de combinatie van inputparameters en naar hun runtime ook verwezen worden met de volgende namen: *datapunten*, *runtimes*, *samplepunten*.

Bij het opbouwen van een testmechanisme moet eerst en vooral de vraag gesteld worden op welke manier de kwaliteit en de eigenschappen van de verschillende predictie methoden liefst uitgetest worden. Een eerste probleem dat zich in dit verband stelt, is de beschikbare hoeveelheid tijd. Wanneer elke methode getest wordt aan de hand van een echte opstelling van een distribeerbare probleemstelling houdt dit in dat elke taak zal moeten uitgevoerd worden. Het probleem dat hier ontstaat, is tweeledig. Enerzijds zal de runtime van de test vooral bestaan uit het uitvoeren van taken en niet zozeer uit het uitvoeren van voorspellingen. Anderzijds zal de runtime lineair stijgen met de runtime van een taak en met het aantal te distribueren taken. Dit brengt met zich mee dat het uitvoeren van predicties met grote executie tijden en met veel taken praktisch onhaalbaar wordt. De oplossing hiervoor wordt besproken in sectie 9.2.4.

Een tweede belangrijke factor in de testmethodiek is de manier waarop de kwaliteit van de verschillende modellen wordt gemeten. Dit topic wordt verder besproken in sectie 9.2.5.

9.2.4 Testmethodieken

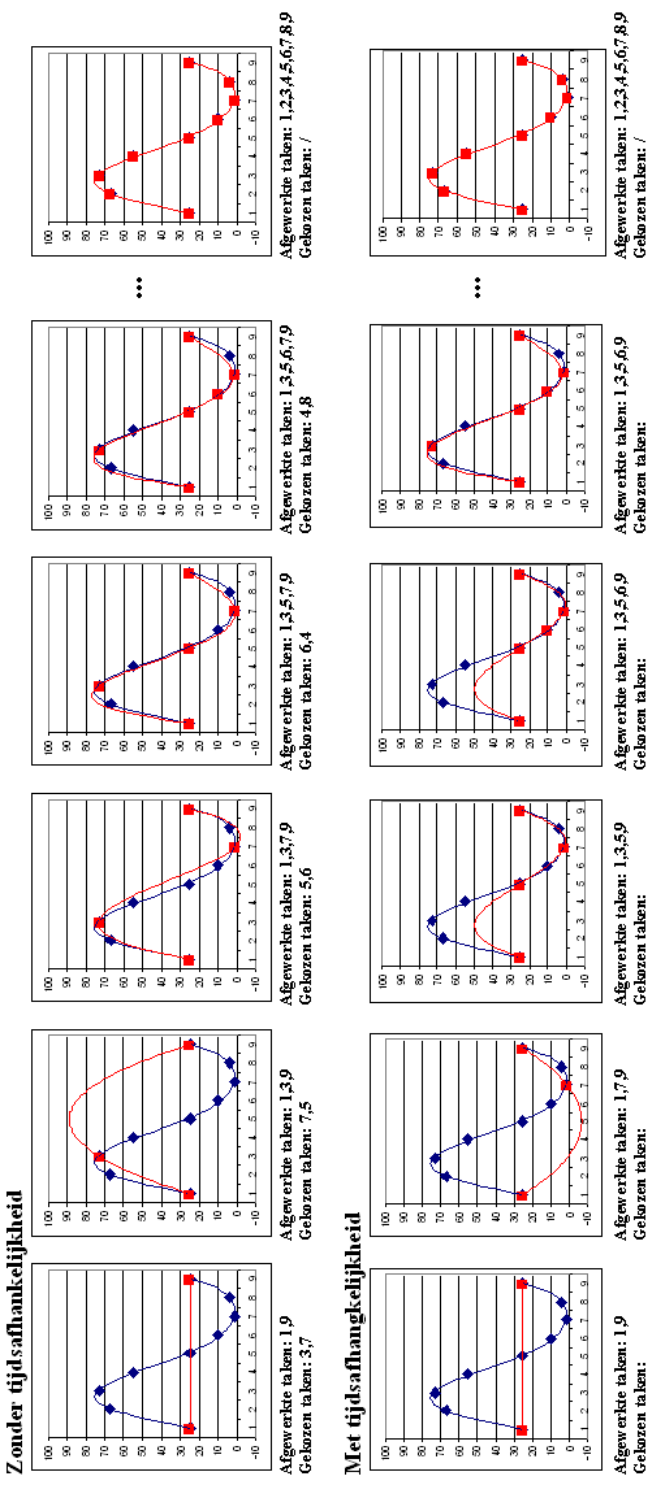
Zoals reeds werd aangehaald, moet er bij het uittesten van technieken steeds rekening gehouden worden met de beschikbare hoeveelheid tijd. Het uitwerken van

testen, op basis van een echte gedistribueerde toepassing, zou in dit geval te tijdrovend zijn. Dit omdat enerzijds het merendeel van de tijd zal gaan naar het uitvoeren van de taken en niet zozeer naar de predictie en anderzijds omdat deze opstelling niet schaalbaar is. De runtime van de tests zal immers lineair stijgen met het aantal taken en met de runtime van de taken.

Om aan dit probleem het hoofd te bieden, is er besloten in een eerste fase het geheel te simuleren om zo de overhead, in dit geval het uitvoeren van de taken, te vermijden. Van deze simulator zijn er twee versies gemaakt. Een eerste versie die geen rekening houdt met de runtime van een taak. Er wordt dus, met andere woorden, tijdens de simulatie geen rekening gehouden met het feit dat het predictiemechanisme, na selectie van een job, nog de runtime van die job moet wachten alvorens het mechanisme over de runtime informatie beschikt. Dit om snel de kwaliteit en de toepasbaarheid van een modelleringstechniek te verifiëren. Een tweede versie doet dit wel om zo, zij het ten koste van de performatie, het realisme van de simulatie op te drijven. Figuur 9.2 maakt dit verschil duidelijk.

Hoewel figuur 9.2 een sterk vereenvoudigd voorbeeld van de situatie geeft, is het duidelijk dat het verloop van de opbouw van de modellen in beide versies anders verloopt. De figuur moet als volgt geïnterpreteerd worden. Op de X-as staan de indexen van de verschillende taken waarvan de runtime voorspeld moet worden en op de Y-as staat de respectievelijke runtime. De blauwe punten zijn de taken die nog moeten uitgevoerd worden en de blauwe lijn is bijgevolg het model dat verkregen zou moeten worden. De rode punten zijn de taken waarvan de runtime reeds gekend is. De rode lijn stelt dan ook het model voor dat, op dat moment, gebruik makend van de rode punten, kan geconstrueerd worden. De verschillende plotjes stellen, van links naar rechts, de verschillende modellen voor die verkregen worden wanneer telkens runtime-informatie omtrent een taak wordt toegevoegd. In de bovenste rij gebeurt dit runtime onafhankelijk in de onderste rij runtime afhankelijk. De plotjes maken duidelijk dat de informatie over enkele punten essentieel zal zijn om een adequaat model op te stellen. Er zal dan ook, zowel in de runtime afhankelijke als in de runtime onafhankelijke methode, gepoogd worden de informatie omtrent deze kritieke punten (punt 1, punt 3, punt 7 en punt 9) zo snel mogelijk te vergaren. Aangezien ervan uitgegaan wordt dat punt 1 en punt 9 het start grid zijn, mag er gesteld worden dat de informatie over deze punten reeds gekend is bij aanvang van het gehele proces. In de praktijk zijn dit meestal de punten die aan de rand van of op de centrale assen van het model liggen. De ware toedracht van deze keuzes wordt later verder toegelicht. Het komt er dus op neer dat beide versies op een zo snel mogelijke manier aan de info over punten 3 en 7 proberen te geraken. Zonder de runtime afhankelijkheid is het evident dat dit in twee stappen kan gebeuren, eerst wordt punt 3 toegevoegd en dan punt 7. Wanneer er wel met de runtime afhankelijkheid gewerkt wordt, is dit minder evident. De 3de taak zal immers nog niet berekend zijn wanneer het systeem al een volgende taak moet kiezen. Sterker nog, deze volgende taken (nummers 5 en 6) zullen reeds voor taak nummer 3 beëindigd zijn waardoor in stappen 2 en 3 nog steeds een model wordt gebouwd zonder het kritieke punt 3. Het is slechts in stap

4 dat de info over punt 3 bekend is en het verkregen runtime afhankelijke model een accuraatheid krijgt die gelijkwaardig is aan die van de runtime onafhankelijke versie.



Figuur 9.2: Vergelijking tussen de tijdsafhankelijke en niet-tijdsafhankelijke versie (X-as = taak nummers, Y-as = tijd)

9.2.5 Kwaliteitsbepaling van modellen

Naast de methode waarop de opeenvolgende modellen in het testframework worden opgebouwd, is het tevens belangrijk een beeld te krijgen van de kwaliteit van deze modellen. Naar de kwaliteitsbepaling zal verder in deze thesis ook verwezen worden met het woord *errorcalculation* waarmee in essentie hetzelfde bedoeld wordt. Deze kwaliteitsbepaling of errorcalculation zal niet enkel nuttig zijn om de verschillende modelleringscomponenten naar waarde te schatten maar zal tevens relevante data verschaffen bij de implementatie van enkele sampleselectors (zie sectie 8.4). Het is immers in vele gevallen essentieel de kwaliteit van een model te kennen alvorens nieuwe samplepunten gekozen kunnen worden. Het concrete gebruik van kwaliteitsbepaling binnen de sampleselectors is terug te vinden in sectie 12.

De gelijkaardige aanpak van deze kwaliteitsbepaling in beide onderdelen van het framework en de verschillende methodieken die hiervoor gebruikt kunnen worden, maken dat het ook hier belangrijk is plugability te ondersteunen. De doelstelling van deze plugability bestaat erin de gebruiker de mogelijkheid te bieden zijn kwaliteitsbepaling aan te passen aan de concrete situatie. Deze noodzaak manifesteert zich op twee vlakken:

- De kwaliteit van een model kan op velerlei manieren gedefinieerd worden. Voor het testframework is het van belang deze verschillende kwaliteitscontroles eenvoudig te kunnen toepassen op de verschillende modellen. De plugability zorgt er in dit geval voor dat deze verschillende kwaliteiten kunnen onderzocht worden door de desbetreffende kwaliteitsbepaling in het systeem te pluggen.
- Het gebruik van verschillende typen sampleselectors maakt dat elk van deze algoritmes nood heeft aan een specifieke kwaliteitseis om nieuwe punten te selecteren. Het optimaliseren van deze selectie op het niveau van de kwaliteitscontrole kan eenvoudig door alternatieve kwaliteitsbepalingen in het systeem te pluggen.

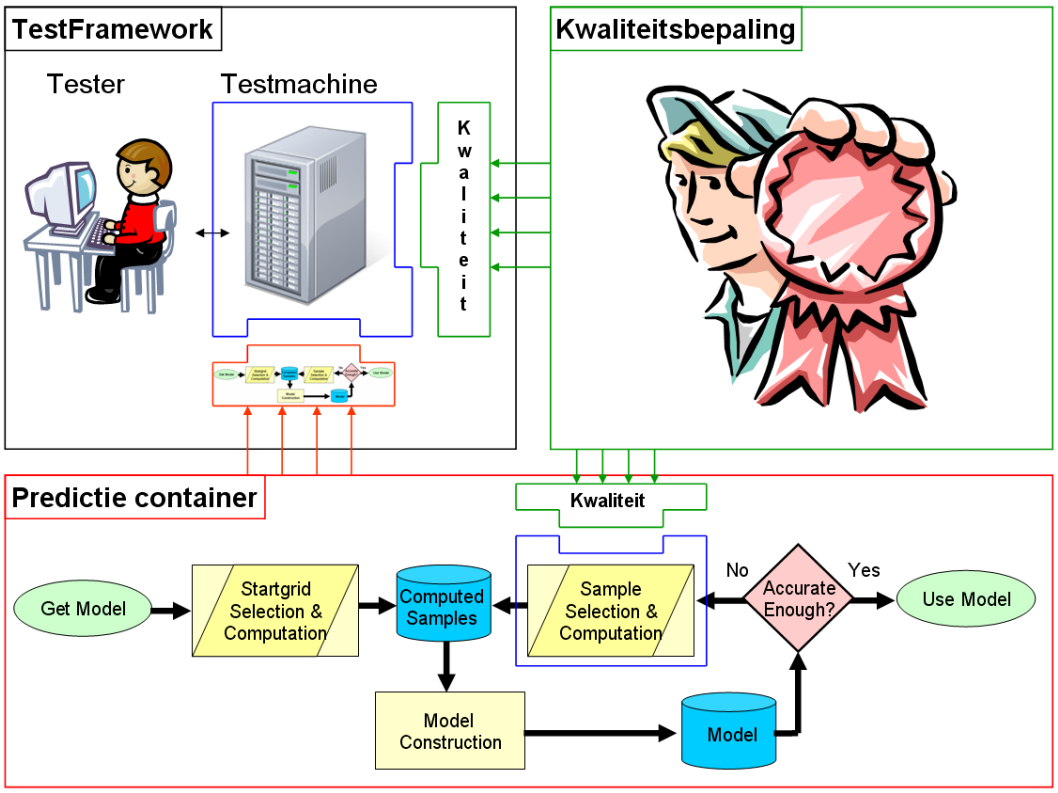
Naast deze twee punten, die de noodzaak van de plugability verantwoorden, is het tevens een groot voordeel dat de kwaliteitscontrole componenten, geïmplementeerd voor de sampleselectie, eveneens in de kwaliteitsbepaling van de modellen geplugd kunnen worden en vice versa. Deze manier van werken bespaart enerzijds veel werk en maakt het anderzijds mogelijk ideeën te transfereren tussen deze twee onderzoeksvlakken.

Figuur 9.3 demonstreert de toepassing van deze plugability op de predictiemethodiek uit figuur 8.1.

Meer informatie omtrent deze methodiek is terug te vinden in sectie 9.4.7.

9.3 Design

Bij de ontwikkeling van software is het belangrijk steeds te vertrekken vanuit een robuust design. Voor er aan een dergelijk design kan begonnen worden, is het essentieel



Figuur 9.3: Toepassing van de plugable kwaliteitsbepaling op de sampleselection en de kwaliteitsbepaling.

te weten wie deze software wil gebruiken en wat deze gebruikers er van verwachten. In dit geval hebben we duidelijk te maken met twee types van gebruikers:

1. *De developer*: Zoals reeds hierboven vermeld, zal de predictie software bestaan uit drie basiscomponenten die in combinatie zullen zorgen voor de gewenste voorspellingen. De implementaties van de verschillende versies van die basiscomponenten verschillen sterk van elkaar. Niet enkel de code op zich maar ook de methodiek om de taak van die component te verwezenlijken zullen meestal verschillen. Aangezien het optimaliseren of het specialiseren van deze basiscomponenten voor verschillende soorten datasets een essentieel deel uitmaakt van de toekomst van deze software, spreekt het voor zich dat de component developer een belangrijke rol zal spelen binnen het design van de software.

2. *De predictie gebruiker*: Zoals het hoort, is ook deze software geschreven om een speciefiek probleem op te lossen voor de gebruiker. In dit geval een voorspelling maken van de runtime van de verschillende taken die de gebruiker wil uitvoeren. Een concreet voorbeeld van zo'n gebruiker is, zoals hierboven reeds aangehaald, een scheduler. In de rest van dit hoofdstuk zal er dan ook steeds naar de *predictie gebruiker* verwezen worden wanneer het woord *scheduler* of gewoon *gebruiker* wordt gehanteerd.

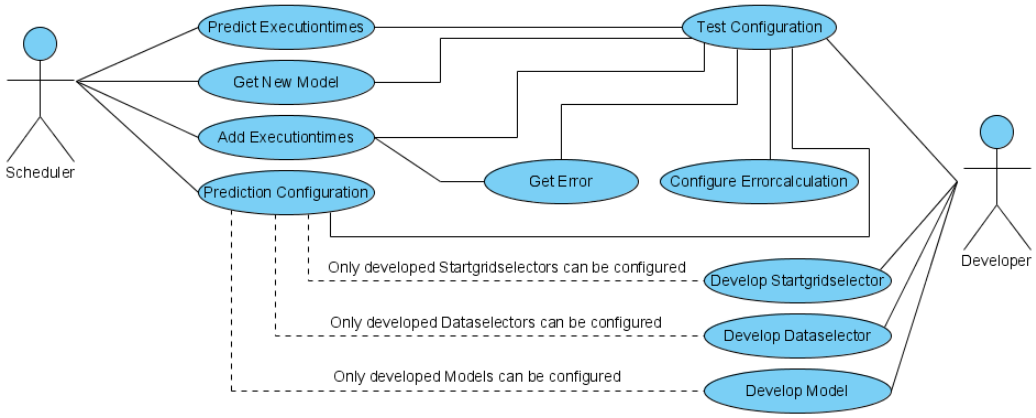
Wanneer de verschillende gebruikers gekend zijn, is het belangrijk de noden en de wensen van de verschillende gebruikers te formuleren.

1. *De developer*:
   (a) Kwaliteitscontrole configureren: Bij de analyse van de werking van de verschillende basiscomponenten is het belangrijk dat er een mogelijkheid bestaat om de kwaliteit op verschillende manieren te controleren. Er wordt enerzijds aan de entiteiten, waartussen de fout berekend wordt, gedacht (bijvoorbeeld tussen verschillende modellen of tussen de originele data en een model). Anderzijds wordt er ook aan de manier waarop de error berekend wordt gedacht (RMS, globale fout, ...).
   (b) Test uitvoeren: Het spreekt voor zich dat de developer in de mogelijkheid moet zijn een test uit te voeren.
   (c) Error Analyseren: Er moet een manier bestaan waarmee de developer de resultaten van de gewenste foutberekening kan verkrijgen.
   (d) Startgridselector ontwikkelen: Het moet mogelijk zijn binnen de software om verschillende manieren om een startgrid te selecteren, te implementeren, te testen en te gebruiken.
   (e) Sampleselector ontwikkelen: Het moet mogelijk zijn binnen de software verschillende technieken om nieuwe datapunten te selecteren, te implementeren, te testen en te gebruiken.
   (f) Model ontwikkelen: Het moet mogelijk zijn binnen de software verschillende modellen te implementeren, te testen en te gebruiken.
2. *De predictie gebruiker*:
   (a) Configureer predictor: Het moet mogelijk zijn te configureren welke startgridselector, dataselector en modellen gebruikt moeten worden bij de predictie.
   (b) Geef uit te voeren punten: De gebruiker moet kunnen opvragen van welke taken de runtime informatie gewenst is.
   (c) Voorspel runtimes: Het moet mogelijk zijn van elke taak op te vragen wat de verwachte runtime is.
   (d) Maak een nieuw model: Het moet mogelijk zijn op elk moment met de aanwezige data een nieuw predictiemodel te bouwen

Op het eerste gezicht blijken de vereisten van deze twee gebruikers vrij ver uit elkaar te liggen. Aan de hand van figuur 9.4 wordt echter het tegendeel bewezen. In essentie komt het erop neer dat het uitvoeren van testen door de developer in pricipe hetzelfde verwacht dan de vereisten van een standaard predictie gebruiker. Dit om de bevindingen van een developer zo nauw mogelijk te laten aansluiten bij de ervaringen van zo'n gebruiker[1]. Er dient dan ook opgemerkt te worden dat de

[1]Wanneer de ervaringen van gebruiker en van developer op elkaar aangesloten worden, dan vereenvoudigt dit de onderlinge communicatie en versnelt dit de developement/verbetering van de desbetreffende component.

'Test Uitvoeren' usecase in figuur 9.4 verbonden zal zijn met al de usecases waarmee de predictie gebruiker (in dit geval een Scheduler) verbonden is.



Figuur 9.4: Gebruiker georienteerde usecase van de ModelBuilder

De vraag blijft dan natuurlijk waar het essentiële verschil zit. Eerst en vooral is de developer, in tegenstelling tot de gebruiker, in staat de verschillende basiscomponenten te ontwikkelen. Dit is terug te vinden onderaan op figuur 9.4. Hiernaast zit de essentie van het verschil hem ook nog in de configuratie, de uitvoering en de analyse van de kwaliteitsbepaling (Errorcalculation). In eerste instantie valt dit op te merken in figuur 9.4 aan de 'Configure Errorcalculation' en de 'Get Error' usecases. Om hier een duidelijker beeld van te scheppen is ervoor gekozen, vanuit de hierboven besproken doelstellingen en de ervaringen uit de scheduling wereld, het use case diagram om te zetten tot twee sequence diagrammen. Een diagram dat de opeenvolgende stappen van de prediction user (de scheduler) toont (zie figuur 9.5) en één diagram dat de opeenvolgende stappen van de developer weergeeft(zie figuur 9.6).

In wat nu volgt, worden eerst de verschillende benodigde objecten, de lus en de te doorlopen stappen in het scheduler sequence diagram besproken, vervolgens worden de verschillen met het developer diagram geanalyseerd.

Er wordt gebruik gemaakt van de volgende zeven componenten:

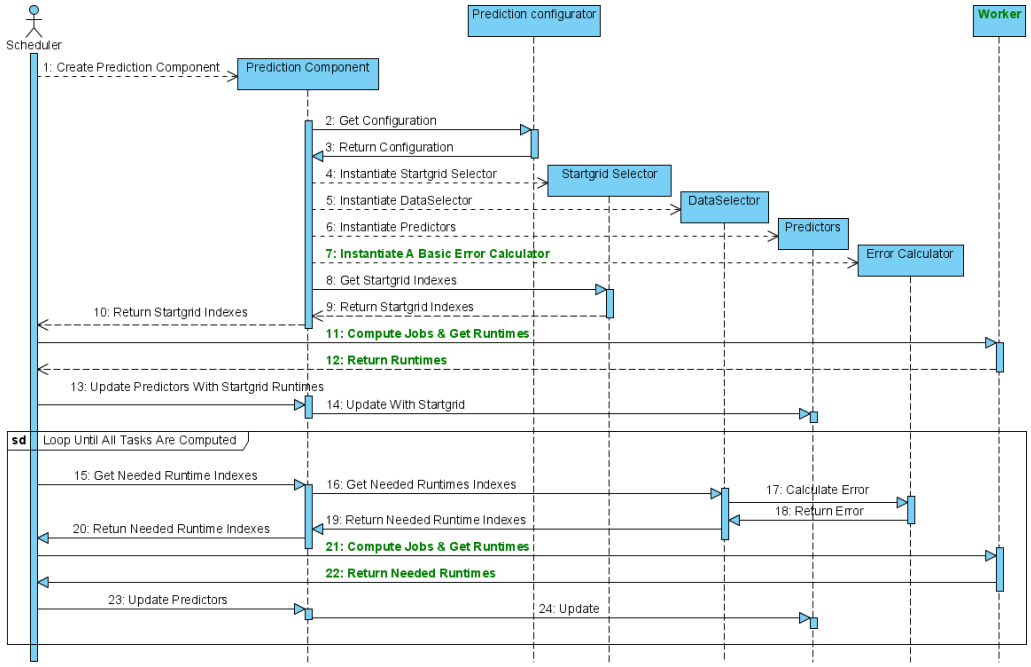
1. *Prediction Component*: Dit is de interface naar de gebruiker toe en wordt daarom ook soms de ModelBuilder interface genoemd. Hij moet voor de gebruiker de volgende dingen mogelijk maken:

   (a) De versie van de verschillende basiscomponenten configureren.

   (b) De voorspellingen voor een arbitreerbare hoeveelheid punten opvragen.

   (c) Extra data toevoegen aan het predictiemodel.

2. *Prediction Configurator*: Deze component moet ervoor zorgen dat de configuratie info, die vereist wordt door de Prediction Component, wordt opgehaald.



Figuur 9.5: Prediction user sequence diagram: Geeft de opeenvolgende stappen weer die een prediction user doorloopt bij het gebruik van de Modelbuilder.

3. *Startgrid Selector* : Deze basiscomponent selecteert, zoals reeds besproken, de punten waarvan de runtimes bekend moeten zijn alvorens de predictie gestart kan worden.

4. *Predictors*: De predictors zijn de verschillende versies van modellen die gebruikt worden om de voorspellingen te maken.

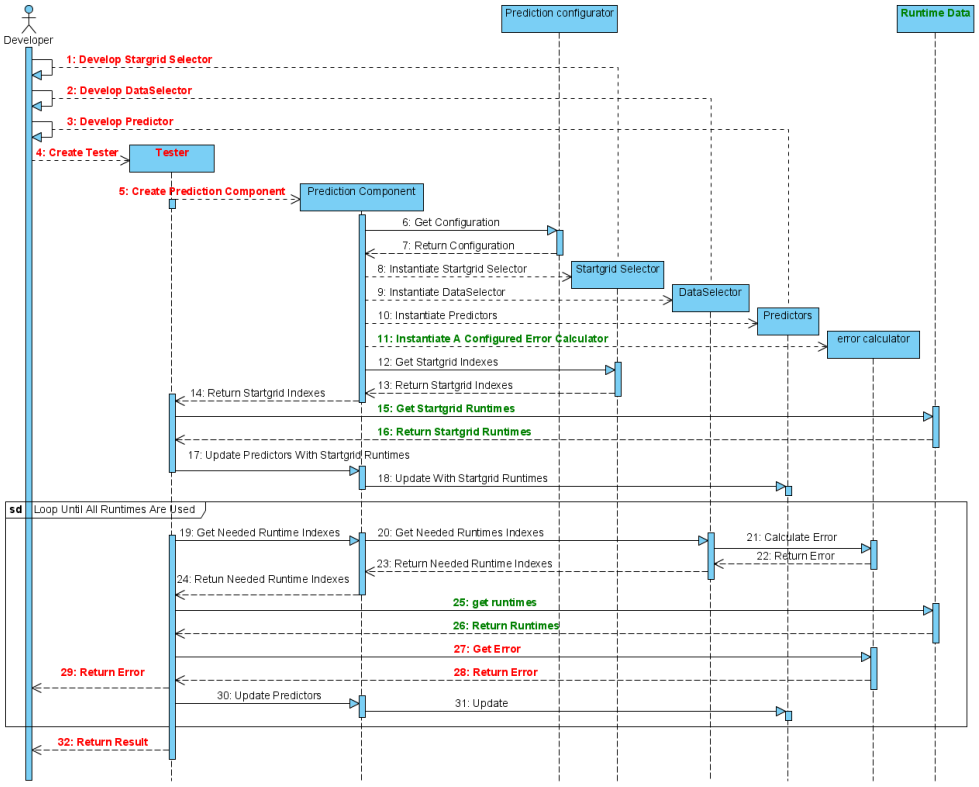
5. *SampleSelector*: Deze basiscomponent selecteert, zoals reeds besproken, de taken waarvan de info omtrent de runtime het meest gewenst is om het huidige model te optimaliseren. (In het verloop van deze tekst wordt deze component, om historische redenen, soms ook *Data Selector* genoemd).

6. *Errorcalculator*: Berekent de fout van het geconstrueerde model, dit kan gebeuren aan de hand van gekende data of aan de hand van andere modellen. Naar deze component wordt in deze tekst ook vaak verwezen met de term *Kwaltiteitscontrole.*

7. *Worker*: Deze component voert de geselecteerde taken uit.

In het onderste gedeelte van figuur 9.5 worden de laatste 10 messages omkaderd door een lus. De lus omvat drie stappen die repetitief worden uitgevoerd totdat er niet langer voldaan wordt aan de lusvoorwaarde:

- Er wordt een nieuw model gebouwd.



Figuur 9.6: Developer sequence diagram: Geeft de opeenvolgende stappen weer die een developer doorloopt bij het gebruik van de Modelbuilder.

- Er wordt berekend welke runtimes essentieel gekend moeten zijn om het bestaande model te verbeteren.
- De runtimes van deze taken worden achterhaald door de taken uit te voeren.

In het verdere verloop van de tekst wordt getoond dat deze lus op drie manieren onderbroken kan worden:

1. De predictie wordt van buitenaf gestopt.
2. De accuraatheid van het model voldoet aan de vereisten.
3. Alle datapunten zijn gekend (Het model kan niet meer verfijnd worden).

Zowel binnen als buiten de lus wordt het verloop van het sequence diagram bepaald door een opeenvolging van messages. Om een zo duidelijk mogelijk beeld te krijgen van dit verloop worden de verschillende messages hieronder toegelicht:

1. *Create Prediction Component*: Als eerste stap zal een gebruiker van de software een Prediction Component aanmaken die, zoals hierboven reeds beschreven, zal zorgen voor de interface tussen de gebruiker en de onderliggende technieken.

2. *Get Configuration*: Als eerste stap binnen het aanmaken van een Prediction Component moeten de verschillende basiscomponenten geconfigureerd worden. Deze message zal ervoor zorgen dat de Prediction configurator deze configuratie informatie zal vergaren.

3. *Return Configuration*: De configuratie informatie wordt doorgegeven aan de Prediction Component.

4. *Instantiate Startgrid Selector*: Gebruik makend van de configuratie informatie wordt de Startgrid Selector geïnstantieerd.

5. *Instantiate DataSelector*: Gebruik makend van de configuratie informatie wordt de DataSelector geïnstantieerd.

6. *Instantiate Predictors*: Gebruik makend van de configuratie informatie worden de nodige modellen geïnstantieerd.

7. *Instantiate A Basic Error Calculator*: Gebruik makend van de configuratie informatie wordt een Basis Error Calculator geïnstantieerd. De basis wil hierin zeggen dat die enkel zal afhangen van de noden van de Dataselector. Later hierover meer.

8. *Get Startgrid Indexes*: De Startgrid Selector krijgt de opdracht te onderzoeken welke datapuntindexen tot de gewenste startgrid behoren.

9. *Return Startgrid Indexes*: De desbetreffende datapuntindexen worden teruggegeven.

10. *Return Startgrid Indexes*: De datapuntindexen worden via de Modelbuilder interface aan de gebruiker doorgespeeld.

11. *Compute Runtimes*: De gebruiker, in dit geval de scheduler, zal de taken die geïndexeerd worden in de verkregen datapuntindexlijst schedulen zodat de runtime van deze taken verkregen wordt.

12. *Return Runtimes*: De verkregen runtimes worden doorgegeven aan de gebruiker.

13. *Update Predictors With Startgrid Runtimes*: De gebruiker geeft de opdracht aan de Modelbuilder interface om het huidige model te updaten met de nieuw verkregen runtime informatie.

14. *Update With Startgrid Runtimes*: De Modelbuilder interface zorgt ervoor dat de betrokken predictors worden geüpdatet met deze nieuwe informatie.

15. *Get Needed Runtime Indexes*: Nu er een nieuw model gecreëerd is geeft de gebruiker, via de Modelbuilder interface, de opdracht te achterhalen welke punten het belangrijkste zijn om het huidige predictiemodel te verfijnen.

16. *Get Needed Runtime Indexes*: De Modelbuilder interface geeft de opdracht aan de DataSelector om de indexen van deze punten te bepalen.

17. *Calculate Error*: De dataselector vraagt via de Errorcalculator de fout van elk punt in het huidige model op.

18. *Return Error*: De fout van al deze punten wordt teruggegeven aan de Dataselector.

19. *Return Needed Runtime Indexes*: Aan de hand van deze fouten worden de indexen van de belangrijkste punten geselecteerd en teruggegeven aan de Modelbuilder interface.

20. *Return Needed Runtime Indexes*: De interface zorgt er dan weer voor dat deze indexen worden doorgespeeld aan de gebruiker.

21. *Compute Runtimes*: De gebruiker, in dit geval de scheduler, zal de taken die geïndexeerd worden in de verkregen datapuntindexlijst schedulen zodat de runtime van deze taken verkregen wordt.

22. *Return Runtimes*: De verkregen runtimes worden doorgegeven aan de gebruiker.

23. *Update Predictors*: De gebruiker geeft opdracht aan de Modelbuilder interface het huidige model te updaten met de nieuw verkregen runtime informatie.

24. *Update*: De Modelbuilder interface zorgt ervoor dat de betrokken predictors worden geüpdatet met deze nieuwe informatie.

Na deze uiteenzetting omtrent het Scheduler sequence diagram wordt dit vergeleken met het Developer usecase diagram. De verschillen tussen de twee diagrammen zijn aangeduid in kleur. De rode tekst staat voor messages/componenten die enkel in het desbetreffende diagram voorkomen en de groene messages/componenten zijn diegene die in beide diagrammen voorkomen maar een andere invulling hebben. In wat nu volgt, worden deze verschillen stuk voor stuk toegelichten. De messages worden geadresseerd met de nummer die ze krijgen in het developer diagram.

Messages 1, 2 en 3 zijn uniek voor de Developer en spreken voor zich. Ze demonstreren dat de developer steeds begint bij het implementeren van de drie basiscomponenten. Hier moet wel bij vermeld worden dat dit zich meestal beperkt tot één van deze componenten. De overige worden dan ingevuld door reeds ontwikkelde versies. De constraint lijnen die deze messages verbinden met respectievelijk de instantiëring van een Startgrids Selector, een DataSelector en de Predictors wijzen op het feit dat de Developer bij de instantiëring verantwoordelijk is voor het aanwezig zijn van de implementatie van deze componenten. Message 4 is ook uniek voor de Developer en zorgt voor de instantiëring van de interface die reeds in het gebruiker georiënteerd usecase diagram besproken werd (Zie figuur 9.4). Deze interface moet ervoor zorgen dat de gebruiker over alle functionaliteit beschikt om zijn implementaties van de basiscomponenten te testen. Zoals de rode kleur in het sequence diagram al aanduidt

spreekt het voor zich dat deze component enkel door de Developer gebruikt wordt. Messages 11, 27 en 28 zijn minder evident. Deze tonen aan dat de Developer in de mogelijkheid moet zijn de foutberekening te configureren, uit te voeren en op te vragen. Er moet hier dus opgemerkt worden dat de Developer, naast de basiscomponenten voor de voorspellingen, ook nood zal hebben aan een eigen implementatie van de foutberekeningscomponent. Hierover later meer in de implementatie fase.

Messages 29 en 32 sluiten nauw aan bij het voorgaande en zijn toegevoegd om duidelijk te maken dat de developer feedback moet krijgen over de grootorde van de fout zowel tijdens het uitvoeren van de test (message 29) als bij het aflopen van de test (message 32). Dit om het mogelijk te maken eventuele inconsistenties of irrelevante tests te detecteren en stop te zetten zonder daarvoor de gehele test te moeten laten uitvoeren. Het vervolg van deze tekst toont aan dat dit bij sommige modellerings- en dataselectie technieken, omwille van de computationele requirements, onontbeerlijk is. Tot slot wordt de Runtime Data component in combinatie met messages 15, 16, 25 en 26 onder de loep genomen. Zoals messages 15 en 25 aanduiden, zorgt deze component voor het vergaren van de runtimes van de taken. Aangezien de Developer werkt met snelle simulaties zullen deze runtimes opgehaald worden uit een data structuur. Vandaar de naam Runtime DATA. Bij de scheduler ligt dit enigszins anders. Figuur 9.5 duidt aan dat de runtimes door middel van messages 11 en 21 niet uit een een datastructuur komen maar dat deze door het uitvoeren van de betrokken taak worden verkregen. Dit brengt ook met zich mee dat we hier niet met een Runtime Data component werken maar met een Worker component die de taken kan uitvoeren.

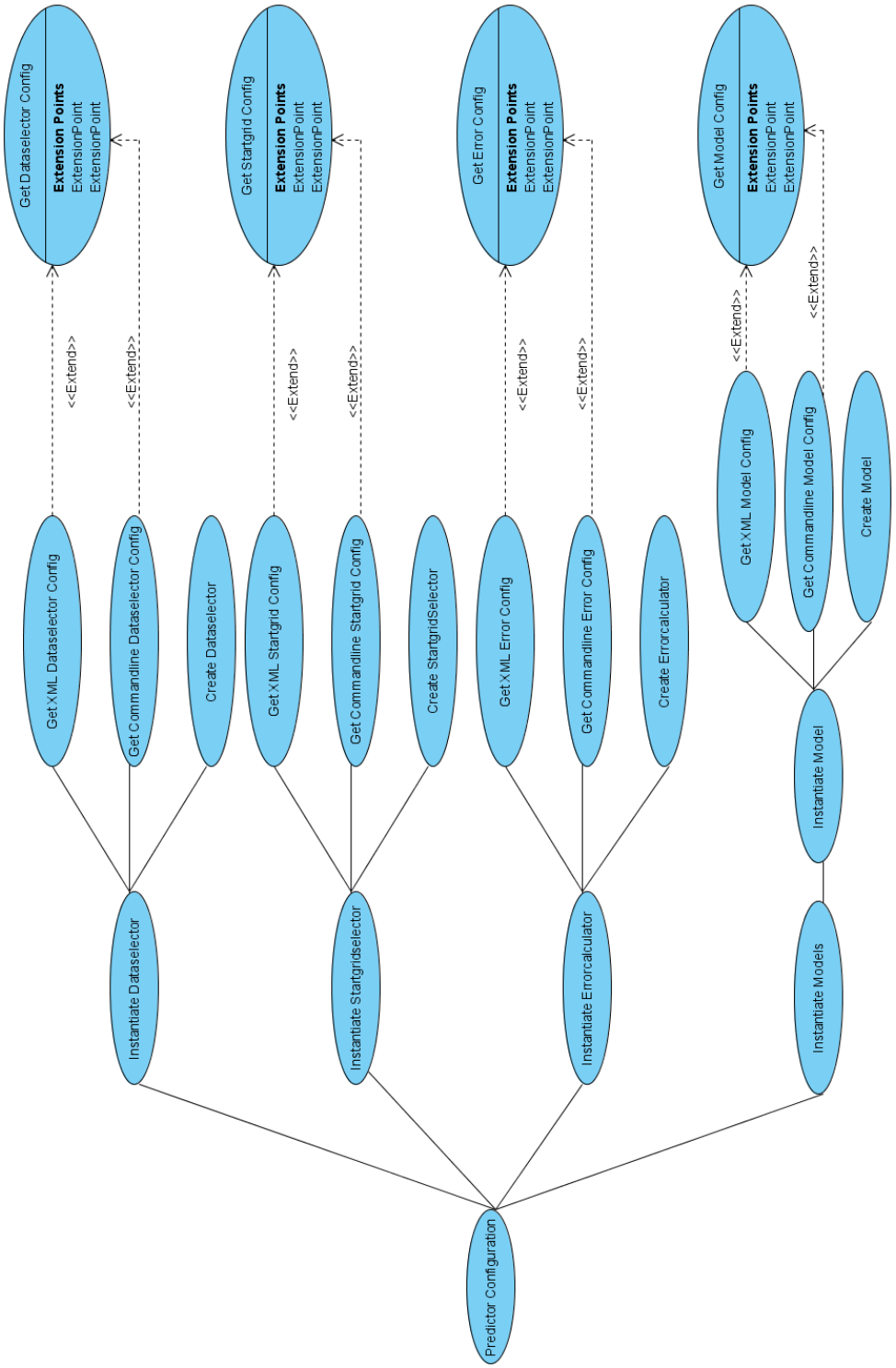
Nu de gebruikers en hun wensen besproken zijn, blijft er nog steeds een belangrijk gedeelte van het design open namelijk het configuratie gedeelte. De “Gebruiker georiënteerde usecase” in figuur 9.4 blijft hier immers veel te vaag. Dit onderdeel van het framework is dan ook verder uitgewerkt in de ‘Prediction Configuration usecase’ (zie figuur 9.7). Zoals reeds eerder werd aangehaald, is het belangrijkste concept binnen heel het framework de plugability. Plugability mogelijk maken, is echter minder evident dan het lijkt. Er dient immers een mechanisme te bestaan dat, at runtime, weet welke implementatie er voor welke component moet ingeladen worden. Zoals al uit de sequence diagrammen kon afgeleid worden, moet dit probleem opgelost worden gebruik makend van de prediction component. Figuur 9.7 toont dat de configuratie van het systeem essentieel bestaat uit de instantiatie van de vier basiscomponenten: de dataselector, de startgridselector, de errorcalculator (of kwaliteitscontrole) en de verschillende modellen. Elk van deze componenten moet op verschillende manieren aangemaakt en geconfigureerd kunnen worden:

- Het moet mogelijk zijn, door middel van constructors, de basiscomponenten aan te maken en te configureren. Dit is essentieel om de uitbreidbaarheid van elk van de implementaties te ondersteunen.

- Het moet mogelijk zijn, door middel van configuratieobjecten, de verschillende componenten te configureren. Het opzet hier is een medium bieden dat elk type van configuratiemethodiek kan omzetten naar een configuratieobject om vervolgens, via standaard constructors en configuratiemethoden de verschillende

componenten naar wens te initialiseren. Concreet wordt er hier gedacht aan configuratie vanuit XML files, configuratie databases, ...

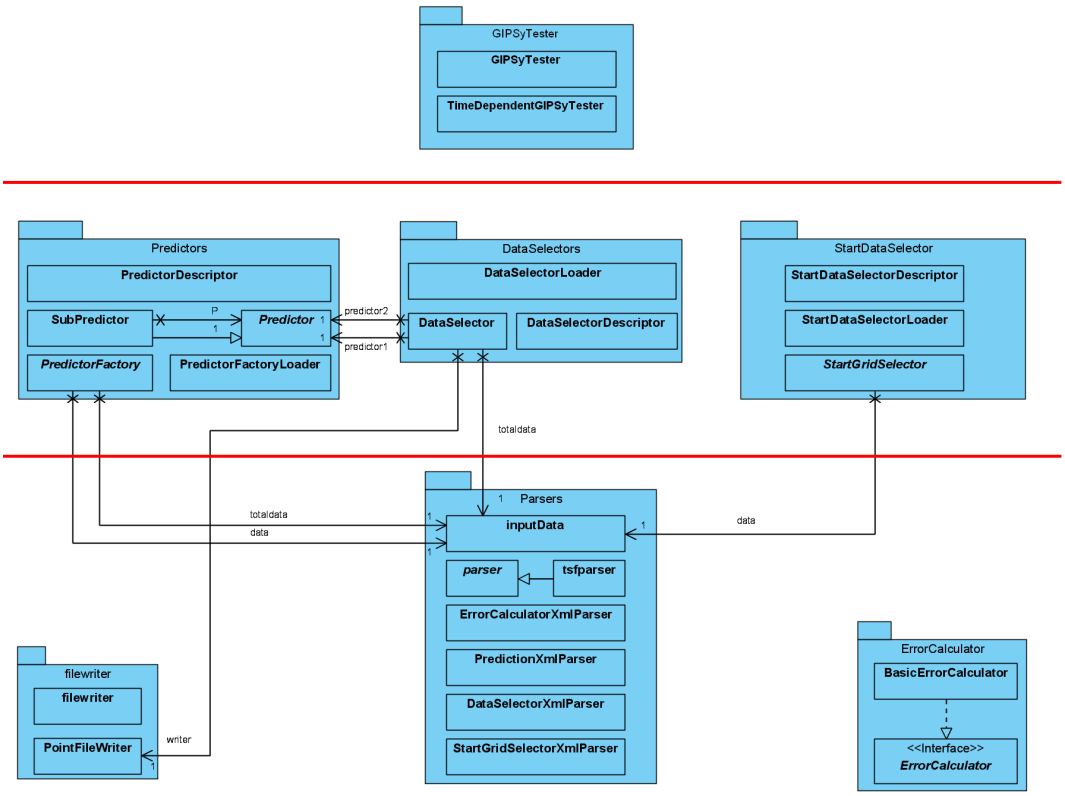
Tot slot dient nog vermeld te worden dat er in vele gevallen meerdere modellen geconfigureerd dienen te worden. Om dit te verwezenlijken wordt onderaan in de usecase een tussenstap opgenomen waarin verschillende modellen geïnstantieerd kunnen worden, gebruik makend van het standaard instantiatie gedeelte dat slechts één model instantieert.



Figuur 9.7: Use case die de Prediction configuration in meer detail beschrijft.

9.4 Implementatie

In deze sectie wordt een concrete implementatie van het gewenste design voorgesteld. Aan de hand van een globaal klassediagram zal een beeld geschetst worden van de verschillende basiscomponentenen en van hun onderlinge afhankelijkheden. Vervolgens zal op elke component van de Modelbuilder worden ingezoomd om zo, aan de hand van meer gedetailleerde diagrammen en informatie, de concrete details van de implementatie toe te lichten.



Figuur 9.8: Algemeen Klassediagram

Het globaal klassediagram op figuur 9.8 is onderverdeeld in drie lagen. De middelste laag bevat duidelijk de drie essentiële basiscomponenten die in de design fase besproken en uitgewerkt werden: De DataSelector, de StartGrid Selector, de Predictors. Naar deze laag wordt in het verdere verloop van de tekst gerefereerd als de *Basis laag* of *Core layer*. De laag onder deze basis laag bevat de verschillende componenten die door de basiscomponenten gebruikt worden maar die los staan van de gebruikte methodiek. In essentie komt het erop neer dat de verschillende versies van een Predictor, een DataSelector of een StartGrid Selector deze componenten gaan gebruiken om enkele basis opdrachten uit te voeren. Meer concreet gaat het

hier over het parsen van inputdata, het wegschrijven van resultaten en het berekenen van fouten. Zowel in het algemene usecase diagram (zie figuur 9.4) als in het Designer sequence diagram (zie figuur 9.6) kwam tot uiting dat er nood was aan een interface die de verschillende implementaties van de basiscomponenten op een eenvoudige maar doorgedreven manier kon evalueren en classificeren. De bovenste laag bevat dan ook de *Test* component die deze interface verzorgt. In wat nu volgt, wordt er dieper ingezoomd op de verschillende individuele componenten om aldus de concrete implementatie te bespreken.

9.4.1 Configuratie structuur

Omwille van de verwevenheid van de configuratie structuur in het volledige framework is het belangrijk een correct beeld de scheppen van deze configuratiemethodiek alvorens elk van de verschillende componenten in detail te bespreken.

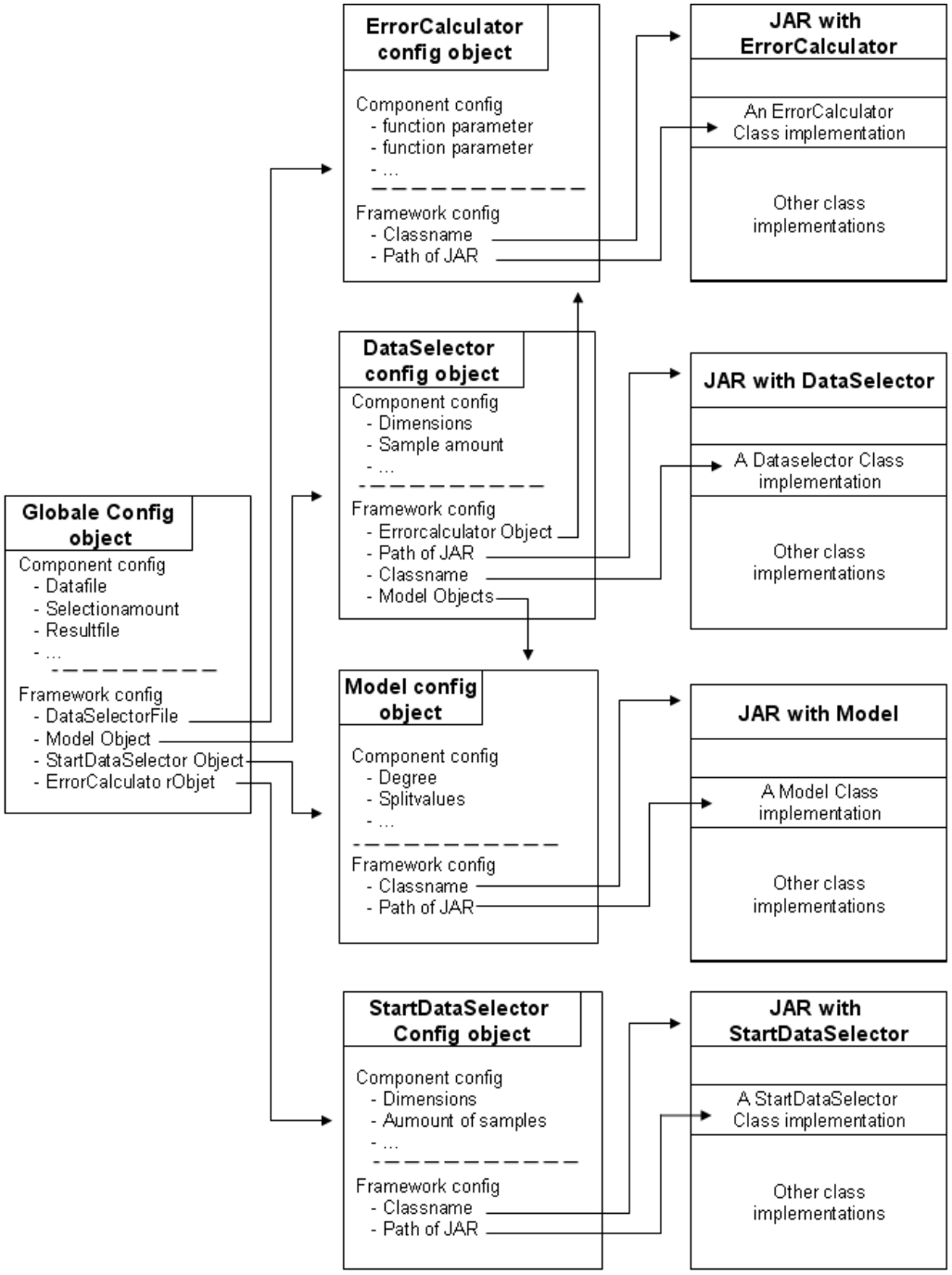
Het doel van de configurator is tweeledig:

1. *Framework configuratie.* Op deze manier wordt de plugability ondersteund. Aan de hand van het configuratie mechanisme wordt de gewenste implementatie van een bepaalde component in het framework geplugd.

2. *Component configuratie.* Naast het inpluggen van componenten is het tevens belangrijk een ingeplugde implementatie te configureren. Zo kan bijvoorbeeld bij een predictor, die gebaseerd is op een polynoom, worden ingesteld welke graad deze polynoom maximaal mag hebben.

Figuur 9.9 geeft een duidelijk overzicht van de structuur van de configuratie. Voor de figuur verder besproken wordt, is het belangrijk op te merken dat wanneer er over configuratieobjecten gesproken wordt de vorm van deze objecten niet vast moet liggen. Zoals het usecase diagram uit figuur 9.7 reeds aangaf, moet de software immers de mogelijkheid bieden verschillende types configuratieobjecten te ondersteunen (zoals configfiles, databases; ...). Om dit te verwezenlijken, is het mogelijk gemaakt verschillende configparsers in te pluggen op het systeem.

Figuur 9.9 bestaat enerzijds uit configuratie objecten (de 1ste en de 2de kolom) en anderzijds uit de jar files die de implementatie van de te configureren grid componenten bevatten. Het eerste configuratieobject is het *globale config object.* Zoals de naam al laat vermoeden, zal het instaan voor de configuratie van de component, die bovenop de andere componenten, het geheel samenhoudt, de GIPSyTester component (zie figuur 9.8). Zoals elk configuratieobject bestaat ook dit object uit twee gedeelten: de Framework configuratie en de Component configuratie. De Component configuratie bevat de informatie die nodig is om de GIPSyTester component te configureren. Voorbeelden van dergelijke informatie zijn: het path van de datafile waarin de runtimes gespecifieerd staan, het aantal datapunten dat per ronde geselecteerd mag worden door de dataselector, het path van de file waarin de resultaten moeten geschreven worden, ... Het framework gedeelte van de configuratiefile bevat de paden naar de configuratiefiles, die elk van de overige objecten van het framework specificeren. Het gaat hier meer bepaald over de ErrorCalculator, de DataSelector,



Figuur 9.9: Overzicht van de configuratiestructuur

het Model en de StartDataSelector configuratie files. In figuur 9.9 wordt dit duidelijk gemaakt door de pijlen die naar de desbetreffende configuratiefiles verwijzen. Een voorbeeld van zulk een Globaal Configuratie object is terug te vinden in Appendix

D.1.

De eerste pijl die vertrekt uit het Globale config object komt aan bij het ErrorCalculator configuratie object. Dit configuratie object zal instaan voor de specificatie van de kwaliteitscontrole die binnen het framework gebruikt wordt. Hoewel er in de figuur staat aangegeven dat ook dit configuratie object de mogelijkheid heeft een component configuratie gedeelte te bezitten, wordt dit tot noch toe zelden of nooit gebruikt. Het Framework configuratie gedeelte daarentegen is wel enorm belangrijk. Het bevat twee onderdelen:

1. *Het Classname onderdeel:* Dit onderdeel bevat de naam van de klasse die gebruikt moet worden om de kwaliteitscontrole uit te voeren.

2. *Het path of JAR onderdeel:* Dit onderdeel bevat het path van de JAR file waarin de gespecificeerde klasse gedefinieerd wordt.

Het framework zal dan, bij het inlezen van de configfile, een `URIclassloader` initialiseren met de huidige classloader en het path van de JAR file als parameters. Aan de hand van deze classloader kan de desbetreffende kwaliteitscontrole geladen en gebruikt worden. De exacte werking hiervan wordt uitgelegd in sectie 9.4.2. Een voorbeeld van zulk een ErrorCalculator configuratie object is terug te vinden in Appendix D.2.

De tweede pijl komt aan bij het DataSelector configuratie object. Dit configuratieobject zal instaan voor de specificatie van de dataselector (ook wel sampleselector genoemd) die binnen het framework gebruikt wordt. Naast het implementatie specifieke component configuratie gedeelte (de informatie die de dataselector zelf initialiseert) bezit dit object ook een framework configuratie gedeelte. In dit geval bestaat dit gedeelte uit vier onderdelen:

1. *Errorcalulator onderdeel:* De kwaliteit van een model zal steeds gemeten moeten worden aan de hand van een maatstaf. De implementatie van die maatstaf wordt aangereikt door de Errorcalculator. In essentie berekent deze component de fout van het huidige model. Dit onderdeel bevat het path van de configuratiecomponent die instaat voor specificatie van deze errorcalculator. Dit wordt aangeduid op figuur 9.9 aan de hand van de pijl die naar het Errorcalculator configuratie object verwijst.

2. *ModelFiles onderdeel:* In vele gevallen zal de selectie van datapunten gebeuren aan de hand van extra modellen. Deze modellen worden gespecificeerd aan de hand van een configfile waarvan het path in dit onderdeel omvat zit. De pijl achter dit onderdeel in figuur 9.9 maakt dit duidelijk.

3. *Het Classname onderdeel:* Dit onderdeel bevat de naam van de klasse die gebruikt moet worden om de DataSelectie uit te voeren.

4. *Het path of JAR onderdeel:* Dit onderdeel bevat het path van de JAR file waarin de gespecificeerde klasse gedefinieerd wordt.

Het framework initialiseert een `URIclassloader`, bij het inlezen van de configfile, met de huidige classloader en het path van de JAR file als parameters. Aan de hand van deze classloader kan de desbetreffende dataselector geladen en gebruikt worden. De exacte werking hiervan wordt uitgelegd in sectie 9.4.2. Een voorbeeld van zulk een Dataselector configuratie object is terug te vinden in Appendix D.3.

De derde pijl komt aan bij het Model configuration object. Dit object zal instaan voor de specificatie van de modellen die binnen het framework gebruikt worden. In het component configuratie gedeelte wordt er in dit geval informatie doorgegeven die essentieel is voor die specifieke implementatie van een model. Bij een polynoom bijvoorbeeld zal worden aangegeven welke graad hij heeft of bij een Kriging functie zal aangegeven worden welke Kernel functie er gebruikt moet worden. Het framework configuratie gedeelte bevat in dit geval, net zoals bij het ErrorCalculator configuratie object, slechts twee componenten:

1. *Het Classname onderdeel:* Dit onderdeel bevat de naam van de klasse die gebruikt moet worden om een model op te bouwen.

2. *Het path of JAR onderdeel:* Dit onderdeel bevat het path van de JAR file waarin de gespecificeerde klasse gedefinieerd wordt.

Het framework zal dan, bij het inlezen van de configfile, een `URIclassloader` initialiseren met de huidige classloader en het path van de JAR file als parameters. Aan de hand van deze classloader kan het desbetreffende model geladen en gebruikt worden. De exacte werking hiervan wordt besproken in sectie 9.4.2. Een voorbeeld van zulk een Model configuratie object is terug te vinden in Appendix D.4.

Tot slot komt de vierde pijl aan bij het StartDataSelector Configuratie object. Dit object zal instaan voor de specificatie van de start data selectie (ook wel startgridselection of initial data selection genoemd). Net zoals bij de Error calculator configuratie component zal ook in dit geval het component configuratie gedeelte veelal leeg blijven. Het is echter wel mogelijk, bij een individuele implementatie van een startgridselector, dit component configuratie gedeelte te gebruiken om alsnog parameters naar deze eigen implementatie door te sluizen. Ook dit configuratie object bevat een framework configuratie gedeelte. Dit gedeelte bevat ook in dit geval twee componenten:

1. *Het Classname onderdeel:* Dit onderdeel omvat de naam van de klasse die gebruikt moet worden om de startgridselectie uit te voeren.

2. *Het path of JAR onderdeel:* Dit onderdeel beschrijft het path van de JAR file waarin de gespecificeerde klasse gedefinieerd wordt.

Het framework zal dan, bij het inlezen van de configfile, een `URIclassloader` initialiseren met de huidige classloader en het path van de JAR file als parameters. Aan de hand van deze classloader kan de desbetreffende StartDataSelector geladen en gebruikt worden. De exacte werking hiervan wordt uitgelegd in sectie 9.4.2. Een

voorbeeld van zulk een Model configuratie object is terug te vinden in Appendix D.5.

9.4.2 Configurator

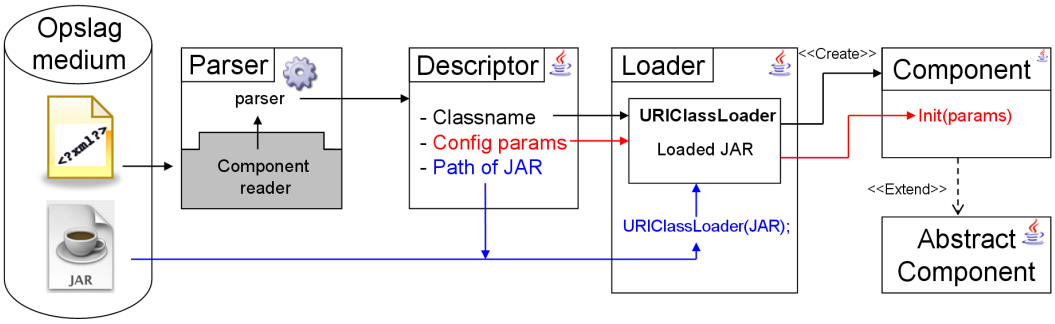
Nu de object structuur, waarbinnen de configuratie van de verschillende componenten van het framework vervat zit, gekend is, resteert enkel nog de taak elk van de componenten op een plugable manier te initialiseren en te gebruiken. Concreet wil dit zeggen dat er een methodiek geïmplementeerd moet worden die, aan de hand van de configuratieobjecten, de gewenste implementatie kiest voor een bepaald object en dit object tevens initialiseert en in het framework plugt.

De gebruikte methodiek is voor elk van de verschillende componenten vrij analoog en gebaseerd op vier onderdelen:

- *Parser*: Deze component zet de data, verworven uit de configuratie component, om naar een component descriptor.
- *Component descriptor*: De Component descriptor bevat al de informatie die verkregen is uit de configuratie component.
- *Component loader*: Deze component verwerft de benodigde classpaths van de componentdescriptor. Hij vervangt de huidige classloader door een URIClassLoader die geïnitialiseerd is met de huidige classloader en met de verworven classpaths. Met deze classloader wordt de gewenste component geladen en als returnwaarde teruggegeven. In de functie definitie staat de returnwaarde steeds gedefinieerd als een entiteit van de abstracte component klasse zodat elke mogelijke afgeleide implementatie van die klasse gebruikt kan worden.
- *Abstracte component klasse*: Deze abstracte klasse beschrijft de basisfunctionaliteit die de component moet bezitten om gebruikt te kunnen worden in het systeem (om te kunnen ingeplugd worden). Nieuwe implementaties van deze componenten zijn per definitie afgeleide klassen van deze klasse waarbij de abstracte functies geïmplementeerd zijn en eventueel de publieke functies vervangen zijn door een nieuwe versie.

De concrete werking van het framework wordt duidelijker op figuur 9.10.

Het plugable component reader onderdeel van de Parser zorgt ervoor dat een configuration object (in dit geval een xml file) in de parser terecht komt. Hier wordt dit omgezet in een Discriptor object. Deze Java klasse bevat, zoals reeds aangehaald, al de informatie omtrent de te configureren component. Gebruik makend van deze info wordt in het Componentloader object een nieuwe `URIclassloader` geïnstantieerd met de in de Descriptor gespecificeerde JAR file als zijn path. Gebruik makend van deze classloader wordt de klasse met de classname, die eveneens in de descriptor gespecificeerd staat, geladen. Deze instantie van een klasse, afgeleid van de Abstracte component klasse, wordt verder geïnitialiseerd door zijn `init` methode aan te roepen met de overige configuratieinformatie als parameters. Om volledig correct



Figuur 9.10: Opeenvolgende stappen tijdens de configuratie van een component

te zijn, dient hierbij vermeld te worden dat de parameters aan de `init` methode worden meegegeven onder de vorm van het volledige descriptor object.

De implementatie van een nieuwe component omvat bijgevolg slechts drie taken:

1. Een configuratiebestand aanmaken waarin het volgende terug te vinden is:
   - Het path van de JAR file die de code van de nieuwe component omvat.
   - De naam van de klasse die de nieuwe component definieert.
   - De informatie die nodig is om de component te initialiseren en te laten werken.
2. Het overloaden van de basisklasse van de desbetreffende component om aldus de gewenste functionaliteit te verwerven.
3. De initialisatie procedure van de nieuwe klasse aanpassen aan de informatie die in het configuratieobject wordt meegegeven en die in de initialisatie methode terecht komt onder de vorm van de componentdescriptor. Zo zal bijvoorbeeld een implementatie van de `DataSelector` component, die gebruik maakt van andere modellen, in deze initialisatie procedure, aan de hand van de gegevens in de componentdescriptor, deze andere modellen moeten laden[2].

Er kan dus geconcludeerd worden dat de gebruiker van een nieuwe component verantwoordelijk is voor de manier waarop het configuratiebestand van deze component is opgebouwd en voor de manier waarop hiermee wordt omgegaan binnen de nieuwe component.

9.4.3 GIPSyTester

Zoals hierboven reeds is aangekaart, staat deze component in voor het evalueren en het classificeren van de verschillende basiscomponenten. In de huidige versie van het systeem zijn er vier verschillende implementaties met enkele subtiele verschillen: de GIPSyTester, de TimeDependentGIPSyTester, de DistributedGIPSyTester en de

[2]Het laden van extra modellen gebeurt meestal aan de hand van een extra configuratieobject waarnaar in het huidige configuratieobject verwezen wordt.

DistributedTimeDependentGIPSyTester. Voor er verder wordt ingegaan op deze verschillen wordt eerst de standaard GIPSyTester behandeld.

Het `GIPSyTester` object is zoals, reeds werd vermeld, het object dat het volledige testframework stuurt door de verschillende componenten te laten samenwerken. Dit wordt verwezenlijkt door een `Main` procedure . In wat nu volgt, wordt, aan de hand van het sequence diagram uit figuur 9.6, de implementatie van de opeenvolgende stappen in de main procedure besproken. Aangezien Messages 1 tot en met 3 instaan voor de implementatie van de verschillende componenten en los staan van het testframework wordt er gestart bij message 4.

1. *Message* 4: Deze message is bij aanvang van het project opgesteld in het sequence diagram omdat het, in de initiële planning, de bedoeling had een Tester object aan te maken dat gebruikt kon worden om testen uit te voeren. Bij de concrete implementatie is van deze visie afgestapt en werd de implementatie verzameld in een statische `Main` procedure. Deze stap kan tijdens de implementatie aldus genegeerd worden.

2. *Message* 5: In deze stap zou, in principe, volgens het design, de prediction component constructor moeten aangeroepen worden. Hoewel dit initieel zo werkte, is later gebleken dat de plugability van individuele componenten veel krachtiger was binnen de verschillende toepassingen (bijvoorbeeld de scheduler). Omwille van dit feit is er van deze gecombineerde plugability afgestapt en werd er gekozen voor het individueel inpluggen van elke component in de verschillende toepassingen. In een implementatie wordt het gebruik van een predictor container vervangen door het gebruik van zowel een sampleselector, als een predictor, als een dataselector en als een error calculator component.

3. *Messages* 6 *en* 7: Deze messages dienen ertoe duidelijk te maken dat er een methode moet bestaan die alle nodige informatie omtrent de huidige predictie opstelling kan vergaren. Concreet zullen de verschillende onderdelen van het Parser package (zie figuur 9.6) gebruikt worden om deze informatie op te halen. De basisinformatie, alle test specificaties en de paden van de basiscomponent gerelateerde configfiles, worden opgehaald uit een Java properties file. De code hiervan is terug te vinden in lijnen 1 tot en met 10 van de onderstaande code. In lijnen 11 tot en met 14 zullen respectievelijk de StartGridXmlParser, de DataSelectorXmlParser en de PredictionXMLParser uit het Parser package gebruikt om de configuratie gegevens aangaande de basiscomponenten te verkrijgen. In lijnen 15 en 16 worden de rekentijd gegevens, die voor deze simulatie gebruikt worden, ingelezen. Dit gebruik makend van de tsfparser uit het Parser package. Deze rekentijd informatie wordt ondergebracht in de inputData klasse die later in dit hoofdstuk besproken wordt.

```
1 Properties properties = new Properties();
2 properties.load(new FileInputStream(args[0]));
3 amount=new Integer(properties.getProperty("SelectionAmount"));
4 String plot = properties.getProperty("Plot");
```

```
5  String model1file = properties.getProperty("Model1File");
6  String datafile = properties.getProperty("DataFile");
7  String dataselectorfile = properties.getProperty("
       DataSelectorFile");
8  String startgridselectorfile = properties.getProperty("
       StartDataSelectorFile");
9  String errorCalculatorFile = properties.getProperty("
       ErrorCalculatorFile");
10 String ResultFile = properties.getProperty("ResultFile");
11 StartGridSelectorDescriptor startGridSelectorDescriptor = new
       StartGridSelectorXmlParser().parseDocument(
       startgridselectorfile);
12 DataSelectorDescriptor dataSelectorDescriptor = new
       DataSelectorXmlParser().parseDocument(dataselectorfile);
13 PredictorDescriptor model1descriptor = new PredictionXMLParser().
       parseDocument(model1file);
14 ErrorCalculatorDescriptor errorCalculatorDescriptor = new
       ErrorCalculatorXMLParser().parseDocument(errorCalculatorFile)
       ;
15 parser theparser = new tsfparser(datafile);
16 inputData data = theparser.parse();
```

4. *Message* 8: In deze message wordt aangeduid dat er een instantie van de StartGridSelector moet aangemaakt worden. Dit gebeurt in de code door de statische *'getStartDataSelector'* methode van de StartGridSelectorLoader klasse te invokeren, met de StartGridSelectorDescriptor en de lijst van uit te voeren taken als parameters. Over de concrete implementatie van deze loader later meer in sectie 9.4.4

```
1 StartGridSelector startgridselector = StartGridSelectorLoader.
      getStartGridSelector(dataleft,startDataSelectorDescriptor)
```

5. *Message* 9: In deze message wordt aangeduid dat er een instantie van de DataSelector moet aangemaakt worden. Dit gebeurt in de code door de statische *'getDataSelector'* methode van de DataSelectorLoader klasse te invokeren met: de DataSelectorDescriptor, de uit te voeren taken, de reeds uitgevoerde taken, het aantal te selecteren taken en de predictor als parameters. Over de concrete implementatie van deze loader later meer in sectie 9.4.5.

```
1 DataSelector selector = DataSelectorLoader.getDataSelector(
      model1predictor, dataleft, computeddata,
      dataSelectorDescriptor);
```

6. *Message* 10: In deze message wordt aangeduid dat er instanties van de gewenste Predictor moeten aangemaakt worden. Dit gebeurt in twee stappen. In een eerste stap wordt een instantie van de gewenste PredictorFactory aangemaakt. In de tweede stap wordt, gebruik makend van deze factory, de gewenste predictor geïnstantieerd. De eerste stap gebeurt, gebruik makend van de statische

'getFactory' methode uit de PredictorFactoryLoader klasse. Deze methode krijgt dan als parameters de naam van de Factory en zowel een lijst van de reeds uitgevoerde taken als de totale lijst van taken. Over de concrete implementatie van deze loader later meer in sectie 9.4.6. In de tweede stap zullen de predictors gecreëerd worden, gebruik makend van de abstracte *'getPredictor'* methode uit de aangemaakte factories. Deze methode krijgt enkel de predictordescriptor mee als parameter. Over de concrete invulling van deze abstracte methode en de uitwerking van de factories later meer in sectie 9.4.6.

```
1 PredictorFactory model1factory = PredictorFactoryLoader.
    getFactory(model1descriptor.getFactoryName(),computeddata,
    data);
2 Predictor model1predictor = model1factory.getPredictor(
    model1descriptor);
```

7. *Message* 11: In deze message wordt aangeduid dat er een instantie van de Errorcaluclator moet aangemaakt worden. Dit gebeurt in de code door de statische *'getErrorCalculator'* methode van de ErrorCalculatorLoader te invokeren met de ErrorCalculatorDescriptor als parameter. Over de concrete implementatie van deze loader later meer in sectie 9.4.7

```
1 ErrorCalculator calculator = ErrorCalculatorLoader.
    getErrorCalculator(errorCalculatorDescriptor);
```

8. *Messages* 12, 13, 14, 15 *en* 16: In deze stap worden de elementen uit de startgrid geselecteerd die als eerste moeten uitgevoerd worden. Vervolgens worden de runtimes van deze taken achterhaald. In een eerste stap om dit te verwezenlijken, wordt de `getStartLines` methode van de StartGridSelector gebruikt. Deze methode vraagt de lijst van indexen op die refereren naar de taken uit de takenlijst, waarvan de runtimes als eerste achterhaald moeten worden. Het sequence diagram schrijft voor dat als volgende stap de runtimes moeten opgevraagd worden uit de runtime data (een datafile). De huidige implementatie brengt echter met zich mee dat, bij het inlezen van de verschillende jobs, tevens de runtime wordt ingelezen. Dit heeft tot gevolg dat deze runtime rechtstreeks beschikbaar is in de lijst van jobs en via de verworven indexen kan opgevraagd worden. De runtimes worden dus eenvoudig verkregen door de desbetreffende jobs uit de lijst te halen met de `popPoints` methode.

```
1 int[] startgridindexes = startgridselector.getStartLines();
2 LinkedList<LinkedList<Double>> startgridpoints = dataleft.
    popPoints(startgridindexes);
```

9. *Messages* 17 *en* 18: De opdracht van deze messages is het aanpassen van het model door toevoeging van de extra gekende datapunten. In dit con-

crete geval de startgrid. Dit gebeurt in de code niet met een update functie, omdat bepaalde predictiemechanismen dit niet ondersteunen, maar wordt geïmplementeerd door telkens een nieuwe instantie van de predictors aan te maken. Dit gebruik makend van de geüpdatet dataset. In essentie komt het erop neer dat de code, die terug te vinden is bij message 10, hier eigenlijk uitgevoerd wordt. Dit omdat de eerste mogelijkheid de modellen te creëeren slechts zal ontstaan na de update van de data met de startgrid. De code hiervan is bijgevolg terug te vinden bij message 10.

10. *De Loop*: In deze fase van het sequence diagram start de loop. Deze heeft tot doel iteratief modellen te verbeteren door selectief runtimes aan de gekende data toe te voegen. Dit houdt in dat: deze punten bepaald moeten worden, de runtimes achterhaald dienen te worden en nieuwe modellen gecreëerd moeten worden. Deze stappen zijn terug te vinden in de onderstaande messages. De lus zal doorgaan tot de runtimes van de volledige lijst van jobs bekend zijn.

```
1 while(dataleft.size() > 0)
2 {
3
4 ...
5
6
7 }
```

11. *Messages* 19, 20, 23 *en* 24: Deze messages gaan, net zoals message 12, 13 en 14, op zoek naar de indexen van de jobs die moeten toegevoegd worden aan de gekende data. De selectie hier zal echter gebeuren op basis van een inschatting van de fout van het model, dit in tegenstelling tot de indexen van messages 12, 13 en 14 die bepaald worden door de vorm van de startgrid. De bepaling van deze indexen gebeurt dan ook gebruik makend van de DataSelector die in meer detail wordt besproken in sectie 9.4.5. Aangezien, zoals in messages 17 en 18 werd besproken, de instanties van de modellen steeds vernieuwd en niet geüpdatet worden en aangezien de constructie van de `DataSelector` aan de hand van deze modellen zal gebeuren, zal ook de `DataSelector` component telkens opnieuw geïnstantieerd worden. Het spreekt dan ook voor zich dat deze instantiëring uit message 9 pas hier zal gebeuren. Voor de duidelijkheid is dit onderdeel van de implementatie eveneens opgenomen in het onderstaande stuk code.

```
1 DataSelector selector = DataSelectorLoader.getDataSelector(
      model1predictor, dataleft, computeddata,
      dataSelectorDescriptor);
2 int[] indexes = selector.getDataIndexes();
```

12. *Messages* 25 *en* 26: Het vergaren van de runtime informatie wordt analoog aan de code voor messages 15 en 16 uit de reeds ingelezen jobs gehaald. Hier hoeft bijgevolg geen verdere uitleg.

13. *Messages* 27 *en* 28: Het berekenen en het vergaren van de fout van het model gebeurt door de errorcaluculator die werd geïnstantieerd in het stukje code van message 11. De gewenste informatie hangt af van de concrete implementatie van de errorcalculator en wordt door de `getError` methode teruggegeven onder de vorm van een `Double[]`. Het spreekt voor zich dat deze methode het huidige model en de data (lijst van jobs) als parameters meekrijgt. De preciese werking hiervan wordt later besproken (zie sectie 9.4.7)

```
1 Double[] errordata = calculator.getError(model1predictor, data);
```

14. *Message* 29: De runtime doorgeven van de foutinformatie aan de gebruiker gebeurt door middel van output op de terminal. In essentie komt het erop neer dat al de info, die vervat zit in de bovenstaande errordata array, wordt afgeprint. De opbouw van deze informatie wordt verder besproken in sectie 13. De eenvoud van deze materie maakt de weergave van het desbetreffende stukje code irrelevant.

15. *Messages* 30 *en* 31: Het updaten van de predictors gebeurt op identiek dezelfde wijze als de code uit messages 17 en 18 en vereist bijgevolg geen verdere toelichting.

16. *Message* 32: Het teruggeven van de foutinformatie aan de gebruiker gebeurt aan de hand van de filewriter. De info, die via deze filewriter wordt weggeschreven, wordt vergaard doorheen het gehele hierboven beschreven proces. Het principe berust op een twee-dimensionale gelinkte lijst van strings. Doorheen het proces wordt aan deze lijst informatie toegevoegd die aan het einde van het proces door een `filewriter` object wordt weggeschreven in een resultfile. De exacte werking van deze filewriter wordt verder besproken in sectie 9.4.9. De code ziet er dan ongeveer als volgt uit:

```
1  ...
2
3  LinkedList<LinkedList<String>> printlist = new LinkedList<
       LinkedList<String>>();
4
5  ...
6
7  LinkedList<String> debugOutputList = new LinkedList<String>();
8  errordata = calculator.getError(model1predictor, data);
9  for(int i=0;i<errordata.length;i++){
10   debugOutputList.add(errordata[i].toString());
11 }
12 printlist.add(debugOutputList);
13
14 ...
```

```
15
16 filewriter writer = new filewriter(ResultFile);
17 try
18 {
19   writer.write(printlist);
20 }
21 catch (IOException e)
22 {
23   e.printStackTrace();
24 }
25
26 ...
```

In de hierop volgende vier hoofdstukken zal de implementatie van elk van de vier basiscomponenten (Startgridselector, DataSelector, Predictor en de Kwaliteitscontrole) verder besproken worden. Meer specifiek worden de volgende vier onderdelen uit sectie 9.4.2, die instaan voor de configuratie van een dergelijke component, toegelicht:

- De parser
- De descriptor
- De loader
- De abstract component

Dit zijn de componenten die nodig zijn om van een configuratie file tot een instantie van de gespecificeerde component te komen.

9.4.4 StartGridSelector

In deze sectie wordt de `StartGridSelector`, ook wel `initial data selector` genoemd, verder besproken. Het concrete doel van deze component is een aantal taken selecteren uit de totale batch alvorens er enige predictie kan gebeuren. Dit om van deze taken als eerste de runtime informatie ter beschikking te hebben. Zoals reeds eerder werd aangehaald, is het in dit pluggable systeem mogelijk verschillende implementaties te maken van elke component. Deze sectie zal daarom ook enkel de structuur van de component binnen het framework en de manier waarop een startgridselector kan geïmplementeerd worden, bespreken. Meer informatie omtrent dit topic is terug te vinden in hoofdstuk 11.
Zoals reeds is aangehaald, berust de configuratie en het gebruik van een implementatie van deze basiscomponent op vier onderdelen. In wat nu volgt, wordt de implementatie van elk van deze onderdelen toegelicht.

StartGridSelector Parser

Om de plugability te ondersteunen, dient, zoals reeds besproken, voor elke implementatie van een component, een configuratieobject gespecificeerd te worden waarin het path van de codebase, de gebruikte klasse en de nodige specificatieparameters

vermeld staan. De Parser heeft als doel dit object (in de meeste gevallen een configuratiefile) om te zetten naar een java object: de `StartGridSelectorDescriptor`. De huidige implementatie die hiervoor gebruikt wordt, is de `StartGridSelectorXmlParser`.

```
1  package be.ac.ua.comp.GIPSy.Parsers;
2
3  import ...;
4
5
6  public class StartGridSelectorXmlParser extends DefaultHandler{
7    private String Path = "";
8    private String startDataSelectorName ="";
9    private Map paramList = new HashMap();
10
11   public void startElement(String uri, String localName, String qName,
         Attributes attributes) throws SAXException {
12     ...
13   }
14
15   public void endElement(String uri, String localName,String qName)
       throws SAXException {
16     ...
17   }
18
19   public void characters(char[] ch, int start, int length) throws
       SAXException
20   {
21     tmpval = new String(ch,start,length);
22   }
23
24   public StartGridSelectorDescriptor parseDocument(String filepath) {
25
26     SAXParserFactory spf = SAXParserFactory.newInstance();
27     try {
28
29       SAXParser sp = spf.newSAXParser();
30       sp.parse(filepath, this);
31
32     }
33     catch(Exception e){
34       e.printStackTrace();
35     }
36     return new StartGridSelectorDescriptor(startDataSelectorName,
           paramList,Path);
37   }
38 }
```

Concreet bestaat deze klasse uit de implementatie van de abstracte `startElement`, `endElement` en `characters` methoden die onder de vorm van een finite state machine de XML configuratiebestanden parsen. Tevens worden de gegevens bijgehouden onder de vorm van het `Path` van de codebase, van de naam van de gebruikte klasse (`startDataSelectorName`) en van de `paramList` map die de configuratieparameters bevat. Tot slot omvat het de `pareseDocument(String filepath)` methode

die gebruik makend van een XML parser en van de bovenstaande methodes het gewenste configuratieobject returnt: de `StartGridSelectrorDescriptor`.

StartGridSelectorDescriptor

De `StartGridSelectrorDescriptor` is een Java klasse die het Path van de codebase, de naam van de gebruikte klasse en de configuratieparameters stockeert ter instantiatie en ter initialisatie van de gespecificeerde StartGridSelector. Het object ziet er als volgt uit:

```
1  package be.ac.ua.comp.GIPSy.StartGridSelector;
2
3  import ...;
4
5  public class StartGridSelectorDescriptor implements Serializable
6  {
7    private String Path = "";
8    private String startGridSelectorName = "";
9    private Map parameterList = null;
10
11   public StartGridSelectorDescriptor(String startGridSelectorName Map
         parameterList, String path){
12     startGridSelectorName = dataSelectorName;
13     this.parameterList = parameterList;
14     this.Path=path;
15   }
16
17   public String getDataSelectorName(){
18     return dataSelectorName;
19   }
20
21   public Map getParameterList(){
22     return parameterList;
23   }
24
25   public URI[] getPath() throws URISyntaxException {
26     return new URI[]{new URI(this.Path)};
27   }
28 }
```

Concreet komt het er dus op neer dat deze klasse bestaat uit drie private velden en drie publieke methoden om de inhoud van deze velden op te vragen. Het `Path` veld omvat het path van de codebase, het `startDataSelectorName` veld de naam van de gebruikte klasse en het `parameterList` veld de Map die de naam van elke parameter mapt op de concrete inhoud. Deze inhoud kan variëren van simpele basistypes tot gelinkte lijsten enz.

StartGridSelectorLoader

De `StartGridSelectorLoader` zorgt ervoor dat, aan de hand van een statische functie, de gewenste component geladen en geïnitialiseerd wordt. Hierbij wordt rekening gehouden met het path van de codebase, met de naam van de `StartGridSelector` en met de initialisatieparameters gespecificeerd in de `StartGridSelectorDescriptor`.

```
1  package be.ac.ua.comp.GIPSy.StartGridSelector;
2
3  import ... ;
4
5  public class StartGridSelectorLoader {
6
7    public static StartGridSelector getStartGridSelector(inputData data,
       StartGridSelectorDescriptor descriptor) throws
       InstantiationException, IllegalAccessException,
       ClassNotFoundException, URISyntaxException{
8      ClassLoader loader = new URIClassLoader(descriptor.getPath(),
         Thread.currentThread().getContextClassLoader());
9      Thread.currentThread().setContextClassLoader(loader);
10     StartGridSelector returnvalue = (StartGridSelector)loader.
         loadClass(descriptor.getDataSelectorName()).newInstance();
11     returnvalue.init(data, descriptor.getParameterList());
12     return returnvalue;
13   }
14
15 }
```

De `URIClassLoader` zal er voor zorgen dat de JAR file, gespecificeerd in het configuratiebestand, als codebase zal gebruikt worden.

Abstracte StartGridSelector Klasse

Naast de omzetting van een configuratieobject tot een concrete instantie van een basiscomponent dient er vanzelfsprekend ook een implementatie van elke nieuwe component gemaakt te worden. Zoals reeds kon afgeleid worden uit figuur 9.10, worden de implementaties van de basiscomponenten steeds afgeleid van een basis klasse. In dit geval de abstracte `StartGridSelector` klasse. Deze ziet er als volgt uit:

```
1  package be.ac.ua.comp.GIPSy.StartGridSelector;
2
3  import java.io.Serializable;
4  import java.util.Map;
5  import be.ac.ua.comp.GIPSy.Parsers.inputData;
6
7  public abstract class StartGridSelector implements Serializable
8  {
9    protected inputData data = null;
10   protected Map paramlist = null;
11
12   public abstract int[] getStartLines();
13
14   public void init(inputData data, Map paramlist)
15   {
16     this.data  = data;
17     this.paramlist = paramlist;
18   }
19 }
```

Deze klasse bevat twee methoden:

- `public absract int[] getStartLines()` : Deze abstracte methode dient geïmplementeerd te worden door de developer en heeft tot doel de lijst van indexen terug te geven die de volgorde bepalen waarin de lijst van jobs/samplepoints moet uitgevoerd worden.
- `public void init(inputData data, Map paramlist)` : Deze methode moet het voor de gebruiker mogelijk maken de informatie, die in de door hem opgestelde configfile omvat zit, op de juiste manier aan te wenden om aldus de component klaar te maken voor gebruik. In essentie komt het erop neer dat de parameters uit de `parameterList` map van de `StartGridSelectrorDescriptor` gebruikt worden om de juiste initialisatie procedure te doorlopen. Deze methode moet bijgevolg ook door de developer geherimplementeerd (overloaded) worden. Tot slot is het belangrijk op te merken dat de bestaande implementatie, in deze geherimplementeerde methode, bij aanvang van de methode, aangeroepen moet worden. Dit om de basisinitialisatie, die voor elke `StartGridSelector` vereist is, steeds uit te voeren.

Besluit

De implementatie van een nieuwe startgridselector component omvat dus enkel de uitbreiding van de abstracte `StartGridSelector` klasse en het opstellen van het configuratieobject dat de initialisatie van deze implementatie ondersteunt. Enkele voorbeelden hiervan zijn terug te vinden in hoofdstuk 11.

9.4.5 DataSelector

In deze sectie wordt de `DataSelector`, ook wel de `SampleSelector` genoemd, verder besproken. Het concrete doel van de `DataSelector` is, na de opbouw van een model, op zoek te gaan naar de datapunten die niet in het huidige model gebruikt worden en die de toekomstigge modellen het best zouden beïnvloeden indien ze wel gebruikt zouden worden. Om dit te verwezenlijken, kunnen verschillende technieken gebruikt worden. Er kan bijvoorbeeld gekeken worden naar de density van de samples in een bepaalde omgeving[3] of naar de fout in elk sample punt of naar nog andere factoren. Net zoals bij de component uit de vorige sectie is het, omwille van de plugability eigenschappen van het systeem, mogelijk verschillende implementaties van deze component te maken. In deze sectie is er dan ook voor gekozen de structuur van de component binnen het framework en de manier waarop een DataSelector kan geïmplementeerd worden, te bespreken. Voor meer informatie over de concrete implementaties ist terug te vinden in hoofdstuk 12.

Net zoals in het vorige hoofdstuk reeds werd aangehaald, berust de configuratie en het gebruik van een implemenatie van een basiscomponent op vier onderdelen. Ook in deze sectie zal de bespreking gebaseerd zijn op de implementatie van deze vier onderdelen:

[3]Het aantal samples per oppervlakte-eenheid.

DataSelector Parser

Ter ondersteuning van de plugability dient ook bij deze component een configuratieobject gespecificeerd te worden. Dit object omvat het path van de codebase, de naam van de dataselector klasse en de nodige specificatieparameters. Het omzetten van deze configuratie informatie naar een Java Object(`DataselectorDescriptor`) gebeurt aan de hand van de DataSelector Parser. De huidige implementatie van deze Parser heeft de naam `DataSelectorXmlParser` gekregen en ziet er als volgt uit:

```
1  package be.ac.ua.comp.GIPSy.Parsers;
2
3  import ...;
4
5  public class DataSelectorXmlParser extends DefaultHandler{
6    private String dataSelectorName ="";
7    private Map paramList = new HashMap();
8    private String Path;
9
10   public void startElement(String uri, String localName, String qName,
         Attributes attributes) throws SAXException {
11     ...
12   }
13
14   public void endElement(String uri, String localName,String qName)
         throws SAXException {
15     ...
16   }
17
18   public void characters(char[] ch, int start, int length) throws
         SAXException {
19     tmpval = new String(ch,start,length);
20   }
21
22   public DataSelectorDescriptor parseDocument(String filepath) {
23
24     SAXParserFactory spf = SAXParserFactory.newInstance();
25     try {
26       SAXParser sp = spf.newSAXParser();
27       sp.parse(filepath, this);
28
29     }
30     catch(Exception e) {
31       e.printStackTrace();
32     }
33     return new DataSelectorDescriptor(dataSelectorName,paramList,Path)
           ;
34   }
35 }
```

In essentie is deze klasse opgebouwd uit 1) de implementatie van drie abstracte methoden (`startElement, endElement en characters`) die ertoe dienen om onder de vorm van een finite state machine de XML parser te ondersteunen, 2) drie globale variabelen (`Path, dataSelectorName` en `parameterList`) die de configuratie

gegevens omvatten en 3) de `parseDocument(String filepath)` methode die, gebruik makend van de bovenstaande methodes en van de XML parser die op deze methoden gebaseerd is, de configuratiegegevens omzet naar een Java object: `DataSelectorDescriptor`.

DataSelectorDescriptor

Deze configuratieklasse omvat het path van de codebase, de naam van de gebruikte klasse en de configuratieparameters ter instantiatie en initialisatie van de gespecificeerde DataSelector of SampleSelector. Het Java object ziet er als volgt uit:

```
1  package be.ac.ua.comp.GIPSy.DataSelectors;
2
3  import ... ;
4
5  public class DataSelectorDescriptor implements Serializable{
6
7    private String dataSelectorName = "";
8    private String Path = "";
9    private Map parameterList = null;
10
11   public DataSelectorDescriptor(String dataSelectorName, Map
         parameterList, String path){
12     this.dataSelectorName = dataSelectorName;
13     this.parameterList = parameterList;
14     this.Path=path;
15   }
16
17   public String getDataSelectorName(){
18     return dataSelectorName;
19   }
20
21   public Map getParameterList(){
22     return parameterList;
23   }
24
25   public URI[] getPath() throws URISyntaxException {
26     return new URI[]{new URI(this.Path)};
27   }
28 }
```

In essentie bestaat deze klasse uit drie private velden en drie publieke methoden om de inhoud van deze velden te raadplegen. In het `Path` veld is het path van de codebase terug te vinden, in het `dataSelectorName` veld de naam van de klasse die de dataSelector implementeert en in het `parameterList` veld de map die de naam van elke configuratieparameter mapt op de concrete inhoud van deze parameter. Deze waarden kunnen variëren van simpele basistypes tot uitgebreide gelinkte lijsten of lijsten van lijsten enz.

DataSelectorLoader

De `DataSelectorLoader` laadt en initialiseert, aan de hand van de statische `getDataSelector` functie, een instantie van de gewenste DataSelector implementatie. Dit

gebeurt op basis van de naam van de implementatie, van het path van de codebase en van de overige parameters die gespecificeerd zijn in de `DataSelectorDescriptor`.

```
1  package be.ac.ua.comp.GIPSy.DataSelectors;
2
3  import ...;
4
5  public class DataSelectorLoader {
6    public static DataSelector getDataSelector(Predictor predictor1,
         inputData data, inputData computeddata, inputData totaldata, int
          selectionamount ,DataSelectorDescriptor descriptor) throws
         InstantiationException, IllegalAccessException,
         ClassNotFoundException, URISyntaxException{
7      URI[] uris = descriptor.getPath();
8      ClassLoader loader = new URIClassLoader(descriptor.getPath(),
           Thread.currentThread().getContextClassLoader());
9      Thread.currentThread().setContextClassLoader(loader);
10     DataSelector returnvalue = (DataSelector)loader.loadClass(
           descriptor.getDataSelectorName()).newInstance();
11     returnvalue.init(predictor1, data, computeddata, totaldata,
           selectionamount, descriptor.getParameterList());
12     return returnvalue;
13   }
14 }
```

De `URIClassLoader` zorgt ervoor dat de in de configuratie gespecificeerde JAR file als codebase gebruikt wordt.

Abstracte DataSelector Klasse

Naast de omzetting van een configuratieobject tot een concrete instantie van een basiscomponent dient er vanzelfsprekend ook een implementatie van elke nieuwe component gemaakt te worden. Zoals reeds kon afgeleid worden uit figuur 9.10, worden de implementaties van de basiscomponenten steeds afgeleid van een basis klasse. In dit geval de abstracte `DataSelector` klasse. Deze ziet er als volgt uit:

```
1  package be.ac.ua.comp.GIPSy.DataSelectors;
2
3  import ...;
4
5  public class DataSelector implements Serializable{
6
7    protected double[] weights = null;
8    protected Predictor predictor1=null;
9    protected inputData idata=null;
10   protected inputData icomputeddata=null;
11   protected inputData totaldata=null;
12   protected double[][] data=null;
13   protected double[][] computeddata=null;
14   protected Map paramlist = null;
15   protected int selectionamount = -1;
16
17   public double[] getWeights(){
```

```
18      return weights;
19    }
20
21    public double[][] getDataWeights(){
22      //returns samplepoints with their weight
23
24    }
25
26    public void init(Predictor predictor1, inputData idata, inputData
          icomputeddata,inputData totaldata, int selectionamount, Map
          paramlist){
27      //initialisation of standard members
28    }
29
30    public int getDataIndex(){
31      // returns index of samplepoint to be calculated
32    }
33
34    public int[] getDataIndexes(){
35      // returns alist of indexes in the orther to be executed
36    }
37
38    public int[] getAllDataIndexes(){
39      // returns alist of all indexes in the orther to be executed
40    }
41  }
```

De methodiek die achter deze sampleselectie zit, is gebaseerd op gewichten. Elke sample zal een bepaald gewicht krijgen dat weerspiegelt in welke mate het van belang is dat de runtime informatie van die sample gekend is om aldus een beter model te genereren. Deze gewichten zijn terug te vinden in de protected `weights` en kunnen opgevraagd worden via:

- de public `getWeights` methode: Deze methode geeft een array van gewichten terug. Het array-element op plaats i geeft het gewicht van de job met index i in de joblijst weer.
- de public `getDataWeights` methode: Deze methode geeft een array van jobs terug. Een job komt hier overeen met een array waarin in de eerste elementen, de parameters die de desbetreffende job definiëren, terug te vinden zijn en in het laatste array-element het gewicht van de job geplaatst wordt.

De doelstelling van elke implementatie van deze component is deze gewichten te bepalen. De overige functionaliteit zit vervat in de basisklasse, meer bepaald in de `getDataIndex`, de `getDataIndexes` en de `getAllDataIndexes` methodes. Deze methoden geven de volgende gegevens terug:

- `getDataIndex` : Het punt waarvan de kennis omtrent de runtime het model vermoedelijk het meest zal verbeteren.
- `getDataIndexes` : Een lijst van punten waarvan de kennis over de runtime het model vermoedelijk het meest zal verbeteren.

- `getAllDataIndexes` : De lijst van alle punten in volgorde zodat de kennis van de runtime van het eerste punt het model vermoedelijk het meest zal verbeteren en deze van het laatste punt het model vermoedelijk het minst zal verbeteren.

Het toekennen van de juiste gewichten aan elke job gebeurt in de `init` methode. Net zoals bij de overige componenten is het noodzakelijk dat bij een implementatie van de `DataSelector` component de `init` functie wordt geherimplementeerd. Het is de bedoeling dat de reeds bestaande `init` methode als eerste in deze herimplementatie wordt aangeroepen en dat de overige code ervoor zorgt dat elke job het juiste gewicht toegekend krijgt.

Besluit

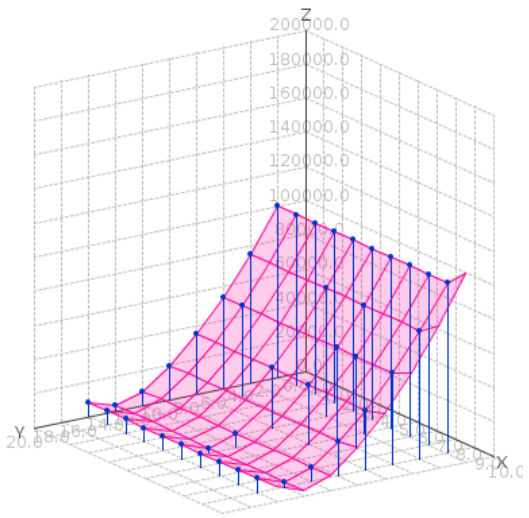
De implementatie van een nieuwe Dataselector kan dus opgevat worden als het opstellen van een configuratiefile die de nodige informatie bevat en het overloaden van de bestaande `DataSelector` klasse. Meer bepaald zal de `init` methode zo moeten herschreven worden dat elke job zijn gewenste gewicht krijgt. De overige functionaliteit, zoals bijvoorbeeld het selecteren van de indexen aan de hand van deze gewichten, zit reeds vervat in de bestaande `DataSelector` basisklasse. Enkele voorbeelden van implementaties van de `DataSelector` zijn terug te vinden in hoofdstuk 12.

9.4.6 Predictors

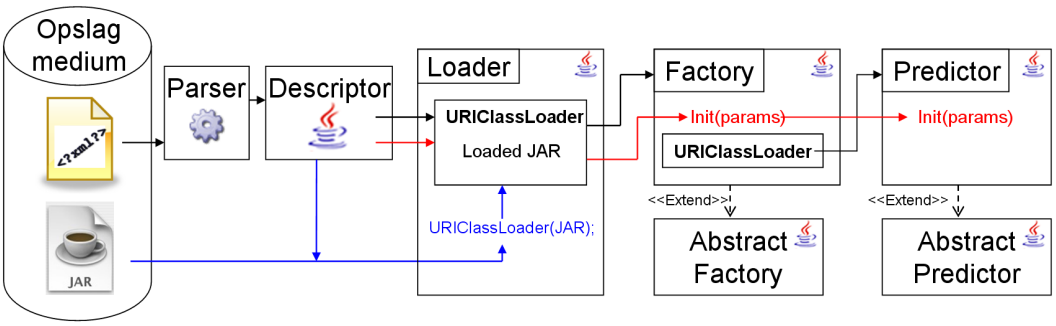
Deze sectie heeft tot doel de `Predictors`, ook wel `Models` genoemd, in meer detail te bespreken. Het opzet van deze objecten is, aan de hand van een set van datapunten of samplepoints (jobs) met gekende runtimes, een model te bouwen van de runtimes van de overige samplepunten. Het spreekt voor zich dat het de bedoeling is de effectieve runtimes van deze overige samplepunten zo nauwkeurig mogelijk te benaderen. Een voorbeeld van een dergelijk model is terug te vinden in figuur 9.11.

In tegenstelling tot de vorige twee basiscomponenten zal de opbouw van deze component lichtjes afwijken van de structuur uit figuur 9.10. Hier zal immers naast de abstracte predictiecomponent nog gewerkt worden met een tussencomponent, namelijk de predictiefactory. Hiervan zal eveneens een abstracte basisversie bestaan die door de developer geïmplementeerd moet worden. Het opzet van deze component bestaat erin groepen van gelijkaardige predicties te groeperen en er, op deze manier, gemeenschappelijke operaties op uit te voeren. Deze structuur wordt mooi geïllustreerd in figuur 9.12. De loader zal in dit geval, aan de hand van de `URIClassLoader`, een factory implementatie, die afgeleid is van de Abstracte Factory klasse, instantiëren. Deze factory zal dan op zich, aan de hand van de URIClassloader, een instantie aanmaken van een Predictorimplementatie die afgeleid is van de abstracte `Predictor` klasse.

Net zoals bij de vorige componenten is het, omwille van de plugability eigenschappen van het systeem, mogelijk verschillende implementaties van de component te maken. In dit geval gaat dit zelfs nog verder. Het is immers mogelijk zowel meerdere implementaties van de PredictorFactory als van de Predictor te voorzien.



Figuur 9.11: Een (Kriging) model opgebouwd aan de hand van de blauwe samplepunten



Figuur 9.12: Opeenvolgende stappen tijdens de configuratie van een predictor. (Let op de verschillen met figuur 9.10).

Omwille van deze oneindige uitbreidbaarheidsmogelijkheden is er ook in deze sectie voor gekozen de structuur van de component binnen het framework en de manier waarop een PredictorFactory en een Predictor kunnen geïmplementeerd worden, te bespreken. Meer informatie omtrent de concrete implementatie van deze twee abstracte klassen staat is terug te vinden in hoofdstuk 10.

PredictorParser

De plugability wordt ook, in dit geval, ondersteund aan de hand van een configuratieobject. Dit object bestaat in essentie uit het path van de codebase, de naam van de predictorfactory klasse, de naam van de predictor klasse en de noodzakelijke specificatieparameters. De `PredictorParser` staat in voor de omzetting van dit configuratieobject naar een Java Object namelijk de `PredictorDescriptor`. De

huidige implementatie van deze parser heeft de naam `PredictionXMLParser` gekregen en ziet er als volgt uit:

```
1  package be.ac.ua.comp.GIPSy.Parsers;
2
3  import ...;
4
5  public class PredictionXMLParser extends DefaultHandler {
6    private String factoryName = "";
7    private String predictorName ="";
8    private Map paramList = new HashMap();
9    private String Path;
10
11
12   public void startElement(String uri, String localName, String qName,
         Attributes attributes) throws SAXException {
13     ...
14   }
15
16   public void endElement(String uri, String localName,String qName)
       throws SAXException {
17     ...
18   }
19
20   public void characters(char[] ch, int start, int length) throws
       SAXException {
21     tmpval = new String(ch,start,length);
22
23   }
24
25   public PredictorDescriptor parseDocument(String filepath){
26
27     SAXParserFactory spf = SAXParserFactory.newInstance();
28     try {
29
30       //get a new instance of parser
31       SAXParser sp = spf.newSAXParser();
32
33       //parse the file and also register this class for call backs
34       sp.parse(filepath, this);
35
36     }catch(Exception se) {
37       e.printStackTrace();
38     }
39     return new PredictorDescriptor(factoryName,predictorName,paramList
         ,Path);
40   }
41 }
```

Concreet bestaat deze klasse uit de implementatie van de abstracte `startElement`, `endElement` en `characters` methoden die onder de vorm van een finite state machine de XML configuratiebestanden parsen. De gegevens worden bijgehouden in vier datamembers: 1) de `Path` variabele die het path van de codebase bevat, 2) de `factoryName` variabele die de naam van de implementatie van de factory klasse

bevat, 3) de `predictorName` variabele die de naam van de implementatie van de predictor klasse bevat en 4) de `paramList` variabele die de noodzakelijke configuratieparameters bevat. De concrete omzetting van een configuratieobject naar het `PredictorDescriptor` Java object gebeurt aan de hand van de `parseDocument(String filepath)` methode.

PredictorDescriptor

Zoals reeds werd aangehaald, zal het `PredictorDescriptor` object de klasse zijn waarbinnen de noodzakelijke configuratieinformatie, zoals het path van de codebase, de naam van de predictorfactory implementatie, de naam van de predictor implementatie en de verschillende configuratieparameters, vervat zitten. Dit ter instantiatie en ter initialisatie van de gespecificeerde PredictorFactory en Predictor. Het object ziet er als volgt uit:

```
1  package be.ac.ua.comp.GIPSy.Predictors;
2
3  import ...;
4
5  public class PredictorDescriptor implements Serializable{
6    private String factoryName = "";
7    private String functionName = "";
8    private Map parameterList = null;
9    private String Path = "";
10
11
12   public PredictorDescriptor(String factoryName, String functionName,
         Map parameterList, String path){
13     this.factoryName = factoryName;
14     this.functionName = functionName;
15     this.parameterList = parameterList;
16     this.Path= path;
17   }
18
19   public String getFactoryName(){
20     return factoryName;
21   }
22
23   public String getFunctionName(){
24     return functionName;
25   }
26
27   public Map getParameterList(){
28     return parameterList;
29   }
30
31   public URI[] getPath() throws URISyntaxException {
32     return new URI[]{new URI(this.Path)};
33   }
34 }
```

Concreet komt het er dus op neer dat deze klasse bestaat uit vier private velden en vier publieke methoden om de inhoud van deze velden op te vragen. Het `Path`

omvat het path van de codebase, de `factoryName` bevat de naam van de factory implementatie, de `functionName` bevat de naam van de predictorimplementatie en het `parameterList` veld bevat de map die de naam van elke parameter mapt op de concrete inhoud van de parameter. Deze inhoud kan variëren van simpele basistypes tot uitgebreide gelinkte lijsten of lijsten van lijsten enz.

PredictorFactoryLoader

De statische `getFactory` methode in de `PredictorFactoryLoader` klasse zorgt ervoor dat de gewenste implementatie van de PredictorFactory wordt geladen en geïnitialiseerd. Dit gebeurt aan de hand van het path van de codebase, de naam van de predictorfactory implementatie en de extra initialisatieparameters die terug te vinden zijn in de `predictorDescriptor` klasse. De `PredictorFactoryLoader` klasse ziet er als volgt uit:

```
1  package be.ac.ua.comp.GIPSy.Predictors;
2
3  import ...;
4
5  public class PredictorFactoryLoader {
6
7    public static PredictorFactory getFactory(PredictorDescriptor
         descriptor, inputData data, inputData totaldata) throws
         InstantiationException, IllegalAccessException,
         ClassNotFoundException, URISyntaxException{
8      URI[] uris = descriptor.getPath();
9      ClassLoader loader = new URIClassLoader(descriptor.getPath(),
           Thread.currentThread().getContextClassLoader());
10     Thread.currentThread().setContextClassLoader(loader);
11     PredictorFactory returnvalue= (PredictorFactory)loader.loadClass(
           descriptor.getFactoryName()).newInstance();
12     returnvalue.Initialise(data,totaldata);
13     return returnvalue;
14   }
15 }
```

De `URIClassLoader` zorgt ervoor dat de, in de configuratie gespecificeerde JAR file, als codebase gebruikt wordt.

Abstracte PredictiorFactory Klasse

Zoals reeds besproken, zullen er in het geval van de Predictors twee abstracte klassen geïmplementeerd moeten worden. De eerste is de `PredictorFactory` klasse. Het opzet van deze tussenklasse, die niet terug te vinden is in de overige basiscomponenten, is tweeledig:

- De predicties die gebeuren in dit framework kunnen, zoals later zal blijken, opgesplitst worden in verschillende typen (vb. Polynomen, Kriging, Radiale Basisfuncties, ...). Deze verschillende typen omvatten echter wel verschillende implementaties van predictors maar delen toch tal van functionaliteit. Deze functionaliteit is op deze manier eenvoudig te centraliseren in de factory klasse zodat codeduplicatie vermeden wordt.

- Naast de centralisatie van code is er tevens nood aan een mechanisme om functies toe te passen op predictors. Er wordt hier bijvoorbeeld gedacht aan het vergelijken van twee predictors, het berekenen van de fout, het nemen van een slice uit een meerdimensionaal model om zo 3D plotting mogelijk te maken, ...

Aangezien de volledige implementatie van de `PredictorFactory` klasse te ver zou leiden, wordt hier een vereenvoudigde versie gegeven. In essentie komt het erop neer dat enkele minder relevante methoden voor de eenvoud zijn weggelaten.

```
1  package be.ac.ua.comp.GIPSy.Predictors;
2
3  import ...;
4
5  public abstract class PredictorFactory implements Serializable
6  {
7    public inputData data;
8    public inputData totaldata;
9    public PredictorFactory(){};
10
11   public PredictorFactory(inputData data){
12     this.data = data;
13   }
14
15   public abstract Predictor getPredictor(PredictorDescriptor
         descriptor);
16
17   public final Predictor getNormalised2DPredictor(Predictor
         xDPredictor, int x, int y, double[] data){
18     return  new SubPredictor(xDPredictor,x,y,data);
19   }
20
21   public final Predictor get2DPredictor(Predictor xDPredictor, int x,
         int y, double[] data) {
22     return  new SubPredictor(xDPredictor,x,y,data);
23   }
24
25   public inputData difference(inputData data, Predictor input){
26     // calculates the distance between samplepoints and their
           predicted values
27   }
28
29   public Double[] getError(Predictor input){
30     // returns a collection of error data
31   }
32
33   public int getBiggestDistanceIndex(Predictor input){
34     // returns the index of the samplepoint with the largest
35     // distance to its nearest neighbour
36   }
37
38   public inputData getPredictordata(inputData data,Predictor input){
39     // returns a list of samplepoints with their predicted runtime
40   }
41
```

```
42    public void Initialise(inputData data, inputData totaldata) {
43      this.data = data;
44      this.totaldata = totaldata;
45    }
46  }
```

De klasse bestaat uit tal van methoden waarvan er twee zeer belangrijk zijn. Deze twee methoden leiden tot de initialisatie en de instantiatie van een gewenst model:

- De `initialise` methode: Deze methode zorgt ervoor dat de predictorfactory geïnitialiseerd wordt met de lijst die de reeds gekende samplepunten bevat en met de lijst van al de samplepunten die afgehandeld zijn of moeten worden. Concreet kan er gesteld worden dat de samplespace hier geïnitialiseerd wordt.

- De abstracte `getPredictor` methode: Deze methode zal, aan de hand van de informatie in de predictor en van de eigenschappen van de huidige predictorfactory, een predictor instantiëren en initialiseren. De grote verschillen tussen de verschillende predictors maken dat deze methode voor elke groep van predictors, of anders gezegd, elke implementatie van de predictorfactory zal verschillend zijn. Meer info over deze concrete implementatie is terug te vinden in hoofdstuk 10.

De overige methoden zoals de `get2Dpredictor` of de `getError` methode, dienen ertoe ten dienste te staan van de extra functionaliteit die rond het framework is gebouwd. Zo zal bijvoorbeeld de `get2Dpredictor` ervoor zorgen dat een multidimensionaal model herleid wordt tot een 3D model zodat dit eenvoudig geplot kan worden met de standaard plotmechanismen.

Abstracte Predictor Klasse

Naast de implementatie van de Factory klasse, die ondermeer zal instaan voor de constructie en de initialisatie van de predictors, moet er ook een implementatie van een predictor voorzien worden. Deze implementatie zal steeds afgeleid zijn van de abstracte `Predictor` klasse. Deze klasse ziet er als volgt uit.

```
1  package be.ac.ua.comp.GIPSy.Predictors;
2
3  import ...;
4
5  public abstract class Predictor implements Serializable{
6    public abstract double Evaluate(Matrix value);
7
8  }
```

Deze klasse is zeer eenvoudig en verplicht de developer enkel de abstracte `Evaluate` methode te implementeren. Deze methode berekent, voor een bepaald samplepunt, de voorspelling van de runtime. Het valt onmiddellijk op dat, in tegenstelling tot de overige componenten, er geen initialisatie gebeurt in deze component. Dit is te wijten aan de factory tussenlaag die eerder reeds besproken werd. Aangezien deze

factory ook de verantwoordelijkheid zal zijn van de gebruiker, zal in deze factory, aan de hand van de predictor descriptor, de juiste implementatie van de predictor gespecificeerd, geïnstantieerd en tot slot geïnitialiseerd worden.

Besluit

De implementatie van een nieuwe predictor brengt drie dingen met zich mee. In de eerste plaats moet er een configuratieobject voorzien worden waarin alle informatie, die noodzakelijk is om de predictor op te bouwen, vervat zit. Ten tweede moet er een predictor factory klasse ontwikkeld worden. Deze moet uitgebreid zijn van de abstracte `PredictorFactory` klasse en moet, aan de hand van de configuratieinformatie, in de mogelijkheid zijn de gewenste predictor implementatie te instantieren en te initialiseren. Tot slot dient er een implementatie gemaakt te worden van deze Predictor. Deze dient afgeleid te worden van de abstracte `Predictor` klasse en dient functionaliteit te voorzien die van elk samplepoint de runtime kan voorspellen. Enkele voorbeelden van dergelijke implementaties zijn terug te vinden in hoofdstuk 10

9.4.7 ErrorCalculator

In deze sectie wordt de `Kwaliteitscontrole`, ook wel `Errorcalculator` genoemd, meer in detail besproken. Deze component heeft als doel een beeld te scheppen van de kwaliteit van een bepaald model om aldus beslissingen te nemen over de verdere stappen die moeten ondernomen worden.

Kwaliteit bestaat onder veel vormen en maten en valt moeilijk te definiëren. Toch is er een definitie die een mooi beeld geeft van wat er in dit hoofdstuk met kwaliteit bedoeld wordt:

"De kwaliteit duidt aan of iets of iemand aan zijn doel beantwoordt."

Ter illustratie een klein voorbeeld. Een Ferrari zal kwalitatief heel goed zijn wanneer er zo snel mogelijk met een voertuig naar een bepaalde bestemming moet gereden worden. Moet er echter een fiets mee in de auto, dan zal de Ferrari kwalitatief veel minder scoren. Om deze verschillen in kwaliteitsbepalling binnen het GIPSy framework te ondersteunen, is deze basiscomponent ontwikkeld. De plugability eigenschappen van het systeem zorgen ervoor dat er op een eenvoudige manier verschillende implementaties van dit object binnen het framework gebruikt kunnen worden. Net zoals bij de overige drie basiscomponenten berust de configuratie en het gebruik van een implementatie van deze basiscomponent op vier onderdelen. De uiteenzetting omtrent de implementatie van deze component zal aldus ook aan de hand van deze vier onderdelen gebeuren.

ErrorCalculator Parser

Om de plugability te ondersteunen, dient in het framework van elke plugable component een configuratieobject gespecificeerd te worden. Deze component omvat, in het geval van de ErrorCalculator, het path van de codebase, de naam van de klasse die de ErrorCalculator implementeert en de noodzakelijke configuratieparameters. Om dit configuratieobject (in vele gevallen een configfile) om te zetten naar een

Java Object, dat binnen het framework ondersteund wordt (de `ErrorCalculatorDescriptor`), is de `ErrorCalculatorParser` ontwikkeld. De huidige implementatie heeft de naam `ErrorCalculatorXMLParser` en ziet er als volgt uit:

```
1  package be.ac.ua.comp.GIPSy.Parsers;
2
3  import ...;
4
5  public class ErrorCalculatorXMLParser extends DefaultHandler{
6
7    private String errorCalculatorName ="";
8    private Map paramList = new HashMap();
9    private String Path;
10
11   public void startElement(String uri, String localName, String qName,
         Attributes attributes) throws SAXException {
12     ...
13   }
14
15   public void endElement(String uri, String localName,String qName)
       throws SAXException {
16     ...
17   }
18
19   public void characters(char[] ch, int start, int length) throws
       SAXException {
20     tmpval = new String(ch,start,length);
21   }
22
23   public ErrorCalculatorDescriptor parseDocument(String filepath) {
24     SAXParserFactory spf = SAXParserFactory.newInstance();
25     try {
26       SAXParser sp = spf.newSAXParser();
27       sp.parse(filepath, this);
28
29     }catch(Exception e) {
30       e.printStackTrace();
31     }
32     return new ErrorCalculatorDescriptor(errorCalculatorName,paramList
         ,Path);
33   }
34 }
```

In essentie bestaat de `ErrorCalculatorParser` klasse uit drie onderdelen. Het eerste onderdeel omvat de `errorCalculatorName`, de `paramList` en de `Path` velden die de noodzakelijke configuratieinformatie bevatten. Het tweede onderdeel bestaat uit de implementatie van de abstracte `startElement`, `endElement` en `characters` die, onder de vorm van een finite state machine, de XML configuratiebestanden parsen. Het laatste onderdeel omvat de `parseDocument` methode die er voor instaat dat het configuratieobject, gebruik makend van de XMLparser en van de hierboven aangehaalde onderdelen, omgezet wordt naar de `ErrorCalculatorDescriptor` klasse.

ErrorCalculatorDescriptor

In de vorige sectie werd reeds vermeld dat de informatie uit het configuratieobject werd omgezet naar een instantie van de `ErrorCalculatorDescriptor` klasse. In deze klasse zit de informatie omtrent het path van de codebase, de naam van de klasse die de ErrorCalaculator implementeert en de noodzakelijke configuratieparameters vervat. Dit object is essentieel ter instantiatie en ter initialisatie van de gewenste errorcalculator implementatie en ziet er als volgt uit:

```
1  package be.ac.ua.comp.GIPSy.ErrorCalculator;
2
3  import ...;
4
5  public class ErrorCalculatorDescriptor implements Serializable{
6
7    private String errorCalculatorName = ;
8    private String Path = ;
9    private Map parameterList = null;
10
11   public ErrorCalculatorDescriptor(String errorCalculatorName, Map
         parameterList, String path){
12     this.errorCalculatorName = errorCalculatorName;
13     this.parameterList = parameterList;
14     this.Path =path;
15   }
16
17   public String getErrorCalculatorName(){
18     return errorCalculatorName;
19   }
20
21   public Map getParameterList(){
22     return parameterList;
23   }
24
25   public URI[] getPath() throws URISyntaxException{
26
27     return new URI[]{new URI(this.Path)};
28   }
29 }
```

In essentie bestaat deze klasse uit drie private velden en drie publieke methoden om de inhoud van deze velden te raadplegen. In het `Path` veld is het path van de codebase terug te vinden, in het `errorCalculatorName` veld de naam van de klasse die de errorcalculator implementeert en in het `parameterList` veld de map die de naam van elke configuratieparameter mapt op de waarde van deze parameter. Deze waarden kunnen variëren van simpele basistypes tot uitgebreide gelinkte lijsten of lijsten van lijsten enz.

ErrorCalculatorLoader

De `ErrorCalculatorLoader` laadt en initialiseert, aan de hand van de statische `getErrorCalculator` functie, een instantie van de gewenste `ErrorCalculator` implementatie. Dit gebeurt op basis van de naam van de klasse die de `ErrorCalculator`

implementeert, van het path van de codebase en van de noodzakelijke configuratieinformatie die vervat zit in het `ErrorCalculatorDescriptor` object. De loader ziet er als volgt uit:

```
1  package be.ac.ua.comp.GIPSy.ErrorCalculator;
2
3  import ...;
4
5  public class ErrorCalculatorLoader {
6
7    public static ErrorCalculator getErrorCalculator(
         ErrorCalculatorDescriptor descriptor) throws
         InstantiationException, IllegalAccessException,
         ClassNotFoundException, URISyntaxException{
8      URI[] uris = descriptor.getPath();
9      ClassLoader loader = new URIClassLoader(descriptor.getPath(),
           Thread.currentThread().getContextClassLoader());
10     ErrorCalculator returnvalue = (ErrorCalculator)loader.loadClass(
           descriptor.getErrorCalculatorName()).newInstance();
11     returnvalue.init(descriptor);
12     return returnvalue;
13   }
14 }
```

De `URIClassloader` zorgt ervoor dat de in de configuratie gespecificeerde JAR file als codebase wordt gebruikt.

Abstracte ErrorCalculator Klasse

Naast de omzetting van het ErrorCalculator configuratieobject tot een concrete instantie van deze basiscomponent dient er vanzelfsprekend ook een implementatie van elke nieuwe component gemaakt te worden. Zoals reeds kon afgeleid worden uit figuur 9.10, worden de implementaties van de basiscomponent steeds afgeleid van een basisklasse. In dit opzicht verschilt deze component van de voorgaande basiscomponenten. In dit geval worden de klassen niet geïmplementeerd door af te leiden van een basisklasse maar wordt er gebruik gemaakt van een interface. Dit omdat het basisobject geen code bevat en het gebruik van een klasse bijgevolg nutteloos is. De gebruikte interface heeft als naam `ErrorCalculator` en ziet er als volgt uit:

```
1  package be.ac.ua.comp.GIPSy.ErrorCalculator;
2
3  import ... ;
4
5  public interface ErrorCalculator extends Serializable{
6      public Double[] getError(Predictor predictor, inputData data);
7      public Double[] getError(Predictor model1predictor, Predictor[]
           model2predictor, inputData data);
8      public String[] getDescription();
9      public void init(ErrorCalculatorDescriptor descriptor);
10 }
```

De implementatie bestaat uit vier methoden:

- `init`: Zorgt er bij elke nieuwe implementatie voor dat de configuratieinformatie uit de `ErrorCalculatorDescriptor` op de juiste wijze aangewend wordt

om de implementatie te initialiseren.

- `getError(Predictor predictor, inputData data)`: Heeft als opdracht de kwaliteit van een predictor te verifiëren aan de hand van de exacte samples die vervat zitten in het `inputData` object. Deze methode resulteert in een array van objecten die de resultaten van verschillende kwaliteitscontroles teruggeeft.
- `getError(Predictor model1predictor, Predictor[] model2predictor, inputData data)`: Werkt in essentie op dezelfde manier als de vorige methode maar baseert zich bij de kwaliteitscontrole niet op exacte samples maar op de resultaten van een tweede model beschreven in model2predictor.
- `getDescription()`: Geeft een array met dezelfde afmetingen terug als deze in de voorgaande twee methoden. In dit geval zit in deze array echter de naam van de verschillende kwaliteitscontroles vervat die in de voorgaande methoden gebruikt werden.

In essentie komt het er voor de implementatie dus op neer dat er, voor elk type kwaliteitscontrole, methoden ontwikkeld dienen te worden die als input ofwel exacte samplepunten ofwel een model krijgen om de kwaliteit van een model te verifiëren.

Besluit

De implementatie van een nieuwe ErrorCalculator brengt twee dingen met zich mee. Enerzijds dient er een configuratieobject (in de meeste gevallen een XML file) ontwikkeld te worden waarin de noodzakelijke informatie ter instantiatie en ter initialisatie van een ErrorCalculator vervat zit. Hiernaast moet er tevens een implementatie voorzien worden van de `ErrorCalculator` interface. Concreet houdt dit in dat er voor verschillende kwaliteitscontrole mechanismen methoden ontwikkeld moeten worden die de kwaliteit van een model kunnen verifiëren aan de hand van andere modellen of aan de hand van een lijst van correcte samplepunten. Een concrete implementatie van deze component is terug te vinden in hoofdstuk 13.

9.4.8 Input Data

Naast het aanbrengen van informatie via de configuratieobjecten is er nog een laatste vorm van informatie die het framework nodig heeft om te functioneren. Het gaat hier over de samplepunten. Het framework heeft in dit verband twee vormen van informatie nodig:

- De lijst van samplepunten die tot de parametersweep behoren.
- De lijst van samplepunten uit de parametersweep waarvan de runtime reeds bekend is.

Aangezien deze twee datatypes een nauw verband vertonen, zullen ze beide omvat worden in eenzelfde type klasse namelijk de `InputData` klasse. Sterk vereenvoudigd[4] ziet deze klasse er als volgt uit:

[4]Enkel de belangrijkste members en enkel de hoofding van de methoden worden getoond.

```
1  package be.ac.ua.comp.GIPSy.Parsers;
2
3  import ...;
4
5  public class inputData implements Serializable{
6      private LinkedList<String> names = new LinkedList<String>();
7      private LinkedList<LinkedList<Double>> data = new LinkedList<
           LinkedList<Double>>();
8      private double[][] ranges;
9
10     synchronized public double[][] getData()
11     synchronized public LinkedList<Double> getPoint(int i)
12     synchronized public double[][] getNormdata()
13     synchronized public double getMax(int dimention)
14     synchronized public double getMin(int dimention)
15     synchronized public double getStepsize(int dimention)
16     synchronized public String[] getNames()
17     synchronized public double[][] get2DData(int degree, int x, int y)
18
19     synchronized public void addPoint(LinkedList<Double> point)
20     synchronized public void addName(String name)
21
22     synchronized public inputData popStartGrid()
23     synchronized public LinkedList<Double> popPoint(int i)
24
25     synchronized public inputData getEmptyInputdata()
26     synchronized public inputData clone()
27 }
```

De members van de klasse kunnen opgesplitst worden in vijf groepen:

1. *De datavelden*: In deze velden zitten de samplepoint data vervat:

   - Het `names` veld: Het is mogelijk elke dimensie in de parametersweep een naam te geven. Deze namen zitten vervat in deze gelinkte lijst van `Strings`.
   - Het `data` veld: Dit veld bevat een gelinkte lijst van samplepunten. Elk van deze samplepunten is dan op zich een gelinkte lijst met voor elke dimensie een waarde. Indien het gaat om samplepunten waarvan de runtime reeds bekend is, zal de gelinkte lijst nog een extra waarde bezitten waarin de runtime vervat zit.
   - Het `ranges` veld: Dit veld bevat voor elke dimensie de minimale waarde, de maximale waarde en de stepsize tussen de opeenvolgende samplepunten. Stel dat bijvoorbeeld het kwadraat van de volgende samplepunten 1, 3, 5 en 7 berekend moet worden dan zal de minimale waarde 1 zijn, de maximale waarde 7 en de stepsize 2.

2. *De get methoden*: Deze methoden vragen de gewenste data op uit de datavelden:

- `getData`: Geeft de samplepunten die in het object zitten terug onder de vorm van een twee-dimensionale array. In elke rij zit een punt. Elke kolom stelt de waarde van een bepaalde parameter voor. De laatste kolom heeft waarde 1, of de echte runtime, indien deze reeds gekend is.
- `getPoint`: Deze methode geeft een samplepunt met een bepaalde index-waarde terug.
- `getNormData`: Deze methode geeft, net als de `getData`, de samplepunten terug die in het object vervat zitten. De samplepunten worden echter eerst genormaliseerd. Dit houdt in dat de waarde van elke parameter zal gemapt worden op een waarde tussen 0 en 1. Dit gebeurt als volgt. Stel dat p_i de waarde is van parameter i en max_i en min_i respectievelijk de maximale en de minimale waarde zijn van deze parameter, dan zal de nieuwe waarde voor deze parameter np_i als volgt berekend worden:

$$np_i = \frac{p_i - min_i}{max_i - min_i} \tag{9.1}$$

- `getMax`: Geeft de maximale waarde van een bepaalde parameter terug.
- `getMin`: Geeft de minimale waarde van een bepaalde parameter terug.
- `getStepsize`: Geeft de stepsize van een bepaalde parameter terug. De stepsize is het verschil tussen twee opeenvolgende waarden van een bepaalde parameter. Bij 1, 3, 5, 7 en 9 bijvoorbeeld zal deze waarde 2 zijn.
- `getNames`: Geeft de naam van een bepaalde parameter terug.
- `get2DData`: Geeft een slice van de data terug waarin slechts twee dimensies variabel zijn. Dit ter ondersteuning van de 3D plotting mechanismen.

3. *De add methoden*: Deze methoden voegen extra data toe aan de datavelden:

   - `addPoint`: Voegt een punt toe aan de lijst van punten die reeds in het object vervat zitten.
   - `addName`: Geeft een bepaalde parameter de gewenste naam.

4. *De pop methoden*: Deze methoden vragen data op van de datavelden en verwijderen deze data:

   - `popStartGrid`: Verwijdert en returnt de lijst van startgridpunten uit de lijst van samplepunten die in dit object vervat zitten.
   - `popPoint`: Verwijdert en returnt een bepaald samplepunt.

5. *De factory methoden*: Deze methoden construeren nieuwe objecten van het inputData type:

   - `getEmptyInputdata`: Deze methode maakt een nieuwe instantie aan van het inputData object. Dit object zal op identieke manier geïnitialiseerd zijn als het huidige object maar geen enkel samplepunt omvatten.

- `clone`: Deze methode maakt een nieuwe instantie aan van het inputData object. Dit object zal volledig identiek zijn aan het huidige object.

Naast de opsomming van de verschillende members is het belangrijk op te merken dat elk van de methoden synchronised is. Aangezien dit `inputData` object gebruikt zal worden door verschillende componenten in verschillende threads is het noodzakelijk dit gesynchroniseerd te laten verlopen.

Naast het object waarin deze data worden verzameld, dient er tevens een mechanisme te zijn dat de data initieel in deze objecten samen brengt. Het spreekt voor zich dat deze functionaliteit zal verschillen tussen de verschillende toepassingen. In het geval van de scheduler zal deze informatie rechtstreeks uit het scheduler object afkomstig zijn. Omwille van deze reden is ervoor gekozen, in deze scheduling component, rechtstreeks gebruik te maken van het `inputData` object en van zijn methoden om het noodzakelijke dataformaat te verkrijgen. In het geval van de tester zal deze informatie vergaard worden aan de hand van een parser, de `InputDataParser`, die afgeleid is van de abstracte `Parser` klasse. Deze extra abstractielaag is toegevoegd om het in de toekomst mogelijk te maken ook alternatieve methoden te voorzien om de gewenste data te verwerven. Deze parser ziet er als volgt uit:

```
1  package be.ac.ua.comp.GIPSy.Parsers;
2
3  import ...;
4
5  public  class InputDataParser extends parser{
6    public InputDataParser(String filename){
7      super(filename);
8    }
9
10   private void getnames(FileInputStream stream, inputData returnvalue)
         {
11     ...
12   }
13
14   private void gettypes(FileInputStream stream, inputData returnvalue)
         {
15     ...
16   }
17
18   private LinkedList<Double> getEntry(FileInputStream stream,
       inputData returnvalue){
19     ...
20   }
21
22   public inputData parse()
23   {
24     inputData returnvalue = new inputData();
25     FileInputStream stream = new FileInputStream(new File(filename));
26     getnames(stream,returnvalue);
27     gettypes(stream,returnvalue);
28     LinkedList<Double> tmpdouble = getEntry(stream,returnvalue);
29     while(tmpdouble!=null)
30     {
```

```
31          returnvalue.addPoint(tmpdouble);
32          tmpdouble = getEntry(stream,returnvalue);
33        }
34        return returnvalue;
35      }
36
37  }
```

De implementatie van de abstracte `parse` methode uit de basisklasse zorgt ervoor dat aan de hand van de `getnames`, `gettypes`, `getEntry` methoden de gewenste data onder de vorm van een inputData object worden ingelezen en gereturned.

9.4.9 File Writer

De filewriter is een component die binnen de GIPSyTester gebruikt wordt en die ertoe dient de kwaliteitsinformatie naar de gebruiker te communiceren. Concreet maakt dit object een tab separated file aan waarin elke kolom een andere kwaliteitswaarde voorstelt en elke rij de informatie is die vergaard is na het aanmaken van een nieuw model.

9.5 Extra Features

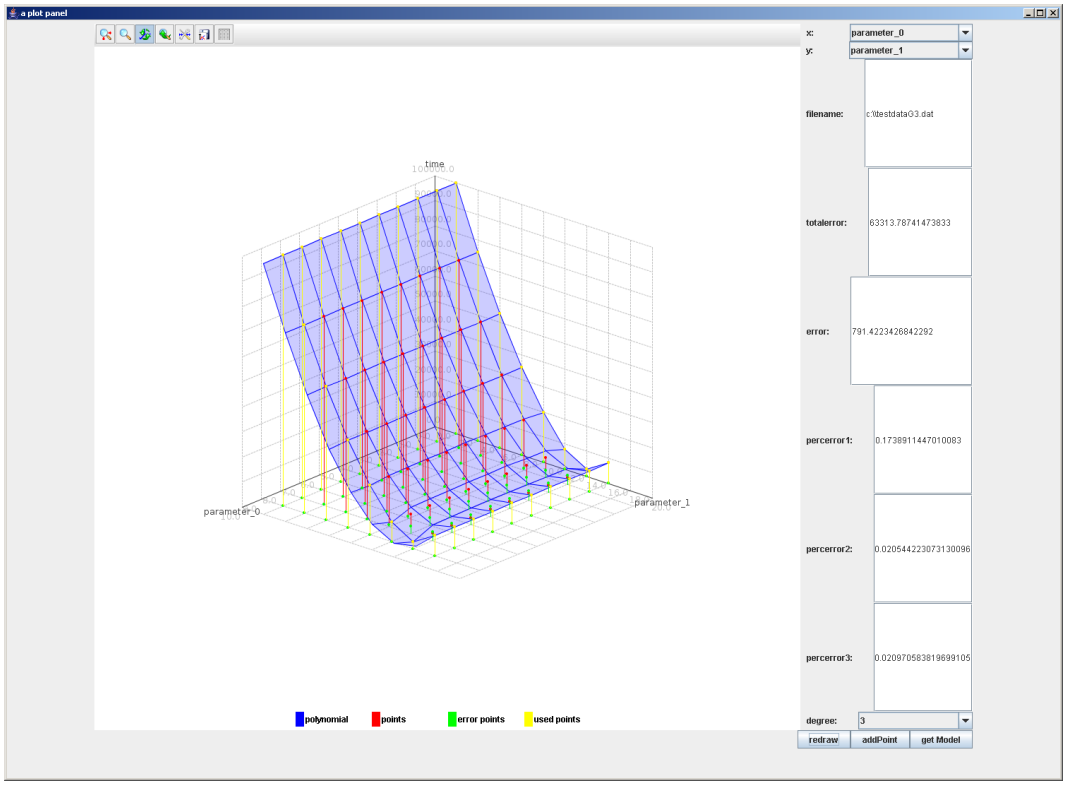
Naast de standaard functionaliteit die het framework voorziet, zijn er nog tal van extra's aan het framework toegevoegd die het werken voor een developer gemakkelijker maken. De belangrijkste twee worden hierna kort besproken.

9.5.1 Grafische User Interface

De grafische interface dient ertoe minder ervaren gebruikers een beeld te geven van de kwaliteit van een model. Om deze doelstelling terdege te verwezenlijken, zijn er twee varianten voorzien van deze interface.

- *Manuele Interface*: Aan de hand van deze interface kunnen er manueel punten aan een model worden toegevoegd. De visualisatie gebeurt aan de hand van 3D plots. Deze zijn opgebouwd uit twee dimensies die de waarden van een bepaalde parameter van de sweep voorstellen en één dimensie die de runtime voorstelt. Aan de hand van de verschillende inputvelden kunnen de gebruikte parameters gewijzigd worden en kan er, voor de overige parameters, een waarde gekozen worden om de te visualiseren slice van het model te specificeren. Dit alles wordt duidelijk in figuur 9.13.

- *Automatische Interface*: Deze interface wordt getoond tijdens de standaard uitvoering van de GIPSyTester. Elke iteratieve stap toont van elke parametercombinatie een plot die een beeld schept van het volledige model dat tot dusver gebouwd is. De Pauze/Continue button maakt het ten allen tijde mogelijk het geheel te pauzeren om zo het model beter te kunnen bekijken. Een afbeelding van deze interface is terug te vinden in figuur 9.14.



Figuur 9.13: Een afbeelding van de Manuele Interface.

9.5.2 Gedistribueerd Testen

De studie van verschillende componenten binnen het framework blijkt al snel computationeel intensief te zijn. Wanneer de vorm van deze berekeningen bestudeerd wordt, blijkt dat het testen van deze componenten met verschillende inputwaarden als een parametersweep kan bekeken worden. Het spreekt voor zich dat het distribueren van deze testen een oplossing zal bieden voor deze computationele vereisten.

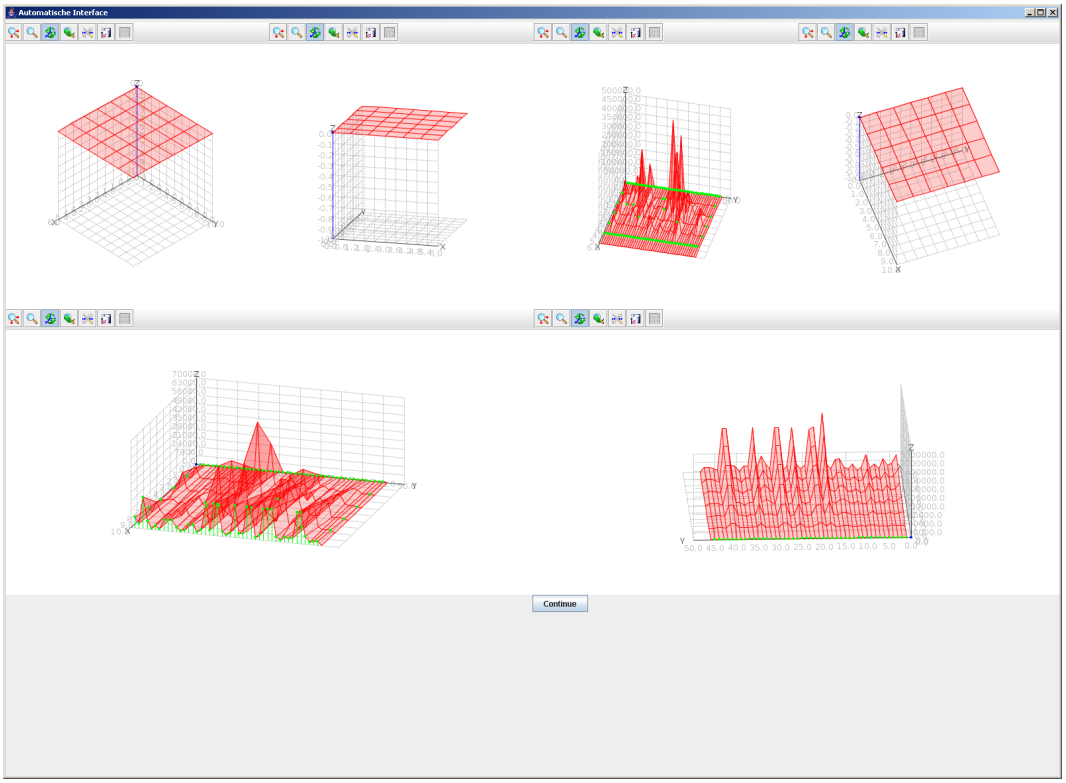
Deze distributie is, in de loop van de ontwikkeling van het GIPSy framework, op twee manieren verwezenlijkt:

- Er is een proof of concept distributie gebeurd aan de hand van het CoBRAS[17] framework.
- Aan de hand van een CoBRA implementatie van de GIPSyTester.

In de volgende twee secties worden deze distributiemethoden kort besproken.

CoBRAS framework

Het CoBRAS framework is een framework dat ontwikkeld is met de doelstelling reeds geschreven Java code te distribueren over verschillende machines. Om dit te bewerkstelligen, worden onderdelen van deze Java code bekeken als een parametersweep job. Dit wil zeggen dat deze stukken code repetitief uitgevoerd zullen



Figuur 9.14: Een afbeelding van de Automatische Interface.

worden met andere parameters als input. Deze verzameling parameters bestaat, naast de klassieke parameters van de uit te voeren methode, tevens uit de globale variabelen die geïnitialiseerd dienen te worden. Het instellen van deze parameters gebeurt op twee manieren. Indien de parameter niet wijzigt gedurende de parametersweep, wordt deze met een vaste waarde geïnitialiseerd. Indien deze parameter moet variëren, kan dit gebeuren aan de hand van een parametersweep functie die voor elke iteratieve stap de parameter anders initialiseert. Meer informatie omtrent het CoBRAS framework is terug te vinden in [17].

In het geval van de GIPSyTester werd elk van de GIPSy componenten (Sampleselector, Predictor, ...) volledig manueel geïnitialiseerd. Uit de resultaten van deze werkwijze bleek dat de distributie een goed idee was maar dat de manier waarop te tijdrovend was. Dit laatste probleem werd opgelost door een concrete gedistribueerde implementatie te maken toegespitst op het GIPSy framework. Een korte beschrijving van deze implementatie is terug te vinden in de volgende sectie die handelt over de CoBRA implementatie.

CoBRA implementatie

Om dit te bewerkstelligen, is er een distributed GIPSyTester ontwikkeld. De implementatie berust volledig op CoBRA en wordt, zoals in sectie 5.6 besproken, gerealiseerd door een klasse, afgeleid van de `Job` klasse, en een klasse, afgeleid van de `Pro-`

`blemSolver` klasse, te implementeren. Concreet houdt dit in dat de functionaliteit, die tot op heden terug te vinden was in de GIPSyTester klasse, geïnjecteerd wordt in de implementatie van de `Job` klasse. De implementatie van de `ProblemSolver` klasse zorgt er dan voor dat voor de verschillende parameters de gewenste instanties van de job klasse geïnstantieerd, geïnitialiseerd en gedistribueerd worden.

Tot slot dient voor de volledigheid nog vermeld te worden dat deze implementatie ook geport is naar de CoBRA componenten die checkpointing ondersteunen. Meer informatie omtrent dit checkpointing framework is terug te vinden in sectie 5.8.3.

HOOFDSTUK 10

Predictie Modellen

10.1 Inleiding

Dit hoofdstuk handelt over de verschillende technieken die aangewend worden om runtimes te voorspellen. In eerste instantie zullen de verschillende technieken, die reeds in de literatuur terug te vinden zijn, kort beschreven worden. De daarop volgende secties pogen technieken uit andere onderzoeksdomeinen te introduceren en aan te wenden om runtime predicties uit te voeren. Elk van deze technieken zal geïmplementeerd worden als uitbreiding van de `Predictor` klasse en getest worden door de klasse in het famework te pluggen. De resultaten van de verschillende technieken worden individueel, per techniek, geanalyseerd en tot slot verwerkt in een algemene conclusie.

10.2 Gerelateerd onderzoek

De literatuur toont aan dat er in het verleden reeds onderzoek naar runtime voorspelling gebeurd is. De belangrijkste technieken worden samengevat in de volgende secties.

10.2.1 Finite state machines

Zoals reeds werd aangehaald in sectie 7.3, wordt er in [28] een predictietechniek voorgesteld die gebaseerd is op finite state machines. Het principe dat hier wordt besproken, is gebaseerd op de volledige verzameling van resource requirements. Hiermee worden zowel de runtime, als de I/O, als de memory requirements bedoeld. Elk van deze requirements wordt gemapt op een dimensie. Gebruik makend van deze

verschillende dimensies zal er gepoogd worden de reeds uitgevoerde taken te clusteren aan de hand van het k-means clustering algoritme. Dit principe werkt als volgt. Er wordt gestart met k clusters. Elk van deze clusters bevat juist één punt uit de reeds uitgevoerde jobs. Dit punt wordt het centrum van de cluster genoemd. De overige jobs worden punt per punt toegevoegd aan de cluster waarvan het centrum het dichtst bij de desbetreffende job ligt. Wanneer alle jobs toegevoegd zijn, wordt per cluster het centrum bepaald en begint het proces opnieuw. Dit gebeurt tot dat de centra van de clusters niet meer of slechts minimaal wijzigen. Formeel uitgedrukt, wordt het centrum van een cluster als volgt bepaald:

$$minimize \sum_{i=1}^{k} \sum_{j} |x_{ij} - \bar{x}_i|^2 \tag{10.1}$$

Hier stelt x_{ij} het j-de element van de i-de cluster C_i voor en is $\bar{x}_i$ het centrum van deze cluster.

Eens de k clusters bepaald zijn, wordt er een transitieschema opgesteld. Dit transitiediagram geeft de kans weer dat de volgende job een job is uit cluster j gegeven dat de huidige job een job is uit cluster i. Het opstellen van dit schema gebeurt op basis van de observaties van de reeds uitgevoerde jobs. De kans p_{ij} dat er wordt overgegaan van state i naar state j wordt als volgt berekend:

$$p_{ij} = \frac{\#\text{transities van state } i \text{ naar state } j}{\#\text{ transities vanuit state } i} \tag{10.2}$$

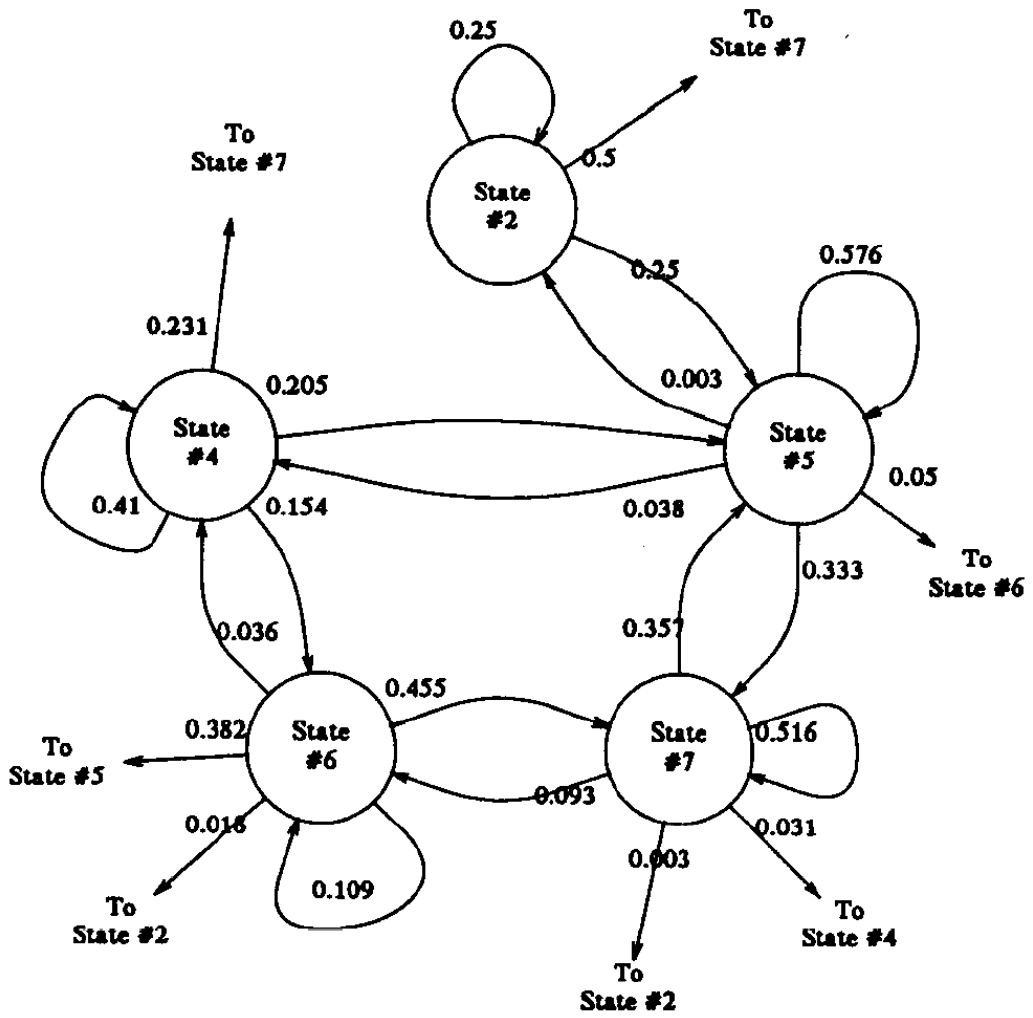
Een voorbeeld van een dergelijk transitiediagram en van een dergelijke transitietabel zijn terug te vinden in figuur 10.1 en tabel 10.1.

Tabel 10.1: Transitietabel

Cluster	1	2	3	4	5	6	7
1	-	-	-	-	-	-	-
2	-	0.250	-	-	0.250	-	0.500
3	-	-	-	-	-	-	-
4	-	-	-	0.410	0.205	0.154	0.231
5	-	0.003	-	0.038	0.576	0.050	0.333
6	-	0.018	-	0.036	0.382	0.109	0.455
7	-	0.003	-	0.031	0.357	0.093	0.516

Eens een transitieschema ter beschikking is, gebeuren de predicties als volgt. Op basis van een bepaald aantal T uitgevoerde jobs wordt er per gedistribueerd programma een centrum d_{ik} per resource type k en per bestaande cluster i gedefinieerd. Er dient wel benadrukt te worden dat dit centrum niet het centrum is dat in het transitieschema gedefinieerd staat. Stel s_{ikj} de benodigde hoeveelheid resources van type k gebruikt door cluster i om taak j uit te voeren dan kan d_{ik} als volgt formeel uitgedrukt worden:

$$d_{i,k} = \frac{1}{T} \sum_{j=1}^{T} s_{ikj} \tag{10.3}$$



Figuur 10.1: Een voorbeeld van een transitiediagram uit paper [28].

Indien de huidige job uit cluster l afkomstig is en p_{lj} de kans is dat er na een job uit cluster l een job uit cluster j volgt, dan kan de resource predictie r_k voor de resource van type k voorspeld worden als:

$$r_k = \sum_{j=1}^{N} p_{lj} \times d_{jk} \tag{10.4}$$

Hoewel deze predictiemethode in vele gevallen blijkt te werken, is ze niet toepasbaar in het huidige onderzoek omwille van de volgende redenen:

- De correlatie tussen verschillende gedistribueerde programmatuur wordt hier gebruikt. Dit is onmogelijk omdat in de studie in deze thesis ook individuele predicties ondersteund moeten worden. De predicties moeten, met andere woorden, kunnen gebeuren zonder informatie van buiten de huidige parametersweep.

- De predictie is gebaseerd op gedistribueerde problemen met zeer specifieke groepen taken, in tegenstelling tot het doel van deze thesis, die ook uniform verdeelde resource requirements wil ondersteunen.
- Dit predictiemechanisme houdt geen rekening met de parametersweep variabelen die in dit onderzoek als extra informatie beschikbaar zijn.
- Het gebruik van het k-means clustering algoritme maakt het opstellen van een finite state machine computationeel heel erg zwaar. Dit heeft als gevolg dat het haast onmogelijk is in een operationele grid, voor elk type applicatie, predictieondersteuning aan te bieden.

10.2.2 Gelijkheids testen

Deze sectie beschrijft een techniek uit de literatuur [107] die op basis van de overeenkomsten tussen de reeds uitgevoerde jobs en de nog uit te voeren jobs de runtime voorspelt. De techniek is gebaseerd op [29] en [37]. In principe komt het er op neer dat er een aantal templates gedefinieerd worden die enkel jobeigenschappen bevatten. Op basis van deze templates wordt de totale verzameling jobs opgesplitst in groepen die, voor de eigenschappen die de template omvat, dezelfde waarde hebben. Het algoritme werkt in drie stappen:

1. Definieer T, de verzameling gebruikte templates, en initialiseer C, de set van verschillende jobgroepen[1]. Deze set zal bij aanvang leeg zijn.
2. Wanneer er een job moet uitgevoerd worden,dan worden de volgende stappen doorlopen:
   (a) De templates uit T worden toegepast op de eigenschappen van job a. Op deze manier worden de groepen C_a waarbinnen de job a zou kunnen thuishoren, bepaald.
   (b) Al de groepen die (nog) niet tot C behoren, worden uit C_a verwijderd.
   (c) Voor elke overblijvende groep in C_a wordt er een runtime voorspelling gedaan en een betrouwbaarheidsinterval[2] berekend.
   (d) De runtime voorspelling met het kleinste betrouwbaarheidsinterval wordt geselecteerd en gebruikt in de applicatie.
3. Wanneer een job beëindigd is, worden de volgende stappen doorlopen:
   (a) De verzameling C_a groepen waarbinnen een job valt wordt wederom bepaald.
   (b) $\forall c_i \in C_a$:

[1] Dit zijn de verschillende groepen die ontstaan door de templates toe te passen op jobs. Stel dat er bijvoorbeeld een template bestaat met daarin enkel de username, dan zullen job a van user Jef en job b van user Jef in dezelfde groep zitten en job c van user Jos in een andere groep.

[2] Er kan met afdoende zekerheid gesteld worden dat de voorspelde waarde binnen dit interval ligt.

i. $c_i \notin C \Rightarrow c_i$ wordt toegevoegd aan C.

ii. $|c_i| =$ maximum history$(c_i) \Rightarrow$ het oudste punt wordt verwijderd uit c_i.

iii. Voeg a toe aan c_i.

De runtime voorspelling in deze techniek kan op twee manieren gebeuren. De eerste manier bepaalt eenvoudigweg de gemiddelde runtime waarde van de jobs in de groep. De tweede methode maakt gebruik van een extra parameter die niet gebruikt is in het opstellen van de template. Stel bijvoorbeeld dat elk punt bepaald wordt door parameters X, Y en Z en dat (X, Z) de gebruikte template is, dan kan Y gebruikt worden om de voorspelling binnen de groep te optimaliseren. concreet wordt er gezocht naar coëfficiënten a en b zodat $aY + b = R$. Lineaire regressie op de reeds gekende runtimes zal ervoor zorgen dat de coëfficiënten a en b gevonden worden die aan de hand van Y de runtime R voor elk punt kunnen bepalen.

Om dit alles mogelijk te maken, dient er tevens een methode ontwikkeld te worden om de geschikte templates op te stellen. In de eerste publicaties omtrent deze techniek [37] gebeurde dit aan de hand van een vast stel templates die opgesteld werden aan de hand van de beschikbare data per job. In een latere fase van het onderzoek werden deze vaste templates vervangen door twee algoritmes die, aan de hand van een dataset W, een optimale set van templates kunnen bepalen:

- Het *Greedy algoritme*: Heeft tot doel een template set $T = t_i$ op te stellen met:

$$t_i = \{(), (h_{1,1}), (h_{2,1}, h_{2,2}), ..., (h_{i,1}, h_{i,2}, ..., h_{i,i})\} \tag{10.5}$$

  Hier staat $h_{j,k}$ voor één van de n jobeigenschappen. Het algoritme gaat als volgt.

  1. Initialiseer de template set $T = \{()\}$.
  2. Voor $i = 1$ tot n
     (a) Stel T_c een nieuwe template set die $\binom{n}{i}$ verschillende templates met i job eigenschappen bevat.
     (b) Voor elke template t_c in T_c
        i. Ontwikkel een mogelijke template set $X_c = T \cup \{t_c\}$
        ii. De gemiddelde fout van de dataset W toegepast op X_c wordt bepaald aan de hand van het hierboven beschreven algoritme.
     (c) De X_c met de laagste gemiddelde fout wordt gedetecteerd en de gerelateerde template t_c wordt aan T toegevoegd ($T = X_c$).

- *Genetisch algoritme*: Het tweede algoritme is een genetisch algoritme. Dit wil zeggen dat het op zoek gaat naar de gewenste template set door verschillende bestaande template sets te kruisen. Deze kruising gebeurt volgens het "survival of the fittest" principe. Een nieuwe generatie wordt steeds ontwikkeld rekening houdend met de kwaliteit van de template sets van de generatie ervoor. Deze

kwaliteit wordt bepaald aan de hand van een *fitness* functie (F). Deze functie ziet er als volgt uit:

$$F = F_{min} + \frac{E_{max} - E}{E_{max} - E_{min}} \cdot (F_{max} - F_{min}) \tag{10.6}$$

E geeft de voorspellingsfout weer van de templateset, E_{min} en E_{max} geven de minimale en de maximale voorspellingsfout weer van de huidige generatie template sets en F_{min} en F_{max} schalen het geheel naar een minimale en een maximale fitness waarde.

Bij het uitvoeren van de kruising wordt elke template set $\left\lfloor \frac{F}{F_{avg}} \right\rfloor$ keer gekozen als ouder bij de voortplanting. Dit wil zeggen dat alle template sets met een fitness waarde, die lager is dan de gemiddelde fitness waarde, niet worden geselecteerd. Om deze kwalitatief mindere template sets alsnog een kans te geven te overleven, worden de overige parents gekozen aan de hand van de *Bernoulli* verdeling. Elke template set heeft dan een kans $F - F_{avg} \left\lfloor \frac{F}{F_{avg}} \right\rfloor$ om als parent geselecteerd te worden.

Bij de kruising van twee parents T_1 en T_2 wordt een template i in T_1 en een template j in T_2 gekozen. Deze templates zijn de positie waar de template sets gesplitst zullen worden. Bovenop deze selectie wordt nog een bit d bepaald waarop de geselecteerde template op zich gesplitst zal worden. Stel dat $T_1 = t_{1,1}, ..., t_{1,n}$ en $T_2 = t_{2,1}, ..., t2, m$ dan zal de nieuwe generatie bestaan uit een nieuwe $T_1 = t_{1,1}, ...t_{1,i-1}, n_1, t_{2,j+1}, ...t_{2,m}$ en een nieuwe $T_2 = t_{2,1}, ...t_{2,j-1}, n_2, t_{1,i+1}, ...t_{1,n}$. Stel dat elke template voorgesteld wordt met b bits dan zal n_1 bestaan uit de eerste d bits van $t_{1,i}$ en de laatste $b - d$ bits van $t_{2,j}$. Analoog zal n_2 dan bestaan uit de eerste d bits van $t_{2,j}$ en de laatste $b - d$ bits van $t_{1,i}$.

Deze kruising wordt repetitief herhaald tot er een template set gevonden wordt waarvan de kwaliteit voldoet aan de vooropgestelde vereisten.

Hoewel ook deze predictie methode in vele gevallen blijkt te werken, is ze niet toepasbaar in het huidige onderzoek omwille van de volgende redenen:

- De correlatie tussen verschillende gedistribueerde programmatuur wordt hier gebruikt. Dit is onmogelijk omdat in deze thesis ook individuele predicties ondersteund moeten worden. De predicties moeten, met andere woorden, kunnen gebeuren zonder informatie van buiten de huidige parametersweep.

- De predictie is gebaseerd op gedistribueerde problemen met zeer specifieke groepen taken, in tegenstelling tot het doel van deze thesis, die ook uniform verdeelde resource requirements wil ondersteunen.

- Dit predictiemechanisme houdt geen rekening met de parametersweep variabelen die in dit onderzoek als extra informatie beschikbaar zijn.

10.2.3 Nearest Neighbour prediction

Een belangrijke groep van runtime predictiemechanismen, die terug te vinden zijn in de literatuur, zijn gebaseerd op de toepassing van een *Kernel* functie op de punten die het dichtst bij het te voorspellen punt liggen. De twee toonaangevende onderzoeken op dit domein worden gevoerd door:

- M. A. Iverson [52][53]
- W. Smith [108] [106]

Iverson techniek

Het doel van deze techniek is, om aan de hand van een niet-geparameteriseerde regressie techniek[3], de runtime van een samplepunt te voorspellen. Iverson realiseert dit aan de hand van het *k-Nearest Neighbour* algoritme [45] [36]. Formeel wordt dit algoritme als volgt gedefinieerd. Stel dat $X = [x^1 x^2 ... x^p]$ een parametervector is en $(t_i, X_i)_{i=1}^n$ een verzameling van n observaties en stel dat:

$$J_X = \{i : X_i \text{ is één van de } k \text{ dichtstbijzijnde buren } X\} \tag{10.7}$$

dan wordt de voorspelling $\hat{m}(X)$, gebruik makend van het *L-Smoothing*[4] algoritme, als volgt gedefinieerd

$$\hat{m}(X) = \frac{1}{k - 2\lfloor \gamma k \rfloor} \sum_{i=\lfloor \gamma k \rfloor}^{k-\lfloor \gamma k \rfloor} W_i(X)\, t_i \tag{10.8}$$

Er wordt vanuit gegaan dat de observaties $\{(t_i, X_i)\} \in J_X$ gesorteerd worden volgens t_i zodat de kleinste en de grootste $\lfloor \gamma k \rfloor$ samplepunten, volgens het L-Smoothing principe, verwaarloosd worden. Hierbij geldt dat $0 \leq \gamma \leq 0.5$. De gewichten $W_i(X)$ bepalen in welke mate de runtime van het reeds uitgevoerde samplepunt i invloed heeft op de te voorspellen runtime. Dit wordt als volgt gedefinieerd:

$$W_i(X) = \frac{K_R(X - X_i)}{\hat{f}_R(X)} \tag{10.9}$$

Hier stelt $\hat{f}_R(X)$ de normalisatiefactor voor die als volgt gedefinieerd is:

$$\hat{f}_R(X) = \frac{1}{k - 2\lfloor \gamma k \rfloor} \sum_{i=\lfloor \gamma k \rfloor}^{k-\lfloor \gamma k \rfloor} K_R(X - X_i) \tag{10.10}$$

Deze normalisatiefactor zorgt ervoor dat een gewicht de verhouding zal zijn tussen de huidige waarde van een kernelfunctie en de gemiddelde waarde over heel de verzameling samplepunten. De multidimensionale kernelfunctie $K_R(U)$, die zowel in

[3] Een niet-geparameteriseerde regressie techniek waarin de predictor geen gepredefinieerde vorm aanneemt maar wordt opgebouwd aan de hand van informatie uit de datapunten.

[4] L-Smoothing zorgt ervoor dat door het verwijderen van outliers een meer robuust gewogen gemiddelde berekend kan worden [43].

formule 10.9 als in formule 10.10 voorkomt, wordt opgebouwd door de kernelfunctie van elke dimensie met deze van de overige dimensies te vermenigvuldigen. Formeel wordt dit als volgt gedefinieerd:

$$K_R(U) = \prod_{j=1}^{p} K_R^j(u_j) \tag{10.11}$$

De kernelfunctie K_R^j van een dimensie j wordt als volgt gedefinieerd:

$$K_R^j(u) = \frac{1}{R^j} K^j\left(\frac{u}{R^j}\right) \tag{10.12}$$

Deze kernelfunctie bestaat uit de *Epanechnikov Kernel*[31][32] K^j en een schalingsfactor R^j. De Epanechnikov Kernel zorgt ervoor dat de samplepunten die het dichtst bij de te voorspellen runtime liggen, de grootste impact hebben op deze voorspelling. De kernel wordt als volgt gedefinieerd:

$$K\left(u\right) = \frac{3}{4}\left(1 - u^2\right) \tag{10.13}$$

De schalingsfactor R^j is de maximale afstand in dimensie j van alle punten in J_x tot het te voorspellen punt. Formeel wordt dit als volgt genoteerd:

$$R^j = \max_{J_x} \left| x^j - x_i^j \right| \tag{10.14}$$

Smith techniek

Deze techniek werkt vrij analoog aan deze voorgesteld door Iverson. Het verschil zit vooral in het feit dat bij deze aanpak ook nominale waarden ondersteund worden en dat er een andere kernel functie gebruikt wordt. Omdat deze techniek zowel lineaire (getallen) als nominale (strings) waarden ondersteunt, wordt de volgende afstandsdefinitie gebruikt:

$$D(x, y) = \sqrt{\sum_f w_f d_f\left(x, y\right) - 2} \tag{10.15}$$

waarin w_f het gewicht, die de invloed van elke inputfactor op de predictie uitoefent, beschrijft. Stel dat f een bepaalde inputfactor aangeeft dan definieert $d_f\left(x, y\right)$ de afstand per inputfactor. $d_f\left(x, y\right)$ wordt als volgt gedefinieerd:

$$d_f\left(x, y\right) = \begin{cases} 1, & \text{als } x_f \text{ of } y_f \text{ onbekend zijn} \\ overlap_f\left(x, y\right), & \text{als } f \text{ een nominale waarde heeft} \\ rn_diff_f\left(x, y\right), & \text{elders} \end{cases} \tag{10.16}$$

met

$$overlap_f\left(x, y\right) = \begin{cases} 0, & \text{als } x_f = y_f \\ 1, & \text{elders} \end{cases} \tag{10.17}$$

en met

$$rn_diff_f = \frac{|x_f - y_f|}{\max_f - \min_f} \tag{10.18}$$

Aan de hand van de afstandsdefinitie $D_f(x, y)$ kan het voorspellen van een outputwaarde f gebeuren gebruik makend van de volgende formule:

$$E_f(q) = \frac{\sum_e K(D(q, e)) V_f(e)}{\sum_e K(D(q, e))} \tag{10.19}$$

Deze definitie geeft het gewogen gemiddelde van de reeds gekende runtimes $V_f(e)$. De gewichten per samplepunt worden berekend aan de hand van de kernelwaarde van hun afstand tot het te voorspellen samplepunt. De kernelfunctie is een functie die ervoor zorgt dat het gewicht van een punt, dicht bij het te voorspellen punt, meer gewicht zal geven aan de voorspelling dan een punt ver weg van het te voorspellen samplepunt. De Gaussische functie is de kernelfunctie die in het onderzoek van Smith gebruikt is. Formeel ziet deze er als volgt uit:

$$K(d) = e^{-\left(\frac{d}{k}\right)^2} \tag{10.20}$$

Het kernelgewicht k maakt het mogelijk de kernelfunctie te verbreden of te versmallen om aldus samplepunten die ver weg liggen respectievelijk meer of minder gewicht te geven.

Besluit

Het is meteen duidelijk dat de methoden van *Iverson* en van *Smith* zeer dicht bij elkaar liggen. Het gebruik van de nominale parameters in de techniek van Smith maakt echter dat deze techniek meer toegespitst is op predicties die zich baseren op taken van meerdere gebruikers. Dit is, zoals reeds vermeld, niet de bedoeling in dit onderzoek. De techniek van *Iverson* daarentegen kan wel volledig gebruikt worden binnen het stramien waarin deze thesis predicties wil doen. Omwille van deze redenen is deze techniek dan ook geselecteerd als referentie om de overige predictietechnieken naar waarde te schatten.

10.2.4 Besluit

Naast de verschillende technieken, die hiervoor beschreven zijn, zijn er in de literatuur nog tal van predictieimplementaties beschreven die specifiek toegespitst zijn op bepaalde gedistribueerde software. Uit de literatuur blijkt dus duidelijk dat er al verschillende technieken ontwikkeld zijn om runtime predictie te ondersteunen. Deze pogingen zijn echter steeds programmaspecifiek opgebouwd of maken gebruik van informatie uit parameter sweep runs. De bedoeling van dit onderzoek is dan ook een predictie mechanisme voor parameter sweeps te ontwikkelen dat enkel gebruik maakt van informatie uit de jobs van de eigen parametersweep. Omwille van het feit dat de techniek van *Iverson* hier eventueel ook voor gebruikt kan worden, zal deze techniek ook binnen het framework geïmplementeerd worden en zullen de resultaten van deze techniek als referentiepunt gebruikt worden.

10.3 Iverson techniek

10.3.1 Theoretische benadering

De werking van deze techniek is reeds beschreven in sectie 10.2.3. Omwille van de toepasbaarheid in dit onderzoek en van de nood aan een referentiepunt is deze techniek geïntegreerd in de huidige sectie. Het vervolg van dit hoofdstuk zal de concrete implementatie en de testresultaten verder toelichten.

10.3.2 Implementatie

Om de implementatie beter te begrijpen, volgt hier een vereenvoudigde versie van deze implementatie.

```
1  package be.ac.ua.comp.GIPSy.Predictors.IversonPredictors;
2
3  import ...;
4
5  public class IversonPredictor extends Predictor {
6
7    private  double[][] points;
8    private  double gamma = 0;
9    private  double[] R;
10   private double[][] totaldata = null;
11
12   public IversonPredictor(Matrix points, double gamma,int amount,
         double[][] totaldata) {
13     super();
14     init(points, gamma,totaldata);
15     this.amount = amount;
16   }
17
18   private void init(Matrix points, double gamma, double[][] totaldata)
         {
19     - sorteert samplepunten op hun runtime
20     - L-Smooth
21     - R wordt bepaald
22   }
23
24   public double Evaluate(Matrix value) {
25     points = getNearest(value.getArray()[0],usedpoints,amount);
26     KR = new double[points.length];
27     knowKR=false;
28     double[] thevalue = value.getArray()[0];
29     double returnvalue=0;
30     frvalue = getfr(thevalue,points);
31     for(int i=0;i<points.length;i++){
32       double weight = getweight(thevalue,points[i],frvalue);
33     }
34     returnvalue/=usedpoints.length;
35     return Math.abs(returnvalue);
36   }
37
```

```
38    private double getKR(double[] value){
39      double returnvalue = 1;
40      for(int i=0;i<value.length;i++){
41        double inputvalue = Math.abs(value[i])/R[i];
42        returnvalue*=((1.0/R[i])*(3.0/4.0)*(1.0-inputvalue*inputvalue));
43      }
44      return returnvalue;
45    }
46
47    private double getweight(double[] fixedvalue,double[] value,double
          frvalue){
48      double Krvalue = getKR(vectorDistance(fixedvalue,value));
49      return Krvalue/frvalue;
50    }
51
52    private double getfr(double[] fixedvalue, double[][] points){
53      int k = points.length;
54      double returnvalue = 0;
55      for(int i=0;i<points.length;i++){
56          returnvalue+=getKR(vectorDistance(points[i],fixedvalue));
57      }
58      return (1.0/k)*returnvalue;
59    }
60  }
```

De implementatie verloopt volgens de theoretische uiteenzetting in sectie 10.2.3 en kan in principe onderverdeeld worden in vijf onderdelen:

1. *De initialisatie fase*: Deze fase wordt doorlopen bij het uitvoeren van de `init(...)` methode. Deze methode heeft drie taken:

   (a) De initialisatie zorgt ervoor dat de noodzakelijke initialisatiedata in het predictor object aanwezig zijn.

   (b) Implementeert het *L-Smooth* algoritme door de $\lfloor \gamma k \rfloor$ datapunten met de hoogste en met de laagste runtime te verwijderen uit de sampleverzameling. Dit zoals formeel beschreven in formules 10.8 en 10.10.

   (c) Berekenen van de schalingsfactor R^j zoals formeel wordt beschreven in formule 10.14.

2. *Evaluatie*: Deze fase wordt ondersteund door de `Evaluate(Matrix value)` methode. Deze methode voorspelt de runtimes gebruik makend van formule 10.8. De gewichten die deze formule vereist, worden opgevraagd aan de hand van de `getweight(...)` methode die verder besproken wordt in het volgende onderdeel van de implementatie.

3. *Gewichten*: Deze fase berekent de gewichten en wordt geïmplementeerd door de `getweight(...)` methode. De gewichten worden berekend aan de hand van formule 10.9, gebruik makend van de normalisatiefactor $\hat{f}_R$ en van de Kernelfunctie K_R die respectievelijk door de `getfr(...)` en de `getKR(...)` methode berekend worden. Beide methoden worden verder besproken in een volgend onderdeel.

4. *Normalisatie*: Deze fase berekent de normalisatiefactor $\hat{f}_R(X)$ aan de hand van formule 10.10. Een lus over de verschillende gebruikte samplepunten berekent de gemiddelde kernelwaarde van het vectorieel verschil tussen het te voorspellen punt en elk van deze samplepunten. De `getfr(...)` methode, die dit geheel implementeert, maakt gebruik van de `getKR(...)` methode die in het volgende implementatieonderdeel behandeld wordt.

5. *KernelFunctie*: Deze fase berekent de output van de kernelfunctie aan de hand van formules 10.11, 10.12 en 10.13. Dit alles wordt geïmplementeerd in de `getKR(...)` methode. Gebruik makend van een lus wordt voor elke dimensie de Epanechnikov waarde van de geschaalde inputwaarde berekend en worden deze met elkaar vermenigvuldigd.

10.4 Polynomiale Modellen

10.4.1 Theoretische benadering

In deze sectie wordt het voorspellen van runtimes aan de hand van polynomiale modellen besproken. Het opzet van deze techniek is aan de hand van de runtimes van de reeds uitgevoerde jobs een polynomiaal model op te stellen dat deze punten benadert. Er wordt immers vanuit gegaan dat een polynoom, die runtimes van de reeds uitgevoerde jobs benadert, dit ook zal doen bij de jobs die nog moeten uitgevoerd worden. De polynoom zal een n dimensionale functie zijn met de n parameters van de parametersweep als inputparameters en de runtime als output. Een polynoom wordt ook wel een algebraïsche uitdrukking genoemd en kan als volgt gedefinieerd worden:

$$P_{n,m}(x) = \sum_{i_j=0}^{m} a_{i_1,\ldots,i_n} x_1^{i_1} \ldots x_n^{i_n} \left(j : 1 \to n; \sum_{k=1}^{n} i_k \leq m \right) \tag{10.21}$$

met n het aantal dimensies en m de graad van de polynoom. Een eenvoudige twee-dimensionale polynoom van de 2de graad krijgt dan de volgende vorm:

$$P_{2,2} = a_{2,0}x_1^2x_2^0 + a_{1,1}x_1^1x_2^1 + a_{0,2}x_1^0x_2^2 + a_{1,0}x_1^1x_2^0 + a_{0,1}x_1^0x_2^1 + a_{0,0}x_1^0x_2^0 \tag{10.22}$$

Het opstellen van de polynoom is het cruciale onderdeel van deze techniek. Een naïeve oplossing voor dit probleem is de graad m van de polynoom zo te kiezen dat het aantal samplepunten $s \leq \frac{(n+m-1)}{m!(n-1)!}$. In dit geval zal het aantal onbekenden groter zijn dan het aantal samplepunten en kan de polynoom door de samplepunten getrokken worden. De gewenste coëfficiënten $a_{i_1,\ldots,i_n}$ om dit te verwezenlijken, worden bekomen door het volgend stelsel op de s samplepunten $p_1 \ldots p_s$ op te lossen:

$$\begin{cases} R(p_1) &= \sum_{i_j=0}^{m} a_{i_1,\ldots,i_n} p_{1_1}^{i_1} \cdots p_{1_n}^{i_n} \quad (j : 1 \to n; \sum_{k=1}^{n} i_k \leq m) \\ &\vdots \\ R(p_s) &= \sum_{i_j=0}^{m} a_{i_1,\ldots,i_n} p_{s_1}^{i_1} \cdots p_{s_n}^{i_n} \quad (j : 1 \to n; \sum_{k=1}^{n} i_k \leq m) \end{cases} \tag{10.23}$$

Omdat in deze toepassing het aantal punten, waaruit de polynomen opgebouwd moeten worden, toeneemt in de tijd zal dit tot gevolg hebben dat de te construeren polynomen steeds van een hogere graad moeten zijn. Het toenemen van deze graad zal twee nadelige gevolgen hebben:

1. De computationele vereisten voor het oplossen van bovenstaand stelsel zullen te groot worden. Het oplossen van het stelsel zal hierdoor niet binnen een aanvaardbaar tijdsframe kunnen gebeuren.

2. De polynoom zal een oscillerend gedrag beginnen vertonen. Dit brengt met zich mee dat het model niet langer representatief zal zijn voor de te voorspellen runtimes.

De oplossing voor deze problematiek bestaat erin de punten te benaderen met de polynoom en de polynoom niet noodzakelijk door de punten te trekken. Dit heeft als voordeel dat er een vaste graad voor de polynoom kan gekozen worden en eventuele oscillaties vermeden kunnen worden. Een veelgebruikte techniek om een dergelijke polynoom op te stellen, wordt de *least squares* of *kleinste kwadraten* [19] techniek genoemd. Het opzet van deze techniek is de afstand van de samplepunten tot de gekozen curve te minimaliseren. Concreet komt dit erop neer dat de som S van de kwadraten van de gekwadrateerde residu's geminimaliseerd wordt. De residu's zijn het verschil tussen de functiewaarde van de polynoom in een bepaald samplepunt en de effectieve runtime in dit samplepunt. Formeel wordt dit als volgt beschreven:

$$S = \sum_{i=1}^{s} (R(x_i) - Pol(x_i))^2 \tag{10.24}$$

Hier zullen $R(x_i)$ en $Pol(x_i)$ respectievelijk de effectieve runtime en de functiewaarde van de polynoom zijn voor een gegeven samplepunt en stelt s het aantal beschikbare samplepunten voor.

In deze techniek wordt er steeds vanuit gegaan dat het aantal samplepunten $s \geq \frac{(n+m-1)}{m!(n-1)!}$. Deze s punten kunnen, net als formule 10.23, uitgeschreven worden als een stelsel:

$$\begin{cases} R(p_1) &= \sum_{i_j=0}^{m} a_{i_1,\ldots,i_n} p_{1_1}^{i_1} \cdots p_{1_n}^{i_n} \quad (j : 1 \to n; \sum_{k=1}^{n} i_k \leq m) \\ & \vdots \\ R(p_s) &= \sum_{i_j=0}^{m} a_{i_1,\ldots,i_n} p_{s_1}^{i_1} \cdots p_{s_n}^{i_n} \quad (j : 1 \to n; \sum_{k=1}^{n} i_k \leq m) \end{cases} \tag{10.25}$$

Stel

$$P = \begin{pmatrix} p_{1_1}^n & p_{1_1}^{n-1} p_{1_2} & \cdots & p_{1_2}^n & p_{1_2}^{n-1} p_{1_3}^{n-2} & \cdots & p_{1_1}^{n-1} & p_{1_1}^{n-2} p_{1_2} & \cdots & p_{1_1} & \cdots & 1 \\ & & & & \vdots & & & & & & & \\ p_{s_1}^n & p_{s_1}^{n-1} p_{s_2} & \cdots & p_{s_2}^n & p_{s_2}^{n-1} p_{s_3}^{n-2} & \cdots & p_{s_1}^{n-1} & p_{s_1}^{n-2} p_{s_2} & \cdots & p_{s_1} & \cdots & 1 \end{pmatrix} \tag{10.26}$$

$$\text{en} \quad A = \begin{pmatrix} a_{n\ 0\ \cdots\ 0} \\ a_{n-1\ 1\ 0\cdots 0} \\ \vdots \\ a_{0\ n\ 0\cdots 0} \\ a_{0\ n-1\ 1\ 0\cdots 0} \\ \vdots \\ a_{n-1\ 0\cdots 0} \\ a_{n-2\ 1\ 0\cdots 0} \\ \vdots \\ a_{1\ 0\cdots 0} \\ \vdots \\ 1 \end{pmatrix} \tag{10.27}$$

$$\text{en} \quad R = \begin{pmatrix} R(p_1) \\ \vdots \\ R(p_s) \end{pmatrix} \tag{10.28}$$

dan kan formule 10.25 herschreven worden als het matrix product:

$$R = P \cdot A \tag{10.29}$$

Aangezien de polynoom niet precies door de samplepunten gaat, bestaat er een verschilvector $R - P \cdot A$. De norm van deze verschilvector is de som van de kwadratische afwijkingen. Er wordt dus op zoek gegaan naar A, waarvoor $\|R - P.A\|$ minimaal is. Dit is het geval wanneer deze verschilvector loodrecht op de kolomruimte van P staat. Twee vectoren staan loodrecht op elkaar wanneer het inwendig product nul is. Bijgevolg kan er gesteld worden dat:

$$\begin{aligned} (PA)^t(R - PA) &= 0 && (10.30)\\ (A^tP^t)(R - PA) &= 0 && (10.31)\\ A^t(P^t(R - PA)) &= 0 && (10.32)\\ A^t(P^tR - P^tPA) &= 0 && (10.33) \end{aligned}$$

zodat:

$$\begin{aligned} P^tR - P^tPA &= 0 && (10.34)\\ P^tPA &= P^tR && (10.35)\\ (P^tP)^{-1}P^tPA &= (P^tP)^{-1}P^tR && (10.36)\\ A &= (P^tP) - 1P^tR && (10.37) \end{aligned}$$

Het concrete polynomiale model wordt verworven door de resulterende A te gebruiken als coëfficiënten voor formule 10.21. Een predictie voor een parameterjob x komt nu overeen met zijn functiewaarde voor formule 10.21.

10.4.2 Implementatie

Zoals uit de theoretische benadering reeds blijkt, zal het merendeel van de operaties bestaan uit matrixbewerkingen. Om dit op een vlotte manier te laten verlopen, is er binnen dit onderzoek gekozen het JAMA [55] (Java Matrix) package te gebruiken.

De implementatie van deze groep predictors gebeurt op twee niveaus onder de vorm van een basisklasse en van uitgebreide klasses. De reden die hierachter schuil gaat is de uniformiteit in evaluatie en de diversiteit in het opstellen van de A (zie formule 10.27) en de P (zie formule 10.26) matrix bij de verschillende polynoomtypes. De functionaliteit van de basisklasse, `Polynomial`, wordt opgebouwd aan de hand van twee matrixen: de coëfficiëntenmatrix en de description matrix. De coëfficiëntenmatrix definieert de waarden van de coëfficiënten. De descriptionmatrix definieert de exponenten van elke dimensie (variabele) die bij deze coëfficiënten horen. De coëfficiëntenmatrix zal bijgevolg een $[1 \times r]$ matrix zijn met r het aantal coëfficiënten en de descriptionmatrix zal een $[r \times n]$ matrix zijn met n het aantal dimensies (parameters in de sweep) en r het aantal coëfficiënten. Beide matrixen worden geïnitialiseerd via de constructor.

Dit alles wordt geïllustreerd aan de hand van een klein voorbeeld. Stel $Pol(x, y) = 4x^2y + 3x + 2y + 1$ dan wordt deze polynoom in de huidige implementatie beschreven door de coëfficiëntenmatrix A en een descriptionmatrix D:

$$A = \begin{bmatrix} 4 & 3 & 2 & 1 \end{bmatrix} \quad D = \begin{bmatrix} 2 & 1 \\ 1 & 0 \\ 0 & 1 \\ 0 & 0 \end{bmatrix}$$

De evaluatie van een punt in een Polynoom gebeurt, gebruik makend van de `Evaluate` methode. Deze krijgt het desbetreffende punt mee als parameter en evalueert het punt volgens formule 10.21. In essentie komt het erop neer dat aan de hand van twee lussen de verschillende componenten van de polynoom worden opgebouwd en berekend. De binnenste lus zal de componenten, beschreven in de descriptor matrix, opbouwen. In de buitenste lus worden deze componenten vermenigvuldigd met de overeenkomstige coëfficiënt en wordt alles opgeteld. Dit alles wordt geïllustreerd aan de hand van het volgende codefragment:

```
1  package be.ac.ua.comp.GIPSy.Predictors.Polynomials;
2
3  import ... ;
4  import Jama.*;
5
6  public class Polynomial extends Predictor{
7
8    protected Matrix description;
9    protected Matrix coefficienten;
10
11   public Polynomial(Matrix description, Matrix coefficienten){
12     super();
13     this.description = description;
```

```
14      this.coefficienten= coefficienten;
15    }
16
17    public double Evaluate(Matrix value){
18      double[][] Parray = new double[1][description.getColumnDimension()
            ];
19      // the current value is put into Matrix P
20      for(int i=0; i< description.getColumnDimension(); i++){
21        Parray [0][i]=1;
22        for(int j=0; j<description.getRowDimension(); j++){
23          double valuepart = value.get(0,j);
24          double exp= Math.pow(valuepart ,description.get(j,i));
25          if((exp!=0)||(valuepart ==0)){
26            Parray [0][i]*= exp;
27          }
28        }
29      }
30      Matrix P= new Matrix(unknownarray);
31      // matrix P is multiplied with matrix a
32      Matrix result = P.times(factors);
33      // absolute value is returned because the runtime can't be
            negatieve
34      return Math.abs(result.get(0,0));
35    }
36  }
```

Aangezien de predicties in dit onderzoek gebeuren aan de hand van samplepunten, zal de opbouw van de A en van de P matrix ook gebeuren op basis van deze samplepunten. Om dit te verwezenlijken, zijn er tot op heden twee relevante uitbreidingen ontwikkeld:

- `MaxDegreeNPolynomial`: Deze implementatie genereert een polynoom van graad $g = min(maxG, M)$ met $maxG$ de maximale graad die bij instantiatie gespecificeerd wordt en $M = \max\left(m : s \geq \frac{(n+m-1)}{m!(n-1)!}\right)$ Hierbij is s het aantal aanwezige samplepunten en n de dimensie van de samplespace (aantal parameters in de parametersweep).

- `SplittedDegreeNPolynomial`: Deze implementatie genereert gelijkaardige polynomen als de voorgaande. In dit geval worden de dimensies echter zo opgesplitst dat de predicties geoptimaliseerd worden. Hierover later meer in sectie 10.11.

De meerwaarde die de `MaxDegreeNPolynomial` biedt ten opzichte van de standaard `Polynomial` klasse is de mogelijkheid deze polynoom te initialiseren met de samplepunten die deze polynoom moet benaderen. In essentie komt het erop neer dat de constructor van deze klasse ervoor zorgt dat de coëfficiëntenmatrix A en een descriptionmatrix D op de gewenste manier geïnitialiseerd worden. Dit gebruik makend van de verzameling samplepunten die de constructor als parameter meekrijgt. Dit alles wordt geïmplementeerd in de `init(Matrix PBasic, Matrix R, int m)` methode. De `PBasic` matrix bevat op elke rij een samplepunt en in elke kolom i de waarde die het desbetreffende punt bezit in dimensie i. De matrix `R` is een $[j \times 1]$

kolom matrix die in elke rij de effectieve runtime van samplepunt j bevat. In eerste instantie wordt de `descriptor` matrix opgesteld voor de meegegeven dimensie m. Deze definitie zal een rij $[i_1 \cdots i_n]$ bevatten voor elk mogelijk $x_1^{i_1} ... x_n^{i_n}$ product dat voldoet aan $\sum_{k=1}^{n} i_k \leq m$. Deze opdracht wordt vervuld door de `getdegreeNDescription` methode. Vervolgens wordt aan de hand van deze descriptor matrix de `PBasic` matrix omgezet naar een standaard P matrix, zoals beschreven in formule 10.26. Tot slot wordt de coëfficiëntenmatrix A berekend aan de hand van formules 10.30 $\rightarrow$ 10.37 gebruik makend van de `solve` methode uit de `Matrix` klasse gedefinieerd in het JAMA [55] pakket. Dit alles wordt duidelijker aan de hand van de volgende codefragmenten:

```
1  public MaxDegreeNPolynomial(Matrix Points, int degree){
2    super(null, null);
3    if(Points != null){
4      degree=Math.min(degree,getMaxDegree(Points.getRowDimension(),
           Points.getColumnDimension()));
5      Matrix PBasic = Points.copy();
6      double[][] Rarray = new double[PBasic.getRowDimension()][1];
7      for(int i =0; i<PBasic.getRowDimension(); i++){
8        Rarray[i][0]=PBasic.get(i,PBasic.getColumnDimension()-1);
9        PBasic.set(i,PBasic.getColumnDimension()-1,1);
10       }
11     Matrix R = new Matrix(Rarray);
12     init( PBasic, R, degree);
13   }
14 }
```

```
1  private void init(Matrix PBasic, Matrix R, int degree){
2    int amount = PBasic.getColumnDimension();
3    int counter =0;
4    double[] values = new double[amount];
5    int combinationamount = 1;
6    for(int k=1;k<=degree;k++){
7      combinationamount += (fac(amount-1 + k - 1)/(fac(k)*fac(amount-1
           -1)));
8    }
9    descriptionarray = this.getdegreeNDescription(degree,amount-1);
10   descriptionarray[amount-1][combinationamount-1]=1;
11   description = new Matrix(descriptionarray);
12   double[][] Parray = new double[PBasic.getRowDimension()][
         combinationamount];
13   for( int i = 0; i< PBasic.getRowDimension(); i++){
14     for( int j = 0; j< combinationamount; j++){
15       Parray[i][j]=1;
16       for( int l =0; l< amount ; l++){
17         Parray[i][j]*=Math.pow(PBasic.get(i,l),descriptionarray[l][j])
               ;
18       }
19     }
20   }
21   Matrix P = new Matrix(Parray);
22   coefficienten = P.solve(R);
23 }
```

10.5 Neurale Netwerken

10.5.1 Theoretische benadering

Neurale Netwerken (NN), soms ook *Artificiële Neurale Netwerken* (ANN) genoemd, zijn netwerken van artificiële neuronen. Het opzet van een dergelijk netwerk is het verwerken van informatie. De kracht van een dergelijk systeem zit meestal vervat in zijn adaptieve eigenschappen. Hiermee wordt bedoeld dat een NN zijn structuur kan aanpassen op basis van de informatie die het verwerkt. Dit adaptief proces wordt leren genoemd. De techniek die hiervoor gebruikt wordt, is gebaseerd op de werking van het brein. De ontwikkelaars van deze techniek hadden het idee dat het modelleren van hersenen in een computer, de mogelijkheid zou bieden de waardevolle eigenschappen van de hersenen te reproduceren. In het bijzonder werd er hier gedacht aan het maken van keuzen en van selecties. In dit onderzoek wordt gepoogd aan de hand van deze neurale netwerken een model op te bouwen van de parameter sweep job runtimes.

Neuron

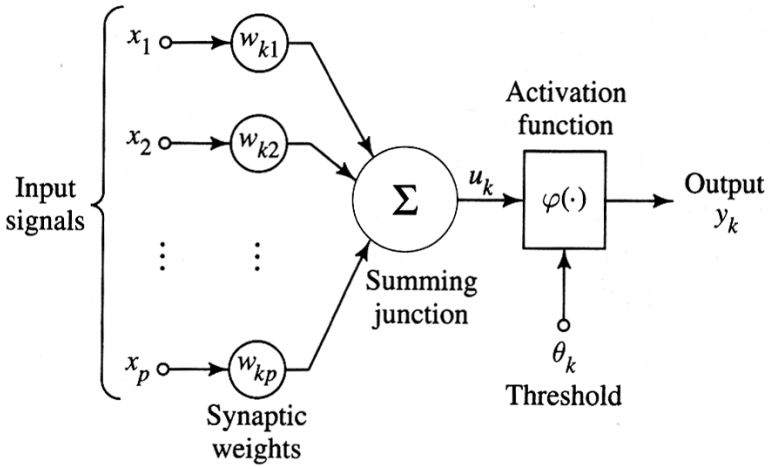
Neuronen zijn, zoals algemeen bekend, de bouwstenen van de hersenen. De basisfunctionaliteit van een biologisch neuron bestaat erin zijn invoer op te tellen en een uitvoer te produceren indien deze waarde groter is dan een drempelwaarde. De invoer van de neuronen is afkomstig van dendrieten die verbonden zijn met de uitvoer van andere neuronen, of prikkels ontvangen van het lichaam. Deze verbindingen, de synapsen genoemd, bepalen de sterkte van het signaal. Sommige synapsen zijn goed verbonden en behouden aldus de signaalsterkte, andere zijn slecht verbonden en verzwakken het signaal.

Een artificieel neuron, die deze functionaliteit wil ondersteunen, moet dus de volgende eigenschappen bezitten:

1. Elk van de invoersignalen moet een verschillende sterkte hebben.
2. De uitvoer van het neuron moet bepaald worden aan de hand van een sommatie van de sterkte van de invoersignalen.
3. Een drempelwaarde moet overschreden zijn alvorens er een uitvoersignaal is.

Deze drie eigenschappen worden in een neuron geïmplementeerd door:

1. De sterkte van de invoersignalen wordt gesimuleerd door elke input i met een gewicht w_i te vermenigvuldigen.
2. De sommatie gebeurt door het product van elk van deze invoersignalen met hun gewicht, bij elkaar op te tellen.
3. Het uitvoersignaal sturen door een drempelwaarde, gebeurt aan de hand van een activatiefunctie of transferfunctie.



Figuur 10.2: Artificieel neuron

Dit alles wordt duidelijk in figuur 10.2

Deze figuur toont dat de verschillende inputsignalen x_i van neuron k vermenigvuldigd worden met het gewicht $w_{k\ i}$ van de desbetreffende synaps. Deze waarden worden vervolgens bij elkaar opgeteld tot de waarde u_k. Het outputsignaal y_k wordt dan berekend aan de hand van de functiewaarde van de activatiefunctie φ met als input het verschil tussen u_k en de threshold θ_k.

Formeel ziet de beschrijving van een neuron er dan als volgt uit:

$$y_k = \varphi\left(u_k - \theta_k\right) \text{ met } u_k = \sum_{j=1}^{p} w_{kj} x_j \tag{10.38}$$

Dit kan ook geschreven worden als:

$$y_k = \varphi\left(v_k\right) \text{ met } v_k = \sum_{j=0}^{p} w_{kj} x_j \tag{10.39}$$

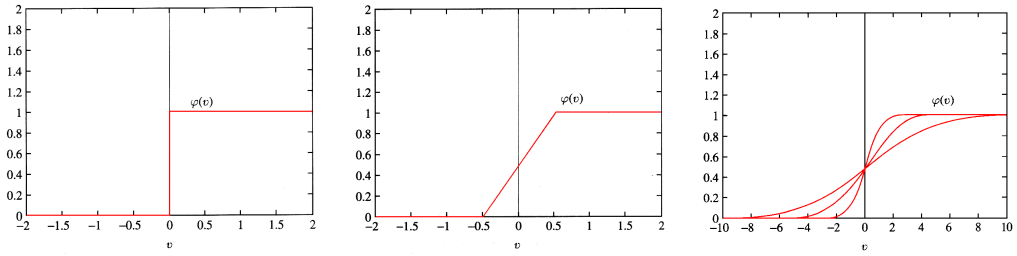
Deze wiskundige gelijkheid herformuleert het model van neuron k door er een nieuwe synaps aan toe te voegen met als invoer $x_0 = -1$ en gewicht $w_{k0} = \theta_k$

De activatiefunctie kan door verschillende functievoorschriften gedefinieerd worden. Een overzicht van de alternatieven is terug te vinden in tabel 10.5.1

De opbouw van een neuron brengt met zich mee dat zijn mogelijkheden beperkt zijn tot de classificatie van lineair separabele problemen. Het zal bijgevolg bijvoorbeeld niet mogelijk zijn een XOR probleem op te lossen. Het is niet mogelijk een lijn te trekken zodat $\{(0,0),(1,1)\}$ en $\{(1,0),(0,1)\}$ gescheiden zijn (zie figuur 10.4). Om hieraan het hoofd te bieden, worden in de volgende subsectie netwerken geïntroduceerd.

naam	functievoorschrift	figuur
Threshold functie	$\varphi(v) = \begin{cases} 1 \text{ als } v \geq 0 \\ 0 \text{ als } v < 0 \end{cases}$	10.3(a)
Dit is de standaard activatiefunctie. De neuronen die deze activatiefunctie gebruiken, worden soms ook McCulloch-Pitts neuronen genoemd. Omwille van het feit dat het soms noodzakelijk is over een output interval $[-1, 1]$ te beschikken, is het mogelijk dit output interval te verschuiven van $[0, 1]$ naar $[-1, 1]$.		
Stuksgewijs lineaire functie	$\varphi(v) = \begin{cases} 1 \text{ als } v \geq 1/2 \\ v \text{ als } 1/2 > v > -1/2 \\ 0 \text{ als } v \leq -1/2 \end{cases}$	10.3(b)
Deze functie ondergaat een lineaire overgang van de waarde 0 tot 1 op het interval $[-0.5, 0.5]$.		
Sigmoide functie	$\varphi(v) = \frac{1}{1+e^{-av}}$	10.3(c)
Dit is de meest gebruikte activatiefunctie. Parameter a definieert de helling van de functie. Door deze te variëren, onstaan er verschillende types sigmoide functies zoals te zien is in figuur 10.3(c). Wanneer een uitvoer interval $[-1, 1]$ gewenst is, kan de functie herschreven worden als $\varphi(v) = \frac{1-e^{-av}}{1+e^{-av}} = tanh\left(\frac{av}{2}\right)$.		

Tabel 10.2: Informatie omtrent verschillende activatiefuncties.



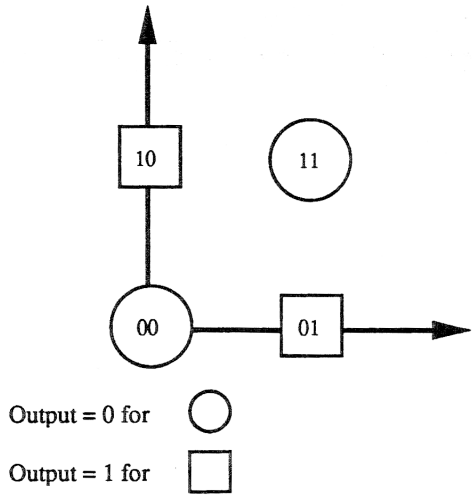
(a) Threshold functie (b) Stuksgewijs lineaire functie (c) Enkele sigmoide functies

Figuur 10.3: Plots van verschillende activatiefuncties.

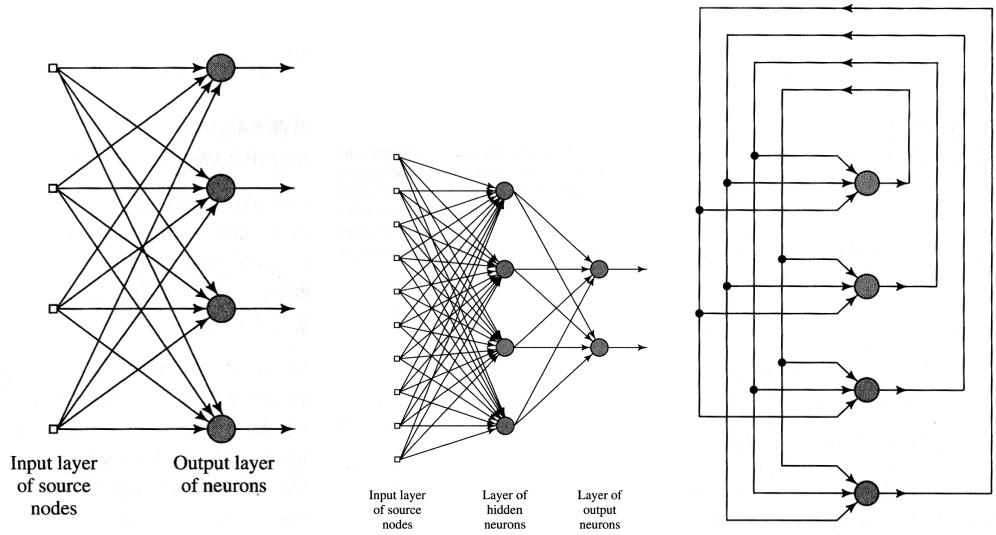
Netwerken

Om de beperkte functionaliteit uit te breiden, worden de input en de output van de verschillende neuronen met elkaar verbonden. Dit kan op verschillende manieren. Enkele voorbeelden zijn Single-layer feedforward netwerken (zie figuur 10.5(a)), Multilayer feedforward netwerken (zie figuur 10.5(b)), recurrente netwerken (zie figuur 10.5(c)), ...

In dit onderzoek wordt uitsluitend het Multi-layer feedforward netwerk gebruikt omdat dit aangewezen is om te modelleren.



Figuur 10.4: Voorstelling van de XOR ruimte



(a) Single-layer feedforward (b) Multi-layer feedforward (c) Recurrent

Figuur 10.5: Verschillende soorten netwerken

Trainen

De technieken die instaan voor het trainen van een netwerk kunnen opgedeeld worden in twee groepen.

- *Gesuperviseerd leren*: Deze groep van technieken werkt onder leiding van een leraar. Deze leraar zal het netwerk, bij het invoeren van een sample, het gewenste resultaat bezorgen. Aan de hand van de fout, die de output van het netwerk heeft ten opzichte van het gewenste resultaat, zal het netwerk aangepast worden. Het is de bedoeling dat op deze wijze het netwerk de kennis

van de leraar zo goed mogelijk overneemt. Eens deze kennis verworven is, kan het netwerk onafhankelijk opereren. Gesuperviseerd leren kan zowel offline als online gebeuren. Offline leren traint een netwerk, alvorens het in gebruik genomen wordt, met een verzameling beschikbare kennis. Op het moment dat deze kennis overgebracht is, kan het netwerk verder gebruikt worden. Bij online leren zal elke iteratie van het netwerk dit netwerk verder trainen. Het netwerk blijft als het ware bijleren. Het spreekt voor zich dat in dit concreet geval de reeds verworven runtimes de ideale bron van kennis zijn voor de leraar. Tevens is er voor offline learning gekozen omwille van synchronisatie redenen en omdat deze manier van werken toelaat de data meerdere keren door het netwerk te sturen alvorens het gebruikt wordt.

- *Niet gesuperviseerd leren*: In deze groep van technieken zullen de verschillende neuronen individueel handelen. Het opzet is het netwerk te laten reageren op bepaalde invoer. Het netwerk zal bijgevolg een specifiek gedrag vertonen bij bepaalde invoer waardoor er bijvoorbeeld bepaalde eigenschappen herkend kunnen worden (feature-detection). Aangezien deze technieken de toegevoegde waarde van parametersweep informatie niet gebruiken, zullen ze ook niet verder toegepast worden in dit onderzoek.

Bij een individueel neuron gebeurt het leerproces als volgt:

1. *Initialisatie*: $w_i(t) = 0$ voor elk inputsynapsis i.
2. *Activatie*: Selecteer het volgende inputpatroon $x(t)$ en de gewenste output $d(t)$.
3. *Neuron berekening*: Het neuron berekent de output.

$$y(t) = \varphi(w^T(t) \cdot (t)) \tag{10.40}$$

4. *Gewicht aanpassing*:

$$w(t+1) = w(t) + \eta\,(d\,(t) - y\,(t))\,x\,(t) \tag{10.41}$$

$$\text{met } d(t) = \begin{cases} +1 \text{ als } x \text{ tot klasse } A \text{ behoort.} \\ -1 \text{ als } x \text{ tot klasse } B \text{ behoort.} \end{cases} \tag{10.42}$$

   η stelt in dit geval de leersnelheid voor die aangeeft in welke mate de gewichten in elke stap aangepast moeten worden.

5. *Lus stap*: t wordt geïncrementeerd met 1 en het proces springt terug naar stap 2.

Hier dient nog bij opgemerkt te worden dat $x_0(t) = -1$ en $w_0(t) = \theta(t)$ steeds ongewijzigd blijven.

Bij een netwerk wordt de *Back propagation* trainingsmethodiek gehanteerd:

1. *Initialisatie*: De gewichten w en de thresholds θ worden met een kleine randomwaarde geïnitialiseerd.

2. *Forward pass*: Stel dat $[x(n), d(n)]$ de invoerwaarden en de gewenste outputwaarden zijn. De uitvoer van de neuronen wordt laag per laag berekend. De interne activiteit van neuron j in laag l wordt als volgt gedefinieerd:

$$v_j^l(n) = \sum_{i=0}^{p} w_{ji}^l(n) y_i^{l-1}(n) \tag{10.43}$$

met $y_i^{l-1}(n)$ de uitvoer van neuron i uit laag $(l-1)$ in iteratie n en $w_{ji}^l(n)$ het gewicht van de synaps ij. Dit is de synaps die neuron i in laag $(l-1)$ verbindt met neuron j in laag l. Stel dat neuronenlaag $(l = L)$ de uitvoerlaag is, dan is de fout $e_j(n)$ van het j-de neuron als volgt gedefinieerd:

$$e_j(n) = d_j(n) - y_j^L(n) \tag{10.44}$$

met $d_j(n)$ de gewenste j-de output (output van neuron j) van het netwerk.

3. *Backward pass*: De gradient waarde δ die de wijzigingen van de gewichten bepaalt, wordt als volgt per laag berekend:

   - voor neuron j in uitvoerlaag L:

$$\delta_j^L(n) = e_j^L(n) y_j^L(n) \tag{10.45}$$

   - voor neuron j in verborgen laag l

$$\delta_j^l(n) = \varphi_j'(v_j(n)) \sum_k \delta^{l+1}(n) w_{kj}^{l+1}(n) \tag{10.46}$$

     met $\varphi_j'(v_j(n))$ de afgeleide activatiefunctie.

   De gewichten in laag l worden dan volgens de delta regel als volgt aangepast:

$$w_{ji}^l(n+1) = w_{ji}^l(n) + \eta \delta_j^l(n) y_i^{l-1}(n) \tag{10.47}$$

   met η de leersnelheid

4. Het doorlopen van de forward en backward pas moet voor elk trainingsvoorbeeld herhaald worden tot de gewichten in het netwerk gestabiliseerd zijn en de fout tot een aanvaardbare waarde is gedaald. Elk van de trainingsvoorbeelden éénmaal doorlopen wordt een *epoch, leercycle* of *trainingscycle* genoemd.

10.5.2 Implementatie

De implementatie van de neurale netwerk predictor (NNPredictor) is net zoals de overige predictors een afleiding op de abstracte `Predictor` en `Predictor` factory klasse. Alvorens deze implementatie gerealiseerd werd is er een grondige studie gemaakt omtrent de verschillende softwarepaketten die de werking van neurale netwerken implementeren. Deze studie is terug te vinden in Appendix E. Het framework dat in deze vergelijking de beste punten scoorde was: *"Joone"* [58]. De initiële implementatie is dan ook op dit framework gebaseerd.

Concreet bestaat de code uit 7 stappen:

1. In de eerste stap wordt aan de hand van de inputparameters (aantal neuronen per layer, type neuronen in elke layer) de structuur van het netwerk opgebouwd (`NeuralNetFactory.create(...)`).
2. In de tweede stap worden de inputbronnen met het netwerk verbonden (`setupInput()`).
3. In de derde stap wordt de monitor (leraar) en de training geconfigureerd (aantal trainingscycles) (`setupLearning()` en `setupMonitor(...)`).
4. In de vierde stap wordt het netwerk gecheckt op fouten (`checknet()`).
5. In de vijfde stap wordt de training afgehandeld (`startLearning()`).
6. In de zesde stap wordt het getrainde netwerk omgevormd tot een predictor die gebruikt kan worden voor runtime predictie (`setupEvaluation()`).
7. In de daarop volgende stappen worden de runtimes van jobs opgevraagd aan de hand van de `Evaluate` methode.

Dit alles wordt duidelijk aan de hand van het volgende vereenvoudigde codefragment:

```
1  package be.ac.ua.comp.GIPSy.SilasNN;
2
3  import ... ;
4
5  public class NeuralNetPredictor extends Predictor implements
       NeuralNetListener{
6
7    public SilasNeuralNetPredictor(int cycles,..., double[][] data)
8    {
9      nnet = NeuralNetFactory.create(layerrows, layertypes);
10     setupInput();
11     setupLearning();
12     setupMonitor(cycles, learningrate, momentum, learner);
13     checknet();
14     startLearning();
15     setupEvaluation();
16   }
17
18   public void setupMonitor(int cycles, ... , String learner){...}
19
```

```
20    public void setupInput(){...}
21
22    public void setupLearning(){...}
23
24    public void checknet(){...}
25
26    public void startLearning(){...}
27
28    public void setupEvaluation() {...}
29
30    public double Evaluate(Matrix value)
31    {
32      value = value.getMatrix(0, value.getRowDimension() - 1, 0, value.
          getColumnDimension() - 2);
33      double[] inputdata = value.getArray()[0]; //evaluate first row
          only
34      double[] result = JooneTools.interrogate(nnet, inputdata);
35      return out;
36    }
37
38  }
```

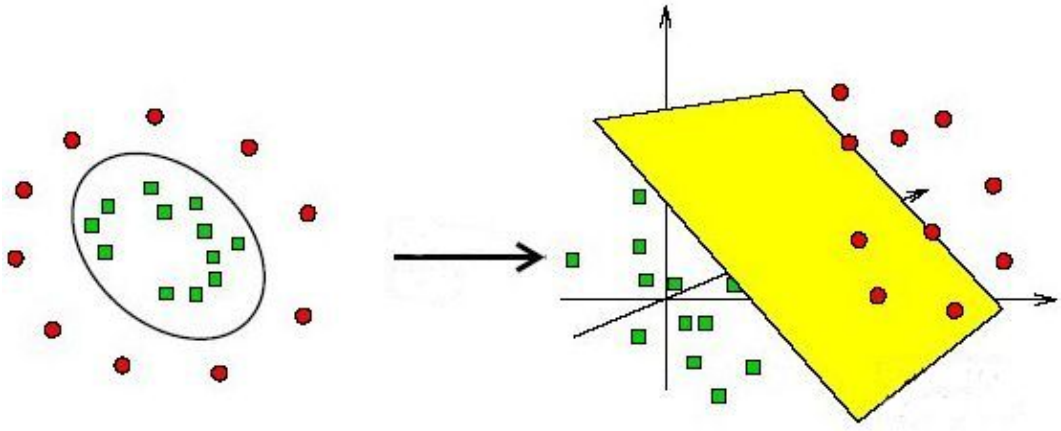
Na afloop van deze implementatie vielen de resultaten van deze implementatie echter onverwacht tegen. Dit zowel op het gebied van performantie als op het gebied van de accuraatheid van de predictie. Om te verifiëren of de gebruikte toolkit aan de oorzaak van deze problemen lag werd er besloten om dezelfde techniek alsnog in Matlab te implementeren. Om dit te verwezenlijken werd er binnen de Matlab toolbox een Jini service gestart die de communicatie tussen de GIPSy Java code en de toolbox verzorgde. Binnen de toolbox werd de Matlab Neural Network Toolbox [114] gebruikt om het neurale netwerk te construeren en te bevragen. Noch de performantie noch de accuraatheid van de predictor bleken hierdoor te verbeteren. Omwille van de onderlinge gelijkheid van deze twee implementaties en de meer gecompliceerde deployment methode van de Matlab implementatie, zullen in de verdere bespreking van de technieken enkel de resultaten van de Joone implementatie gebruikt worden.

10.6 Support Vector Machines

10.6.1 Theoretische benadering

Support Vector machines (SVM) is een techniek die op basis van regressie, functies kan benaderen. Het grote voordeel van SVM is dat het een soort van “black box” is die, onafhankelijk van de volgorde van de trainingsvoorbeelden, steeds zal evolueren naar een optimaal resultaat. Het basisprincipe van SVM is gebaseerd op *linear classifiers* wat later is uitgebreid naar *niet-linear classifiers*. Deze classifiers verdelen een ruimte in twee delen aan de hand van een hypervlak. Het spreekt voor zich dat lineaire classificatie steeds beperkt is tot eenvoudige invoerruimten. De classificatie van samplepunten, zoals wordt afgebeeld in de linkse plot van figuur

10.6, is immers onmogelijk. Om dit alsnog te verwezenlijken, maakt SVM gebruik van kernelfuncties. Deze functies zullen een n-dimensionale ruimte afbeelden op een m-dimensionale ruimte (met $m > n$) zodat de classificatie wel kan gebeuren aan de hand van zo'n hypervlak.



Figuur 10.6: Een lager dimensionale ruimte waarin lineaire classificatie niet mogelijk is wordt afgebeeld op een feature-space (hoger dimensionale ruimte) waarin lineaire classificatie wel mogelijk is.

Theoretisch ziet dit er dan als volgt uit. Stel een n-dimensionale ruimte χ en een verzameling samplepunten:

$$(x_1, y_1), \ldots, (x_m, y_m) \in \chi \times \{+1, -1\} \tag{10.48}$$

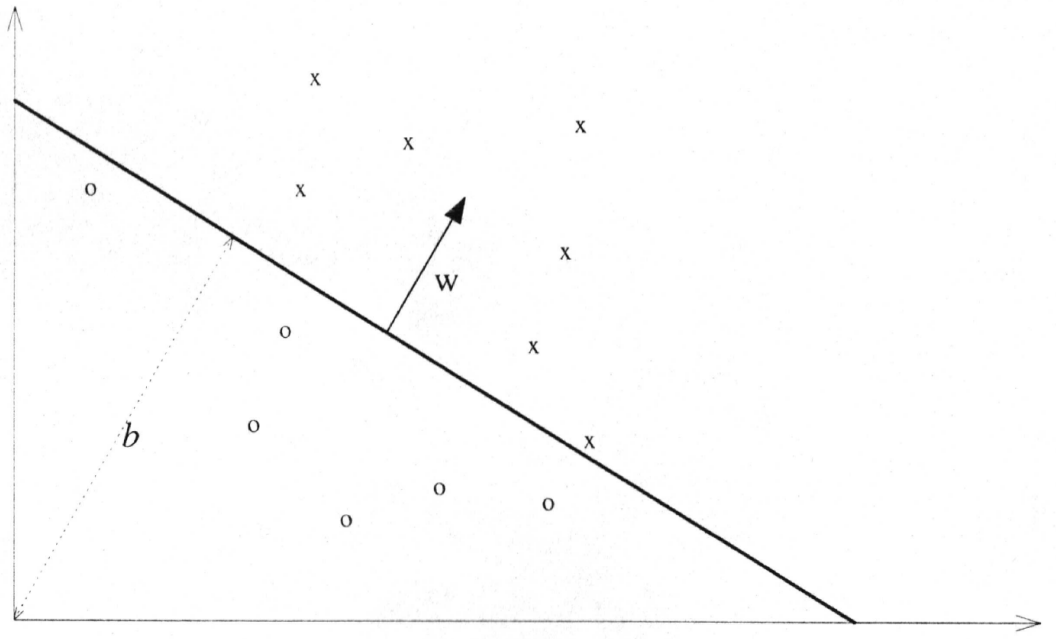
met x_i de invoerdata uit de verzameling χ en y_i de klasse -1 of 1 waartoe het samplepunt behoort. De bedoeling is nu een methode te ontwikkelen die voor de overige elementen uit verzameling χ bepaalt tot welke klasse ze behoren. De methode die hiervoor gebruikt wordt, ontwikkelt een functie die weergeeft in welke mate de punten, die voorspeld moeten worden, overeenkomen met de reeds gekende samplepunten uit χ. Om dit te bewerkstelligen, wordt een functie K gedefinieerd die de maat van overeenkomst tussen twee samplepunten bepaalt.

$$K : \chi \times \chi \to \mathbb{R} : (x, x') \mapsto K(x, x') \tag{10.49}$$

Hier zullen x en x' de twee gegeven invoerpatronen zijn en K de functie die de overeenkomst tussen deze twee patronen aanduidt. Een voorbeeld van een dergelijke functie is het dotproduct, dat als volgt wordt gedefinieerd:

$$\langle x, x' \rangle = \sum_{i=1}^{n} x_i x'_i \tag{10.50}$$

Aan de hand van deze overeenkomstfunctie K kan, zoals aangeduid in figuur 10.7, op een eenvoudige manier een hypervlak opgesteld worden om de lineaire classificatie



Figuur 10.7: Een classificatie aan de hand van een hypervlak(een rechte) in een 2-dimensionale ruimte

te bewerkstelligen. Het hypervlak f wordt als volgt gedefinieerd:

$$\begin{aligned} f(x) &= K(w,x) + b && (10.51)\\ &= \langle w, x\rangle + b && (10.52)\\ &= \sum_{i=1}^{n} w_i x_i + b && (10.53) \end{aligned}$$

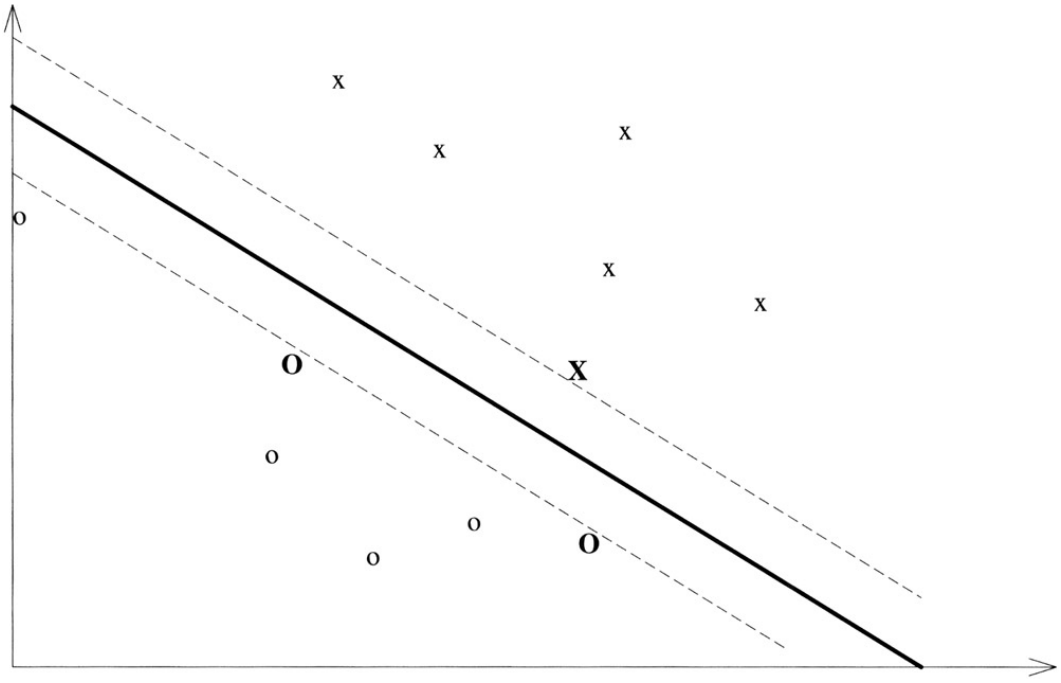
met w de normaal op het hypervlak en b de factor die de plaats van het hypervlak in de ruimte bepaalt.

Figuur 10.7 maakt duidelijk dat er verschillende hypervlakken gekozen kunnen worden. Het optimale vlak is echter dit vlak met de grootste afstand of *margin* tot de omliggende punten. Er wordt bijgevolg een *maximum margin classifier* gebruikt om de classificatie te bekomen met een maximale margin. Gebruik makend van de trainingsvoorbeelden x_j, zal deze als volgt gedefinieerd worden:

$$\max_{w\in\chi, b\in\mathbb{R}} min\left\{\|x - x_j\| \, x \in \chi, \langle w, x\rangle + b = 0, j = 1, \ldots, m\right\} \qquad (10.54)$$

De maximum margin classifier zal enkel afhankelijk zijn van die trainingsvoorbeelden die het dichtst bij het bekomen hypervlak liggen. Deze vectoren worden dan ook de *support vectors* genoemd. Deze worden aangeduid met de vette letters in figuur 10.8.

Aangezien, zoals reeds werd aangehaald, twee groepen niet steeds onderscheiden kunnen worden door een hypervlak (zie figuur 10.6), wordt een functie ϕ gedefinieerd



Figuur 10.8: Een hypervlak wordt bepaald aan de hand van de support vectors aangeduid met de vette letters.

die de invoerdata afbeeldt op een hoger dimensionale ruimte ϑ. Deze nieuwe ruimte wordt ook wel de feature-space genoemd. Deze functie ziet er als volgt uit:

$$\phi : \chi \to \vartheta : x \mapsto \phi(x) \tag{10.55}$$

De nieuwe overeenkomstfunctie K' wordt dan als volgt gedefinieerd:

$$K'(x, x') = K\left(\phi(x), \phi(x')\right) \tag{10.56}$$

De concrete functie, die in de meeste gevallen het beste resultaat haalt, is de zogenaamde "Radiale Basis Functie" die als volgt gedefinieerd wordt:

$$K'(x, x') = e^{-\gamma \|x - x'\|^2} \tag{10.57}$$

waarbij γ een schalingsfactor is die, afhankelijk van het gemodelleerde probleem, gekozen moet worden.

10.6.2 Implementatie

De implementatie van de SVM predictor start, zoals bij de overige predictors, bij het implementeren van de abstracte `Predictor` klasse. Zoals reeds werd aangehaald, is de support vector machine een soort blackbox techniek die, aan de hand van enkele inputparameters, de gewenste classificaties zal uitvoeren. Deze benodigde functionaliteit zit reeds vervat in de LibSVM [25] library. Deze library wordt dan ook gebruikt bij de opbouw van de predictor. Essentieel bestaat de werking van de predictor uit drie stappen:

- Het zetten van de benodigde parameters zoals het type kernelfunctie.
- Het trainen van het netwerk met de beschikbare samplepunten.
- Het voorspellen van de overige samplepunten aan de hand van de blackbox support vector machine.

10.7 Radiale basis functies Modellen

Theoretische benadering

In deze sectie worden de runtimes voorspeld door de runtime functie te benaderen aan de hand van radiale basis functies. Een radiale functie f is een functie die voldoet aan $f(x) = f(\|x\|)$. Deze functies rond de oorsprong kunnen eenvoudig herschreven worden als een functie $f'(x, c) = f(\|x - c\|)$ rond een centraal punt c. Aan de hand van deze techniek kan in elk samplepunt een dergelijke functie geconstrueerd worden. Gebruik makend van de juiste gewichten w_i kan ervoor gezorgd worden dat het model

$$y(x) = \sum_{i=1}^{N} w_i f(\|x - p_i\|) \tag{10.58}$$

de reeds gekende samplepunten p_i benadert. Dit heeft tot gevolg dat deze functie, naar alle waarschijnlijkheid, de runtimes ook zal benaderen in de niet gekende samplepunten.

In dit onderzoek worden testen gedaan met een selectie uit de groep van belangrijke radiale basis functies:

- De Gausische functie:
$$f(x) = e^{\left(-\frac{x^2}{s^2}\right)} \tag{10.59}$$
- De Multiquadric functie:
$$f(x) = \sqrt{x^2 + s^2} \tag{10.60}$$
- De Inverse (Reciprocal) Multiquadric functie:
$$f(x) = \frac{1}{\sqrt{x^2 + s^2}} \tag{10.61}$$
- De Epanechnikov Kernel functie [31][32]:
$$f(x) = \frac{3}{4}\left(1 - \left(\frac{x}{s}\right)^2\right) \tag{10.62}$$

De schaling van deze functies gebeurt steeds aan de hand van de schalingsfactor s.

Het verwerven van de gewichten w_i gebeurt, net zoals bij de polynomiale modellen, aan de hand van de *least squares* of *kleinste kwadraten* methode. De matrix P

uit formule 10.26 zal hier echter vervangen worden door een matrix waarin op elke rij i het gekende samplepunt i ingevoerd zal worden in de verschillende kernelfuncties. De j-de kolom in een dergelijke rij zal dan overeenkomen met de kernelfunctie die opgebouwd is aan de hand van samplepunt j. Concreet ziet dit er dan als volgt uit.

$$P = \begin{pmatrix} f_{p_1}(p_1) & f_{p_2}(p_1) & \cdots & f_{p_n}(p_1) \\ & & \vdots & \\ f_{p_1}(p_n) & f_{p_2}(p_n) & \cdots & f_{p_n}(p_n) \end{pmatrix} \tag{10.63}$$

Met f_p de radiale basis functie opgesteld rond het punt p en n het aantal gekenden samplepunten.

Implementatie

De implementatie gebruikt evenals de polynomiale predictor, het JAMA pakket om de gewenste matrix bewerkingen uit te voeren. De code werkt in 3 stappen:

- De nieuwe P matrix wordt opgesteld aan de hand van de `getP` procedure en van een `RBF` klasse.
- De gewichten w_i worden berekend aan de hand van de `solve` methode van de P matrix en de R matrix.
- Nieuwe voorspellingen gebeuren door formule 10.58 uit te voeren met de verkregen gewichten en met het te voorspellen samplepunt als input.

Dit alles wordt duidelijk aan de hand van het volgende codefragment:

```
1  package be.ac.ua.comp.GIPSy.Predictors.RBFPredictors;
2
3  import ... ;
4
5  public class RBFPredictor extends Predictor{
6
7    RBF rbf = null;
8    Matrix w;
9    double scale=1;
10
11
12   public RBFPredictor(Matrix Points, int n,RBF rbf, Matrix points,
         double scale){
13     this.rbf=rbf;
14     this.scale=scale;
15     adaptedPoints=Points.getMatrix(0,Points.getRowDimension()-1,0,
           Points.getColumnDimension()-2).getArrayCopy();
16     Matrix R=Points.getMatrix(0,Points.getRowDimension()-1,Points.
           getColumnDimension()-1,Points.getColumnDimension()-1);
17     Matrix P = getP(adaptedPoints,adaptedPoints);
18     w=P.solve(R);
19   }
20
21   Matrix getP(double[][] pointsA, double[][] pointsB){
22     double[][] returnvalue = new double[pointsA.length][pointsB.length
           ];
```

```
23      for(int i=0;i<pointsA.length;i++){
24        for(int j=0;j<pointsB.length;j++){
25          returnvalue[i][j]=rbf.evaluate(pointsA[i], pointsB[j],scale);
26        }
27      }
28      return new Matrix(returnvalue);
29    }
30
31    public double Evaluate(Matrix value) {
32      value=value.getMatrix(0, value.getRowDimension()-1, 0, value.
            getColumnDimension()-2);
33      Matrix P = getP(value.getArray(),adaptedPoints);
34      return Math.abs( P.times(w).get(0,0);
35    }
36
37  }
```

De implementatie van de verschillende radiale basis functies gebeurt door de RBF klasse uit te breiden. In deze uitbreiding moet enkel de `RBFimpl` methode geïmplementeerd worden. Deze verschillende implementaties van deze methode berekenen, aan de hand van een value en van een schalingsfactor, respectievelijk de formules 10.59, 10.60, 10.61 en 10.62.

10.8 Kriging Modellen

10.8.1 Theoretische benadering

De Kriging modelleringstechniek is vergelijkbaar met deze gebaseerd op de radiale basis functies. Het verschil zit in de plaats waar de radiale basis functies gebruikt worden. Waar deze, bij de RBF gebaseerde methode, werden gebruikt om een functie te creëren rond een samplepunt i, zullen ze bij kriging gebruikt worden om een functie te creëren rond elke dimensie j van een samplepunt i. Dit alles wordt duidelijk aan de hand van de formele beschrijving:

$$y(x) = \sum_{i=1}^{N} \left(w_i \prod_{j=0}^{dim} f(x_j - c_{ij}) \right) \tag{10.64}$$

De gebruikte kernelfuncties f zijn dezelfde als diegene beschreven in formules 10.59, 10.60, 10.61 en 10.62.

10.8.2 Implementatie

De implementatie zal, bijgevolg, ook bijna identiek ogen. Enkel de `getP` methode verschilt.

```
1   Matrix getP(double[][] pointsA, double[][] pointsB){
2     double[][] returnvalue = new double[pointsA.length][pointsB.length];
3     for(int i=0;i<pointsA.length;i++){
4       for(int j=0;j<pointsB.length;j++){
```

```
5           returnvalue[i][j]=K.evaluate(pointsA[i], pointsB[j],scales);
6       }
7   }
8   return new Matrix(returnvalue);
9 }
```

De kernel functie wordt hier toegepast in elke dimensie, in tegenstelling tot de RBF gebaseerde implementatie.

10.9 Naaste Buren Model

10.9.1 Theoretische benadering

Een naaste buren model is, zoals zijn naam al laat vermoeden, gebaseerd op de samplepunten die in de nabijheid van het te voorspellen punt gelokaliseerd zijn. Bij de voorspelling van een runtime worden de k dichtstbijzijnde samplepunten geselecteerd. Op basis van de runtimes en van de afstand van deze samplepunten tot het te voorspellen punt, wordt de runtimevoorspelling berekend. Deze voorspelling kan gebaseerd zijn op verschillende technieken. Naast de technieken van Smith [108] [106] en van Iverson [52][53], die terug te vinden zijn in de literatuur (zie sectie 10.2.3), worden in deze thesis enkele nieuwe technieken geïntroduceerd:

- *Gemiddelde waarde*: In dit geval wordt het gemiddelde genomen van de k dichtstbijzijnde punten. Stel dat N_i de verzameling is van de k punten x_j die het dichtst bij het te voorspellen samplepunt x_i liggen en stel dat $R(x_j)$ de effectieve runtime van job x_j is, dan wordt $P(x_i)$, de predictie in punt x_i, formeel beschreven als:

$$P(x_i) = \frac{\sum_{j=0}^{k} R(x_j)}{k} \tag{10.65}$$

- *Afstandsgemiddelde*: In dit geval wordt het gewogen gemiddelde van de k dichtstbijzijnde punten genomen. Het gewicht van een samplepunt wordt bepaald aan de hand van zijn afstand tot het te voorspellen punt. Stel $D_i(x_j)$ de afstand van een samplepunt $x_j = (x_j^1 ... x_j^n)$ tot het te voorspellen punt $x_i = (x_i^1 ... x_i^n)$ in een n-dimensionale samplespace, dan wordt deze afstand als volgt gedefinieerd:

$$D_i(x_j) = \sqrt{\sum_{l=1}^{n} \left(x_j^l - x_i^l\right)^2} \tag{10.66}$$

  en wordt de predicitie $P(x_i)$ als volgt gedefinieerd:

$$P(x_i) = \frac{\sum_{j=0}^{k} D_i(x_j) R(x_j)}{\sum_{j=0}^{k} D_i(x_j)} \tag{10.67}$$

  Het is belangrijk hierbij op te merken dat, binnen de algemene `PredictorFactory` klasse, nog extra functionaliteit voorzien is om de waarden van de

verschillende dimensies te normaliseren tussen 0 en 1, zodat de hierboven gebruikte afstanden niet langer afhankelijk zijn van de grootorde van de inputparameters.

- *Afstandsfunctie*: De voorspelling gebeurt ook hier aan de hand van een gewogen gemiddelde. Dit gemiddelde is echter niet recht evenredig met de afstand van het te voorspellen punt tot de k dichtsbijzijnde punten, maar wordt bepaald aan de hand van een functie op de afstand. Deze afstanden worden ook hier bepaald aan de hand van $D_i(x_j)$ uit formule 10.66. De gebruikte functies definiëren in principe de impact die een punt, op een bepaalde afstand $D_i(x_j)$, heeft op de predictie. Zoals in sectie 10.8 reeds werd besproken, zullen de verschillende kernelfuncties de ideale functies zijn om deze impact uit te drukken. In dit onderzoek werden de volgende kernelfuncties gebruikt:
  - Exponential kernel zie formule 10.59.
  - Multiquadric kernel zie formule 10.60.
  - Inverse multiquadric kernel zie formule 10.61.
  - Epanechnikov kernel zie formule 10.13.
- *Lokale Predictors*: De voorspelling gebeurt aan de hand van één van de standaard predictors die reeds beschreven zijn:
  - Iverson (Zie sectie 10.3).
  - Polynomiale modellen (Zie sectie 10.4).
  - Neurale netwerken (Zie sectie 10.5).
  - Support Vector Machines (Zie sectie 10.6).
  - Radiale basisfuncties (Zie sectie 10.7).
  - Kriging modellen (Zie sectie 10.8).

  Het verschil met de voorafgaande technieken is dat de voorspellingsfuncties niet langer op heel de dataset gedefinieerd worden, maar dat voor elke voorspelling een lokale voorspellingsfunctie wordt opgesteld aan de hand van de k dichtsbijzijnde punten. Zo kan de voorspelling bijvoorbeeld gebeuren door een polynomiale functie of een Kriging functie op te stellen aan de hand van de runtimes van de k dichtsbijzijnde punten. De voorspelling gebeurt dan door de functiewaarde te berekenen voor het te voorspellen samplepunt.

10.9.2 Implementatie

De implementatie bestaat uit twee onderdelen:

- De `kNNEvaluator` klasse: Deze klasse is afgeleid van de `Predictor` klasse en implementeert de interface van deze predictor.

```
1 package be.ac.ua.comp.GIPSy.Predictors.NNEvaluation;
2
3 import ...;
```

```
4
5  public class kNNEvaluator extends Predictor{
6
7    private Matrix points = null;
8    private ValueComposer valueComposer = null;
9    private int k;
10
11   public kNNEvaluator(Matrix points, int k,ValueComposer
         valueComposer){
12     this.points=points;
13     this.valueComposer = valueComposer;
14     this. k = k;
15   }
16
17   public double Evaluate(Matrix value){
18     for(int i = 0; i<points.getRowDimension(); i++){
19       - Bepaal de afstand van point i tot het punt gedefinieerd
             door value
20       - voeg dit punt met zijn afstand toe aan de valuecomposer
21     }
22     return valueComposer.getValue(value);
23   }
24
25 }
```

De taak van deze klasse bestaat erin de afstand van elk gekend samplepunt tot het te voorspellen punt te bepalen en dit toe te voegen aan zijn `ValueComposer` klasse. Tot slot wordt de voorspellingswaarde bekomen door deze bij de `ValueComposer` klasse op te vragen.

- De `ValueComposer` klasse: Deze klasse selecteert de k dichtstbijzijnde samplepunten en berekent, aan de hand van deze samplepunten ,de voorspelling. Om de diversiteit te ondersteunen, zal elk van de ValueComposers in principe een implementatie zijn van de abstracte `ValueComposer` klasse. De abstracte klasse ondersteunt hierin de selectie van de k dicthstbijzijnde punten en de implementatie voegt de gewenste predictieberekening toe. De `ValueComposer` klasse ziet er als volgt uit:

```
1  package be.ac.ua.comp.GIPSy.Predictors.NNEvaluation;
2
3  import ...;
4
5  public class kNNEvaluator extends Predictor{
6
7    private Matrix points = null;
8    private ValueComposer valueComposer = null;
9    private int k;
10
11   public kNNEvaluator(Matrix points, int k,ValueComposer
         valueComposer){
12     this.points=points;
13     this.valueComposer = valueComposer;
14     this. k = k;
```

```
15    }
16
17    public double Evaluate(Matrix value){
18      for(int i = 0; i<points.getRowDimension(); i++){
19        - Bepaal de afstand van point i tot het punt gedefinieerd
              door value
20        - voeg dit punt met zijn afstand toe aan de valuecomposer
21      }
22      return valueComposer.getValue(value);
23    }
24
25  }
```

Het is duidelijk dat de samplepunten, bij toevoeging, gesorteerd worden op hun afstand tot het te voorspellen punt. Tevens wordt ervoor gezorgd dat enkel de k dichtstbijzijnde punten worden bijgehouden. Om de gewenste technieken uit de inleiding te ondersteunen, zijn er vier types `ValueComposer` geïmplementeerd:

1. *Gemiddelde waarde*: Deze techniek wordt geïmplementeerd in de `MeanValueComposer`. De implementatie van de abstracte `getValue` methode is zeer eenvoudig. De methode implementeert formule 10.65 en bevat enkel een lus die de runtimes van de verschillende samplepunten optelt en deelt door het aantal samplepunten. Concreet ziet deze implementatie er als volgt uit:

```
1  public Double getValue(Matrix value){
2    Double returnvalue = 0.0;
3    for(int i =0; i< this.queuevalue.size();i++){
4      Matrix tmpmatrix = queuevalue.get(i);
5      returnvalue+=tmpmatrix.get(0,tmpmatrix.getColumnDimension
           ()-1);
6    }
7    return  returnvalue/=queuevalue.size();
8  }
```

2. *Afstandsgemiddelde*: Deze techniek wordt geïmplementeerd in de `WeightedMeanValueComposer`. De implementatie van de abstracte `getValue` methode is zeer eenvoudig. De methode implementeert formule 10.67 en bevat enkel een lus die, de runtimes van de verschillende samplepunten vermenigvuldigt met hun gewicht, optelt, en deelt door de som van de gewichten. De gewichten worden bepaald aan de hand van de afstand van het desbetreffende samplepunt tot het te voorspellen punt. Concreet ziet deze implementatie er als volgt uit:

```
1  public Double getValue(Matrix value) {
2    Double returnvalue = 0.0;
3    Double divider = 0.0;
4    for(int i =0; i< this.queuevalue.size();i++){
5      Matrix tmpmatrix = queuevalue.get(i);
6      Double weight = queuekey.get(i);
7      returnvalue+=weight * tmpmatrix.get(0,tmpmatrix.
           getColumnDimension()-1);
```

```
8       divider+=weight;
9     }
10    returnvalue/=divider;
11    return returnvalue;
12  }
```

3. *Afstandsfunctie*: Deze techniek wordt geïmplementeerd in de `KernelValueComposer`. Deze valuecomposer zal, net zoals de `WeightedMeanValueComposer`, voorspellingen maken aan de hand van een gewogen gemiddelde. De gewichten per samplepunt worden hier echter bepaald door een gekende en reeds geïmplementeerde kernelfunctie toe te passen op de afstandswaarden $D_i(x_j)$. De `KernelValueComposer` laadt de Kernelimplementatie, die gespecificeerd wordt in het configuratieobject, en past deze toe op de afstanden. Dit alles wordt duidelijk aan de hand van de concrete implementatie:

```
1   public Double getValue(Matrix value)
2   {
3     Double returnvalue = 0.0;
4     Double divider = 0.0;
5     // inladen van een kernelfunctie
6     Kernel kernel = (Kernel)Thread.currentThread().
          getContextClassLoader().loadClass(this.kernelpath).
          newInstance();
7     for(int i =0; i< this.queuevalue.size();i++)
8     {
9       Matrix tmpmatrix = queuevalue.get(i);
10      double[] vector1= value.getArrayCopy()[0];
11      double[] vector2= tmpmatrix.getArrayCopy()[0];
12      Double weight = kernel.evaluate(vector1, vector2, scales)
            ;
13      returnvalue+=weight * tmpmatrix.get(0,tmpmatrix.
            getColumnDimension()-1);
14      divider+=weight;
15    }
16    returnvalue/=divider;
17    return returnvalue;
18  }
```

4. *Lokale Predictors*: Deze techniek wordt geïmplementeerd in de `PredictorValueComposer`. Deze valuecomposer berekent de voorspelling van een job aan de hand van een predictor functie die wordt opgesteld, gebruik makend van de k dichtstbijzijnde punten. Gebruik makend van het standaard configuratiemechanisme (`PredictorDescriptor`, `PredictorFactoryloader`,...) wordt deze predictor geladen en geïnitialiseerd met de matrix die de k dichtstbijzijnde punten omvat. Een predictie gebeurt dan door deze predictor te evalueren in het te voorspellen punt. De concrete implementatie ziet er als volgt uit:

```
1   public Double getValue(Matrix value) {
2     //het path bevat de lokatie van de predictor configuratie
          file
```

```
3     PredictorDescriptor tmpdescriptor = new PredictionXMLParser
          ().parseDocument(path);
4     inputData data = new inputData();
5     // matrix bevat de k geselecteerde samplepunten.
6     data.addPoints(matrix.getArrayCopy());
7     PredictorFactory factory = PredictorFactoryLoader.
          getFactory(tmpdescriptor,data,null);
8     P= (Predictor)factory.getPredictor(tmpdescriptor);
9     return P.Evaluate(value);
10  }
```

10.10 Gecombineerde Modellen

10.10.1 Inleiding

Naast de implementatie van verschillende modelleringstechnieken zijn er binnen het framework tevens technieken ontwikkeld die verschillende modelleringstechnieken combineren:

- *MeanFunctionPredictor*: In dit geval wordt een voorspelling gedaan in een punt aan de hand van een aantal predictors. Het resultaat van deze predictor zal het gemiddelde zijn van deze voorspellingen (al dan niet mits het verwijderen van outliers).

- *SimpleOutLierReductionPredictor*: Deze predictor is niet zozeer een combinatie van verschillende predictors, maar doet een poging de outliers van een bestaande predictor te verwijderen. Dit gebeurt op basis van het min-max principe. Dit principe stelt dat voorspellingen van een model nooit de minimale of de maximale runtime waarde van de gebruikte samplepunten mogen overschrijden. Dit wordt verantwoord door het feit dat de gebruikte startgrid en sampleselector, waarop deze techniek wordt toegepast, steeds pogen zo snel mogelijk zowel de buigpunten als de minima en de maxima te selecteren. Indien een voorspelling niet in dit interval ligt, resulteert deze predictor in de minimale of de maximale runtime van de gekende samplepunten al naargelang welke waarde het nauwst aansluit bij de voorspelling.

- *NearestFunctionOutLierReductionPredictor*: In de lijst van predictors wordt op zoek gegaan naar de eerste predictor die een voorspelling doet, die binnen het minimum en het maximum van de gekende samplepunten ligt. Indien zulk een predictor niet wordt gevonden, resulteert de predictie, net zoals de *SimpleOutLierReductionPredictor*, in de minimale of de maximale runtime van de gekende samplepunten al naargelang welke waarde het nauwst aansluit bij de voorspelling.

10.10.2 Implementatie

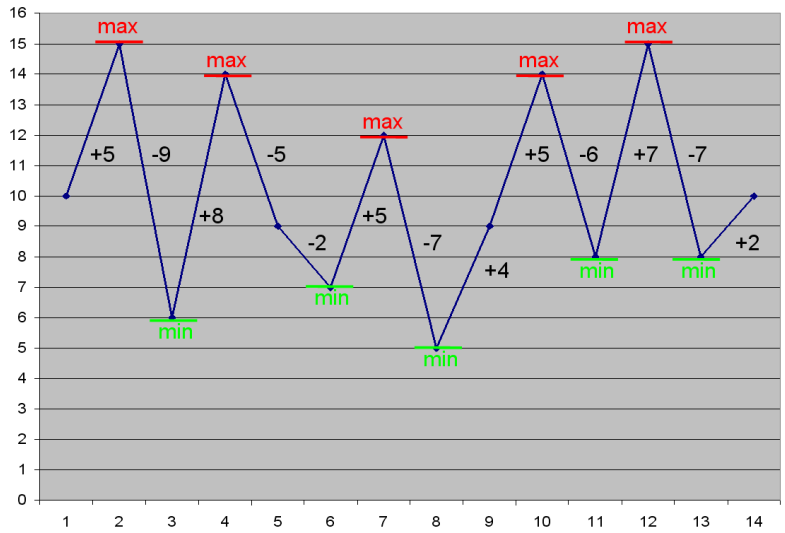
De implementatie is straightforward. Tijdens de constructie worden de verschillende onderliggende predictors aangemaakt en, indien nodig, worden de minimale en maximale waarden berekend. De evaluatie bestaat dan uit het evalueren van de onderliggende predictors en van het gebruik van de minimale en maximale waardes indien deze predictie niet binnen de gewenste grenzen valt.

10.11 Gesplitte Modellen

10.11.1 Inleiding

Bij de constructie van modellen bleek dat, in sommige gevallen, de invloed van een bepaalde dimensie volledig random verdeeld is. De impact van deze ruisfactor is enorm nadelig bij het construeren van een model en dient daarom omzeild te worden. Om dit mogelijk te maken, is er een mechanisme ontwikkeld dat deze ruis detecteert en vervolgens de modellen opsplitst in deze dimensie.

De detectie van de ruis gebeurt op basis van een eenvoudig principe. Ruis kenmerkt zich door het gebrek aan trends. Wanneer, via interpolatie, een functie door de samplepunten wordt getrokken, zal bijna elk punt een lokaal minimum of maximum zijn. Dit wordt duidelijk aan de hand van figuur 10.9.



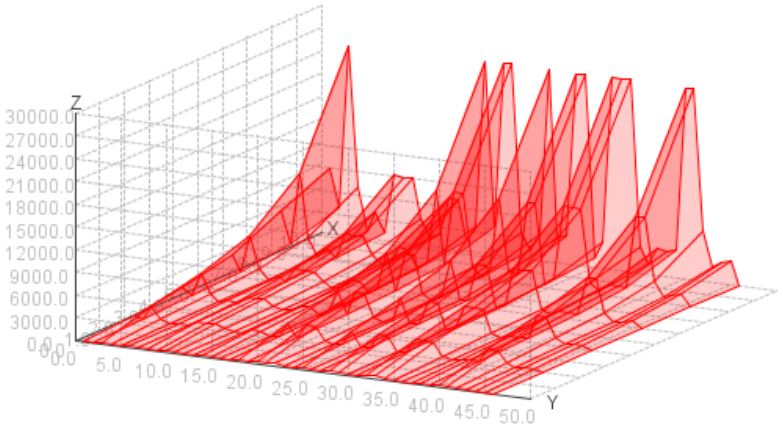
Figuur 10.9: Voorbeeld van de lokale minima en maxima in een interpolatie tussen ruisdata.

Proefondervindelijk werd vastgesteld dat er van een ruisfunctie kan gesproken

worden als:

$$\frac{\#\text{samplepunten}}{\#\text{lokale maxima} + \#\text{lokale minima}} \leq 4 \tag{10.68}$$

De opsplitsing van de predictors gebeurt door, voor elke waarde in de ruisdimensie, een individuele predictor te specificeren, die een model bouwt van de overige dimensies. Wanneer er meer ruisdimensies zijn, spreekt het voor zich dat deze techniek wordt toegepast voor elke mogelijke combinatie van inputwaarden in in elk van deze dimensies. Een mooi voorbeeld van de toepassing van een dergelijke techniek is terug te vinden in figuur 10.10.



Figuur 10.10: Een gesplit model van een inputruimte met ruisdimensie Y.

10.11.2 Implementatie

De implementatie bestaat uit twee onderdelen:

- Het eerste onderdeel zal de ruisdimensies detecteren. Bepalen of een dimensie i een ruisdimensie is, start bij het opstellen van de verzameling P_i. Stel P de verzameling van alle gekende samplepunten en stel A_i de verzameling van alle verschillende punten uit P wanneer dimensie i buiten beschouwing wordt gelaten:

  $$A_i = \{a = \{a_1, \ldots, a_{i-1}, a_{i+1}, \ldots, a_n\} \,|\exists p \in P, \forall j \neq i : a_j = p_j\} \tag{10.69}$$

  Stel dan P_{ia} de verzameling van alle punten p waarvan de waarde in elke dimensie $j \neq i$ overeenkomt met het punt a:

  $$P_{ia} = \{p \in P, \forall j \neq i | a_i = p_i\} \tag{10.70}$$

  Dan wordt P_i, de verzameling waarin voor elke $a \in A_i$ de overeenkomstige verzameling P_{ia} vervat zit, gedefinieerd als:

  $$P_i = \{P_{ia} | P_{ia} \subset P, a \in A_i\} \tag{10.71}$$

Na het opstellen van de verzameling P_i wordt voor elke groep punten P_{ia} in P_i het aantal minima en maxima bepaald. Dit zijn de samplepunten waarvan de runtime van het voorgaande en van het volgende punt, volgens dimensie i, beide respectievelijk groter of kleiner zijn dan de runtime van het huidige punt. Indien 1/4 of meer van de samplepunten een dergelijk minimum of maximum zijn, wordt de dimensie i als ruisdimensie beschouwd.

- De effectieve splitsing gebeurt aan de hand van een `SplittedPredictor`. Dit is een klasse die afgeleid is van de abstracte `Predictor` klasse en die voor elke mogelijk inputwaarde x_i van dimensie i een predictor aanmaakt van de vooropgestelde modelleringstechniek (Polynomiaal, Kriging, ...). Deze predictor krijgt, bij constructie, als input elk samplepunt met waarde x_i in de i-de dimensie. Deze waarde x_i wordt vanzelfsprekend verwijderd aangezien de groep submodellen een dimensie minder bestrijkt. Bij evaluatie van een punt p wordt de predictor, die overeenkomt met waarde p_i, geselecteerd om een voorspelling op te vragen. Ook hier wordt de waarde p_i verwijderd om evaluatie in de desbetreffende predictor mogelijk te maken.

HOOFDSTUK 11

Startgrid Selector

11.1 Inleiding

Zoals reeds werd aangehaald, brengt het gebrek aan voorkennis bij aanvang van een gedistribueerde applicatie met zich mee dat het, bij aanvang, moeilijk te beslissen is welke runtime informatie essentieel is om een goed model te bouwen. Toch blijkt uit onderzoek dat het gebruik van welbepaalde patronen in deze selectie van samples [85] tot betere resultaten leidt dan random sample selectie. Om aan te tonen dat het inpluggen van een dergelijke startgrid selectie techniek straightforward is, zijn er binnen dit framework reeds twee technieken geïmplementeerd.

11.2 Random Startgrid Selector

11.2.1 Theoretische benadering

De Random startgrid selector zal, zoals zijn naam al laat vermoeden, een arbitreerbaar aantal samples selecteren uit de volledige verzameling van uit te voeren jobs. Deze implementatie dient als referentie om eventuele vergelijkende studies te maken met meer geavanceerde startgridselectie technieken.

11.2.2 Implementatie

De implementatie gebeurt eenvoudig door de `StartGridSelector` klasse uit te breiden en de abstracte `getStartSamples()` methode te implementeren. Deze methode selecteert de samplepunten die als eerste uitgevoerd dienen te worden en plaatst de

desbetreffende punten in een indexlijst. In dit concrete geval zullen een arbitreerbaar aantal indexen random gegenereerd worden en in de lijst worden geplaatst. Het is hier wel belangrijk dat erop gelet wordt dat een index geen twee maal in de indexlijst terecht kan komen.

11.3 Basic Startgrid Selector

11.3.1 Theoretische benadering

De techniek die in deze basisversie van een sampleselector gebruikt wordt zal in elke dimensie al de uiterste punten selecteren. Visueel voorgesteld, kunnen deze punten gedefinieerd worden als de punten die samenvallen met een ballonwand wanneer deze strak rond de inputruimte gespannen wordt. Bij een kubusvormige inputruimte zullen dit bijvoorbeeld de samplepunten op de wanden van de kubus zijn. Stel P de verzameling van samplepunten $p = (p_1, \ldots, p_n)$ dan wordt de verzameling stargridpunten S formeel als volgt beschreven:

$$\begin{aligned} S = \{p : p \in P \& \exists i : p_i \quad &\in \quad \{\{\max a_i | a \in P \& \forall j \neq i : a_j = p_j\} \\ &\cup \quad \{\min a_i | a \in P \& \forall j \neq i : a_j = p_j\}\} \end{aligned} \tag{11.1}$$

Het voordeel van deze techniek is dat, zelfs indien de splittechnieken uit sectie 10.11 gebruikt worden, de uiterste punten van elk te construeren model gekend zijn. Dit brengt met zich mee dat de verschillende technieken steeds een interpolatie zijn en geen moeilijk voorspelbare extrapolatie.

11.3.2 Implementatie

De implementatie van zulk een selectie is niet straightforward en gebeurt op basis van een recursieve methode. Deze methode bepaalt voor elke dimensie de samplepunten met een minimum of een maximum waarde en roept deze methode vervolgens per dimensie opnieuw op, zonder deze dimensie en met de minimale en maximale punten van deze dimensie. Dit wordt recursief herhaald tot de methode wordt opgeroepen met slechts één dimensie. Al de punten die op dit niveau overblijven, worden geselecteerd. In pseudocode ziet dit er als volgt uit:

1. Indien de inputverzameling slechts één dimensie heeft of, slechts één punt bevat, return de gehele inputverzameling.
2. Zoniet lus over elke dimensie i van de input.
3. Bepaal de punten $p = \{p_1, ..., p_n\}$ met een minimum en een maximum in de huidige dimensie i en voeg deze toe aan lijst newinput.
4. Start de methode opnieuw in stap 1 met newinput als input en zonder de huidige dimensie i. De samplepunten in newinput zien er dan als volgt uit $p = \{p_1, \ldots, p_{i-1}, p_{i+1}, \ldots, p_n\}$.
5. Return de samplepunten die deze methode voor elke dimensie terug geeft.

HOOFDSTUK 12

De Sampleselector

12.1 Inleiding

Zoals reeds werd aangehaald in sectie 8.4, zal de set van beschikbare samplepunten een enorme invloed hebben op de kwaliteit van de geconstrueerde modellen. Het spreekt voor zich dat de volgorde waarin de jobs uitgevoerd worden en waarin bijgevolg hun runtimes beschikbaar worden het voornaamste stuurmiddel is van dit proces. Om onderzoekers de mogelijkheid te geven dit proces zelf te sturen, bestaat de mogelijkheid een eigen implementatie van deze selectie in het framework te pluggen. Een eenvoudige uitbreiding van de `SampleSelector` klasse en een kleine aanpassing van de configuratiefiles maken dit mogelijk. Om aan te tonen dat dit enerzijds effectief mogelijk en anderzijds ook nuttig is, zijn er een aantal implementaties binnen dit framework voorzien. De verschillende geïmplementeerde technieken, die in het vervolg van dit hoofdstuk worden besproken, tonen duidelijk aan dat een verschillende aanpak tot uiteenlopende resultaten kan leiden.

De sampleselectors die in dit hoofdstuk beschreven worden, kunnen in twee groepen onderverdeeld worden. De eerste groep bestaat uit de sampleselectors die de evolutie naar het optimale model nastreven. Hiermee wordt bedoeld dat er zo snel mogelijk getracht wordt de punten, die essentieel zijn voor de opbouw van een goed model, te selecteren. Deze punten komen in vele gevallen overeen met de punten waarin het model het meeste afwijkt van de te voorspellen waarden. De tweede groep is ontstaan uit de toepassing van GIPSy in een prediction based scheduler (Deel III van deze thesis). Hier is het streefdoel niet langer het optimale model maar wel de accuraatheid van de predicties van de geselecteerde punten. Het blijkt immers dat de globale afwijking van de geselecteerde punten geminimaliseerd wordt wanneer telkens de punten, waarvan verwacht wordt dat ze minimaal afwijken van

de reële waarde, geselecteerd worden. De invloed die de kwaliteit van het globale model, bij gebruik van deze selectiemethode, heeft op de globale afwijking blijkt minder krachtig te zijn dan de selectietechniek op zich. Hier dient bij opgemerkt te worden dat de selectieprocedures van de beide groepen elkaars inversen zijn. De selectieprocedures zijn bijgevolg bijna identiek, enkel de volgorde van de punten dient geïnverteerd te worden. Omwille van deze reden worden deze twee sampleselectors telkens samen besproken. Tabel 12.1 geeft de sampleselectors van de eerste groep weer met hun inversen uit de tweede groep.

Model optimalisatie	Predictie optimalisatie
LargestError	SmallestError
MultiPredictorLargestError	MultiPredictorSmallestError
LargestDistance	SmallestDistance
LargestDistanceAndError	SmallestDistanceAndError
SelectionNeighbourPenalisation	-
Fluctuation	NoFluctuation

Tabel 12.1: Tabel van de model optimalisatie sampleselectors en hun inverse sampleselctors die de runtime predictie van de geselecteerde punten optimaliseert.

12.2 RandomSampleSelector

12.2.1 Theoretische benadering

De Random sampleselector zal, zoals zijn naam al laat vermoeden, een arbitreerbaar aantal samples selecteren uit de volledige verzameling nog uit te voeren jobs. Deze implementatie dient als referentie om eventuele vergelijkende studies te maken met meer geavanceerde sample selectie technieken.

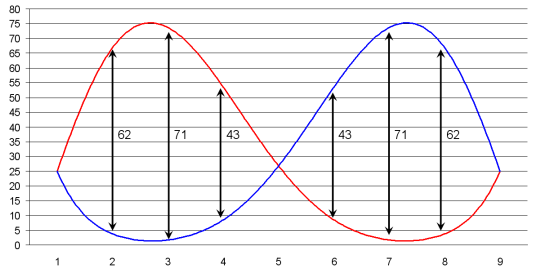
12.2.2 Implementatie

De implementatie gebeurt eenvoudig door de `SampleSelector` klasse uit te breiden. Het is bij de implementatie van een sampleselector immers voldoende voor elk samplepunt een gewicht in de globale `weights` lijst te plaatsen. De `SampleSelector` basisklasse zorgt er dan automatisch voor dat de samples met de grootste gewichten geselecteerd worden. In het geval van random sampleselectie komt het er bijgevolg op neer dat er in de initialisatie procedure voor elk samplepunt een random waarde in deze weights lijst gezet wordt.

12.3 ErrorSampleSelector

12.3.1 Theoretische benadering

Deze sectie beschrijft de LargestErrorSampleSelector & de SmallestErrorSampleSelector. Ze zijn gebaseerd op een extra model. De runtime predicties voor elk samplepunt worden vergeleken met de evaluatie van dit extra model in de desbetreffende samplepunten. De punten waarin deze twee voorspellingen het meest/minst van elkaar verschillen, worden geselecteerd en krijgen het grootste gewicht. In de LargestErrorSampleSelector worden de gewichten gelijk gesteld met de absolute waarde van dit verschil, bij de SmallestErrorSampleSelector is dit gewicht bijgevolg de negatie van deze absolute waarde. Zie figuur 12.1.



Figuur 12.1: Gewichten bepaling door verschil tussen twee predictor curves.

12.3.2 Implementatie

De implementatie is straightforward. De `SampleSelector` klasse wordt uitgebreid en de `init` methode wordt geherdefinieerd. In deze methode wordt elk punt geevalueerd met zowel de huidige predictor als met de extra predictor. Deze absolute waarde van de verschillen tussen deze evaluaties, of de negatie hiervan, worden in de `weights` lijst geplaatst.

12.4 MultiPredictorErrorSampleSelector

12.4.1 Theoretische benadering

Deze sectie beschrijft de MultiPredictorLargestErrorSampleSelector & de MultiPredictorSmallestErrorSampleSelector. Zoals de naam van de sampleselector al laat vermoeden zal deze sampleselector gebruik maken van meerdere extra predictors. Elk van de nog uit te voeren samplepunten zal geëvalueerd worden in elk van deze predictors. Elk samplepunt krijgt de gemiddelde waarde van de verschillen tussen deze predicties als gewicht. Het spreekt voor zich dat deze selector nog verbeterd kan worden door de outliers uit deze predicties te filteren. Er zijn tevens mogelijkheden om alternatieve implementaties te ontwikkelen. In sommige situaties kan

het immers raadzaam zijn de minimale, de maximale of de mediaan waarde van de verschillen tussen deze predicties te gebruiken als gewicht.

12.4.2 Implementatie

De implementatie is analoog aan deze van de LargestErrorSampleSelector. In de geherimplementeerde `init` methode wordt elk samplepunt geëvalueerd met elke predictor. Voor elk punt wordt dan elk verschil tussen twee predictors berekend. Van deze waardes wordt vervolgens het gemiddelde genomen.

12.5 DistanceSampleSelector

12.5.1 Theoretische benadering

Deze sectie beschrijft de LargestDistanceSampleSelector & de SmallestDistanceSampleSelector. Intuïtief is het vanzelfsprekend dat punten in een model, die in de omgeving liggen van de samplepunten waarmee het model is opgebouwd, naar alle waarschijnlijkheid het best voorspeld kunnen worden. De accuraatheid van een model dreigt immers te verminderen op locaties ver van de informatie die het model ondersteunt. DistanceSampleSelectors gaan dan ook op zoek naar die punten waarvan de afstand tot de nabijgelegen gekende samplepunten het grootst/kleinst is. De afstand tot de x meest nabijgelegen gekende samplepunten, of de negatie hiervan, worden hier gebruikt als gewicht. Het aantal samplepunten x wordt door de gebruiker bepaald bij initialisatie (via het configuratie mechanisme uit sectie 9.4.1). Hier dient nog bij opgemerkt te worden dat er, in het geval van splitted predictors, rekening wordt gehouden met het feit dat punten, die aan elkaar grenzen in een gesplitte dimensie, niet als buren beschouwd kunnen worden. Ze hebben immers geen invloed op elkaar.

12.5.2 Implementatie

De implementatie gebeurt, evenals bij de vorige sampleselectors, door de herimplementatie van de `init` methode. Voor de afstandsbepaling kan gebeuren, moet de inputdata eerst genormaliseerd worden. Het is immers mogelijk dat de afstand tussen twee opeenvolgende samplepunten in dimensie i groter is dan in dimensie j. Voor elk punt wordt de waarde in elke dimensie i verminderd met de minimale waarde in die dimensie en vervolgens gedeeld door het verschil tussen de minimale en de maximale waarde in die dimensie. Vervolgens worden, brute force, de x meest nabijgelegen gekende samplepunten gedetecteerd. De afstand van elk punt tot al de gekende samplepunten wordt berekend en de x meest nabij gelegen punten worden geselecteerd. De som van deze afstanden, of de negatie ervan, worden vervolgens gebruikt als gewicht.

12.6 DistanceAndErrorSampleSelector

12.6.1 Theoretische benadering

Deze sectie beschrijft de LargestDistanceAndErrorSampleSelector & de SmallestDistanceAndErrorSampleSelector. Deze sampleselector is, in essentie, de combinatie van de ErrorSampleSelector uit sectie 12.3 en van de DistanceSampleSelector uit sectie 12.5. Aan de hand van een $error factor$ en van een $distance factor$ wordt de invloed van de error en van de distance op het totale gewicht gedefinieerd. Beide factoren worden door de gebruiker bepaald bij initialisatie (via het configuratie mechanisme uit sectie 9.4.1).

12.6.2 Implementatie

De implementatie gebeurt, evenals bij de vorige sampleselectors, door de herimplementatie van de `init` methode. De implementatie kwam tot stand door de combinatie en de refactoring van de code uit de ErrorSampleSelectors en van de code uit de DistanceSampleselectors. Deze refactoring was nodig om de lussen uit beide predictors samen te voegen om aldus de computationele vereisten te minimaliseren.

12.7 FluctuationSampleSelector

12.7.1 Theoretische benadering

Deze sectie beschrijft de FluctuationSampleSelector & de NoFluctuationSampleSelector. Het spreekt voor zich dat, bij de constructie van een model meer samplepunten nodig zijn in regio's waar de data sterk fluctueren. Hierbij kan ook gesteld worden dat de voorspellingen van punten in deze regio's minder accuraat zijn. Dit maakt dat het enerzijds essentieel is meer samplepunten in die regio's te selecteren wanneer getracht wordt een optimaal model te creëren (FluctuationSampleSelector) en dat het anderzijds raadzaam is punten uit deze regio's te mijden indien getracht wordt de predicties van deze punten te optimaliseren(NoFluctuationSampleSelector). Om dit te bewerkstelligen zal de FluctuationSampleSelector techniek voor elke predictie op zoek gaan naar de naburige x gekende samplepunten en de som van het verschil tussen de runtimes van deze samplepunten als maat nemen voor de fluctuatie. Deze fluctuatiemaat kan dan als gewicht gebruikt worden om de juiste selectie te maken. Het aantal samplepunten x wordt door de gebruiker bepaald bij initialisatie (via het configuratie mechanisme uit sectie 9.4.1)

12.7.2 Implementatie

De implementatie gebeurt, evenals bij de vorige sampleselectors, door de herimplementatie van de `init` methode. Het bepalen van de buren gebeurt analoog aan de techniek uit de DistanceSampleSelector (sectie 12.5). De fluctuatie wordt berekend door de runtime verschillen tussen de verschillende buren bij elkaar op te tellen. Deze fluctuatiemaat wordt dan tot slot per punt in de `weights` lijst geplaatst.

12.8 SelectionNeighbourPenalisationSelector

12.8.1 Theoretische benadering

In tegenstelling tot de voorgaande sampleselectors kan deze enkel toegepast worden voor model optimalisatie doeleinden. Proefondervindelijk werd vastgesteld dat de voorgaande technieken, die gebruik maken van extra modellen (ErrorSampleSelector, MultiPredictorErrorSampleSelector, DistanceAndErrorSampleSelector en FluctuationSampleSelector), goed functioneren binnen hun toepassingsdomein wanneer tijdens elke selectieronde slechts één nieuw samplepunt geselecteerd wordt. In reële toepassingen zal een dergelijk gebruik van de software echter zelden of nooit voorkomen. In de meeste gevallen worden de punten in batches van enkele tientallen tot enkele honderdtallen opgevraagd. Bij het selecteren van batches van punten dreigen groepen van punten uit dezelfde omgeving geselecteerd te worden. Deze clusterselectie wordt veroorzaakt door het feit dat afstand tussen twee curves bij punten in dezelfde omgeving vergelijkbaar is. Als twee of meerdere modellen in het punt p het verst van elkaar liggen, dan zullen de punten, waar deze afstand het tweede, derde, vierde, ... grootst is, naar alle waarschijnlijkheid buren zijn van het punt p. Deze punten dienen echter niet geselecteerd te worden aangezien de aanwezigheid van p in het volgende model er, naar alle waarschijnlijkheid, voor zal zorgen dat de predictie in deze punten ook drastisch zal verbeteren.

In een eerste poging dit probleem te vermijden, werd de SelectionNeighbourPenalisationSelector ontwikkeld. Deze sampleselector zal, net zoals de DistanceAndErrorSampleSelector, op zoek gaan naar een evenwicht tussen de fout in een punt en de afstand van dit punt tot de reeds gekende samplepunten. In deze selector zullen de gewichten echter niet langer opgesteld worden op basis van een waarde, die een maat is voor de afstand en voor de error, maar zal een selectieprocedure doorlopen worden. Het gewicht van het punt wordt bepaald door het volgnummer van de selectieronde waarin het geselecteerd wordt. Het punt, geselecteerd in de eerste ronde, krijgt het grootste gewicht, het laatst geselecteerde punt krijgt het laagste gewicht. De selectie gebeurt op basis van een waarde die wordt opgesteld aan de hand van de fout en van de afstand tot de naburige geselecteerde punten (deze berekening is identiek aan de gewichtsberekening uit de DistanceAndErrorSampleSelector). Het punt met de grootste waarde wordt geselecteerd en krijgt het gewicht dat overeenkomt met het aantal selectieronden − de index van de huidige selectieronde. Na deze selectie worden alle buren van het geselecteerde punt gedetecteerd en wordt hun gewicht gedeeld door een penalisation factor. Deze factor wordt bij initialisatie bepaald door de gebruiker en zorgt ervoor dat de naburige punten minder snel geselecteerd worden.

12.8.2 Implementatie

De berekening van de error en van de distance waarden gebeurt analoog aan de DistanceAndErrorSampleSelector. Naast deze functionaliteit wordt een lus toegevoegd

die telkens het samplepunt met de hoogste error-distance waarde selecteert en de buren van dit punt penaliseert door deze error-distance waarde te delen door de penalisation factor.

12.9 Universele ClusterPrevention

Naast de SelectionNeighbourPenalisationSelector, die bewees dat het oplossen van het clusterprobleem positieve gevolgen had op de predictie, is er een algemene oplossing ontwikkeld die de problematiek oplost in elk van de betrokken sampleselectors (ErrorSampleSelector, MultiPredictorErrorSampleSelector, DistanceAndErrorSampleSelector en FluctuationSampleSelector). In deze oplossing wordt de selectieprocedure uit de SelectionNeighbourPenalisationSelector overgenomen. De gewichten worden bijgevolg punt per punt berekend. De penalisation techniek wordt hier echter vervangen door een andere techniek. Deze techniek simuleert dat de punten, geselecteerd in de voorafgaande selectieronden, reeds tot de gekende samplepunten behoren. Deze toevoeging heeft twee gevolgen die de clusterproblematiek vermijden:

- De afstand tot een punt p zal ook in rekening worden gebracht in de distancefactor van de overige punten. De buren van punt p zullen bijgevolg dicht bij een samplepunt liggen. Dit resulteert in een gevoelige verlaging van hun distancefactor.

- Elk van de betrokken modellen wordt opnieuw geconstrueerd, rekening houdend met de voorspelde informatie van het geselecteerde punt p. Dit simuleert dat de predicties van de verschillende modellen in de buren van punt p, naar alle waarschijnlijkheid, dicht bij elkaar zullen liggen wanneer de informatie in punt p effectief gekend is. De errorfactor in deze punten zal bijgevolg gevoelig verlagen. Hier dient wel bij opgemerkt te worden dat een lokale toevoeging van informatie in sommige modelleringstechnieken, zoals polynomen, globale gevolgen kan hebben op het model. In deze gevallen is deze techniek niet toepasbaar.

De daling van zowel de distance als de van errorfactor in de nabij gelegen punten, vermijdt eventuele clustering zonder onoverkomelijke invloed uit te oefenen buiten de omgeving van punt p.

HOOFDSTUK 13

De Kwaliteitscontrole

13.1 Inleiding

Zoals reeds eerder werd besproken, is het bepalen van de kwaliteit van een model essentieel bij het modelleren. Deze kwaliteitscontrole gebeurt aan de hand van de fouten die op een model kunnen gedefiniëerd worden. In essentie komt het erop neer dat er wordt bepaald in welke mate een model afwijkt van het te bereiken model. Alvorens de concrete implementatie van de huidige kwaliteitscontrole wordt besproken, is het belangrijk eerst de verschillende foutmaten te bespreken. Meer informatie is terug te vinden in de doctoraatsthesis van Wouter Hendrickx die later dit jaar zal verschijnen.

Het vervolg van dit hoofdstuk ziet er dan ook als volgt uit. In sectie 13.2 wordt een theoretisch overzicht gegeven over de verschillende soorten foutmaten die gebruikt kunnen worden tijdens de kwaliteitscontrole. In sectie 13.3 wordt de tot dusver enige implementatie van de kwaliteitscontrole component beschreven (de `BasicErrorCalculator`). Tot slot volgt in sectie 13.5 een korte reflectie over de huidige implementatie.

13.2 Foutmaten

Een model M kan in deze context beschouwd worden als een functie die een set van parameters (een samplepoint) afbeeldt op een reële waarde (de runtime predictie):

$$M : \mathbb{R}^N \rightarrow \mathbb{R} : (x_1, ..., x_N) \mapsto M(x_1, ..., x_N) \tag{13.1}$$

Op de zelfde manier kunnen de effectieve runtime waarden van de punten uitgezet worden op een functie R die een samplepunt afbeeldt op zijn effectieve runtime.

$$R : \mathbb{R}^N \rightarrow \mathbb{R} : (x_1, ..., x_N) \mapsto R(x_1, ..., x_N) \tag{13.2}$$

De fouten[1] van een Model M kunnen bestudeerd worden op twee verschillende niveaus.

- *Lokale kwaliteitscontrole*: De lokale kwaliteitscontrole bekijkt de errorwaarden die verkregen worden in elk van de samplepunten en wordt verder besproken in sectie 13.2.1.
- *Globale kwaliteitscontrole*: De globale kwaliteitscontrole bestudeert de kwaliteit van het volledige model, gebaseerd op de individuele waarnemingen in elk samplepunt. Dit wordt verder besproken in sectie 13.2.2.

Elk van deze controlemechanismen wordt gebruikt in het huidige framework. Op het niveau van de sampleselectie zal, aan de hand van de lokale kwaliteitscontrole, bepaald worden wat de kwaliteit is van het model in elk punt, om aldus een selectie te maken van de punten waarvan de runtime data het meest noodzakelijk zijn om het huidige model te verbeteren.

De onderlinge verwantschap van deze twee niveaus wordt als volgt aangetoond. Stel dat E^l de errorwaarde is die een maat is voor de lokale kwaliteit van een model in een punt x_k, dan kan dit als volgt geschreven worden:

$$E^l : \mathbb{R} \times \mathbb{R} \rightarrow \mathbb{R} : (M(x_k), R(x_k)) \mapsto E^l(M(x_k), R(x_k)) \tag{13.3}$$

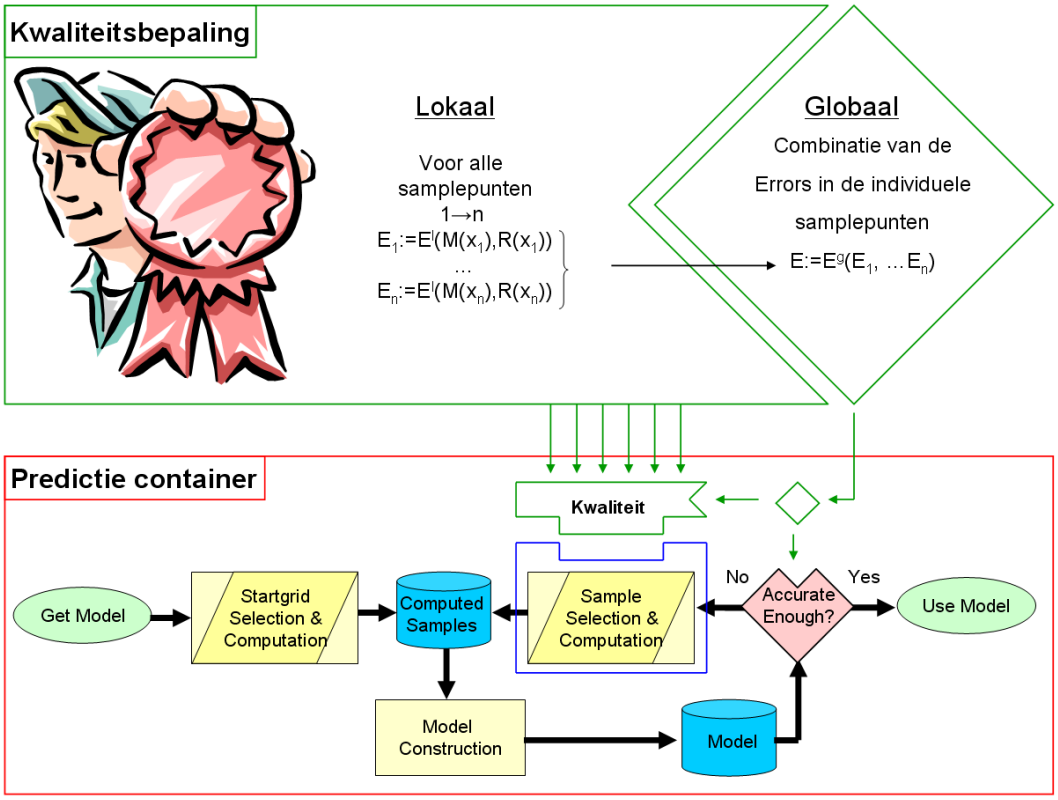
Wanneer nu een globale kwaliteitsanalyse moet gebeuren, dan zal deze gebaseerd moeten zijn op de verschillende lokale kwaliteitsverschijnselen. In essentie komt het erop neer dat de globale error E^g steeds zal afhangen van de lokale errors E^l in elk van de samplepunten. Stel dat de volledige parametersweep uit N samplepunten bestaat, dan wordt de maat van de globale kwaliteit uitgedrukt met een errorwaarde E^g die er als volgt uitziet:

$$E^g : \mathbb{R}^n \rightarrow \mathbb{R} : \left(E^l(M(x_k), R(x_k))\right)_{k=1}^{N} \mapsto E^g\left(\left(E^l(M(x_k), R(x_k))\right)_{k=1}^{N}\right) \tag{13.4}$$

Het komt er in principe op neer dat de globale error E^g berekend wordt door een combinatiemethode toe te passen op de lokale errorwaarde E^l van elk punt in de parametersweep.

Figuur 13.1 schetst de manier waarop deze lokale en globale kwaliteitscontroles binnen het framework passen. Het wordt hier duidelijk dat de lokale controle zal gebruikt worden binnen de sampleselector om zo te bepalen welke informatie het best vergaard kan worden in de volgende stappen. De globale kwaliteitscontrole wordt dan weer gebruikt om te beslissen of het huidige model accuraat genoeg is en of er aldus geen modellen meer gebouwd moeten worden. Hier dient nog bij vermeld te worden dat, zoals figuur 9.3 aangeeft, zowel de lokale als de globale kwaliteitsbepalingen ook gebruikt worden als feedback voor de GIPSyTester.

[1] De verschillen met de effectieve runtime functie R.



Figuur 13.1: Deze figuur toont de plaats van de globale en lokale kwaliteitscontroles binnen het plugable framework.

13.2.1 Lokale kwaliteitscontrole

Traditioneel bestaan er drie belangrijke types lokale errors in de literatuur, ook wel puntsgewijze fouten (pointwise errors) genoemd:

- *De absolute fout*: Geeft de absolute waarde van het verschil tussen twee waarden in dit punt weer. In dit concreet geval het verschil tussen de gemodelleerde waarde en de correcte waarde.

$$E^l_{abs} : \mathbb{R} \times \mathbb{R} \rightarrow \mathbb{R} : (M(x_k), R(x_k)) \mapsto |M(x_k) - R(x_k)| \tag{13.5}$$

- *De relatieve fout*: De relatieve fout schaalt de fout ten opzichte van de runtime die het model teruggeeft. Dit heeft als resultaat dat de afwijking van kleine runtimes evenzeer wordt afgestraft als die van grote runtimes.

$$E^l_{rel} : \mathbb{R} \times \mathbb{R} \rightarrow \mathbb{R} : (M(x_k), R(x_k)) \mapsto \frac{|M(x_k) - R(x_k)|}{|R(x_k)|} \tag{13.6}$$

- *De gecombineerde fout*: In het geval de runtimes dicht bij nul liggen, kan de relatieve foutmaat problemen geven. De deling door de runtime zal er dan

immers voor zorgen dat de schaling enorm kan worden. Hoewel deze situatie, in het geval van runtimes, zelden voorkomt, kan ze opgelost worden door de *gecombineerde fout* te gebruiken. Deze error zal de absolute fout benaderen voor $R(x_k) << 1$ en de relatieve fout benaderen voor $R(x_k) >> 1$.

$$E^l_{com} : \mathbb{R} \times \mathbb{R} \to \mathbb{R} : (M(x_k), R(x_k)) \mapsto \frac{|M(x_k) - R(x_k)|}{1 + |R(x_k)|} \tag{13.7}$$

Er dient hier wel rekening gehouden te worden met het feit dat de gecombineerde fout slechts half zo groot zal zijn als de absolute fout wanneer de runtimes nabij 1 liggen.

13.2.2 Globale kwaliteitscontrole

Zoals reeds werd aangehaald en duidelijk geïllustreerd wordt in figuur 13.1, zal de globale kwaliteitscontrole afhangen van de lokale kwaliteitscontrole in elk samplepunt. Stel dat er N samplepunten zijn, dan zal de globale error een afbeelding zijn van de vorm $\mathbb{R}^N \to \mathbb{R}$. Stel verder dat $e_k = E^l(M(x_k), R(x_k))$ de lokale error is in samplepunt x_k, dan kunnen de belangrijkste afbeeldingen voor de runtime predictie als volgt uitgedrukt worden:

- *De maximum error*: Deze error zal de maximale waarde nemen van elk van de lokale errors en ziet er als volgt uit:

$$E^g_{max} : \mathbb{R}^N \to \mathbb{R} : e_k \mapsto max^N_{k=1} e_k \tag{13.8}$$

- *De totale error*: Deze error wordt bepaald door de error van elk punt bij elkaar op te tellen:

$$E^g_{tot} : \mathbb{R}^N \to \mathbb{R} : e_k \mapsto \sum_{k=1}^{N} e_k \tag{13.9}$$

- *De totale relatieve error*: Deze error wordt bepaald door de totale error te delen door de som van de correcte runtimes:

$$E^g_{rel} : \mathbb{R}^N \to \mathbb{R} : e_k \mapsto \frac{\sum_{k=1}^{N} e_k}{\sum_{k=1}^{N} R(x_k)} \text{ of } \frac{E^g_{tot}}{\sum_{k=1}^{N} R(x_k)} \tag{13.10}$$

- *De gemiddelde error*: Deze error wordt bepaald door de gemiddelde waarde van alle lokale errors te nemen:

$$E^g_{avg} : \mathbb{R}^N \to \mathbb{R} : e_k \mapsto \frac{1}{N} \sum_{k=1}^{N} e_k \text{ of } \frac{1}{N} E^g_{tot} \tag{13.11}$$

- *De Root-mean-square(RMS)* : Deze foutmaat geeft een beeld van de magnitude waarmee de fout varieert en wordt, omwille van zijn vorm, ook wel kwadratisch gemiddelde genoemd:

$$E^g_{rms} : \mathbb{R}^N \to \mathbb{R} : e_k \mapsto \sqrt{\frac{1}{N} \sum_{k=1}^{N} {e_k}^2} \tag{13.12}$$

13.3 Implementatie

Aangezien er tot op heden slechts één implementatie van deze basiscomponent geïmplementeerd is, wordt deze kort besproken in deze sectie. De implementatie heeft als naam `BasicErrorCalculator` gekregen. Zoals reeds besproken in de vorige sectie geeft de `getDescription()` de beschrijving van de verschillende kwaliteitscontroles terug in een array van strings. De concrete implementatie van deze methode in de `BasicErrorCalculator` ziet er als volgt uit:

```
1  public String[] getDescription()
2  {
3    String[] returnvalue = new String[7];
4    returnvalue[0]="totalerror";
5    returnvalue[1]="gemiddelde error per punt";
6    returnvalue[2]="gemiddelde percentage error per punt";
7    returnvalue[3]="totale error tov totale exacte tijd";
8    returnvalue[4]="totale error tov totale voorspelde tijd";
9    returnvalue[5]="RMS";
10   returnvalue[6]="RelRMS";
11   return returnvalue;
12 }
```

Deze implementatie omvat dus de volgende manieren van kwaliteitscontrole:

- *totalerror*: De totale error wordt in dit geval berekend volgens de *totale error* techniek zoals beschreven in formule 13.9, gebruik makend van de *absolute fout*, zoals beschreven in formule 13.5. Deze globale kwaliteitscontrole berekent het verschil tussen de echte runtime in een punt en de voorspelde runtime en telt deze voor alle punten in de parametersweep op. Stel dat $R(x_k)$ de runtime is van een job x_k en dat $M(x_k)$ zijn voorspelde (Expected) runtime is, in een sweep met N jobs, dan kan de totale error uitgedrukt worden als:

$$totalerror = \sum_{k=1}^{N} |M(x_k) - R(x_k)| \tag{13.13}$$

- *gemiddelde error per punt (gepp)*: Geeft weer hoeveel de runtime voorspelling voor een job $M(x_k)$ gemiddeld afwijkt van de werkelijke runtime van die job $R(x_k)$. Ze wordt berekend aan de hand van de *gemiddelde error* techniek, besproken in formule 13.11, gebruik makend van de *absolute fout*, zoals beschreven in formule 13.5. Stel dat N het aantal jobs in de volledige sweep van jobs is, dan kan dit als volgt uitgedrukt worden:

$$gepp = \frac{1}{N} \sum_{k=1}^{N} |M(x_k) - R(x_k)| \tag{13.14}$$

- *gemiddeld percentage error per punt (gpepp)*: Dit is de gemiddelde van de errorpercentages in alle punten van de parametersweep. Het errorpercentage is

de percentuele verhouding tussen de error en de correcte runtime van een job. Deze kwaliteitswaarde wordt berekend aan de hand van *gemiddelde error* techniek, besproken in formule 13.11, maar er wordt in dit geval gebruik gemaakt van de *relatieve fout*, zoals beschreven in formule 13.6. Tot slot wordt in de postprocessing fase van de lokale kwaliteitscontrole elke error omgezet in een percentage. Dit kan ook als volgt uitgedrukt worden:

$$gpepp = \frac{\sum_{k=1}^{N} \left(\frac{|M(x_k) - R(x_k)|}{100 \cdot R(x_k)} \right)}{N} \% \tag{13.15}$$

- *totale error tov totale exacte tijd (ErvEx)*: Dit is de verhouding van de som van alle errors $|M(j) - R(j)|$ op de som van alle exacte tijden $R(j)$. Deze kwaliteitswaarde wordt berekend aan de hand van de *totale relatieve error* techniek, besproken in formule 13.10, gebruik makend van de de *absolute fout*, zoals beschreven in formule 13.5. Dit wordt als volgt uitgedrukt:

$$ErvEx = \frac{\sum_{k=1}^{N} |M(x_k) - R(x_k)|}{\sum_{k=1}^{N} R(x_k)} \tag{13.16}$$

- *totale error tov totale voorspelde tijd (ErvPred)*: Dit is de verhouding van de som van alle errors $|M(j) - R(j)|$ op de som van alle voorspelde tijden $M(j)$. Deze kwaliteitswaarde wordt berekend aan de hand van een alternatieve versie van de *totale relatieve error* techniek, besproken in formule 13.10. De effectieve runtime $R(x_k)$ in de noemer van de formule wordt vervangen door de voorspelde waarde $M(x_k)$. De lokale kwaliteitscontrole gebeurt aan de hand van de *absolute fout*, zoals beschreven in formule 13.5. Dit wordt als volgt uitgedrukt:

$$ErvPred = \frac{\sum_{k=1}^{N} |M(x_k) - R(x_k)|}{\sum_{K=1}^{N} M(x_k)} \tag{13.17}$$

- *Root Mean Square (RMS)*: Deze foutmaat geeft een beeld van de magnitude waarmee de error varieert en wordt, omwille van zijn vorm, ook wel kwadratisch gemiddelde genoemd. De implementatie van deze kwaliteitswaarde gebeurt aan de hand van de *Root-mean-square(RMS)* methode, besproken in formule 13.12, gebruik makend van de *absolute fout*, zoals beschreven in formule 13.5. Dit kan als volgt uitgedrukt worden:

$$RMS = \sqrt{\frac{1}{N} \sum_{k=1}^{N} (M(x_k) - R(x_k))^2} \tag{13.18}$$

- *Relative Root Mean Square (RelRMS)*: Dit is dezelfde foutmaat als de RMS met in dit geval de relatieve error per punt. De implementatie van deze kwaliteitswaarde gebeurt aan de hand van de *Root-mean-square(RMS)* methode,

besproken in formule 13.12, gebruik makend van de *relatieve fout*, zoals beschreven in formule 13.6. Dit kan als volgt uitgedrukt worden:

$$RelRMS = \sqrt{\frac{\sum_{k=1}^{N} \left(\frac{M(x_k)-R(x_k)}{R(x_k)}\right)^2}{N}} \tag{13.19}$$

13.4 Inputwaarden

Tot slot is het belangrijk op te merken dat de waarden die in deze kwaliteitsbepaling gebruikt worden niet steeds de runtime waarden moeten zijn. Het is eveneens mogelijk gebruik te maken van de locatie van het huidige samplepunt om zo de kwaliteit van dat punt te bepalen. Het spreekt immers voor zich dat in de meeste gevallen een punt, dat dicht bij een gekend samplepunt ligt, nauwkeuriger voorspeld kan worden dan een punt dat ver ligt van de reeds gekende samplepunten. De introductie van deze verschillende waarden maakt dat dit systeem enorm krachtig wordt bij het gebruik binnen de sampleselectors.

13.5 Besluit

De huidige implementatie van de kwaliteitscontrole is summier maar afdoende krachtig om de huidige wensen te verwezenlijken. Deze kracht is niet in het minst te wijten aan de verschillende inputtypes of combinaties van inputtypes die als kwaliteitswaarden kunnen gebruikt worden (runtime waarde, locatie, ...). Het spreekt dan ook voor zich dat bij de implementatie van meer toepassingsspecifieke en geavanceerde sampleselectietechnieken deze component ook uitgebreid zal moeten worden.

Er kan dus besloten worden dat de huidige implementatie afdoende uitbreidbaar en krachtig is.

HOOFDSTUK 14

Testen

14.1 Introductie

De doelstelling van deze sectie is tweeledig. Enerzijds wordt nagegaan of het GIPSy framework de gewenste functionaliteit en het gewenste gebruiksgemak voorziet. Anderzijds wordt een kwaliteitsanalyse gedaan van de verschillende technieken die aangewend kunnen worden in elk van de Predictieonderdelen. Concreet worden verschillende predictietechnieken en Sampleselectiemethoden vergeleken. De eerste sectie beschrijft de testtechniek die aangewend wordt. In een volgende sectie wordt beschreven op welke manier de kwaliteit gedefinieerd wordt en op welke manier de kwaliteitsanalyse gebeurt. Tot slot worden de verschillende testen beschreven en de resultaten besproken.

14.2 Testtechniek

Het GIPSy framework biedt een medium dat de toepasbaarheid van een predictiemechanisme binnen een bepaalde predictie doelstelling kan toetsen, alvorens het desbetreffend predictiemechanisme in een scheduler geïntegreerd wordt. De informatie die aangewend wordt om dit geheel te testen, bestaat uit een lijst taakspecificaties met de overeenkomende runtimes. Aan de hand van deze informatie en van de reeds beschreven kwaliteitscontrole (zie hoofdstuk 13) kunnen de verschillende kwaliteitswaarden van een techniek bepaald worden. De testmethodiek die gebruikt wordt om de essentiële informatie voor deze kwaliteitscontrole te bekomen, werd reeds besproken in hoofdstuk 8 aan de hand van figuur 8.1. Concreet komt het er op neer dat in een volledige test een serie van modellen wordt geconstrueerd. Na de

constructie van een model wordt telkens een nog niet uitgevoerde Job geselecteerd en de runtime-informatie van deze Job bij de gekende runtime-informatie gevoegd. Met deze extra informatie wordt het volgend model opgebouwd. Aan de hand van de kwaliteitscontrole in elk van deze opeenvolgende stappen (voor elk opeenvolgend model) wordt een globaal beeld gevormd van de kwaliteit van elk van de gebruikte componenten in deze test (Modelleringstechniek, Sampleselector).

14.3 Kwaliteitscontrole

Alvorens het uitvoeren van testen kan aanvangen, dienen de testmethodiek en de bijhorende kwaliteitscontrole afgesteld te worden op de concrete doelstelling. Om dit terdege te begrijpen, wordt de toepassing van de GIPSy predictor in een scheduler hier kort herhaald. Aan de hand van reeds uitgevoerde Jobs wordt een model opgesteld dat de runtime van de resterende taken voorspelt. Een sampleselectietechniek bepaalt de uitvoervolgorde van de resterende taken. Deze uitvoervolgorde wijzigt bijgevolg telkens er een nieuw model ontwikkeld is. Het concrete doel is, in dit geval, *ervoor zorgen dat de fout op de runtime predictie voor elke taak, op het moment dat ze wordt uitgevoerd, minimaal is.*

In eerste instantie werd aangenomen dat deze doelstelling bereikt werd door het *gemiddelde percentage error per punt (gpepp)* uit vergelijking 13.15 te minimaliseren. Testresultaten tonen echter aan dat de *gpepp* waarden onafhankelijk van de *error van de uitgevoerde punten (eup)* kunnen evolueren. De *eup* wordt bepaald door de verhouding van, het verschil tussen de runtime van een Job en de predictie hiervan op het moment dat deze Job wordt uitgevoerd, en de effectieve runtime (uitgedrukt in een percentage). Het feit dat de *gpepp* waarden een beeld geven van de globale kwaliteit van het model en dat dit onderzoek enkel interesse toont in de kwaliteit van de voorspellingen voor de Jobs die effectief uitgevoerd worden, liggen aan de basis van deze onderlinge onafhankelijkheid. De misconceptie wordt verholpen door de *eup* waarden toe te voegen aan de kwaliteitscontrole[1] en de verdere evaluatie te baseren op deze kwaliteitsfactor. De eenvoud van deze toevoeging illustreert de kracht en de gebruiksvriendelijkheid van het GIPSy famework.

Naast de kwaliteitscontrole is het tevens belangrijk een bepaalde performantie norm te halen. Concreet betekent dit dat zowel het construeren van het model, als de bepaling van de uitvoervolgorde van de Jobs, binnen een bepaald tijdsframe uitgevoerd moeten kunnen worden. Indien deze tijd te lang zou worden zal de predictie reeds achterhaald zijn wanneer ze aangewend wordt binnen de scheduler.

[1] De basis errorcalculator klasse wordt uitgebreid en de *eup* waarden worden toegevoegd.

14.4 Testen en resultaten

14.4.1 Testverloop

De test analyse wordt opgesplitst in drie delen. In een eerste deel wordt de kwalitatief meest geschikte modelleringstechniek bepaald. In het tweede deel wordt nagegaan welke sampleselectie techniek het best wordt aangewend om de vooropgestelde doelstelling te bereiken. Het derde deel bestaat uit een performantie analyse die bepaalt of de afhandeling van de verschillende technieken binnen het gewenste tijdsframe kan plaats vinden. De analyse in elk van deze groepen gebeurt op basis van de uitgevoerde testen die besproken worden in de volgende sectie.

14.4.2 Uitgevoerde testen

Om een zo volledig mogelijk beeld te krijgen van de kwaliteit van de verschillende technieken is ervoor gekozen zo uitgebreid mogelijk te testen. Dit wil zeggen dat elke combinatie van testset, sampleselectietechniek en modelleringstechniek aan de besproken testmethodiek onderworpen is. Dit is eenvoudig, gebruiksvriendelijk en zonder beperkingen te realiseren aangezien een test volledig gespecificeerd wordt aan de hand van eenvoudig uitbreidbare en aanpasbare hiërarchische configfiles. Concreet komt het erop neer dat elke mogelijke combinatie van de volgende specificaties getest is:

- *TestSets*: Om de testen doorheen deze thesis te uniformeren, werd ook in dit geval getest op basis van de testsets uit hoofdstuk 5.8.7:

	Aantal taken(T)	Sequentiële runtime($T \times t_m$)	Beschrijving
QF Test	5670	18271,706 s	Veel kleine taken + enkele grote
Test 1	4000	23907,2033 s	Evenveel kleine als grote taken
Test 2	2457	100769,1452 s	Veel grote taken + enkele kleine

Tabel 14.1: Eigenschappen van de drie gebruikte testsets.

- *Modelleringstechnieken*: Met de intentie zo volledig mogelijk te zijn, werd gepoogd de verschillende technieken, beschreven in hoofdstuk 10, uit te testen. Concreet zijn de volgende technieken uitgetest:
  - *Iverson techniek*: Meer informatie is terug te vinden in sectie 10.3.
  - *Polynomiale modellen*: Testen werden uitgevoerd met polynomiale modellen van de 2de tot de 10de graad. Meer informatie is terug te vinden in sectie 10.4.
  - *Radiale Basis functies*: Testen werden uitgevoerd met zowel de Gausische (Exp) als met de Multiquadric (MQ) en met de Inverse Multiquadric (IMQ) functie. Meer informatie is terug te vinden in sectie 10.7.

  - *Kriging modellen*: Testen werden uitgevoerd met zowel de Gausische (Exp) als met de Multiquadric (MQ) en met de Inverse Multiquadric (IMQ) functie. Meer informatie is terug te vinden in sectie 10.8
  - *Naaste Buren Model*: Hier werden zowel testen uitgevoerd met gemiddelde waarden, als met afstandsgemiddelden als met afstandsfuncties, als met lokale predictors. De gebruikte lokale predictors komen overeen met de bovenstaande Modelleringstechnieken.

  Voor de volledigheid dient hier bij opgemerkt te worden dat er ten allen tijde gebruik gemaakt wordt van de gesplitte versie van bovenstaande predictors zoals beschreven in sectie 10.11. Dit omdat deze methode zowel de performantie als de kwaliteit ten goede komt. Het gebruik van onafhankelijke waarden kan de kwaliteit van een model immers niet positief beïnvloeden.

  De beperkte kwaliteit en de performantie van zowel SVM als van Neurale netwerken, die werden aangetoond in [87], hebben ertoe geleid dat deze technieken in dit werk niet verder uitgetest werden.

- *Sample selectie technieken*: Volgende technieken uit sectie 12 werden uitgetest:
  - RandomSampleSelector: Meer informatie is terug te vinden in sectie 12.2.
  - LargestErrorSampleSelector: Meer informatie is terug te vinden in sectie 12.3.
  - SmallestErrorSampleSelector: Meer informatie is terug te vinden in sectie 12.3.
  - LargestDistanceSampleSelector: Meer informatie is terug te vinden in sectie 12.5.
  - SmallestDistanceSampleSelector: Meer informatie is terug te vinden in sectie 12.5.
  - FluctuationSampleSelector: Meer informatie is terug te vinden in sectie 12.7.
  - NoFluctuationSampleSelector: Meer informatie is terug te vinden in sectie 12.7.

  De MultiPredictorSampleSelectors, de DistanceAndErrorSampleSelectors en de NeighbourPenalisationSelector werden om performantieredenen niet verder uitgetest. De errorsampleselecters werden enkel getest op basis van de exacte runtime data. Eventuele alternatieven, die een tweede of een derde model gebruiken om deze fout te berekenen, werden omwille van hun ondermaatse prestaties en performantie niet verder uitgetest. Aangezien er meer dan één model dient gecreëerd te worden, zal de performantie dalen en zullen de fouten, die ook in dit tweede model vervat zitten, de prestaties negatief beïnvloeden.

Concreet bestaan deze tests bijgevolg uit 3 verschillende testsets, 30 verschillende modelleringstechnieken en 7 sampleselectietechnieken. Dit maakt een totaal van 630

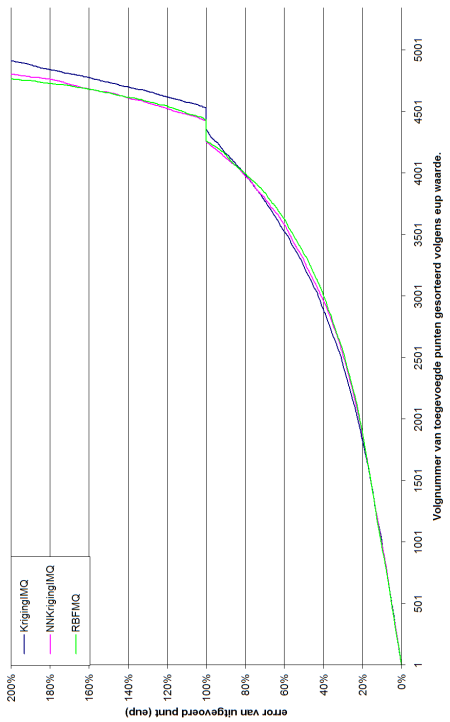
testen die gemiddeld 20 uren duren. Deze sequentiële duurtijd van 525 dagen, werd door middel van de CoBRA implementatie (zie sectie 9.5.2), gespreid over een 50 tal processoren zodat de duurtijd verkort werd tot 2 weken.

14.4.3 Deel 1: Modelleringstechnieken

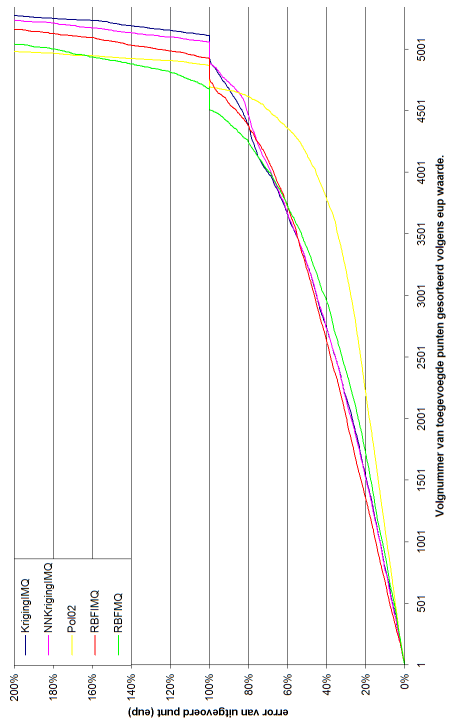
Om de kwaliteit van elke modelleringstechniek te bepalen, wordt voor elke test de lijst van *eup* waarden volgens grootte gesorteerd. Wanneer deze waarden gesorteerd geplot worden, spreekt het voor zich dat de curve die de X-as het dichtst benadert kwalitatief de beste is.

Met als doel de beste modelleringstechniek te selecteren, wordt voor elke testset (quantum fysica, testset 1 en testset 2) een figuur gebruikt (figuur 14.1, figuur 14.2 en figuur 14.3). Elke figuur bestaat uit vier plots, die een beeld vormen van de impact van de verschillende relevante[2] sampleselectie technieken (fluctuation sampleselector, random dataselector, smallest error sampleselector en smallest distance sampleselector). Elke plot geeft een weergave van de modelleringstechnieken die, gebruik makend van de desbetreffende specificaties, de X-as het meest benaderen (de modellen met de beste kwaliteit).

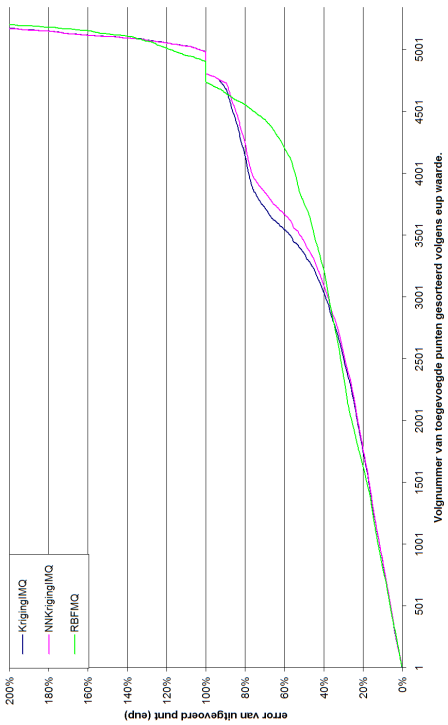
[2]Met relevante sampleselectietechnieken worden die technieken bedoeld die tot de beste resultaten leiden.



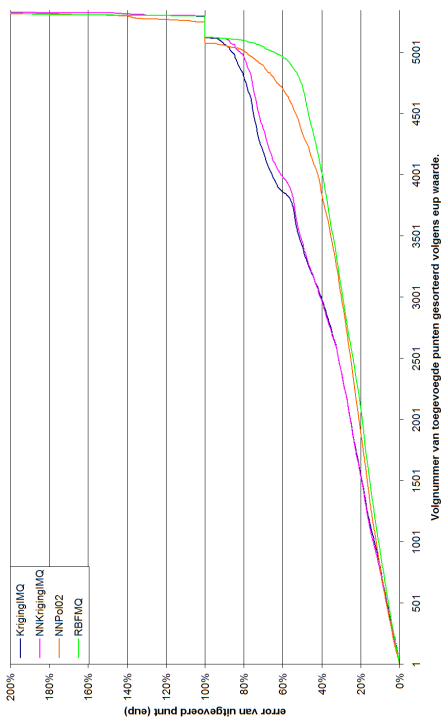
(b) Random Sampleselector



(a) Fluctuation Sampleselector

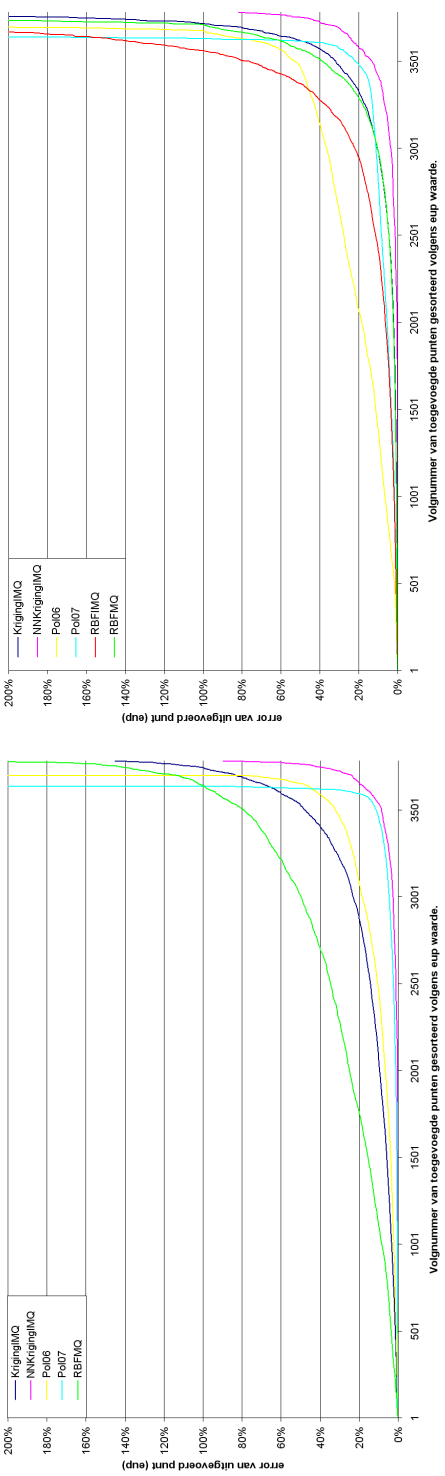


(d) Smallest Distance Sampleselector



(c) Smallest Error Sampleselector

Figuur 14.1: Vergelijking van modelleringstechnieken aan de hand van een selectie *eup* plots voor de quantum fysica testset.



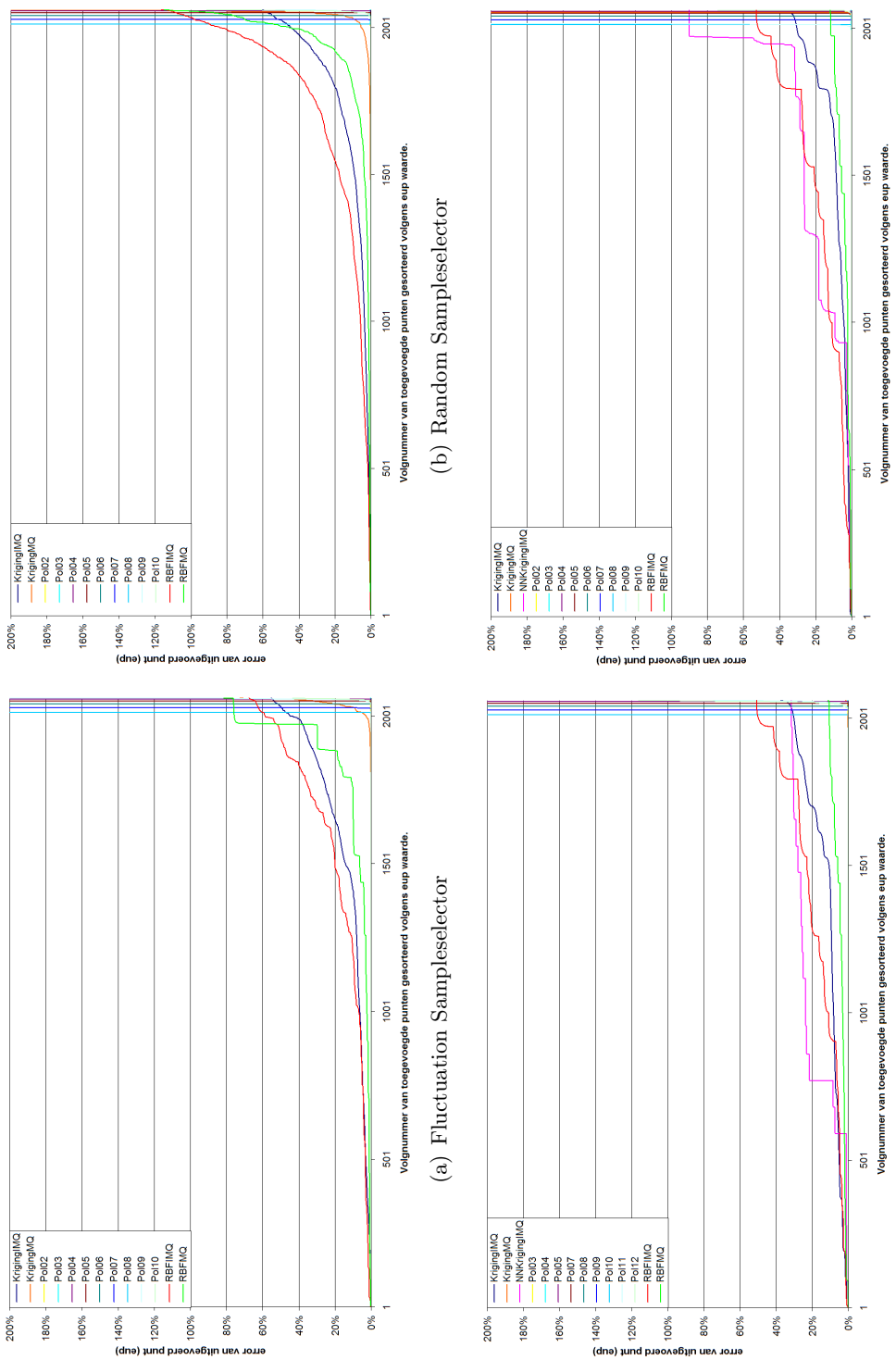
(a) Fluctuation Sampleselector

(b) Random Sampleselector

(c) Smallest Error Sampleselector

(d) Smallest Distance Sampleselector

Figuur 14.2: Vergelijking van modelleringstechnieken aan de hand van een selectie *eup* plots voor testset 1.



(a) Fluctuation Sampleselector

(b) Random Sampleselector

(c) Smallest Error Sampleselector

(d) Smallest Distance Sampleselector

Figuur 14.3: Vergelijking van modelleringstechnieken aan de hand van een selectie *eup* plots voor testset 2.

In eerste instantie valt het onmiddellijk op dat de polynomiale modellen testset 2 bijna perfect benaderen. Deze perfecte benadering is niet te wijten aan de kwaliteiten van deze modelleringtechniek maar aan de manier waarop testset 2 opgebouwd is. De opbouw berust immers op combinaties van polynomen die vanzelfsprekend eenvoudig te fitten zijn met een polynomiaal model.

Verder onderzoek van de verschillende plots brengt aan het licht dat er slechts drie modelleringstechnieken in elke plot terug te vinden zijn. Er kan dus gesteld worden dat enkel deze drie technieken in elke situatie toepasbaar zijn:

1. Kriging model met een inverse multiquadric kernel (KrigingIMQ).
2. Radiale basis functie met een multiquadric kernel (RBFMQ).
3. Nearest Neighbour model op basis van Kriging modellen met een inverse multiquadric kernel (NNKrigingIMQ).

Deze technieken worden aldus geselecteerd om in het verdere verloop van dit werk de gewenste predicties uit te voeren.

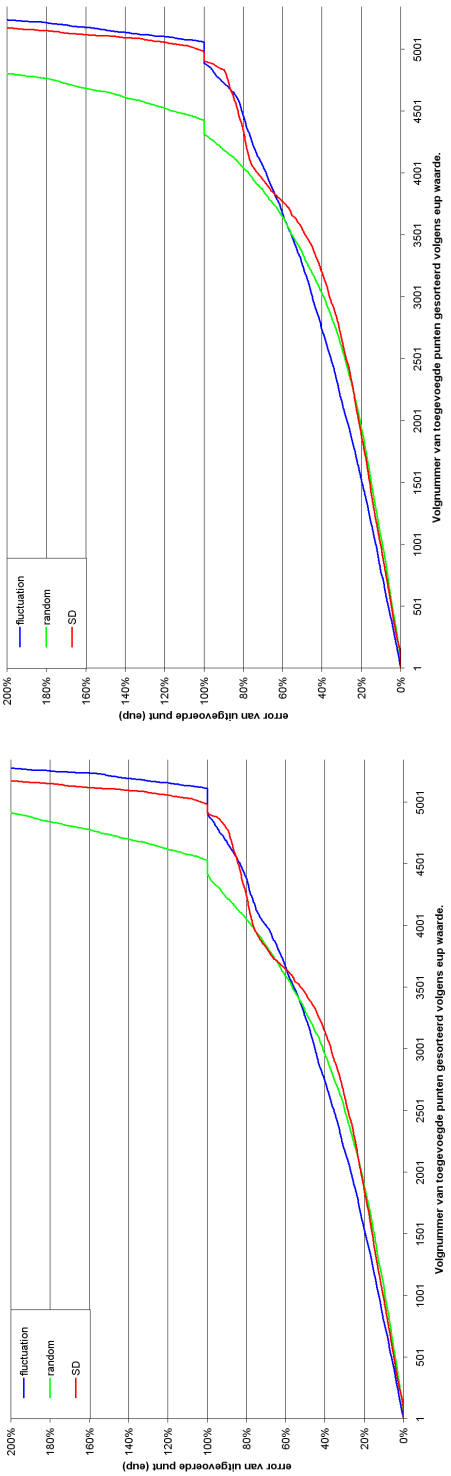
14.4.4 Deel 2: Sampleselectietechnieken

Aangezien het aantal modelleringstechnieken, die toepasbaar zijn in prediction based scheduling, op basis van de predictiekwaliteit kan herleid worden tot drie, kan de analyse van de verschillende sampleselectietechnieken ook op basis van deze drie technieken gebeuren. De Errorbased dataselectors dienen niet verder behandeld te worden. Ze zijn immers gebaseerd op onbeschikbare voorkennis of berusten op een tweede of op een derde model die het predictiemechanisme, zoals reeds besproken, negatief beïnvloeden. Er moet bijgevolg een keuze gemaakt worden tussen:

- De Random sampleselector (RaSS).
- De Fluctuation sampleselector (FlSS).
- De Smallest distance sampleselector (SDSS).

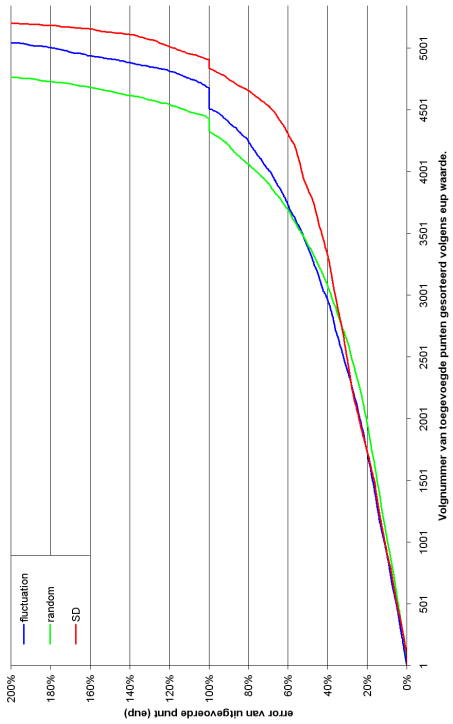
Figuren 14.4 , 14.5 en 14.6 vergelijken deze technieken voor het KrigingIMQ, het NNKrigingIMQ en het RBFMQ model. Uit de resultaten blijkt duidelijk dat de SDSS kwalitatief het best scoort in het merendeel van de testsets. Aangezien deze techniek steeds analoge of betere resultaten verwezenlijkt dan de Random dataselector kan er besloten worden dat het gebruik van de *SDSS* aan te raden is.

Naast deze vaststelling is het tevens belangrijk de knik op 100% op te merken in de plots uit figuur 14.4. Deze knik ontstaat door situaties waarin een runtime niet voorspelt kan worden door een gebrek aan informatie. De runtime wordt in dit geval 0 en resulteert in een fout van 100%. Uit de resultaten blijkt duidelijk dat het gebruik van de *SDSS* dit fenomeen ook zal minimaliseren en in een minimum van tijd voor de gewenste data zal zorgen.



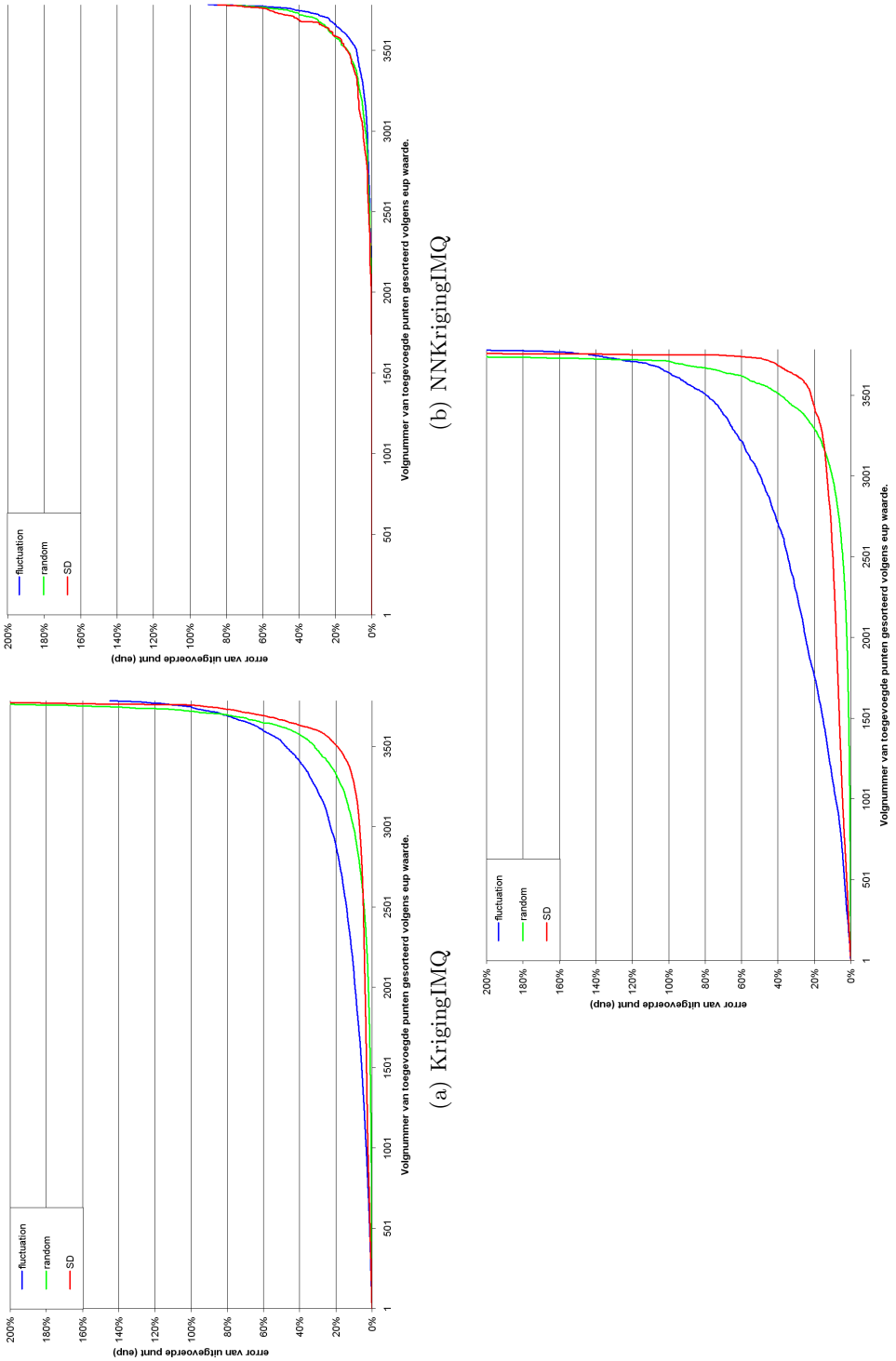
(a) KrigingIMQ

(b) NNKrigingIMQ



(c) RBFMQ

Figuur 14.4: Vergelijking van sampleselectors aan de hand van een selectie *eup* plots voor de quantum fysica testset.

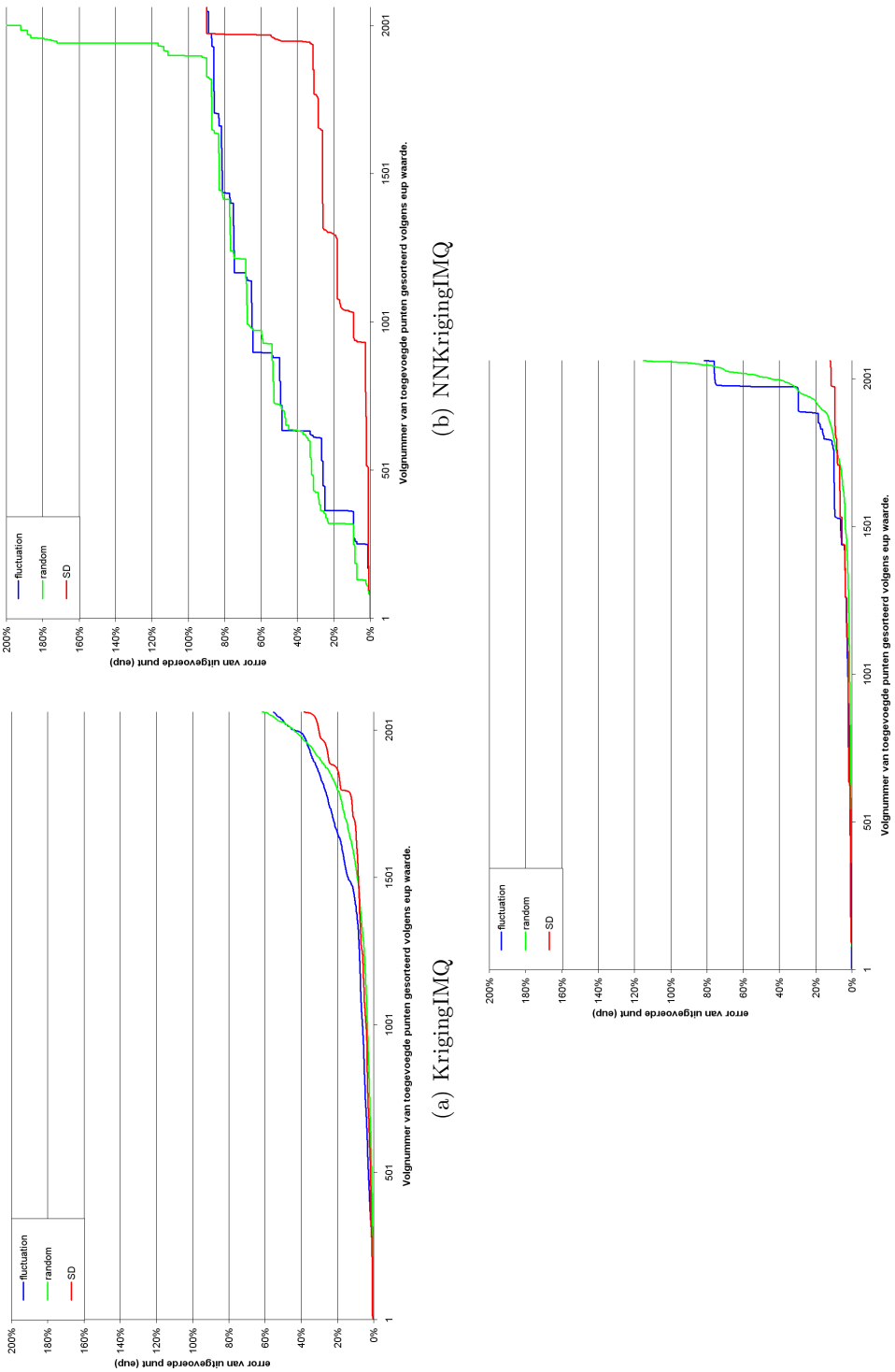


(a) KrigingIMQ

(b) NNKrigingIMQ

(c) RBFMQ

Figuur 14.5: Vergelijking van sampleselectors aan de hand van een selectie *eup* plots voor testset 1.



(a) KrigingIMQ

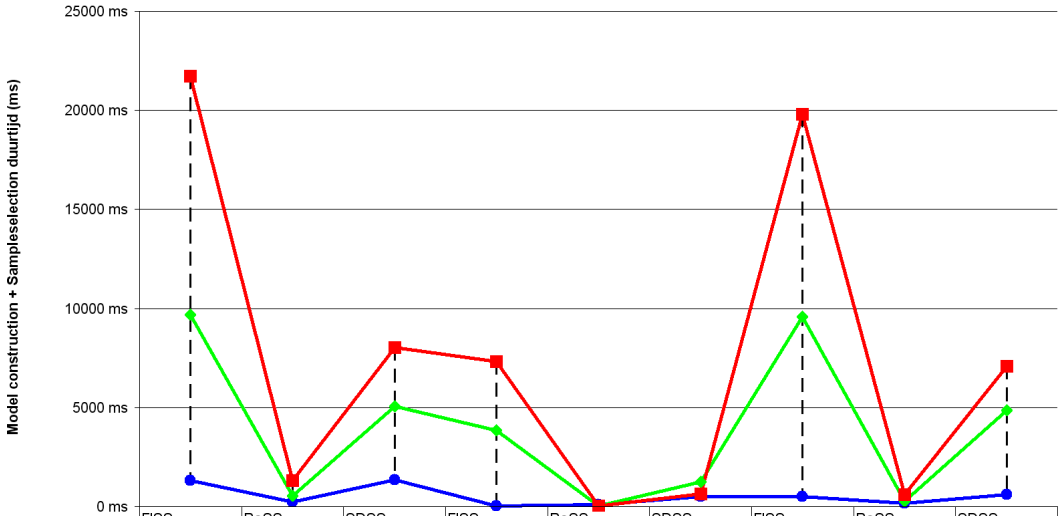
(b) NNKrigingIMQ

(c) RBFMQ

Figuur 14.6: Vergelijking van sampleselectors aan de hand van een selectie *eup* plots voor testset 2.

14.4.5 Deel 3: Performantieanalyse

Naast de kwaliteitsvereisten is het tevens belangrijk een beeld te krijgen van de performantie van de verschillende gebruikte technieken. Het spreekt voor zich dat kwaliteit belangrijk is, maar onbevredigend kan zijn, wanneer dit ten koste gaat van de performantie. De resultaten van de performantie analyse worden getoond in figuren 14.7, 14.8 en 14.9.

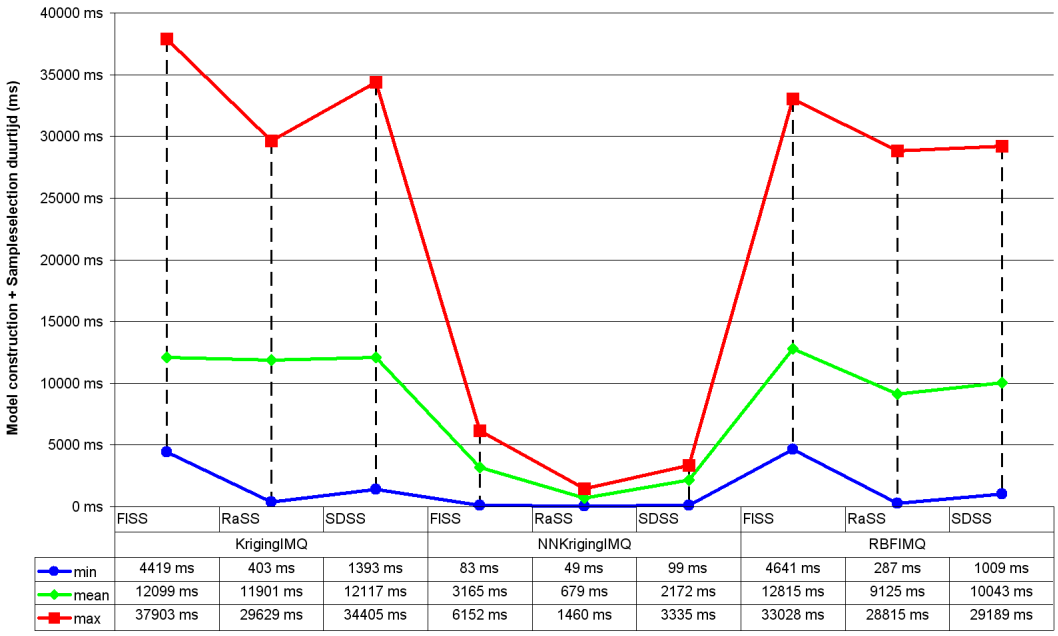


	FISS	RaSS	SDSS	FISS	RaSS	SDSS	FISS	RaSS	SDSS
	KrigingIMQ			NNKrigingIMQ			RBFIMQ		
QF min	1307 ms	249 ms	1357 ms	26 ms	111 ms	498 ms	491 ms	183 ms	613 ms
QF mean	9687 ms	526 ms	5068 ms	3855 ms	22 ms	1237 ms	9578 ms	266 ms	4859 ms
QF max	21714 ms	1330 ms	8032 ms	7305 ms	38 ms	645 ms	19790 ms	612 ms	7083 ms

Figuur 14.7: De minimale, de gemiddelde en de maximale tijd nodig om een model te construeren en de samples te selecteren voor de quantum fysica testset.

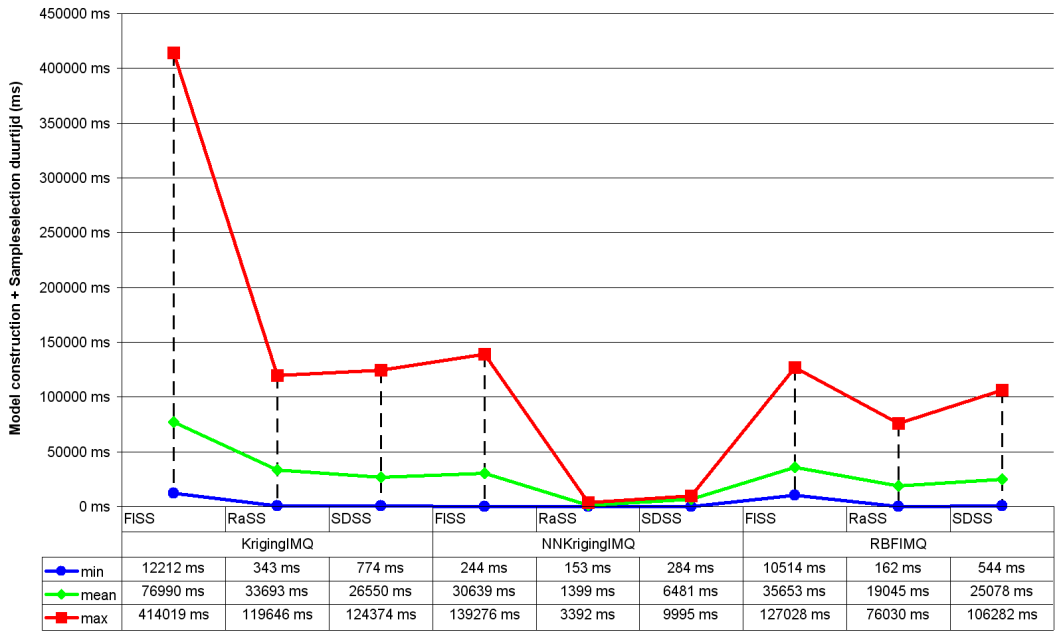
Om de resultaten terdege te begrijpen, dient de volgende kennis in het achterhoofd gehouden te worden. De quantum fysica testset heeft vier inputdimensies. Het globale model wordt op basis van de splitting strategie uit sectie 10.11 opgesplitst in modellen met twee dimensies. Testset 1 bezit 4 input dimensies die niet verder worden opgesplitst. Testset 2 bezit 3 input dimensies die ook niet verder worden opgesplitst.

In eerste instantie worden de verschillende modeltypes met elkaar vergeleken. Het valt meteen op dat de performantie van het RBFMQ en van het KrigingIMQ model vergelijkbaar zijn. Het NNKriging IMQ model scoort daarentegen veel beter. De oorzaak van deze snelheidswinst ligt in de gebruikte hoeveelheid data om een model op te bouwen. De computationele vereisten voor het oplossen van de least squares benadering, zullen sterk stijgen bij een toenemend aantal samplepunten. In de NNKriging variant wordt in elk punt een model opgebouwd aan de hand van een beperkte verzameling punten, die in de omgeving van het te voorspellen punt liggen (meestal 20 punten). In de andere gevallen wordt één globaal model



	FISS	RaSS	SDSS	FISS	RaSS	SDSS	FISS	RaSS	SDSS
	KrigingIMQ			NNKrigingIMQ			RBFIMQ		
min	4419 ms	403 ms	1393 ms	83 ms	49 ms	99 ms	4641 ms	287 ms	1009 ms
mean	12099 ms	11901 ms	12117 ms	3165 ms	679 ms	2172 ms	12815 ms	9125 ms	10043 ms
max	37903 ms	29629 ms	34405 ms	6152 ms	1460 ms	3335 ms	33028 ms	28815 ms	29189 ms

Figuur 14.8: De minimale, de gemiddelde en de maximale tijd nodig om een model te construeren en de samples te selecteren voor de testset 1.



	FISS	RaSS	SDSS	FISS	RaSS	SDSS	FISS	RaSS	SDSS
	KrigingIMQ			NNKrigingIMQ			RBFIMQ		
min	12212 ms	343 ms	774 ms	244 ms	153 ms	284 ms	10514 ms	162 ms	544 ms
mean	76990 ms	33693 ms	26550 ms	30639 ms	1399 ms	6481 ms	35653 ms	19045 ms	25078 ms
max	414019 ms	119646 ms	124374 ms	139276 ms	3392 ms	9995 ms	127028 ms	76030 ms	106282 ms

Figuur 14.9: De minimale, de gemiddelde en de maximale tijd nodig om een model te construeren en de samples te selecteren voor de testset 2.

geconstrueerd, gebruik makend van al de beschikbare punten(300 → 5670). De

mindere kwaliteit van het NNKrigingIMQ model maakt echter dat het RBFMQ en het KrigingIMQ nog steeds geprefereerd worden. Indien het opstellen van een model echter computationeel te veel vereist, kan er overgeschakeld worden naar het NNKrigingIMQ model. Deze situaties kunnen zich voordoen wanneer het aantal inputdimensies of het aantal samplepunten te omvangrijk worden.

Wanneer de verschillende sampleselectietechnieken vergeleken worden, blijkt onmiddellijk dat er grote verschillen zijn tussen de drie gebruite technieken. Het random selecteren van punten zal, zoals te verwachten, nagenoeg geen invloed hebben op de performantie. Het gebruik van de fluctuation dataselector daarentegen zal de performantie enorm verminderen. De oorzaak van de gecreëerde overhead ligt in het feit dat de dataselector gebruik maakt van de predicties in een punt. In elk punt moet de runtime eerst voorspeld worden alvorens de sampleselectie kan uitgevoerd worden. In de smallest distance dataselector daarentegen wordt enkel gebruik gemaakt van de input parameters die reeds voorhanden zijn. De overhead van die laatste zal bijgevolg niet veel groter zijn dan diegene van de random dataselector en ligt gemiddeld gezien om en bij de milliseconde per samplepunt. Er kan bijgevolg besloten worden dat er tussen de kwalitatief vergelijkbare fluctuation en de smalest distance dataselector verschillen bestaan op het niveau van de performantie. Deze verschillen en de beperkte kwaliteit van de random dataselector maken dat de smallest distance sampleselector globaal het best scoort.

Naast de vergelijking van de technieken is het belangrijk op te merken dat de vorm van de inputdata de belangrijkste factor is die de performantie beïnvloedt. Deze invloed is veel sterker dan deze van de gebruikte sampleselectie of de modelleringstechniek. Het aantal inputdimensies dat wordt gebruikt bij de opbouw van de individuele modellen en het aantal samplepunten blijken hier de voornaamste invloeden. Dit wordt meteen duidelijk wanneer naar de schalen op de assen van figuren 14.7, 14.8 en 14.9 wordt gekeken. Testset 1, die 4000 samplepunten bezit en modellen opstelt voor vier inputdimensies, heeft om en bij de halve minuut nodig om een KrigingIMQ of RBFIMQ model op te stellen en nieuwe samples te selecteren. Het quantum fysica probleem, waarbij het aantal samplepunten 5670 bedraagt, maar het aantal dimensies tot 2 wordt gereduceerd heeft hier slechts 5 seconden voor nodig. Testset 2, waarin drie dimensies gebruikt worden om een model te bouwen voor 2457 samplepunten, doet er dan weer 12 seconden over. Omwille van deze reden is het verstandig, bij een groot aantal inputdimensies, de modellen op te splitsen in lagere dimensies. Voorlopig wordt hiervoor enkel de techniek uit sectie 10.11, die dimensies met een random invloed detecteert, gebruikt. Hier kunnen in de toekomst echter technieken aan toegevoegd worden die nagaan voor welke dimensie de variantie van de invloed het grootst is. Deze techniek kan dan aangewend worden om een dimensie volgorde op te stellen, die aangeeft volgens welke dimensies de modellen het best gesplitst kunnen worden.

14.5 Conclusie

De uitgevoerde testen tonen aan dat, aan de hand van het GIPSy framework, op een eenvoudige manier de kwaliteit van de verschillende technieken kan bepaald worden om een specifiek doel te bereiken. Zowel het specifieke doel als de verschillende technieken zijn eenvoudig in te voegen in het kwalitatief onderzoek.

Uit het concrete kwalitatief onderzoek naar de minimalisatie van de *eup* waarden is gebleken dat slechts drie technieken kwalitatief voldeden: Het Kriging model met een inverse multiquadric kernel (KrigingIMQ), het RBF model met een multiquadric kernel (RBFMQ) en het nearest neighbour model op basis van Kriging modellen met inverse multiquadric kernels (NNKrigingIMQ). De kwaliteit is merkbaar beter in de eerste twee modellen, die bijgevolg geprefereerd worden. Het derde model is echter veel performanter en kan bijgevolg aangewend worden in situaties waar de performantie zwaar doorweegt. Uit de verschillende sampleselectors, die onder de loep werden genomen, blijkt dat de smallest distance sampleselector zowel kwalitatief als op het niveau van de performantie het best scoort. Deze techniek wordt aldus aangeraden om het vooropgestelde doel te bereiken. Tot slot dient er rekening gehouden te worden met de invloed van het aantal samplepunten en van het gebruikte aantal input dimensies bij het opstellen van de verschillende modellen. De impact van deze factoren is zo groot dat er, in geval van performantie problemen, best gekozen kan worden voor het opsplitsen van een model in een bepaalde dimensie. Het bepalen van deze dimensies kan, tot op heden, gebeuren aan de hand van de techniek uit sectie 10.11. Naar de toekomst toe is het wel interessant deze niche verder uit te werken.

HOOFDSTUK 15

Conclusie

In deel II van deze doctoraatsthesis werd het GIPSy framework voorgesteld. De introductie van dit nieuwe framework had twee concrete doelstellingen:

1. Een medium bieden om op eenvoudige en op eenduidige wijze nieuwe predictietechnieken te testen en in de praktijk toe te passen.

2. Een proof of concept bieden dat aantoont dat het mogelijk is , gebruik makend van een weloverwogen verzameling predictietechnieken, tot adequate runtime predicties te komen.

De eerste doelstelling werd bereikt door, bij de opbouw van het framework, de plugability vereisten als primaire eis naar voor te schuiven. Deze eis werd ingewilligd door het volledig predictiemechanisme op te bouwen aan de hand van de drie basiscomponenten uit figuur 8.1 (Startgrid selection, Modelbuilding, Sample Selection). Deze werkwijze maakt het mogelijk elk van deze componenten eenvoudig te vervangen door een alternatieve implementatie. Op deze manier kan de kwaliteit van verschillende implementaties van een component vergeleken worden.

In hoofdstuk 14 werd, volgens deze werkwijze, op basis van de eveneens plugable kwaliteitscontrole component, de kwaliteit van de verschillende predictiecomponenten geëvalueerd en besproken. Dit kwalitatief onderzoek naar de minimalisatie van de *eup* waarden (fout op het moment van uitvoering) toont aan dat drie van de geëvalueerde modelleringstechnieken voldoen aan de vereisten runtimes van parametersweep jobs te voorspellen. Meer specifiek gaat het hier om het Kriging model met een inverse multiquadric kernel (KrigingIMQ), het RBF model met een multiquadric kernel (RBFMQ) en het nearest neighbour model op basis van Kriging modellen met inverse multiquadric kernels (NNKrigingIMQ). Hoewel het merendeel van de sampleselectietechnieken valabel bleek, scoort de smallest distance sampleselector, zowel kwalitatief als op het niveau van de performantie, ruimschoots het best.

Concreet kan er dus vastgesteld worden dat het voorgestelde GIPSy framework de mogelijkheid biedt verschillende predictietechnieken te vergelijken en dat er, binnen deze range van technieken, valabele kandidaten bestaan om kwalitatieve runtime predicties te verwezenlijken. De doelstellingen van dit gedeelte van de thesis zijn aldus bereikt.

Deel III

Prediction Based Scheduling

HOOFDSTUK 16

Inleiding

Deel II van deze doctoraatsthesis gaf reeds aan dat het GIPSy project tot doel had een framework te ontwikkelen om predictie mechanismen enerzijds uit te testen via simulatie en anderzijds toe te passen in een prediction based scheduler. Het doel van een prediction based scheduler is jobs optimaal te verdelen over non-dedicated resources. Non-dedicated betekent in dit geval dat de gebruiker van de grid geen zekerheid heeft over de uptime van de verschillende resources. Deze problematiek doet zich voor in Desktop grids [62]. Aangezien de resources in dit geval beheerd en gebruikt worden door een derde onafhankelijke partij, zullen deze resources niet ten allen tijde beschikbaar zijn. Dit kan tot gevolg hebben dat een job, die op een resource uitgevoerd wordt, plots onderbroken wordt omdat de eigenaar van de machine deze resource nodig heeft. Het werk van de desbetreffende job zal aldus verloren gaan. De performantie analyse uit [127] toont aan dat het dergelijk falen van jobs een grote impact kan hebben op de totale runtime bij het gebruik van failure unaware schedulers[1]. Een mogelijke oplossing voor dit probleem is het gebruik van tijdelijke data en checkpointing [70] (Zie ook sectie 5.8.3). Deze methodiek vereist echter dat de gebruiker zijn code aanpast. In het geval van Condor [113] [27] bijvoorbeeld, dient een gebruiker zijn code te linken met een system call library om checkpointing mogelijk te maken. Naast het feit dat deze technieken niet altijd toepasbaar zijn, voldoen ze niet aan de vooropgestelde doelstelling. Het CoBRA framework vereist immers dat de gebruiker geen kennis dient te hebben van de gebruikte scheduling techniek en dat de implementatiecomplexiteit geminimaliseerd dient te worden. De problematiek kan echter ook op een alternatieve manier opgelost worden. Deze techniek vereist echter meer informatie: het resource availability window[2] en de runtime informatie van de jobs. Aan de hand van deze informatie kunnen de Jobs zo gescheduled worden, dat het zelden of nooit voorkomt dat een

[1] Schedulers die geen rekening houden met het feit dat jobs falen wanneer een resource niet langer beschikbaar is.

[2] Het tijdsframe waarin een resource beschikbaar is.

resource, gedurende de jobexecutie, onbeschikbaar wordt. Op deze manier moeten de Jobs niet herstart worden en gaat er geen werk verloren. Het resource availability window en de runtime informatie van een job zijn echter niet voorhanden. Er kan enkel gepoogd worden deze te voorspellen. Resource availability predictie technieken bestaan reeds in projecten zoals [98] en [21]. Omwille van deze reden worden ze in het verder verloop van dit werk als gekend beschouwd. Job runtime predicties daarentegen zijn minder vanzelfsprekend maar kunnen voorspeld worden aan de hand van het GIPSy framework (Zie deel II van deze doctoraatsthesis).

Dit deel van de thesis bespreekt een scheduling techniek die gebruik maakt van job runtime predicties en van gekende availability windows om een optimale verdeling na te streven. Optimaal betekent in dit geval het minimaliseren van het aantal gereschedulde taken, het minimaliseren van de round-trip time en indien mogelijk het maximaliseren van de throughput.

De doelstelling wordt gerealiseerd door het GIPSy framework te integreren in de PGS [122] (Prediction based Grid Scheduler). PGS is een prediction based scheduler ontwikkeld bovenop het CoBRA framework. De structuur en de werking van deze scheduler worden toegelicht in hoofdstuk 17. In hoofdstuk 18 wordt de scheduler getest en worden de resultaten van deze testen besproken. Hoofdstuk 19 vormt tot slot een algemeen besluit omtrent de ontwikkelde prediction based scheduler.

HOOFDSTUK 17

PGS (Prediction based Grid Scheduling)

17.1 Introductie

PGS of Prediction based Grid Scheduling werd geïntroduceerd in [122]. De doelstelling van dit project bestaat erin Bag of Task (BoT) Jobs optimaal te verdelen over non-dedicated resources, gebruik makend van uptime en van runtime informatie. Het project tracht de runtimes van de BoT te verdelen over het availability window van de beschikbare resources. Een optimale verdeling optimaliseert de throughput en de roundtriptime door ervoor te zorgen dat:

- de idle time van elke resource minimaal is.
- er geen jobs gerescheduled moeten worden. Dit wil zeggen dat elke job op een resource afgerond moet zijn wanneer deze niet langer beschikbaar is.

Stel It_x een idle time van machine x tijdens de distributie van een probleem dan moet $\max(It_x)$ geminimaliseerd worden in $0 < It_x < \max(It_x)$ voor elke machine x.

17.2 Principe

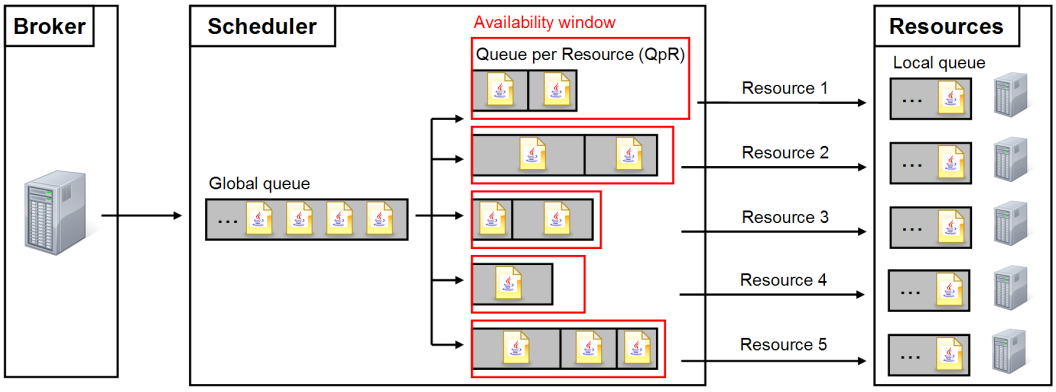
Zoals reeds werd aangehaald, wordt er in PGS vanuit gegaan dat de availability windows van de verschillende resources en de runtimes van de verschillende Jobs gekend zijn. In essentie komt het erop neer dat de lijst van de nog te verdelen taken de runtime informatie van de taken bevat en de resource lijst de resterende uptime van de resources bevat.

17.2.1 Availability window

Het availability window is de periode waarin een resource ter beschikking staat van de CoBRA grid en er dus Jobs uitgevoerd kunnen worden. Dit window wordt bepaald aan de hand van de uptime van de resource. Enkel deze uptime is echter niet voldoende om te bepalen of er nog plaats is voor extra Jobs in het window. Er dient immers rekening gehouden te worden met de Jobs die reeds in het window geplaatst zijn. Dit zijn de Jobs die zich reeds op de Resource bevinden en de Jobs die in de queue per resource (QpR) op de Scheduler geplaatst zijn (Zie figuur 17.1). Stel et_i de execution time van een job i en qr_R het aantal Jobs op de local queue van een Resource R, stel qs_R het aantal Jobs op de QpR (queue per resource) in de Scheduler van een Resource R, stel et_R en eu_R respectievelijk de execution time en de huidige uitvoeringstijd[1] van de Job, die op het huidige moment op resource R wordt uitgevoerd, en stel ru_R de resterende uptime van resource R, dan kan de tijd die nog vrij is in het availability window, van resource R uitgedrukt worden als:

$$tl_R = ru_R - (et_R - eu_R) - \sum_{i=0}^{qr_R} et_i - \sum_{j=0}^{qs_R} et_j \tag{17.1}$$

De availability windows worden in figuur 17.1 aangeduid met de rode kaders. Het is duidelijk dat in het eerste en in het vierde window nog plaats vrij is en dat de overige windows bijna vol gepland zijn.



Figuur 17.1: De opstelling van de verschillende queues binnen de PGS scheduler.

17.2.2 Scheduler design

Om de verdeling van de taken zonder veel communicatie overhead te laten verlopen, is het essentieel dat de taak verdelingen en herverdelingen over de resources lokaal op de scheduler gebeuren. Om dit te bewerkstelligen, is geopteerd voor een design met een globale Jobqueue en met een lijst van QpRs. Dit wordt duidelijk aan de hand van figuur 17.1. De globale lijst bestaat uit de Jobs die aangeleverd worden door de Broker. Deze Jobs zijn nog niet verdeeld over de resources. De QpRs

[1] De tijd die verstreken is sinds een Job gestart is op een worker.

stellen de Jobs voor die reeds binnen het availability window van de desbetreffende resource gescheduled zijn. Het is hier duidelijk dat de lengte van de QpRs binnen het rode availability window valt. Wanneer het aantal jobs in de lokale queue van een resource onder een bepaald minimum zakt, plaatst de Scheduler een extra Job op deze queue. De scheduler selecteert hiervoor de eerste Job uit zijn QpR op de Scheduler. Deze Job wordt vervolgens uit deze queue verwijderd.

17.2.3 Scheduling techniek

Het schedulen in PGS bestaat essentieel uit drie delen:

- Het plaatsten van Jobs uit de QpRs op de desbetreffende resource.
- Het plaatsen van Jobs uit de globale queue in de QpRs.
- Omgang met foute runtimes.

De eerste taak is straightforward en gebeurt analoog aan de standaard CoBRA scheduler. Van zodra het aantal Jobs in de lokale queue op een resource onder een bepaald minimum daalt, wordt deze aangevuld met de eerste job van zijn QpR op de scheduler.

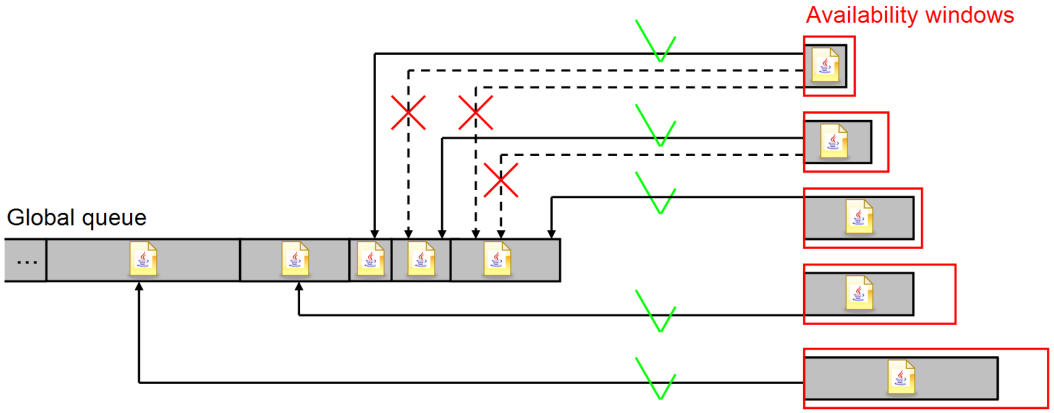
De tweede taak is echter minder vanzelfsprekend. Er dient immers enerzijds voor gezorgd te worden dat het availability window zo goed mogelijk gevuld wordt, maar er dient anderzijds ook voorkomen te worden dat de Jobs in de QpR dit availability window overschrijden. Dit om te voorkomen dat Jobs gerescheduled moeten worden en aldus uitgevoerd werk verloren gaat. Om dit te bewerkstelligen, bestaat deze scheduling stap uit vier taken die sequentieel en repetitief worden uitgevoerd:

- De eerste stap rangschikt de resources van klein naar groot volgens de lengte van hun availability window (Zie figuur 17.2). Dit is nuttig voor elk van de volgende stappen. Het zorgt ervoor dat elke taak wordt gescheduled op de resource met het kleinst passend availability window[2].

Figuur 17.2: De availability windows worden gesorteerd volgens grootte.

[2]Een availability window past wanneer een Job in de nog resterende tijd kan uitgevoerd worden.

- De tweede stap zorgt ervoor dat er, indien mogelijk, op elke Resource een Job in de lokale queue terecht komt, en dat er zich minstens één taak in zijn QpR op de Scheduler bevindt. Dit gebeurt door sequentieel twee Jobs te zoeken in de globale queue en die op zijn QpR te plaatsen. De eerste job wordt immers meteen naar de desbetreffende Resource gestuurd om daar uitgevoerd te worden. Figuur 17.3 toont hoe dit proces voor één job in zijn werk gaat. Het is hier onmiddellijk duidelijk dat de sorteerstap uit stap één ervoor zorgt dat de kleinste jobs, vooraan in de globale queue, in het kleinst passend availability window worden geplaatst.

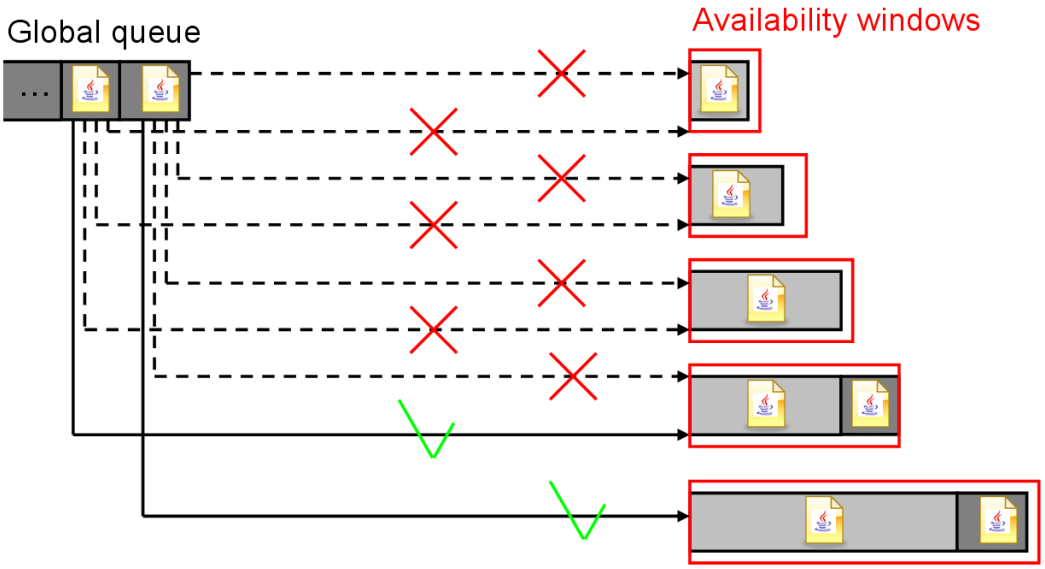


Figuur 17.3: Er wordt een Job geplaatst op elke QpR in de Scheduler.

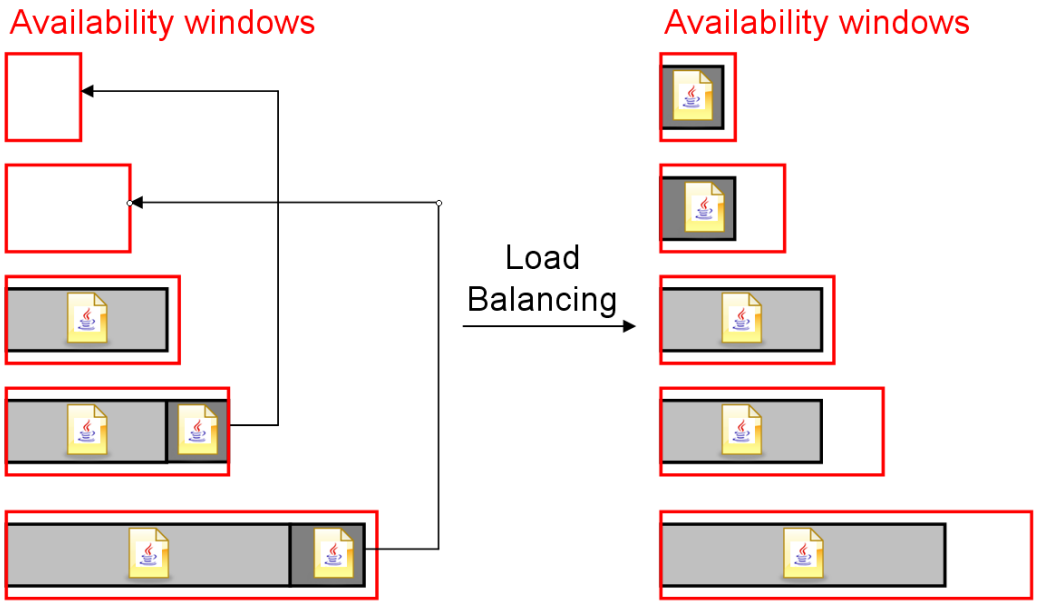
- In de derde stap wordt getracht de Jobs in de lokale queue over de verschillende QpRs te verdelen. Dit gebeurt door job per job in de eerste QpR met passend availability window te plaatsen. Aangezien de lijst van resources gesorteerd is volgens availability window, zal elke Job bijgevolg terecht komen in de QpR met het kleinst passend availability window. Deze stap wordt herhaald tot de globale queue leeg is of tot een QpR leeg blijft na het overlopen van de globale Job queue. Dit alles wordt duidelijk aan de hand van figuur 17.4.

- De vierde stap wordt slechts bereikt wanneer de globale jobqueue leeg is of wanneer er een QpR leeg is. In dit geval wordt getracht te voorkomen dat een resource idle wordt. Dit gebeurt door jobs, die zich achteraan op een QpR bevinden, te verplaatsen naar een andere QpR die leeg is. Op deze manier wordt voorkomen dat een reeks van resources idle blijft wanneer andere resources nog tal van taken moeten uitvoeren. Figuur 17.5 toont de werking van dit proces.

De derde taak tracht om te gaan met inexacte runtimes. Wanneer de Job runtimes exact gekend zijn, kunnen zich in principe geen problemen voordoen. In het geval de runtimes echter voorspeld worden, kunnen deze voorspellingen afwijken van de effectieve waarde.



Figuur 17.4: Er wordt sequentieel voor elke job een plaats gezocht in een availability window volgens de gesorteerde window lijst.



Figuur 17.5: Jobs in de QpRs worden verplaatst naar lege queues om aldus de load te balancen.

Er zijn twee situaties waarin erachter kan gekomen worden dat een runtime niet correct is:

1. Indien een job reeds langer wordt uitgevoerd dan de vooropgestelde runtime.

De vooropgestelde execution time *pet* wordt in dit geval vervangen door:

$$pet = \min(pet + meanerror, pet * meanerrorfactor, maxuptime). \quad (17.2)$$

Hier stelt $meanerror = \frac{\sum_{i=0}^{n} |et_i - pet_i|}{n}$ het gemiddelde verschil tussen de vooropgestelde en de effectieve runtime voor, met n het aantal reeds uitgevoerde Jobs, et_i de executiontime van Job i en pet_i de vooropgestelde execution time van Job i. De $meanerrorfactor = \frac{\sum_{i=0}^{n} et_i/pet_i}{n}$ stelt de factor voor waarmee de vooropgestelde runtime gemiddeld van de effectieve afwijkt. *maxuptime* geeft de langste uptime van de resterende resources weer. Het gebruik van deze drie factoren maakt dat de aanpassingen realistisch blijven en in de aanwezige resources blijven passen.

2. Indien de job runtimes worden aangepast door nieuwe verwachtingen (bijvoorbeeld door GIPSy). De vooropgestelde runtime *pet* wordt in dit geval vervangen door de nieuwe verwachtingswaarde.

Deze runtime aanpassingen kunnen tot gevolg hebben dat de geschedulde Jobs het availability window van een worker overschrijden. Indien dit het geval is, worden de volgende stappen ondernomen:

- De laatste Job van de QpR wordt verwijderd tot de geschedulde Jobs binnen het availability window passen.
- Indien de QpR leeg is, wordt de laatste job in de lokale queue van de resource verwijderd.
- Indien al de voorgaande queues leeg zijn, wordt de Job, die op de Worker uitgevoerd wordt, verder afgehandeld in de hoop dat ze alsnog binnen het availability window afgehandeld zal zijn. Reschedulen heeft in dit geval geen zin omdat het teniet doen van het reeds uitgevoerde werk en de rescheduling kost niet opwegen ten opzichte van de kans dat deze predictie ook niet correct is en de Job, in tegenstelling tot de voorspelling, wel succesvol beëindigd kan worden.

17.3 Implementatie

Zoals reeds werd aangehaald, is PGS een implementatie van een scheduler voor het CoBRA framework. In essentie komt de implementatie dus neer op een uitbreiding van de standaard Scheduler component. De uitgebreide implementatie omvat vier nieuwe componenten die in twee groepen kunnen opgesplitst worden:

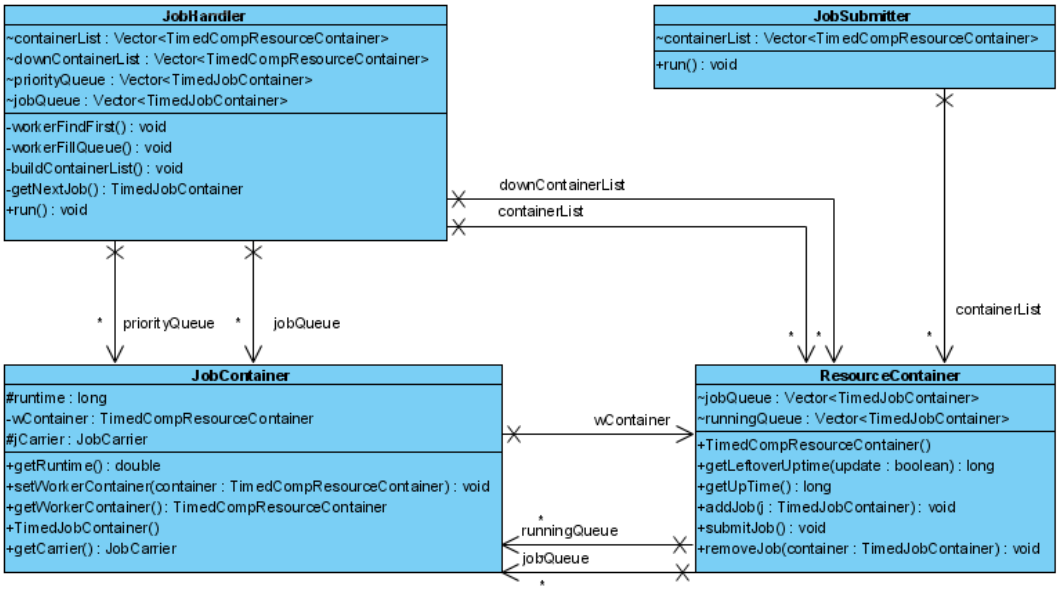
- *Data georiënteerde componenten*: Deze componenten bevatten de extra informatie om deze geavanceerde scheduling techniek te ondersteunen:

  - De `JobContainer`: Dit object bevat de essentiële Job gerelateerde informatie zoals: de Job zelf, onder de vorm van een JobContainer, de Resource waarop deze Job gescheduled wordt (indien dit al bekend is) en de Runtime informatie van de Job.
  - De `ResourceContainer`: Dit object bevat de essentiële Resource gerelateerde informatie zoals: een interface naar de resource, onder de vorm van een ResourceCarrier, het availability window en de QpR.

- *Uitvoerings georiënteerde componenten*: Zoals reeds werd aangegeven, dient een Scheduler twee taken uit te voeren. Het plaatsen van de Jobs uit de globale queue in de QpRs en het verplaatsen van de Jobs in de QpRs naar de desbetreffende resource. Deze twee functies worden opgesplitst in twee threads:

  - De `JobHandler`: Deze thread maakt gebruik van de JobContainer en van de ResourceContainer informatie om de Jobs vanuit de Globale queue in de QpRs te plaatsen volgens de techniek besproken in sectie 17.2.3.
  - De `JobSubmitter`: Deze thread verstuurt Jobs uit de QpRs naar de lokale queues in de Resources zelf, wanneer het aantal Jobs in deze lijst onder een vooraf bepaald minimum zakt.

De dataobjecten bevinden zich in lijsten in de implementatie van de nieuwe Scheduler klasse. De twee threads worden tijdens de initialisatie van de Scheduler gestart met deze lijsten als parameters. Het klassediagram van deze vier componenten wordt getoond in figuur 17.6.



Figuur 17.6: Vereenvoudigd model van het PGS klassediagram.

17.4 Integratie van GIPSy

Het uiteindelijk doel van de Prediction based scheduler is taken verdelen volgens het PGS principe op basis van Job runtime predicties. Om dit te bewerkstelligen, dient het predictieframework GIPSy geïntegreerd te worden in de PGS scheduler.

Het integreren van GIPSy in PGS houdt drie belangrijke aspecten in:

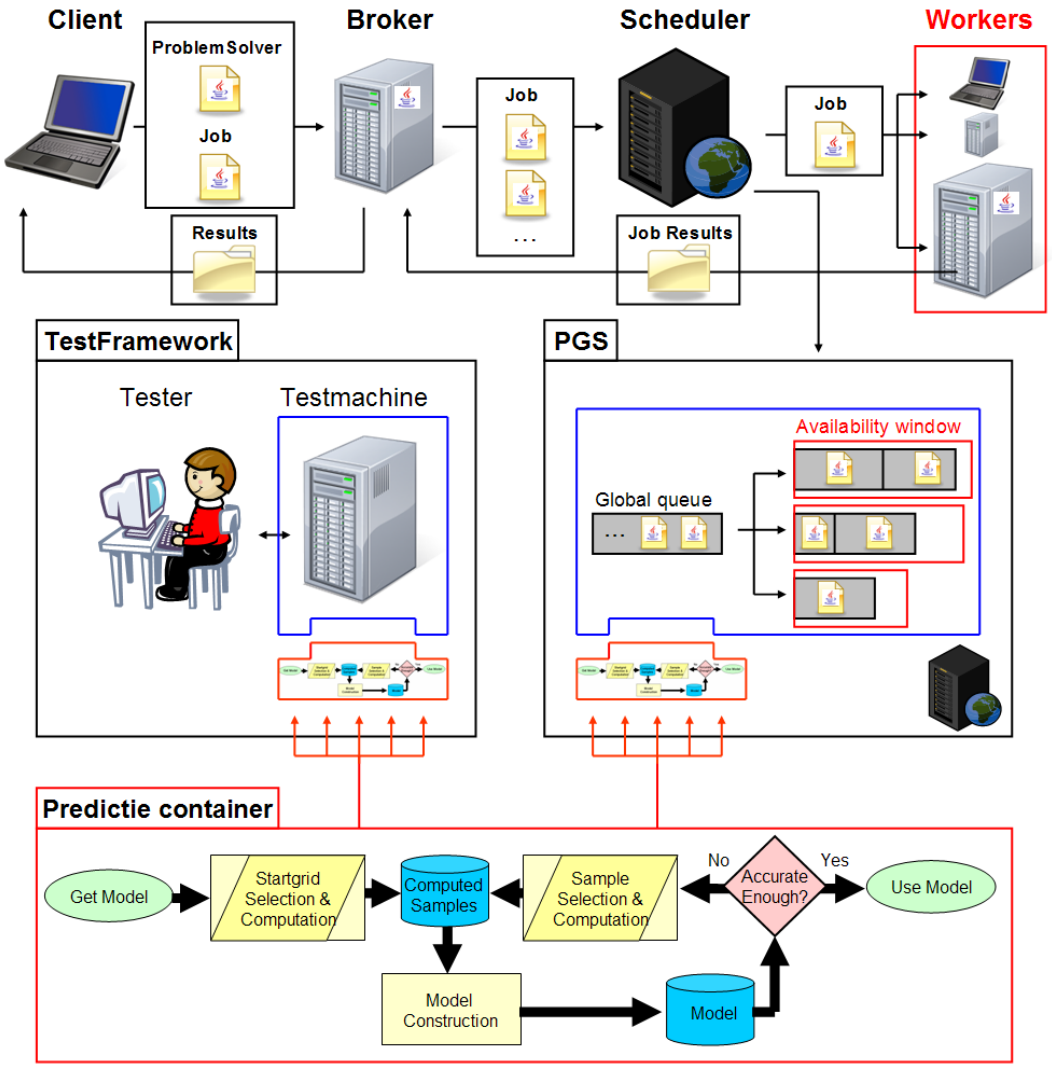
- De functionaliteit van PGS spitst zich toe op Desktop Grids. Dit heeft tot gevolg dat er over het algemeen gewerkt wordt met heterogene grids. De runtimes zullen bijgevolg verschillen van resource tot resource. Om de predicties optimaal te laten verlopen, worden de effectieve runtimes herschaald naar de runtime op een modelmachine. Dit kan statisch gebeuren door elke machine vooraf te onderwerpen aan een grondige analyse en deze resultaten te gebruiken bij de herschaling. Dit kan ook dynamisch gebeuren door op vaste tijdsintervallen een Job uit te voeren op de desbetreffende machine om aldus zijn schalingsfactor te bepalen. Deze laatste methode heeft als voordeel dat er rekening gehouden wordt met de load van het systeem.
- Om GIPSy optimaal te laten functioneren, dienen de taken in de volgorde, die het framework aangeeft, uitgevoerd te worden. Wanneer dit wordt vertaald naar de PGS scheduler, dienen de Jobs in de globale Jobqueue gesorteerd te worden volgens dit principe.
- PGS gaat ervan uit dat de runtimes van de verschillende Jobs bekend zijn en terug gevonden kunnen worden in de `ResourceContainer`. Deze vereiste kan door GIPSy worden ingevuld door de voorspelde runtimes van deze Jobs in de desbetreffende `JobContainer` te plaatsen.

Dit gehele proces kan geïmplementeerd worden door een extra thread. Deze thread dient, in eerste instantie, een model te bouwen, gebruik makend van het GIPSy framework. De data die het framework hiervoor nodig heeft, zijn de runtimes van de reeds uitgevoerde Jobs. Deze informatie is reeds in PGS aanwezig en kan eenvoudig opgevraagd worden. Om synchronisatieredenen wordt, na de opbouw van dit model, een kopie gemaakt van de Globale queue (een `JobContainer` lijst). Vervolgens wordt in de `JobContainer` van elke Job de desbestreffende runtime informatie geplaatst. Gebruik makend van een sampleselector uit het GIPSy framework, wordt de `JobContainer` lijst gesorteerd. De globale queue wordt uiteindelijk vervangen door deze gesorteerde `JobContainer` lijst.

Omdat de implementatie van deze integratie binnen het concept van de CoBRA grid dient te passen, is ervoor gekozen deze thread onafhankelijk te maken van de PGS scheduler. Dit gebeurt door de thread te starten in een aparte `CoBRAPluglet` implementatie. De PGS scheduler connecteert bijgevolg met deze Pluglet om gebruik te maken van een bepaalde Predictor. Dit heeft als belangrijk voordeel dat PGS kan

kiezen welke implementatie hij voor deze functionaliteit wil gebruiken. Concreet kunnen verschillende Predictor pluglets werken met verschillende modellen enz... .

Figuur 17.7 geeft een duidelijk overzicht van de CoBRA-PGS-GIPSy integratie:



Figuur 17.7: Integratie van CoBRA, PGS en GIPSy.

HOOFDSTUK 18

Testen en resultaten

18.1 Inleiding

De doelstelling van dit hoofdstuk is tweeledig. In de eerste plaats wordt er aangetoond dat het mogelijk en nuttig is GIPSy runtime predicties aan te wenden in een failure aware scheduling techniek (PBS). Hiernaast wordt de relevantie van het GIPSy testmechanisme geëvalueerd. Dit door de resultaten van de GIPSy tester te vergelijken met de resulaten van de prediction based scheduler (namelijk PBS). Om dit te bewerkstelligen, beschrijft sectie 18.2 de testmethodiek die gebruikt wordt om de prediction based scheduler te evalueren. Sectie 18.3 bespreekt vervolgens de desbetreffende resultaten en bevindingen. In sectie 18.4 tenslotte worden deze resultaten vergeleken met die van de GIPSy tester uit hoofdstuk 14.

18.2 Testtechniek

De techniek, die in dit hoofdstuk wordt voorgesteld, tracht te voorkomen dat er taken op een resource uitgevoerd worden wanneer deze resource onbeschikbaar wordt. Een evidente keuze om de kwaliteit van een dergelijke techniek te valideren, is het aantal reschedulingen[1] te monitoren. Het probleem met deze kwaliteitsmaat is dat deze niet steeds garant staat voor een betere performantie. Het blijkt immers zo te zijn dat het reschedulen van lange taken en van taken, die zich nabij het einde van de volledige uitvoering bevinden, een significant grotere impact hebben dan de overige taken. Om de kwaliteitscontrole te optimaliseren, zal de kwaliteitsmaat bepaald worden

[1] Het aantal keer dat een job zich op een resource bevindt die onbeschikbaar wordt en op een alternatieve resource gescheduled moet worden.

door de round-trip time van de volledige distributie. Deze methodiek reflecteert rechtstreeks de manier waarop een gebruiker de werking van het systeem ervaart.

18.3 Testen en resultaten

Net zoals in deel I omtrent CoBRA en in deel II omtrent GIPSy wordt ook in dit hoofdstuk gebruik gemaakt van het quantum fysica probleem en van de twee testsets gegenereerd door de testsetgenerator. In tegenstelling tot de GIPSy testen wordt, in dit geval, wel de effectieve uitvoering van het quantum fysica probleem als testcase gebruikt. Meer informatie is terug te vinden in tabel 18.1.

	Aantal taken(T)	Sequentiële runtime($T \times t_m$)	Beschrijving
QF Test	5670	$\pm$ 1290000 s	Veel kleine taken + enkele grote
Test 1	4000	23907,2033 s	Evenveel kleine als grote taken
Test 2	2457	100769,1452 s	Veel grote taken + enkele kleine

Tabel 18.1: Eigenschappen van de drie gebruikte tests.

Elk van deze testen werd tien maal uitgevoerd voor verschillende scheduling instellingen. Naast een distributie met een standaard *first-come, first-served* versie, werden de verschillende predictiecomponent implementaties, voorgesteld in hoofdstukken 10 en 12 omtrent GIPSy, achtereenvolgens in de prediction based scheduler geplugd. Omdat een volledige beschrijving van al de mogelijke combinaties van deze implementaties te ver zou leiden, worden in dit hoofdstuk de combinaties met de beste resultaten vergeleken. In concreto komt het erop neer dat volgende modellen:

- Kriging model met een inverse multiquadric kernel (KrigingIMQ).
- Radiale basis functie model met een multiquadric kernel (RBFMQ).
- Nearest neighbour model op basis van kriging modellen met een inverse multiquadric kernel (NNKrigingIMQ).

worden gecombineerd met de volgende sampleselectors:

- Largest distance sampleselector (LDSS).
- Smallest distance sampleselector (SDSS).
- Sorted sampleselector [2](SoSS).

Het valt meteen op dat deze sampleselectors slechts gedeeltelijk overeenkomen met deze besproken in sectie 14 van het GIPSy framework. De reden hiervoor wordt in detail besproken in sectie 18.4.

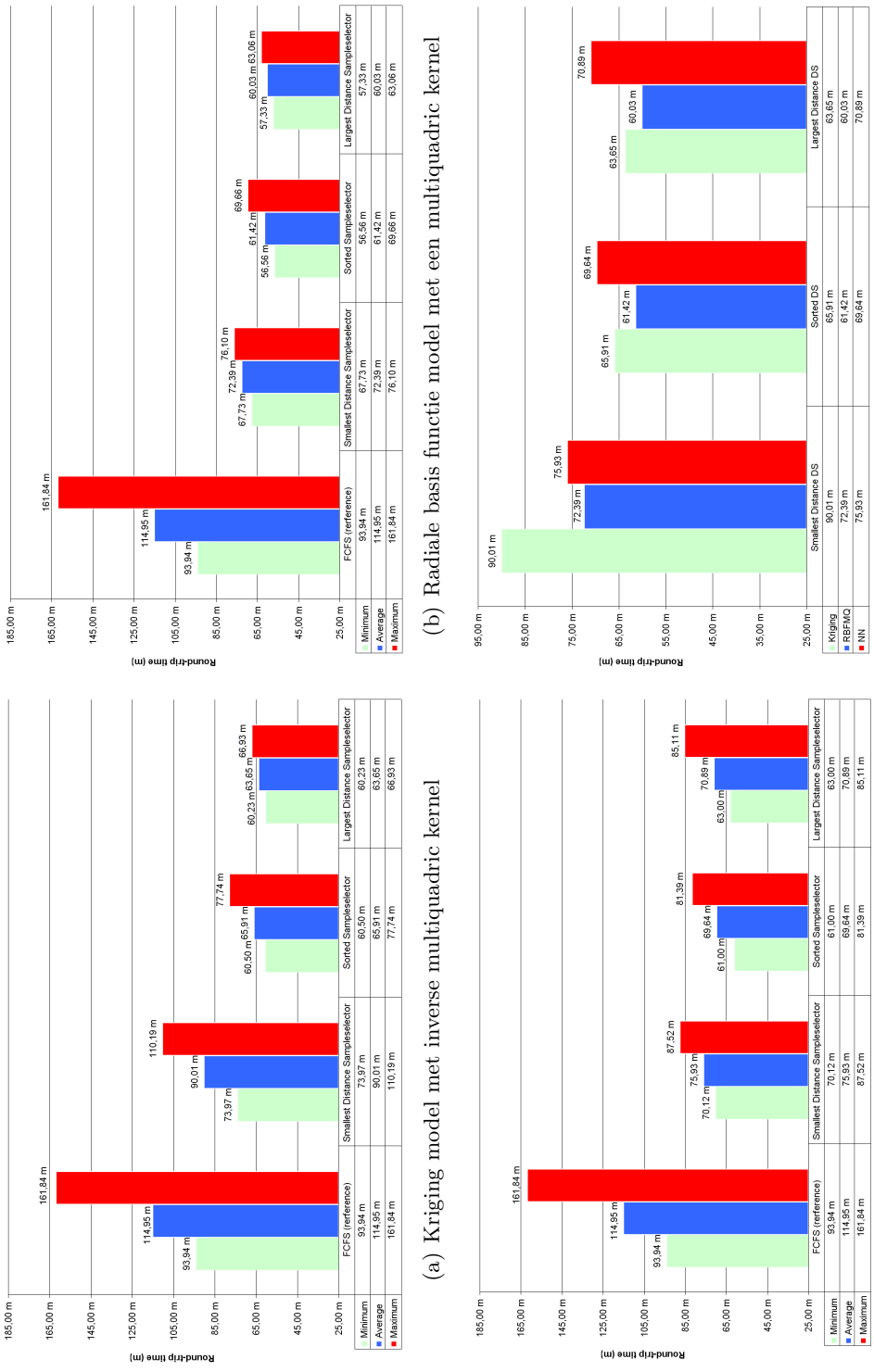
[2]Deze sampleselector verspreidt steeds de taken met de grootste voorspelde runtime.

Om een realistisch beeld te krijgen van een desktop grid systeem, waarin de beschikbaarheid van desktop machines onzeker is, wordt elk van de verschillende resources volgens een vast interval gestopt en vervolgens weer gestart. Concreet krijgt elke resource een vaste uptime gedurende welke hij beschikbaar is en waarna hij herstart wordt. Deze uptime bepaalt dan ook onmiddellijk het availability window dat in de scheduler gebruikt wordt. De uptime, ut_i, van worker i, worden bepaald aan de hand van een evenredige spreiding, tussen een minimale uptime mu en een maximale uptime Mu, over het aantal beschikbare resources n:

$$ut_i = i\frac{Mu - mu}{n} \tag{18.1}$$

In deze concrete tests wordt de uptime, die zich in het interval $mu = 25$ s en $Mu = 1250$ s bevindt, evenredig verspreid over 50 resources. Deze opstelling geeft een correct beeld van een gridsysteem waarop de vooropgestelde probleemstellingen op een verantwoorde manier gedeployed kunnen worden. De ondergrens wordt vastgelegd op de kortste bruikbare uptime. De bovengrens wordt vastgelegd omdat de relevantie van machines die langer up zijn, enkel getest kan worden door distributies met langere uitvoeringstijden. Dit is als gevolg van de tijdsconstraints vrijwel onmogelijk te testen.

Om de beste modellerings- en sampleselectietechniek te selecteren worden voor elke testset (quantum fysica, testset 1 en testset 2) vier plots gehanteerd(figuur 18.1, figuur 18.2 en figuur 18.3). Elke figuur bestaat uit vier plots. De eerste drie plots vergelijken elk de resultaten van de verschillende sampleselectietechnieken (SDSS, SoSS, LDSS) voor een welpebaalde modelleringstechniek (KrigingIMQ, RBFMQ, NNKrigingIMQ). In elk van deze plots wordt het resultaat van een first-come, first-served scheduler als referentiepunt toegevoegd. De vierde plot plaatst de gemiddelde round-trip time van elk van de verschillende technieken op één grafiek om aldus de performantie globaal te kunnen vergelijken.



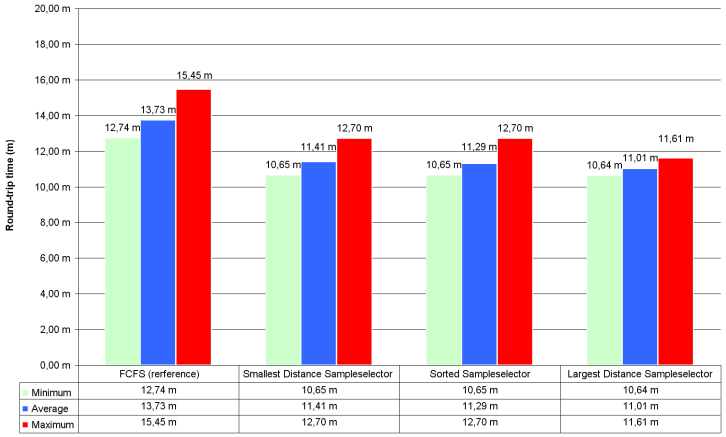
(a) Kriging model met inverse multiquadric kernel

(b) Radiale basis functie model met een multiquadric kernel

(c) Nearest neighbour model op basis van kriging modellen met een inverse multiquadric kernel

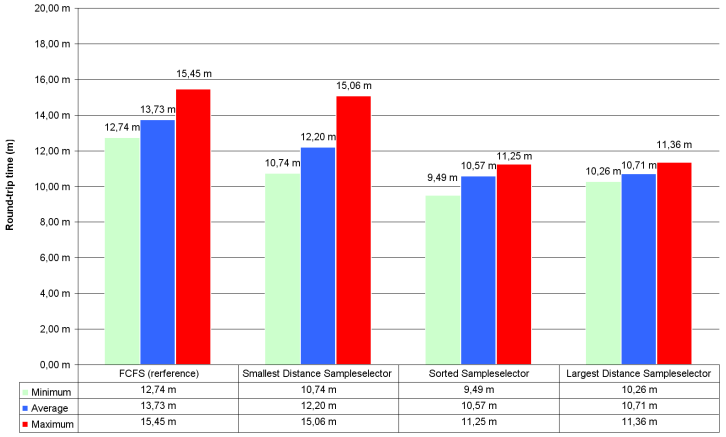
(d) Vergelijking van de modellen per sampleselector

Figuur 18.1: Testresultaten met betrekking tot de distributie van het quantum fysica probleem.



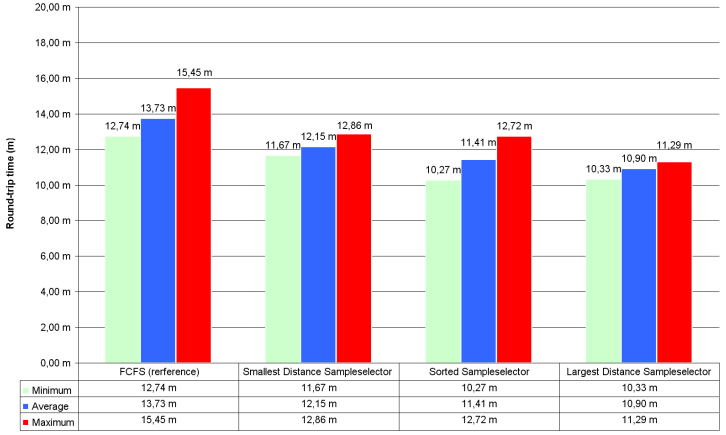
	FCFS (rerference)	Smallest Distance Sampleselector	Sorted Sampleselector	Largest Distance Sampleselector
Minimum	12,74 m	10,65 m	10,65 m	10,64 m
Average	13,73 m	11,41 m	11,29 m	11,01 m
Maximum	15,45 m	12,70 m	12,70 m	11,61 m

(a) Kriging model met inverse multiquadric kernel



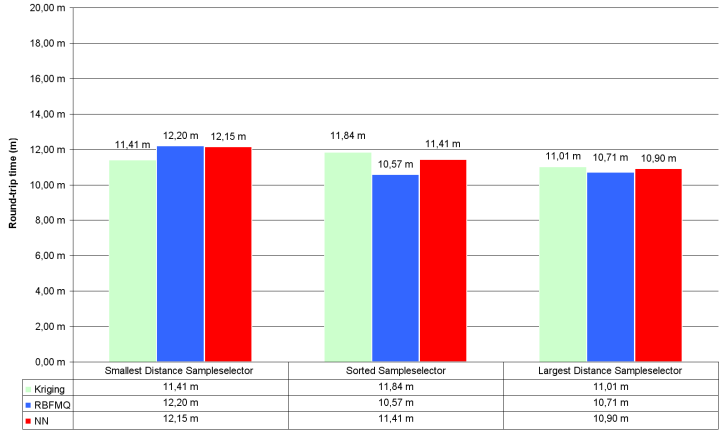
	FCFS (rerference)	Smallest Distance Sampleselector	Sorted Sampleselector	Largest Distance Sampleselector
Minimum	12,74 m	10,74 m	9,49 m	10,26 m
Average	13,73 m	12,20 m	10,57 m	10,71 m
Maximum	15,45 m	15,06 m	11,25 m	11,36 m

(b) Radiale basis functie model met een multiquadric kernel



	FCFS (rerference)	Smallest Distance Sampleselector	Sorted Sampleselector	Largest Distance Sampleselector
Minimum	12,74 m	11,67 m	10,27 m	10,33 m
Average	13,73 m	12,15 m	11,41 m	10,90 m
Maximum	15,45 m	12,86 m	12,72 m	11,29 m

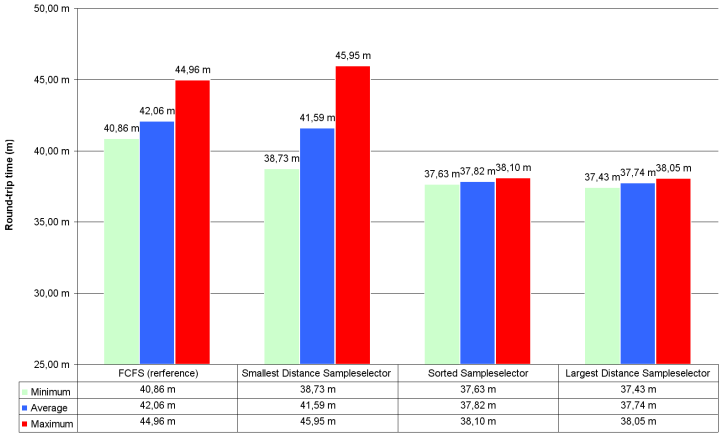
(c) Nearest neighbour model op basis van kriging modellen met een inverse multiquadric kernel



	Smallest Distance Sampleselector	Sorted Sampleselector	Largest Distance Sampleselector
Kriging	11,41 m	11,84 m	11,01 m
RBFMQ	12,20 m	10,57 m	10,71 m
NN	12,15 m	11,41 m	10,90 m

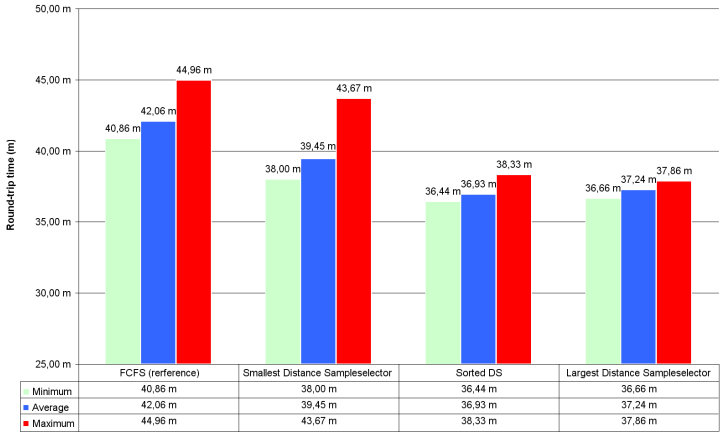
(d) Vergelijking van de modellen per sampleselector

Figuur 18.2: Testresultaten met betrekking tot testset 1.



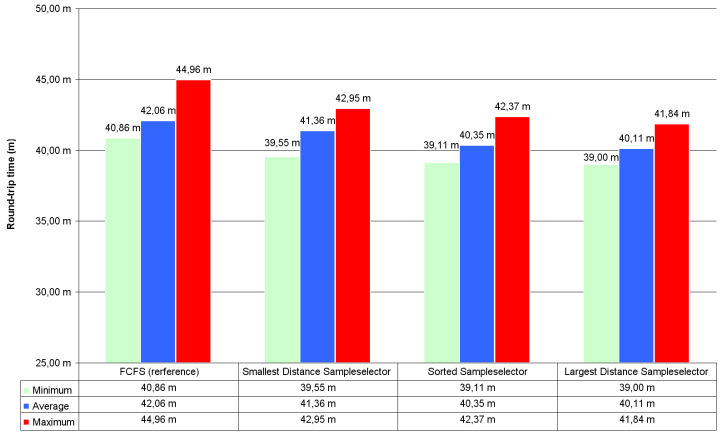
	FCFS (rerference)	Smallest Distance Sampleselector	Sorted Sampleselector	Largest Distance Sampleselector
Minimum	40,86 m	38,73 m	37,63 m	37,43 m
Average	42,06 m	41,59 m	37,82 m	37,74 m
Maximum	44,96 m	45,95 m	38,10 m	38,05 m

(a) Kriging model met inverse multiquadric kernel



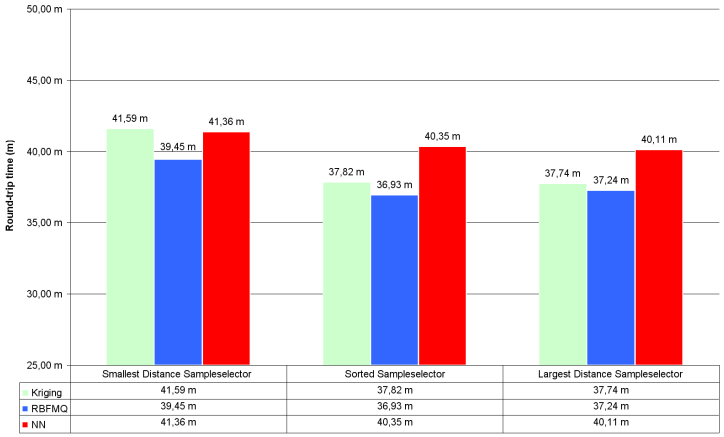
	FCFS (rerference)	Smallest Distance Sampleselector	Sorted DS	Largest Distance Sampleselector
Minimum	40,86 m	38,00 m	36,44 m	36,66 m
Average	42,06 m	39,45 m	36,93 m	37,24 m
Maximum	44,96 m	43,67 m	38,33 m	37,86 m

(b) Radiale basis functie model met een multiquadric kernel



	FCFS (rerference)	Smallest Distance Sampleselector	Sorted Sampleselector	Largest Distance Sampleselector
Minimum	40,86 m	39,55 m	39,11 m	39,00 m
Average	42,06 m	41,36 m	40,35 m	40,11 m
Maximum	44,96 m	42,95 m	42,37 m	41,84 m

(c) Nearest neighbour model op basis van kriging modellen met een inverse multiquadric kernel



	Smallest Distance Sampleselector	Sorted Sampleselector	Largest Distance Sampleselector
Kriging	41,59 m	37,82 m	37,74 m
RBFMQ	39,45 m	36,93 m	37,24 m
NN	41,36 m	40,35 m	40,11 m

(d) Vergelijking van de modellen per sampleselector

Figuur 18.3: Testresultaten met betrekking tot testset 2.

Het valt onmiddellijk op dat elk van de resultaten beter presteert dan het first-come, first-serve mechanisme. Hier dient onmiddellijk bij opgemerkt te worden dat de resultaten voor deze first-come, first-served methode afkomstig zijn van 10 test-runs en dat de bovengrens van deze methode, in principe, onbepaald is. Het is immers zo dat een job telkens gerescheduled kan worden op een machine die onbeschikbaar wordt alvorens de job beëindigd is. Een Prediction based scheduler voorkomt deze situatie. Een voorspelling moet immers steeds binnen het availability window van een resource passen alvorens de taak op een resource gescheduled kan worden en de verwachte runtime wordt steeds verhoogd indien de huidige predictie te kort bleek te zijn.

Wanneer de verschillende modelleringstechnieken met elkaar vergeleken worden, blijkt, in eerste instantie, dat de resultaten van de verschillende modellen voor test-set 1 vrijwel vergelijkbaar zijn. De variantie is hier immers groter dan het verschil tussen de verschillende resultaten. Dit fenomeen is te verklaren door de aard van de testset. De gemiddelde runtime van een job is immers kleiner dan 6s en het eerste model kan slechts geconstrueerd worden nadat elke job gesubmit is ($\pm$4 minuten). In deze tijdspanne zijn er echter al meer dan 2000 jobs uitgevoerd. De modellen die, aan de hand van deze punten van de desbetreffende dataset gemaakt kunnen worden, zijn vergelijkbaar, wat maakt dat het verdere verloop van de distributie voor elk van deze modelleringstechnieken vrij analoog zal verlopen. Hetzelfde kan gezegd worden over de sampleselectietechnieken.

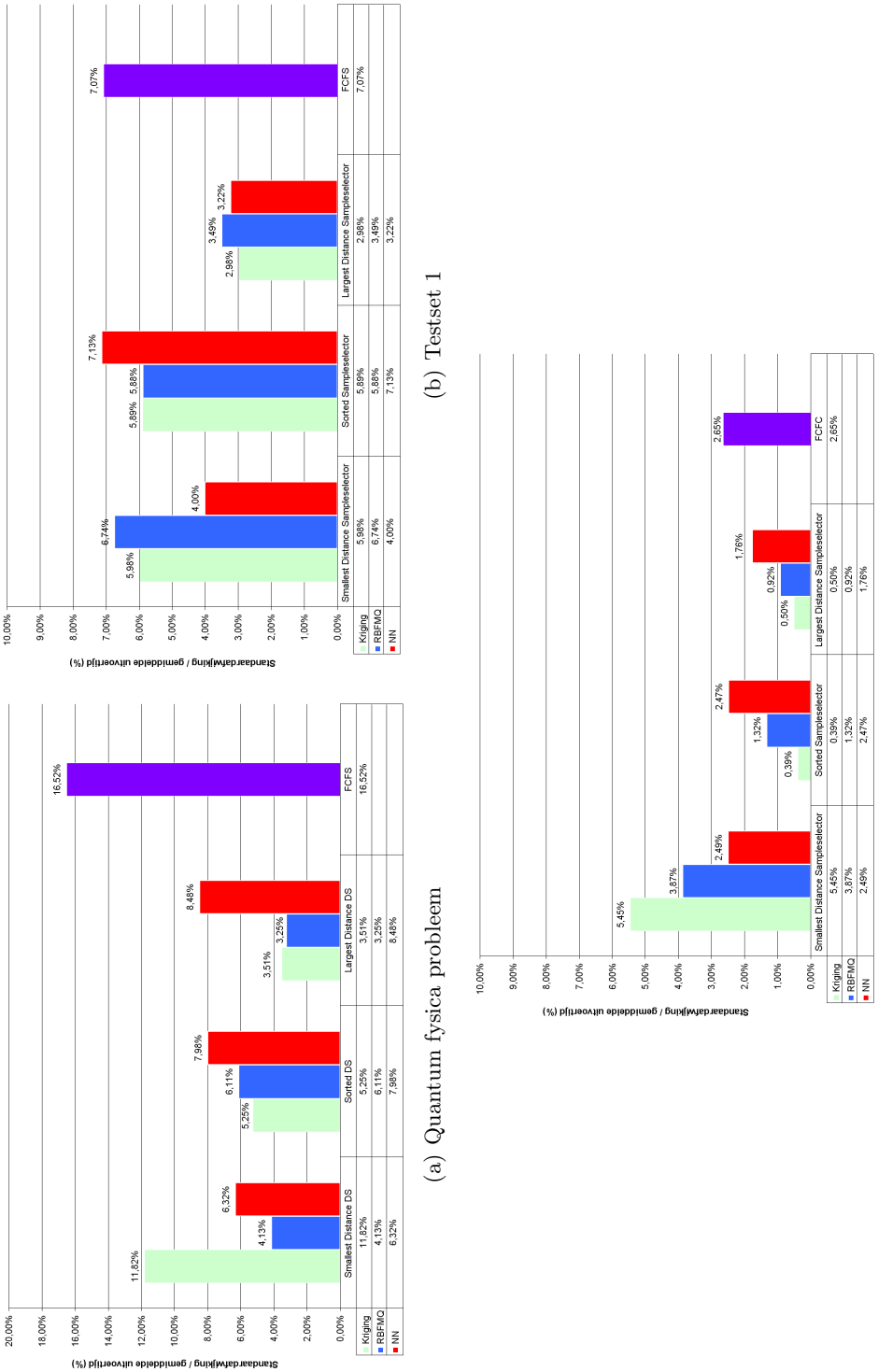
Een vergelijking van de testen, op basis van de overige datasets, wijst uit dat de predicties, aan de hand van radiale basisfunctie modellen met een multiquadric kernel, steeds het beste resultaat opleveren. De kriging techniek met inverse multiquadric kernel blijkt eveneens goed te functioneren wanneer de juiste sampleselectietechniek wordt aangewend (largest distance sample selectie). De NN-Predictietechniek daarentegen presteert significant slechter maar blijft, zoals reeds aangehaald in sectie 14.4.5, performanter bij het opbouwen van modellen. In concreto betekent dit dat deze techniek gebruikt dient te worden wanneer het opbouwen van een model met de overige technieken computationeel te zwaar wordt.

Wanneer de verschillende sampleselectietechnieken worden vergeleken, blijkt dat de sorted sampleselector en de largest distance sampleselector vergelijkbare resultaten behalen. De analogie in deze resultaten is niet zozeer te wijten aan de vergelijkbaarheid van de technieken, maar aan de unieke kracht die elk van deze sampleselectors bezit. De sorted sampleselector haalt de verwezenlijkte tijdswinst uit het principe: “Lange taken, met hogere prioriteit schedulen”. Dit heeft tot gevolg dat, bij het eind van de distributie, enkel korte taken overblijven en bijgevolg de verschillende availability windows mooi evenredig opgevuld kunnen worden. De largest distance sampleselector heeft dan weer als voordeel dat de sampleselectiepunten evenredig over heel de input space verspreid worden. Dit heeft tot gevolg dat er vrij snel runtime informatie aanwezig is in elke regio van de input space en er aldus vrij snel accurate voorspellingen kunnen gebeuren. Deze accuraatheid bevordert de

scheduling techniek en minimaliseert bijgevolg automatisch de round-trip time. Een combinatie van de krachtpunten van deze twee methoden zou, naar de toekomst toe, tot nog betere resultaten kunnen leiden. In concreto dient er getracht te worden, aan de hand van de largest error sampleselector, te werken naar een afdoend precies model. Van zodra deze precisie verwezenlijkt wordt, dienen de overige punten geselecteerd te worden, aan de hand van de sorted sampleselector, zodat de overgebleven jobs met de langste runtimes als eerste afgehandeld kunnen worden. Deze combinatie verbindt de kracht van beide methoden en zal aldus, naar alle waarschijnlijkheid, tot betere resultaten leiden.

Tot slot is het belangrijk de kwaliteit van deze testen te verifiëren. Aangezien elk van deze tests 10 of meerdere keren uitgevoerd is, kan deze kwaliteit het best bepaald worden aan de hand van de standaardafwijking. Figuur 18.4 toont de verhouding van deze standaardafwijking ten opzichte van de effectieve uitvoertijd uitgedrukt in percentages. Uit deze resultaten blijkt dat deze afwijking bij prediction based testen, op twee uitzonderingen na, steeds ruim onder de 8% blijft. Aangezien de performantie winst varieert van 12%, bij dataset 2, tot 48%, bij het quantum fysica probleem, kan besloten worden dat de conclusies uit deze testen onafhankelijk zijn van de variantie. Hierbij dient nog opgemerkt te worden dat de standaard afwijking bij gebruik van de beste technieken (RBFMQ met een Largest Distance sampleselector) minimaal is. Deze afwijking is echter nog lager (onder de 4%) indien steeds, het RBFMQ model in combinatie met de largest distance sampleselector, wordt gebruikt. Omwille van deze reden wordt in deze thesis dan ook deze combinatie aangeraden.

Concreet geeft het RBFMQ model, in combinatie met de largest distance sampleselector, de beste resultaten. De snelheidswinst, ten opzichte van de gemiddelde runtime van de first-come, first-served scheduler, varieert van 12% bij testset 2, over 23% bij testset 1, tot 48% bij het quantum fysica probleem. Dit bewijst de toepasbaarheid van de prediction based scheduling techniek.



(a) Quantum fysica probleem

(b) Testset 1

(c) Testset 2

Figuur 18.4: De verhouding standaard afwijking / gemiddelde runtime.

18.4 Relevantie van GIPSy

In deze sectie worden de resultaten van de voorgaande testen vergeleken met hetgeen het GIPSy framework in hoofdstuk 14 had voorspeld. In concreto worden de resultaten uit Deel 1 (sectie 14.4.3) en Deel 2 (sectie 14.4.4) vergeleken met de prediction based scheduler testen uit sectie 18.3.

Het eerste deel van de GIPSy testen handelde over de verschillende modelleringstechnieken en hun verwachtte prestaties. Er werd geconcludeerd dat slechts drie modelleringstechnieken in aanmerking kwamen om de gewenste predicties te realiseren. Wanneer deze conclusies vergeleken worden met de conclusies omtrent de prediction based scheduling testen, blijkt dat deze modelleringstechnieken overeenkomen. Het blijkt dus dat er effectief slechts drie van de voorgestelde modelleringstechnieken in aanmerking komen om de prediction based scheduling te ondersteunen, namelijk, deze voorgesteld door de GIPSy tester:

- Kriging model met een inverse multiquadric kernel (KrigingIMQ).
- Radiale basis functie model met een multiquadric kernel (RBFMQ).
- Nearest neighbour model op basis van kriging modellen met een inverse multiquadric kernel (NNKrigingIMQ).

Dit toont aan dat de GIPSy tester een perfect beeld kan geven van de modelleringstechniek die best gebruikt wordt om een bepaalde doelstelling te bereiken.

Het tweede deel van de GIPSy testen handelde over de verschillende sampleselectietechnieken en hun verwachte prestaties. Er werd geconcludeerd dat slecht drie technieken in aanmerking kwamen voor de prediction based scheduling:

- De Random sampleselector (RaSS).
- De Fluctuation sampleselector (FlSS).
- De Smallest distance sampleselector (SDSS).

Uit deze groep van technieken bleek de Smallest distance sampleselector veruit de beste keuze te zijn. Wanneer deze resultaten worden vergeleken met de resultaten uit de prediction based scheduler, blijken deze resultaten echter totaal verschillend. Enkel de SDSS is in de lijst van valabele sampleselectietechnieken terug te vinden:

- Largest distance sampleselector (LDSS).
- Smallest distance sampleselector (SDSS).
- Sorted sampleselector (SoSS).

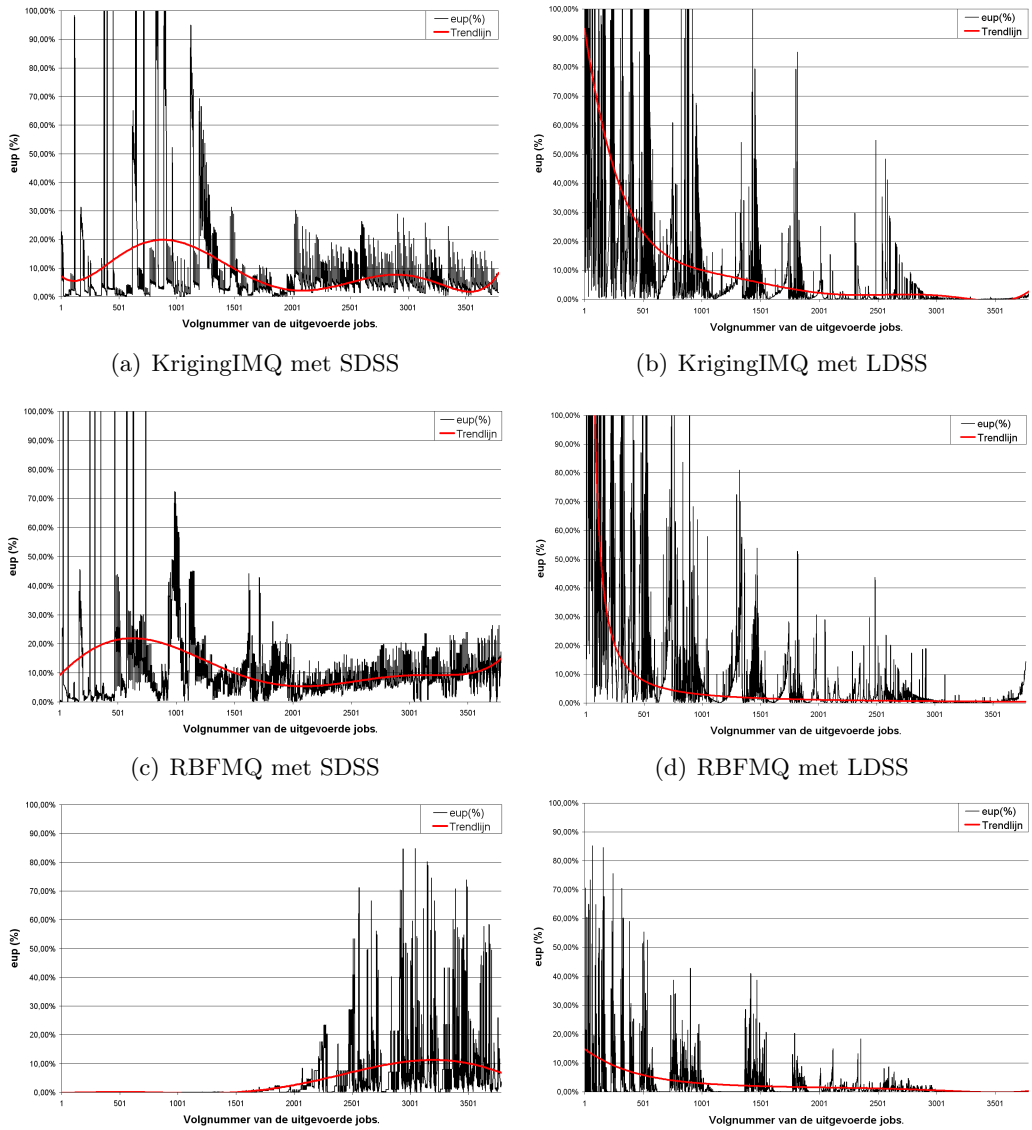
In eerste instantie kan er aangenomen worden dat de GIPSy tester niet in staat is valabele sampleselectors te selecteren. Deze conclusie is echter fout. De oorzaak van de inconsistentie is terug te vinden in een denkfout. In hoofdstuk 14 werd aangenomen dat de kwaliteit van een prediction based scheduler het best bepaald kon worden aan de hand van de *error van de uitgevoerde punten (eup)*. Deze veronderstelling is gedeeltelijk correct. Er zijn echter twee extra factoren die de round-trip time van een prediction based scheduler beïnvloeden. Het is enerzijds zo dat de *eup* waarden van de voorspelde punten veel belangrijker zullen zijn aan het einde van de uitvoering van een gedistribueerd probleem. Een rescheduling van de laatste taak zal er immers voor zorgen dat de round-trip time van het volledige probleem met de desbetreffende taaklengte wordt verlengd. Anderzijds is het ook zo dat de uitvoeringstijd van de taken op het einde van de distributie de round-trip time beïnvloedt. De availability windows van de verschillende resources kunnen immers eenvoudiger gelijkmatig opgevuld worden wanneer de taken die zich aan het einde van de distributie bevinden kort zijn.

Hoewel de largest distance sampleselector globaal verantwoordelijk is voor slechtere *eup* waarden, blijkt dat de *eup* waarden, naar het einde van de distributie toe, aanzienlijk lager liggen dan bij elke andere gebruikte sample selectietechniek (figuur 18.5 toont dit voor de RBF). Dit is eenvoudig te verklaren omdat de samplepunten, door deze selector, evenredig verdeeld worden over de inputspace en er bijgevolg, aan het eind van de distributie, in elk gedeelte van de inputspace afdoende informatie aanwezig is om kwaliteitsvolle predicties uit te voeren.

Ook de sorted sampleselector zal globaal verantwoordelijk zijn voor slechtere *eup* waarden. De sampleselector zal er echter voor kiezen ten allen tijde de taken, met de grootste voorspelde uitvoeringstijd te verdelen. Dit heeft tot gevolg dat aan het einde van de distributie al de grote taken reeds uitgevoerd zijn en de resterende taken eenvoudig evenredig verdeeld kunnen worden over de verschillende resources. Wat de round-trip time, zoals reeds aangehaald, positief beïnvloedt.

Uit deze uiteenzetting blijkt ook meteen dat de combinatie van de largest distance sampleselector en de sorted sampleselector een goed idee zou zijn. Op deze manier zullen de runtimes op het einde van de distributie zowel klein als exact voorspeld zijn.

Er kan dus bevestigd worden dat de GIPSy tester uitstekend werkt. Er dient echter ten allen tijde rekening gehouden te worden met elke mogelijke inputfactor. Iets wat duidelijk niet gebeurde in de voorafgaande GIPSy testen.



(a) KrigingIMQ met SDSS

(b) KrigingIMQ met LDSS

(c) RBFMQ met SDSS

(d) RBFMQ met LDSS

(e) NNKrigingIMQ met SDSS

(f) NNKrigingIMQ met LDSS

Figuur 18.5: Verschil in eup spreiding tussen smallest distance sampleselector (SDSS) en largest distance sampleselector (LDSS).

18.5 Conclusie

De verschillende testen tonen aan dat het gebruik van een prediction based scheduler de round-trip time van een distributie drastisch vermindert. Het gebruik van een radiaal basis model op basis van een multiquadric kernel blijkt hierbij de beste resultaten te geven. De round-trip time wordt in dit geval verminderd met 12% bij testset 2, over 23%, bij testset 1, tot 48% bij het quantum fysica probleem.

Naast deze vaststelling werd tevens aangetoond dat de kwaliteit van zowel de modellen als van de sampleselectors, voor een specifieke scheduling techniek, bepaald kan worden aan de hand van de GIPSy tester. Dit echter enkel op voorwaarde dat de juiste kwaliteitsvoorwaarden gehanteerd worden. Concreet betekent dit dat zowel het opzet van de prediction based scheduler als het opzet van het GIPSy framework als geslaagd beschouwd kunnen worden.

HOOFDSTUK 19

Conclusie

In deel III werd het nut van een prediction based scheduler bewezen. Concreet werd PGS, een prediction based scheduling techniek, voorgesteld. Deze techniek tracht, op basis van runtimevoorspellingen uit het GIPSy framework, tot een optimale schedule te komen. In essentie komt het erop neer dat de voorspelde runtimes weloverwogen op de availability windows worden gemapt, om aldus rescheduling tegen te gaan.

De ontwikkeling van de scheduler beruste op drie principes. In de eerste plaats dienen de taken optimaal verdeeld te worden over de verschillende availability windows. In de tweede plaats dient rekening gehouden te worden met eventuele fouten in de voorspellingen. Tot slot dient de scheduling te gebeuren op basis van de GIPSy predictie technieken. Om dit optimaal te laten verlopen, dient er een wisselwerking te zijn tussen GIPSy en de scheduler. Dit impliceert dat de scheduler rekening moet houden met de waarde die het GIPSy framework hecht aan een gekende runtime. De scheduler zal, met andere woorden, Jobs met een meer waardevolle runtime informatie trachten voorrang te geven.

In hoofdstuk 18 werd aangetoond dat het gebruik van de juiste predictiemechanismen leidt tot een kortere round-trip time. Er werd meer bepaald aangetoond dat de roundtrip time, verwezenlijkt aan de hand van predicties gebaseerd op een radiaal basisfunctie model met multiquadric kernel, met 12% tot zelfs 48% verkort kan worden.

Naast deze vaststelling bleek tevens dat de kwaliteit van de predictiemodellen overeenkomt met deze voorspeld door de GIPSy tester uit deel II van deze thesis. De voorspelling omtrent de sampleselectie bleek echter niet correct. Het gebruik van een foute veronderstelling lag aan de basis van deze inconsistentie. Het gebruik van de juiste kwaliteitsbepaling leverde echter wel goede resultaten op. Er werd bijgevolg besloten dat het gebruik van de GIPSy tester adequaat kan oordelen over de relevantie van de verschillende predictiemechanismen binnen een bepaalde scheduling

techniek.

Concreet werd aangetoond dat prediction based scheduling op basis van GIPSy predictie componenten tot goede resultaten leidt en dat de kwaliteit van de gebruikte predictiecomponenten succesvol vooraf bepaald kan worden door de GIPSy tester. Het opzet van zowel de prediction based scheduler als van het GIPSy framework kan bijgevolg als geslaagd beschouwd worden.

Deel IV

Conclusie en toekomst perspectief

HOOFDSTUK 20

Conclusie

Zoals de inleiding reeds aangaf, situeert deze thesis zich in de evolutie naar het gebruik van resources uit een desktop grid. Concreet werden drie uitdagingen, die met deze evolutie gepaard gingen, aangegaan en tot een goed einde gebracht.

In de eerste uitdaging werd op zoek gegaan naar een voedingsbodem voor het huidige desktop grid onderzoek. Concreet diende een synergie gecreëerd te worden die de implementatieprocedure van een gebruiker versnelt en die de ontwikkeling van nieuwe technieken door een developer, met feedback van de gebruiker, ondersteunt. Er werd aangetoond dat zowel de HWG (heavy weight grid) als de LWG (light weight grid) over onvoldoende capaciteiten beschikten om dit te verwezenlijken. In het eerste gedeelte van deze thesis werd daarom een nieuw framework voorgesteld. De CoBRA grid, zoals dit framework werd genoemd, combineert de essentiële eigenschappen van zowel de LWGs als van de HWGs. Dit wordt gerealiseerd door door, gebruik makend van plugability, de essentiële eigenschappen van een HWG (monitoring en robuustheid) aan een LWG (flexibiliteit, aanpasbaarheid) toe te voegen. De synergie werd bewezen aan de hand van testen die zowel de ervaring van de gebruiker als die van de developer weerspiegelen. De performantietesten reflecteerden zowel de bruikbaarheid in concrete distributies als het realisme in proof of concept implementaties. Er kan aldus gesteld worden dat de eerste uitdaging tot een goed einde werd gebracht.

De tweede uitdaging poogde een concreet probleem bij gridsystemen op te lossen. Veel technieken gaan er immers vanuit dat de runtimes van de jobs gekend zijn. Dit is echter een foutieve veronderstelling. Deze informatie kan enkel verschaft worden door de gebruiker, die niet steeds betrouwbaar is. Om deze problematiek de wereld uit te helpen, werd het GIPSy framework voorgesteld. De doelstelling van dit framework is tweeledig. Het wil enerzijds een medium bieden om op een eenvoudige en een eenduidige wijze nieuwe predictietechnieken te testen en in de praktijk toe te

passen. Anderzijds wil het een proof of concept bieden dat aantoont dat het mogelijk is, gebruik makend van een weloverwogen verzameling predictietechnieken, tot een adequate runtime predictie te komen. De eerste doelstelling werd verwezenlijkt door het GIPSy framework op te bouwen rond het plugability concept. Elk vitaal onderdeel in het predictiemechanisme (intial sampleselectie, sampleselectie, modelleringstechniek) tot zelfs de kwaliteitscontrole kan zo vervangen worden door een alternatieve implementatie. Op deze manier kan elke mogelijke kwaliteitscontrole uitgevoerd worden op de verschillende implementaties van de predictie componenten. Aan de hand van deze testtechniek werd bewezen dat drie modelleringstechnieken geschikt zijn om de runtimes van parameter sweep jobs te voorspellen. Het gaat hier meer bepaald om het radiale basisfunctie model met multiquadric kernel (RBFMQ), het kriging model met inverse multiquadric kernel (KrigingMQ) en het nearest neighbours model op basis van kriging modellen met een inverse multiquadric kernel (NNKrigingMQ). Op analoge manier werd aangetoond dat vrijwel elke sampleselectietechniek voldeed aan de vereisten, maar dat de smallest distance sampleselector duidelijk beter scoorde dan de overige selectors. Er kan aldus gesteld worden dat het GIPSy framework de mogelijkheid biedt verschillende predictie technieken te vergelijken en dat er binnen de range van bestaande technieken valabele kandidaten bestaan om kwalitatieve runtime predicties uit te voeren. Dit maakt dat ook de tweede uitdaging tot een goed einde werd gebracht.

De verschillende resources in een desktop grid zijn niet ten allen tijde beschikbaar. Jobs die uitgevoerd worden op een resource die niet langer beschikbaar is voor de grid, zullen aldus herstart moeten worden op een ander resource. Het minimaliseren van de verloren tijd die hieruit voortvloeit, vormde de derde uitdaging in dit doctoraat. Concreet werd een prediction based scheduler voorgesteld die heuristieken combineert met de GIPSy predictie componenten en aldus deze rescheduling tracht te vermijden. De plugability van de GIPSy componenten impliceert dat ook hier de verschillende implementaties van elk van deze componenten gehanteerd kunnen worden. Vergelijkende testen wezen uit dat scheduling op basis van een radiale basis functie model, in combinatie met een largest distance dataselector, de round-trip time van een distributie minimaliseert. Dit zijn tevens de resultaten die voorspeld werden door de GIPSy tester wanneer de juiste kwaliteitscontole toegepast werd. Er kan aldus besloten worden dat prediction based scheduling op basis van GIPSY predictie componenten tot goede resultaten leidt en dat de kwaliteit van de gebruikte predictiecomponenten, succesvol, vooraf bepaald kan worden door de GIPSy tester. Dit bewijst enerzijds dat de derde doelstelling is bereikt en bevestigt anderzijds nogmaals het slagen in de tweede uitdaging.

Er kan tot slot besloten worden dat met deze doctoraatsthesis drie belangrijke uitdagingen op het gebied van desktopgrids tot een goed einde zijn gebracht.

HOOFDSTUK 21

Toekomst perspectief

Hoewel deze thesis slaagt in de drie vooropgestelde uitdagingen kan er, naar de toekomst toe, nog veel werk gebeuren. Voor de eenvoud wordt de bespreking eveneens opgedeeld volgens de drie delen die in deze thesis aan bod kwamen.

In het eerste gedeelte, omtrent de CoBRA grid, dient er naar de toekomst toe onderzoek gedaan te worden in drie specifieke onderdelen:

- *Backbone:* Er dient op zoek gegaan te worden naar een methode die de schaalbaarheid van de backbone onafhankelijk maakt van het aantal mogelijke socketconnecties. Er kan een poging gedaan worden om Jini en H2O uit te breiden zodat de connecties snel genoeg verdwijnen en weer opgebouwd kunnen worden. Er kan ook op zoek gegaan worden naar technieken die los staan van H2O of Jini en als robuuste CoBRA backbone dienst kunnen doen.

- *Filesysteem:* Er dient naar de toekomst toe een volledig gedistribueerd filesysteem ontwikkeld te worden zodat elk data object op elke resource aanwezig lijkt te zijn. Op deze manier kan er een vorm van redundantie gecreëerd worden. Het wordt tevens veel eenvoudiger, voor de gebruiker, bepaalde data objecten aan te spreken. De gebruiker zal immers niet langer instaan voor de transfer van de desbetreffende objecten.

- *Security:* Tot op heden is er binnen het CoBRA framework geen rekening gehouden met eventuele security problemen. Bij uitbreiding naar een gridsysteem gespreid over verschillende administratieve domeinen dient dit aangepakt te worden.

Binnen het tweede gedeelte, omtrent het GIPSy framework, liggen de onderzoeksmogelijkheden vooral in de richting van de verschillende predictiecomponenten. Er dient zowel meer onderzoek gedaan te worden naar sampleselectie technieken, als

naar startgidselectie technieken, als naar modelleringstechnieken. Tevens dienen de mogelijkheden in de kwaliteitscontrole verder uitgebreid te worden om aldus de kwaliteit voor meer diverse doelstellingen te verifiëren.

In het derde deel, omtrent de prediction based scheduler, zullen de volgende aandachtspunten belangrijk zijn:

- De scheduler zou geïntegreerd moeten worden in een gridsimulator, zoals GESS [118], GridSim [23] of SimGrid [67], om aldus de schaalbaarheid van de verschillende technieken in meer detail te kunnen uittesten.
- De scheduler dient tevens geïntegreerd te worden in andere gridsystemen om aldus de portability en toepasbaarheid van de techniek te verifiëren.
- Zoals reeds werd aangehaald, is het nuttig de largest distance sampleselector en de sorted sampleselector naar de toekomst toe te combineren. Dit zal wellicht tot betere resultaten leiden.
- Tot slot is het zeer belangrijk aan de hand van GIPSy concreet onderzoek te doen naar alternatieve predictietechnieken en deze technieken na de GIPSy evaluatie effectief uit te testen op de scheduler.

Appendices

BIJLAGE A

H2O

A.1 Inleiding

In dit hoofdstuk wordt een uiteenzetting gegeven over de werking van de H2O middleware. De kennis werd vergaard uit verschillende H2O papers en via rechtstreekse communicatie met de H2O developers. Het H2O project is ontstaan binnen het Harness project. Dit project had als doel een pluggable[1] virtuele machine op te zetten gespreid over een heterogene groep van resources. Om dit te bewerkstelligen moet elke machine in deze virtuele machine in de mogelijkheid zijn componenten toe te voegen en te verwijderen (pluggable zijn). Deze onderliggende functionaliteit wordt verzorgt door het H2O project.

Uit deze korte inleiding blijkt reeds dat de fundamentele motivatie voor deze projecten identiek is. Beide projecten willen op elke niveau pluggabiliteit ondersteunen.

[1] Pluggable betekent dat nieuwe functionaliteit ten allen tijde in of uit de virtuele machine kan *geplugd* worden.

Deze motivatie vindt zijn oorsprong in twee vaststellingen:

1. De technologische evolutie heeft zulk een snelheid bereikt dat nieuwe hardware technieken of software vereisten er vaak toe leiden dat aanpassingen aan de distributie technieken en aan de cluster technologieën noodzakelijk worden. Denk maar aan: de nieuwe mogelijkheden die hedendaagse multicore machines hebben, de nieuwe technologieën in netwerk verbindingen, de nieuwe protocols, de strengere software vereisten, ... Dit leidt al snel tot veroudering van de traditionele systemen. Om deze veroudering tegen te gaan, is het noodzakelijk dat de middleware layer aangepast wordt. Deze aanpassingen hebben echter de tendens steeds complexer te worden en de interoperabiliteit te verstoren. Het gebruik van een *plug-in* systeem omvat de mogelijkheid at runtime het systeem te herconfigureren. Het aanvullen of vervangen van de bestaande services biedt een oplossing tegen deze veroudering van het systeem. Het is immers mogelijk de verouderde componenten aan te vullen of te vernieuwen.

2. Het opzetten van een gedistribueerde applicatie op een traditioneel systeem is niet altijd even evident. Het deployment[2] brengt meestal met zich mee dat de applicatie moet aangepast worden zodat het in de filosofie van een traditioneel gridsysteem past. Sterker nog, bij het uitvoeren van bijvoorbeeld omvangrijke simulaties wordt er meestal gewerkt in verschillende fases: data verkrijgen, opzetten van het systeem, berekeningen, analyse van de berekeningen, visualisatie van de berekeningen ,... . Dergelijke opsplitsing brengt vaak met zich mee dat resources, die in één van de fases gebruikt worden, vaak onbenut zijn in andere fases en vice versa. Dit moet ten allen tijde vermeden worden omdat ongebruikte resources verloren resources zijn. Hier komt nog bij dat wanneer een service, die niet bij de start van de berekening voorzien is, alsnog noodzakelijk blijkt te zijn voor de applicatie, er geen eenvoudig systeem is om deze service alsnog te voorzien. Om aan deze problematiek tegemoet te komen, biedt het *plug-in* mechanisme de mogelijkheid het gedistribueerde systeem dynamisch aan te passen aan de noden van de gedistribueerde applicatie. Dit brengt met zich mee dat:

   - De applicatie niet langer zal moeten aangepast worden aan het gedistribueerde systeem.
   - De resources in de verschillende fases van een applicatie optimaal benut kunnen worden.
   - Er dynamisch alsnog extra functionaliteit voorzien wordt als dit nodig blijkt te zijn.

Er kan dus gesteld worden dat het gebruik van de *plug-in* technologie, als gevolg van de toegenomen flexibiliteit en van zijn eenvoud in het toevoegen van nieuwe technologieën in het systeem, zorgt voor een globale verbetering in het gebruik van de resources. Concreet betekent dit dat de noodzaak tot de ontwikkeling van

[2] De applicatie op het gedistribueerde systeem plaatsen zodat de code uitgevoerd kan worden.

Harness en H2O gebaseerd was op drie tekortkomingen van de toenmalige klassieke metacomputing frameworks:

1. Het niet flexibel zijn van deze systemen.

2. Het onvermogen om nieuwe technologieën aan deze systemen toe te voegen.

3. De snelle veroudering van deze systemen.

Om de herkomst van het H2O project te schetsen zal in de tweede sectie het Harness project besproken worden. De daaropvolgende sectie geeft een globaal overzicht van de principes, de werking en de opbouw van de H2O middleware. Sectie A.4 bespreekt dan in meer detail de verschillende ondersteunende concepten die aan de basis liggen van het H2O framework (bv. RMIX). Tot slot wordt dit hoofdstuk afgesloten met een operationeel model en een voorbeeld toepassing.

A.2 Harness

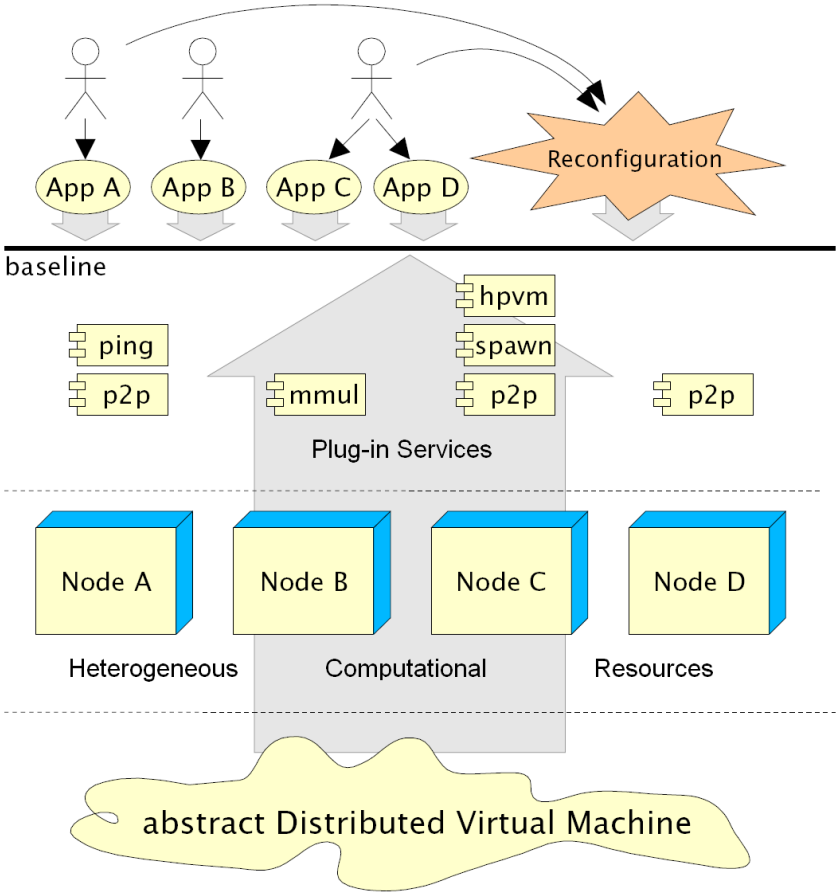
A.2.1 Filosofie

Het Harness Metacomputing Framework is een project dat tot stand kwam van uit een samenwerking tussen de Emory University te Atlanta, het Oak Ridge National Laboratory(ORNL) en de University of Tennessee. Het doel van het project was, aan de hand van de principes van herconfigureerbare virtuele machines op het netwerk, een experimenteel metacomputing[3] framework op te zetten. De term "herconfigureerbaarheid" omvat hier zowel de configuratie van de resources (netwerk, computers) als de mogelijkheid de functionaliteit van de virtuele machine dynamisch aan te passen. De methode die hiertoe dient en die centraal staat in Harness is het reeds aangehaalde *plug-in* mechanisme. Het hele basis idee achter het Harness metacomputing framework is, zoals hierboven reeds aangehaald, het opzetten van een pluggable *Distributed Virtual Machine*. Het ontwikkelde systeem zal zoals blijkt uit figuur A.1, bestaan uit vier lagen:

1. *Abstract Distributed Virtual Machine*: Dit is de fundamentele abstractielaag van het Harness metacomputing framework. Elke instantie van dit abstract concept is in principe een verzameling die een naam heeft en de verschillende resources groepeert binnen één virtuele machine.

2. *De Heterogeneous Computational Resources*: Deze laag bestaat, zoals zijn naam al laat uitschijnen, uit een heterogene verzameling resources die zich bij de virtuele machine hebben gevoegd. Er dient hier echter wel de nadruk gelegd te worden op het feit dat deze abstractielaag enkel de resources die

[3]Metacomputing wordt gedefinieerd als: "the use of powerful computing resources transparently available to the user via a networked environment" [105].



Figuur A.1: Het Harness lagen model [64]

zich binnen een virtuele machine bevinden beschrijft maar niet de mogelijkheid biedt tot rechtstreekse communicatie tussen de individuele resources en de gebruikers.

3. *De plug-in services*: Deze interactie tussen gebruiker en resource wordt ondersteund door de plug-in services. Elk van deze services zal instaan voor het uitvoeren van een specifieke opdracht, gebruik makend van de resources waarop hij geplugd is. Aan de hand van deze plugins wordt een consistente baseline[4] ontwikkeld.

4. *Aplication/Reconfiguration layer*: De bovenste laag bestaat uit de herconfiguratie mechanismen en de applicaties die een gebruiker ontwikkelt gebruik makend van de baseline.

Naast dit vierlagen model, blijkt uit figuur A.1 ook dat de gebruiker steeds in de mogelijkheid is de virtuele machine te herconfigureren. Dit zowel op het gebied van

[4]De basisfunctionaliteit waarop applicaties geconstrueerd kunnen worden.

resources (Resources kunnen toegevoegd en verwijderd worden) als op het gebied van de aangeboden services (Services kunnen ingeplugd en uitgeplugd worden).

Een plug-in is de eigenlijke implementatie van een service. Elk van deze plug-ins kan gekarakteriseerd worden aan de hand van een set van attributen die de volgende eigenschappen specificeren:

- *Shareability*: Een plug-in is shareable wanneer verschillende gebruikers éénzelfde instantie ervan mogen gebruiken.
- *Exportability*: Een plug-in is exportable wanneer het mogelijk is toegang tot deze plug-in te verlenen aan gebruikers van verschillende Distributed Virtual Machines (DVM's).
- *Threadability*: Een plug-in is threadable wanneer de plug-in naast andere plug-ins of andere instanties van deze plug-in zonder conflicten uitgevoerd kan worden.

De beschikbaarheid van de plug-ins voor de heterogene computationele resources hangt af van twee eigenschappen van het framework:

- *Portability*: De portability beschrijft in welke mate de plug-ins compatibel zijn met de verschillende resources.
- *Searchable plug-in repositories*: Het type plug-in repositorie definieert op welke manier en bijgevolg in welke mate bepaalde plug-ins opzoekbaar zijn voor eventuele gebruikers.

Één van de hoofddoelen van Harness is een heterogene set van resources te groeperen in een gedistribueerde virtuele machine en het mogelijk te maken dat elk van deze resources services kan aanbieden aan de gebruikers. Om dit te bewerkstelligen, is het essentieel de verschillende componenten van het framework zo portable mogelijk te maken over een zo groot mogelijke verscheidenheid aan resource types. In dit opzicht was het evident het wiel niet opnieuw uit te vinden en gebruik te maken van de reeds beschikbare Java technologie. Het is immers mogelijk op een Java Virtuele Machine (JVM)[69] code te ontwikkelen die volledig portable is over verschillende architecturen. Hierbij dient wel opgemerkt te worden dat een gereduceerde performantie de prijs is die hiervoor betaald wordt. Aangezien de portability voor sommige componenten enerzijds niet eenvoudig te realiseren is, en deze anderzijds voor sommige componenten ook niet essentieel is, werd er binnen het Harness framework gekozen om drie levels van portability te introduceren:

1. *Kernel level services*: Deze services implementeren enerzijds de functionaliteit om de DVM status dynamisch te updaten en anderzijds de basisfunctionaliteit om services te laden. Het spreekt voor zich dat deze services 100% portable moeten zijn. In essentie komt dit erop neer dat deze services ondersteunt moeten worden door een resource om zich bij de DVM te mogen voegen.

2. *Basic services*: Deze tweede categorie omvat services die op zeer regelmatige basis gebruikt worden, zoals messaging of event notification services. Van deze services wordt aangenomen dat ze op elke resource beschikbaar zijn. Hier mag echter in het uitzonderlijke geval van afgeweken worden indien deze resource niet in de mogelijkheid is deze service te ondersteunen.

3. *Specialized services*: Deze laatste categorie bevat twee types van services:

   - *Higly architecture specific services*: Deze services zijn afhankelijk van de specifieke architectuur van de niet-standaard resource. Zo zullen bijvoorbeeld bepaalde message passing services gebruik maken van zeer specifieke netwerk interfaces.
   - *Performance optimization services*: Deze groep services is sterk afhankelijk van architecturale optimalisaties om aan zeer specifieke performantie eisen te voldoen.

   Het spreekt voor zich dat ook dit type van services moet streven naar een zo groot mogelijke portability. Het is echter wel aanvaardbaar dat, indien deze portability niet haalbaar is, de service slechts door een beperkte groep van resources ondersteund wordt.

Deze verschillende graden van portability maken het mogelijk de gebruikte Java infrastructuur te combineren met byte code en met systeem afhankelijke native code. De ondersteuning van dit principe berust op de Java Native Interface (JNI). Deze interface laat toe dat non-Java code binnen de JVM aangeroepen wordt. Op deze manier kunnen specifieke kritieke of performantie gevoelige onderdelen van het systeem geïmplementeerd worden in C of in FORTRAN. Hierbij dient wel opgemerkt te worden dat dit steeds nadelige gevolgen heeft voor de ontwikkelingskost en in het bijzonder voor de portability.

Naast de portability voordelen die Java als ontwikkelingsplatform met zich meebrengt, biedt Java/Harness nog tal van voordelen:

- Het is mogelijk het framework te ontwikkelen onder de vorm van een wel afgelijnde groep van samenwerkende objecten. Dit maakt dat er naar de gebruiker toe een OO ontwikkelingsomgeving kan aangeboden worden.

- Het concept plug-in kan naar de Java omgeving omgezet worden. Elke plug-in kan immers als Java klasse aan het systeem aangeboden worden.

- De robuste multithreading architectuur van Java is gebruikt bij de ontwikkeling van het Harness framework.

- De gebruiker is in de mogelijkheid extra services te ontwikkelen. Dit zowel onder de vorm van een library als onder de vorm van een thread of process.

- Java-RMI kan gebruikt worden als RPC mechanisme om services op remote resources aan te spreken.

- Java object serialisatie kan gebruikt worden om data in een consistent formaat over het netwerk te transfereren.

- Het is mogelijk de definitie van de door de basis plug-ins geïmplementeerde interfaces door te geven aan de gebruikers.

- Het Java Class en Library loading mechanisme kan gebruikt worden om dynamisch en op aanvraag libraries toe te voegen aan het framework. Dit concept wordt het inlinken van libraries genoemd.

A.2.2 Architectuur

Zoals uit figuur A.1 reeds gebleken is maakt het Harness metacomputing framework het mogelijk om een DVM te definiëren. De kernel level services waaruit deze DVM is opgebouwd worden aangereikt door een gedistribueerd systeem dat opgebouwd is uit twee soorten entiteiten:

- Een DVM status server die uniek is voor elke DVM.

- Een verzameling Harness kernels waarvan er op elke resource, die tot de DVM behoort, of die er in de nabije toekomst toe wil behoren, één draait.

Zowel de status server als de Harness kernels zullen, om de reeds besproken portability te garanderen, volledig geïmplementeerd zijn in Java. De communicatie binnen het volledige systeem berust voor een groot gedeelte op asynchrone event notificatie. Concreet zendt de status server controle en status change berichten naar elke aanwezige Harness kernel om zo een consistente evolutie van de DVM te garanderen. Om dit alles in goede banen te leiden, bevat Harness een stel van protocols die uitgevoerd worden tijdens de verschillende fases waarin de DVM zich kan bevinden:

- *Enrollment protocol*: Behandelt het toevoegen en het verwijderen van resources en is gebaseerd op IP multicast of op een centrale name server.

- *Event notification protocol*: Zorgt ervoor dat alle events bezorgd worden aan alle partijen die er een boodschap aan hebben (vb. join, leave).

- *Service versioning protocol*: Dit protocol zorgt ervoor dat elke component in een DVM uniek is en er slechts één enkele versie van bestaat.

- *DVM status reconstruction protocol*: Zorgt ervoor dat de DVM status server geen *single point of failure* is. Het is in de mogelijkheid, bij het eventuele falen van de status server, de server na het herstarten terug volledig te herstellen in zijn oorspronkelijke staat. Dit door de status van elke resource, die de DVM omvat, incrementeel toe te voegen. Ook dit protocol is gebaseerd op IP multicast of op een centrale naming server.

A.3 H2O

A.3.1 Inleiding

Het H2O project werd in 2002 opgestart door het Distributed Computing Laboratory aan de Emory University te Atlanta. Het opzet was de realisatie van een component gebaseerd, service georiënteerd, distributed computing framework. De originele 'mission statement' werd als volgt samengevat:

> *The development of a generalized plug-in paradigm for distributed virtual machines that allow users or applications to dynamically customize, adapt and extend the distributed computing environment's features to match their needs without compromising security or the consistency of the programming environment* (afgeleid uit [83] en [39]).

Vrij vertaald wordt het H2O project gedefinieerd als een implementatie van het plug-in paradigma dat ertoe dient een distributed virutele machine te ondersteunen. De concrete toepassing waarbinnen deze implementatie in eerste instantie zal gebruikt worden is de gedistribueerde Harness virtuele machine.

Om de werking van het H2O principe beter te begrijpen, kunnen we H2O best vergelijken met het servlet principe. Servlets zijn kleine Java programma's die functionaliteit aan een webserver toevoegen. Om dit te verwezenlijken, dienen ze gedeployed[5] te worden in een servlet container. Deze servlet container zal dan de functionaliteit voorzien om de dienst, die de servlet aanbiedt, te koppelen aan de webserver en aan de client. Populaire voorbeelden van dergelijke servlet containers zijn de Apache Tomcat container[1], de Winston Servlet Container[11],... .
Op dezelfde manier zoals bijvoorbeeld Tomcat een container is voor servlets, zal de H2O beschikken over een kernel welke een container zal zijn voor de H2O pluglets (vergelijkbaar met servlets). Het grote verschil tussen de H2O kernel en elk ander traditioneel model zit hem echter in het feit dat naast de eigenaar van een container (H2O-kernel) ook gebruikers, die voldoen aan de lokaal gedefinieerde constraints, services (pluglets) mogen deployen op de H2O kernel. Dit wordt duidelijk geïllustreerd in figuur A.3. In tegenstelling tot de traditionele statische containers maakt dit principe het mogelijk op een dynamische manier gedistribueerde resources te delen. Aangezien de termen Kernel en Pluglet uit H2O vergelijkbaar zijn met de respectievelijke termen Container en Service uit het Jiniframework zullen deze termen in het verdere verloop van de tekst door elkaar gebruikt worden.

A.3.2 Definitie

Alvorens de basisprincipes uit te diepen, is het belangrijk de officiële H2O definitie [112] kort te bespreken:

> H2O is a novel *component-based*, *service-oriented* framework for distributed metacomputing. Adopting a provider-centric view of resource

[5] Code ter beschikking stellen zodat deze kan uitgevoerd worden.

> sharing, it provides a *lightweight*, *general-purpose*, *configurable platform* for *resource sharing*, keeping *minimal state*. Resource owners host a software backplane (called a *kernel*) onto which owners, clients, or third-party resellers may load components (called *pluglets*) or component-suites that deliver value added services without compromising owner security or control. *Standards based descriptions* of services facilitate publication and discovery via established schemes.

Het eerste wat in deze definitie onmiddellijk opvalt, is de overeenkomst met het Harness framework. De component gebaseerde en service georiënteerde eigenschappen vinden inderdaad hun oorsprong in Harness. Omwille van deze reden zal H2O, evenals Harness, het plug-in paradigma gebruiken. Dit brengt met zich mee dat H2O van dezelfde voordelen zal genieten als deze van Harness. Deze verwevenheid met Harness heeft eveneens als gevolg dat de doelstellingen van H2O niet liggen in large scale of performance critical networks maar zich vooral richten op light weight, loosely coupled netwerken waar de resource sharing berust op P2P (Peer to Peer) communicatie. Hier dient wel bij opgemerkt te worden dat de designers van H2O ervan overtuigd zijn dat, mits doelgerichte aanpassingen aan de onderbouw van H2O, het hele framework eenvoudig kan geoptimaliseerd worden om verschillende distributed computing principes te ondersteunen: metacomputing(High Performance Distributed Computing of HPDC), client-server, peer-to-peer, task farm, distributed component composition, ...

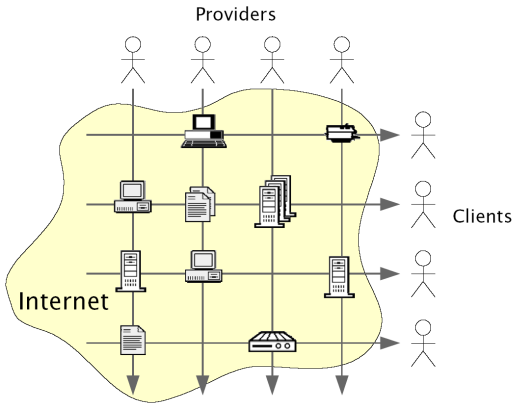
De software component waarop het hele systeem berust, wordt een *kernel* genoemd. Deze component is op elke resource aanwezig om het plug-in principe te ondersteunen. De verschillende componenten die dynamisch op dit kernel gebaseerde systeem kunnen geplugd worden, zijn de *pluglets*[6]. Een belangrijke eigenschap van een dergelijke kernel is dat hij ten allen tijde onder de volledige controle van de eigenaar van de resource blijft. Het is echter wel mogelijk, voor clients die over voldoende rechten beschikken, pluglets op de kernel te gebruiken of zelfs zelf te deployen[7]. Tot slot dient bij deze definitie nog opgemerkt te worden dat getracht wordt een gewenst niveau van interoperability en usability te bereiken door het gebruik van standaarden zoals WSDL & UDDI bij het opstellen van pluglet descriptions en bij het publishen van services.

A.3.3 Container architectuur: kernels en pluglets

Het H2O framework gaat ervan uit dat resource providers volledig onafhankelijk van elkaar handelen en dat deze handelingen gebaseerd zijn op lokale gebeurtenissen en principes. Wanneer dit uitgangspunt gecombineerd wordt met een minimalisatie of met een eliminatie van een globale coördinatie (coördinatie middleware en/of globale status van het systeem) opent dit de mogelijkheid tot het vormen van een *self-organizing distributed computing system*.

[6] De naam vindt zijn oorsprong in een combinatie van het woord *plugging* en van de Java termen *servlet* en *applet*.

[7] Het ter beschikking stellen van software in een kernel zodat deze door arbitraire clients gebruikt kan worden.



Figuur A.2: De services, Resource providers en clients binnen het H2O framework [112].

Bij de vorming van een globaal beeld van H2O valt het op dat de bovenste laag van het H2O framework enkel bestaat uit (zie figuur A.2):

- Een dynamische verzameling van providers die, onafhankelijk van elkaar, hun resources volgens individueel bepaalde constraints ter beschikking stellen.
- Een dynamische verzameling van gebruikers (clients) die deze beschikbare resources onafhankelijk van elkaar ontdekken en gebruiken.

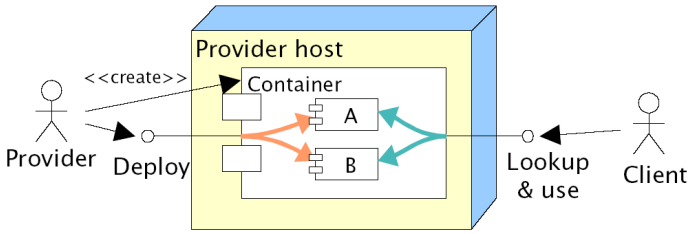
Aangezien elke gebruiker tevens een provider kan zijn, is het mogelijk bepaalde services afhankelijk te laten zijn van andere services (de provider van service A is bijvoorbeeld client bij service B en C; meer concreet kan een service, die instaat voor het afspelen van een film, gebruik maken van een service die video bestanden decodeert, van een service die de geluidsfragmenten decodeert en van een service die de ondertiteling verzorgt). Op deze manier wordt de distributie abstractie zo dicht mogelijk bij de gebruiker gelegd.

Zoals reeds vermeld in sectie A.3.2 worden de services die gedeployed worden *pluglets* genoemd en wordt de container waarin deze pluglets gedeployed worden de *kernel* genoemd. Een kernel is verantwoordelijk voor:

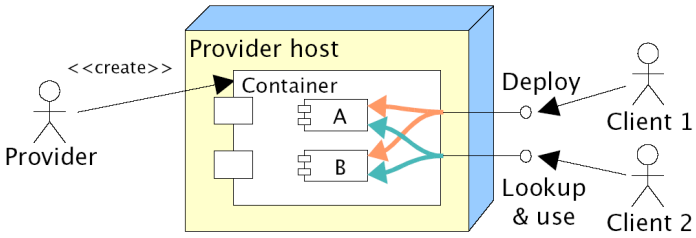
- Het deployen van services.
- Lifetime management van een service.
- Afhandeling van constraints.
- Communicatie.

A.3.4 DVM architectuur

Zoals in sectie A.3.3 reeds werd aangehaald, wordt de distributie abstractie zo dicht mogelijk bij de gebruiker gelegd. Concreet houdt dit in dat H2O enkel de functionaliteit levert om de verschillende onafhankelijke entiteiten (gebruikers en resources) toe



(a) traditional model

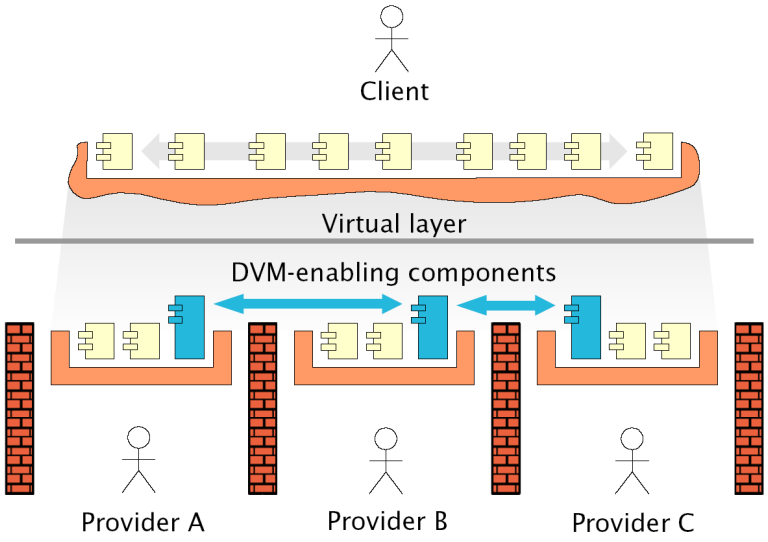


Figuur A.3: Verschil tussen het traditionele container model en het H2O container model [112].

te laten binnen de H2O omgeving samen te werken en zo Metacomputing te ondersteunen. Metacomputing op zich steunt typisch op een mechanisme dat verschillende resources groepeert en deze in parallel verschillende opdrachten laat uitvoeren. Dit mechanisme om resources te groeperen, wordt vaak vernoemd onder de term gegroepeerde *context*, en het bestaat meestal onder de vorm van een expliciete of impliciete DVM. Om het opzetten van een dergelijke context mogelijk te maken, voorziet H2O niet alleen de functionaliteit om componenten te construeren die uitgevoerd worden op de individuele machine's maar ook de coördinatie die een consistent beeld van een virtuele machine verzorgt (zie figuur A.4). Het is dan ook dit netwerk van DVM componenten op zich, dat instaat voor de hard distributed state[8]. Het isoleren van deze state in één enkel component type maakt dat de onafhankelijkheid en de flexibiliteit over het algemeen behouden blijven.

Het is belangrijk hierbij op te merken dat de verantwoordelijkheid voor het opzetten en voor de implementatie van deze *context* volledig bij de gebruiker ligt. De "DVM-enabling component zijn met andere woorden een specificatie waarvan geen implementatie terug te vinden is in de H2O source code. Ze verwijzen naar een H2O opstelling waarbinnen een verzameling van interconnected pluglets een gemeenschappelijke gedistribueerde status delen en beheren met als doel een gecoördineerde service aan te bieden.

[8]Hiermee wordt bedoeld dat de status van de DVM niet onderhevig is aan time-outs en bijgevolg enkel kan gewijzigd worden wanneer er expliciet de opdracht toe gegeven wordt.



Figuur A.4: DVM enabeling components [112]

A.4 Ondersteunende Concepten

A.4.1 RMIX

Inleiding

Een eerste stap in de bespreking van de ondersteunende concepten is logischerwijze de onderste communicatie laag. De belangrijkste doelstelling van H2O is de ontwikkeling van een pluggable service paradigma. Het spreekt voor zich dat het doortrekken van deze pluggability naar het communicatiemechanisme één van de basisdoelstellingen was van H2O. Bij de aanvang van het H2O project bestonden er in de Java gebaseerde distributed computing drie standaard communicatie protocols namelijk:

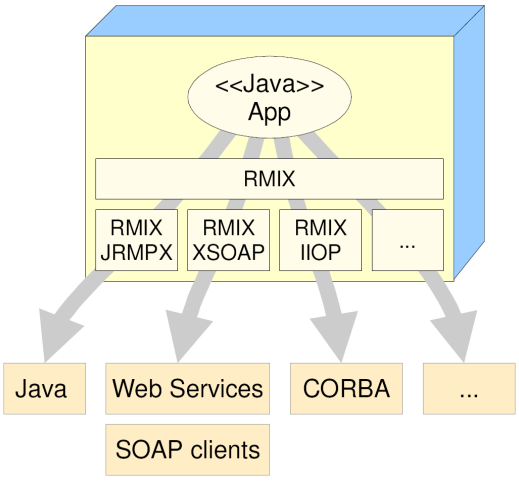
- *JRMP*: Het protocol waarop standaard RMI [41] [80] gebaseerd is.
- *IIOP*: Dit protocol ondersteunt interoperabiliteit via CoRBA (RMI-IIOP [81]).
- *SOAP*: Dit protocol ondersteunt interoperabiliteit via Webservices (JAX-RPC [56] [103] [115]).

Naast deze drie standaarden bestonden er nog tal van performantie georiënteerde RMI implementaties [63] [74] [73] [89]. Het grote nadeel van deze verschillende implementaties van het RMI paradigma was hun mutual exclusivity en het gebrek aan voorzieningen voor onderlinge communicatie.

Om het H2O project te doen slagen, diende er een nieuw communicatie mechanisme te komen dat voldeed aan de vereisten van de H2O specificaties. Om dit te bewerkstelligen, werd het RMIX project opgestart. De doelstellingen van het RMIX

project waren tweeledig en zijn terug te vinden in zijn naamgeving. RMIX staat voor *RMI miXture* en *RMI eXtended*.

1. *Mixture*: De eerste doelstelling bestaat erin een pluggable RMI framework op te zetten waarin er voor de communicatie tussen verschillende objecten kan gekozen worden uit een *mix* van protocols. De keuze zal zich niet enkel beperken tot reeds bestaande protocols maar ook dynamisch uitgebreid kunnen worden naar nieuwe protocols. Er kunnen als het ware nieuwe protocols *ingeplugd* worden. Het gehele principe zal werken via een geüniformiseerde API, gebruik makend van de standaard remote method invocation semantiek. Zoals figuur A.5 duidelijk maakt, leidt dit principe tot een omgeving waarin interoperabiliteit en heterogene toegang tot een service ondersteund worden. Het zal op deze manier bijvoorbeeld mogelijk worden een service via verschillende protocols ter beschikking te stellen. De service zal voor iedereen toegankelijk zijn via een protocol met maximale interoperabiliteit, zoals SOAP[109], en tegelijkertijd geoptimaliseerd toegankelijk zijn voor gespecialiseerde clients die een geoptimaliseerd protocol kunnen gebruiken. Het kan echter nog straffer. Het wordt tevens mogelijk aan services, die reeds in gebruik zijn, at runtime protocols toe te voegen zodat er ook aan de noden van clients, die nieuwere protocols ondersteunen, wordt voldaan. Een ander niet te verwaarlozen voordeel van deze methodiek is de eenvoud in het hergebruik van code. Het was in het verleden vaak zo dat programmatuur bij het ontstaan van nieuwe protocols volledig moest herschreven worden om dit protocol te kunnen ondersteunen. Met deze methodiek is dit probleem echter volledig verleden tijd. Het nieuwe protocol kan immers dynamisch in het RMIX framework geplugd worden zodat het invoeren van een nieuw protocol zelf niet langer moet impliceren dat de componenten van het software systeem moeten herstart worden. Zo zou bijvoorbeeld de migratie van IIOP naar SOAP niet langer veel tijd in beslag nemen.



Figuur A.5: Interoperability in RMIX [65]

2. *Extended*: De tweede doelstelling streeft naar het opwaarderen van de standaard RMI semantiek met de volgende eigenschappen:

   - Service providers controle geven over de manier waarop remote endpoints aangemaakt worden.
   - Flexibel en configureerbaar zijn.

   Concreet was er nood aan *API's enabling services* die de mogelijkheid bieden verschillende combinaties van service methodes te exporteren. Dit door de endpoints met verschillende interface subsets te associëren. Dit moest gecontroleerd worden aan de hand van *invocation interceptors* die de access policies per client zouden bepalen. Bovenop deze verbeteringen aan het communicatiemodel moest er nog asynchrone en one-way invocation semantiek voorzien worden.

Deze aanpassingen in het standaard RMI communicatiemodel moesten ervoor zorgen dat RMIX een meer robuuste basistechnologie voor multi-user, heterogene, coöperatieve loosely coupled en geografisch gedistribueerde systemen zou worden.

RMIX Design & extra features

In deze sectie wordt besproken aan welke eigenschappen de verschillende protocols die RMIX ondersteunt, moeten voldoen. Tevens wordt er besproken welke features er aan standaard RMI zijn toegevoegd om aan de vereisten uit de inleiding te voldoen en om tot RMIX te komen.

Het was een bewuste keuze bij aanvang van het RMIX project, het basis model en de semantiek die gedefinieerd worden door de Java RMI specifications [80] te behouden. Concreet betekent dit dat het gebruik van remote objecten gerealiseerd wordt, gebruik makend van client-side stubs van deze remote objecten. De stubs implementeren dezelfde set van remote interfaces[9] als hun doelobjecten. Een ander eigenschap die in de Java RMI specifications gedefinieerd staat, en door het merendeel van de alternatieve RMI implementaties gedeeld wordt, bestaat erin dat alle parameters als waarde worden doorgegeven (*passed by value*). Dit met uitzondering van de remote objecten die, aangezien ze voorgesteld worden door stubs, doorgegeven worden als referentie (*passed by reference*). Omwille van het feit dat deze beide eigenschappen door de verschillende RMI implementaties[10] en tevens door het merendeel van de alternatieve RMI oplossingen[11] ondersteund worden, zijn deze eigenschappen binnen de RMIX specificaties gepromoveerd tot de minimale functionele vereisten waaraan een protocol provider implementatie[12] moet voldoen.

[9] Dit zijn interfaces die *java.rmi.Remote* implementeren waarvan de specificatie van elk van de methodes definieert dat ze *java.rmi.RemoteExeption* throwen.

[10] RMI implementaties die voldoen aan de RMI specificaties.

[11] RMI implementaties die niet altijd letterlijk aan de RMI specificaties voldoen.

[12] Een protocol implementatie dat door RMIX gebruikt kan worden om de communicatie tussen een client en een remote object te ondersteunen. Dit kunnen, zoals eerder besproken, reeds bestaande standaarden zijn zoals de standaard *RMI over JRMP* als nieuwe implementaties die in RMIX worden ingeplugd.

In tegenstelling tot de reeds vermelde gelijkenissen tussen het merendeel van de RMI implementaties blijkt dat, afhankelijk van de gebruikte data encoding techniek, de serialisatie mogelijkheden drastisch van elkaar kunnen verschillen. Omwille van de beperkte encoding technieken die gebruikt worden in enkele van de meest populaire RMI/RPC protocols, zoals bijvoorbeeld XDR[82], is er doelbewust voor gekozen de vereisten in deze specificatie niche zo op te stellen dat er geen protocols worden uitgesloten. Om deze reden wordt de verzameling types, die door elke protocol provider verplicht geserialiseerd moet kunnen worden, beperkt tot de volgende types:

- Primitieve types (Integer, Reals, ...).
- Strings (Begrensde reeks karakters).
- Array van Primitives.
- Array van Strings.
- Final Classes[13] die enkel Primitives of Strings als velden hebben en waarbij elk niet-publiek veld vergezeld wordt van een *getter* en *setter* methode conform het Java bean design pattern [79].

Bovenop deze standaard Java types moeten RMIX providers tevens de serialisatie van elke remote reference ondersteunen onafhankelijk van de protocol provider die de referentie aangemaakt heeft. Het spreekt voor zich dat deze eigenschap essentieel is om, at-runtime, protocol switching te ondersteunen. Aangezien het noch designmatig aan te raden is, noch realistisch is (het aantal stijgt exponentieel met het aantal providers) expliciet inter-provider dependencies te introduceren voor elke protocolprovider, introduceert RMIX een unified remote reference format dat als protocol switching *lingua franca*[14] dient.

Naast de mogelijkheid remote referenties door te geven tussen clients en servers is er bij RMIX, net zoals bij standaard RMI, nood aan een bootstrap mechanisme dat voor het initiële contact tussen de beide partijen zorgt. In standaard RMI wordt deze bootstrap functionaliteit verzorgd door een eenvoudige naming service: De *rmiregistry*. Ook CORBA maakt gebruik van een vergelijkbare naming service. Webservices (SOAP) daarentegen maken dan weer gebruik van meer geavanceerde mechanismen zoals WSDL[15] service descriptors [26], WS-Inspection documents [18], UDDI registries [72] en vele andere. Omwille van de grote verscheidenheid in deze bootstrap mechanismen is er binnen RMIX voor gekozen de transport providers niet te verplichten een specifiek discovery mechanisme te ondersteunen. Het is de bedoeling dat deze keuze applicatieafhankelijk gemaakt wordt. Omwille van de backward compatibility is er echter wel compatibiliteit met de *rmiregistry* voorzien.

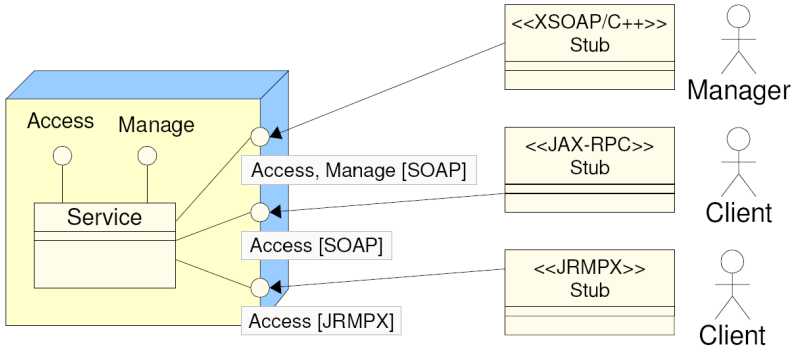
Naast deze vereisten, waaraan protocol providers moeten voldoen, voegt RMIX nog een aantal semantische verbeteringen toe aan RMI. De meest significante hiervan

[13] Final classes zijn classes waarvan de specificatie vooraf gegaan wordt door de *final* identifier en bijgevolg niet meer uitgebreid kunnen worden.

[14] Een taal die door twee partijen, die traditioneel een verschillende taal gebruiken, gebruikt wordt om onderling te communiceren. Zo spreekt een Chinees Engels tegen een Belg op een conferentie.

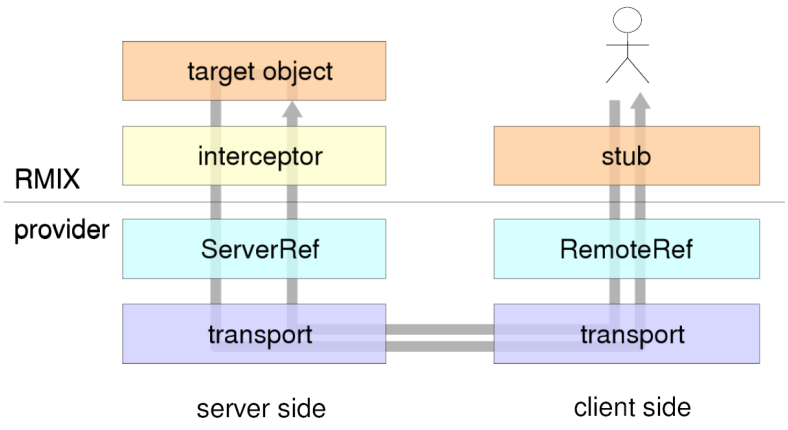
[15] WebService Description Language

is wellicht de mogelijkheid remote objecten verschillende malen te exporteren met verschillend export parameters. Er kunnen dus verschillende endpoints aangemaakt worden die gepersonaliseerde toegang tot een object mogelijk maken (Zie figuur A.6). Dit principe wordt in vele gevallen gerealiseerd door de informatie omtrent deze verschillende endpoints te verspreiden aan de hand van een registry service.



Figuur A.6: Verschillende aanpasbare endpoints [65]

Bovenop deze functionaliteit is het mogelijk de toegang tot de endpoints te beheren met *invocation interceptors*(Zie figuur A.7). De interceptors hebben de bevoegdheid, gebruik makend van de runtime policy en van de status van het systeem, te beslissen of een methode oproep aanvaard of tegengehouden wordt.



Figuur A.7: De RMIX oproep stack [66]

Bovenop de standaard blocking remote message invocation introduceert RMIX asynchrone en one-way calls die direct naar de caller returnen nadat de parameters geserialiseerd zijn. Deze methodiek staat toe de performantie te verhogen door communicatie en berekeningen van een pc te laten overlappen. Dit heeft vooral zijn nut in geografisch zeer sterk verspreidde applicaties waarbij de communicatie overhead

enorme vertragingen veroorzaakt [94]. Het is voor sommige protocol providers mogelijk one-way calls op een zeer efficiënte wijze te implementeren aangezien het einde van een call niet aangegeven moet worden. Asynchrone calls daarentegen melden het einde van een call wel. Het is zelfs mogelijk deze notificatie te vergezellen van een return value. Dit gebeurt aan de hand van een *future* object of een notificatie op een callback die door de gebruiker gespecificeerd is.

Een andere belangrijke verbetering die RMIX met zich mee bracht, was het verdwijnen van de stub generatie. In traditionele RMI was het vereist, na de standaard compilatie, elke remote klasse nogmaals te compileren, dit gebruik makend van de `rmic` compiler. Die maakte voor elk van deze remote klassen de gewenste stub klassen aan. Naast het feit dat deze compilatie niet eenvoudig en uiterst vervelend was, had deze methodiek nog twee enorme nadelen:

- De gebruiker was met deze statische compilatie gebonden aan een vast protocol (hardcoded in de gecompileerde stubs).
- Het deployen van gedistribueerde applicaties wordt aanzienlijk bemoeilijkt. De gecompileerde stubs moeten immers op elk van de deelnemende machines aanwezig zijn [78].

Deze problematiek wordt in RMIX verholpen via dynamische proxy klassen [77] en via dynamische stubs die dynamisch aangemaakt worden at run time.

Tot slot dient nog vermeld te worden op welke manier protocol providers op een eenduidige manier, realtime, en op een uniforme wijze toegevoegd en verwijderd kunnen worden. Om dit te realiseren, is het verplicht de vereisten van elke protocol provider te bundelen in een JAR file waarin de benodigde meta-data in het manifest worden opgeslagen. De installatie van de provider gebeurt dan eenvoudigweg door de JAR file in een vooraf gespecificeerde directory te plaatsen. Na deze installatie worden de beschikbare protocols op een dynamische manier zichtbaar gemaakt voor de gebruiker aan de hand van een dynamisch discovery mechanisme. Bij de aanvang van deze thesis waren er slechts twee protocol providers beschikbaar: de JRMPX provider en de XSOAP provider. De eerste is een eenvoudige extensie bovenop standaard Java RMI en behoudt bijgevolg de volledige RMI semantiek:

- De volledige Java serialisatie mogelijkheden.
- Dynamische remote classloading.
- Ondersteuning voor class annotation.
- Distributed garbage collection.

Omwille van het feit dat JRMPX niet afhangt van enige niet publieke APIs zal het ook, naar de toekomst toe, compatibel blijven met eventuele nieuwe Java RMI releases. De tweede provider is gebaseerd op de XSOAP toolkit(De voormalige SoapRMI[104]). Dit mechanisme gebruikt SOAP 1.1 als protocol. De aanpassingen die nodig waren om XSOAP binnen het RMIX framework te laten passen, waren enorm maar het resultaat is bevredigend. Zowel JRMPX als XSOAP ondersteunen

de verschillende RMIX-specifieke features zoals asynchrone invocation, multiple and customizable object endpoints, ...

A.4.2 Plugging

Zoals reeds eerder aangehaald werd, berust de functionaliteit van H2O op het *plugging* principe. Het plugging principe is gebaseerd op twee componenten:

- *Pluglet*: Een service die op een resource kan ingeplugd worden.
- *Kernel*: Een omgeving die op een resource aanwezig is en het inpluggen van een service realiseert.

Alvorens een pluglet in een kernel geplugd of aangesproken kan worden, moet er een veilige connectie tussen de client (deployer of gebruiker van de service) en de Kernel tot stand gebracht worden (zie figuur A.8(a)). Zoals besproken wordt in sectie A.4.4, zal een component, die de *gatekeeper* genoemd wordt, instaan voor de authenticatie en de autorisatie van een client. Deze stap omvat de verificatie van de client security credentials en de aflevering van een *kernelcontext* [16]. Deze staat in voor de client permissies op de kernel.

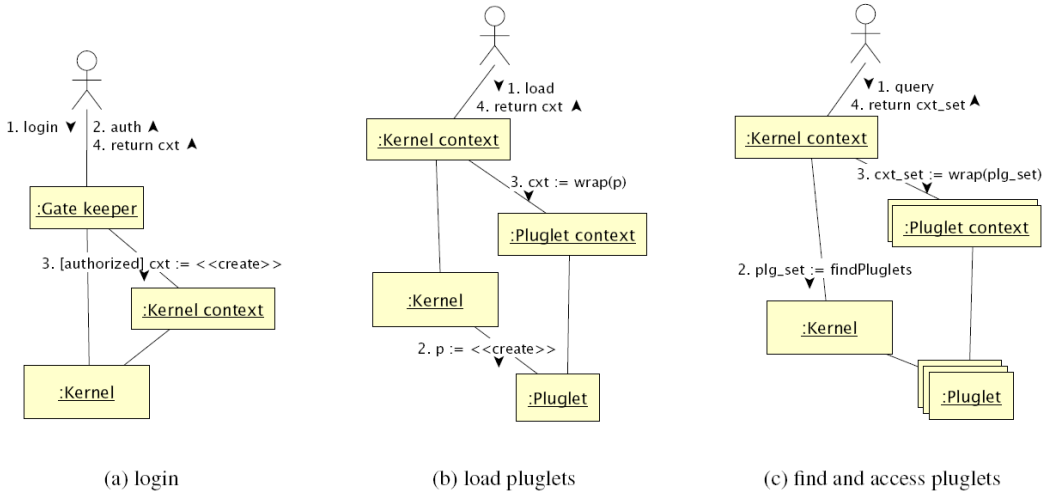
Clients die over de benodigde permissies beschikken, mogen nieuwe pluglets inpluggen op een kernel (zie figuur A.8(b)). Deze plugin procedure vereist de volgende informatie:

- Het path (URL) van de class files van de pluglet (Meestal een jar file).
- De naam van de plugletklasse.
- De naam van de pluglet: De naam die aan een pluglet gegeven wordt wanneer hij in de kernel geplugd wordt. Deze mag niet verward worden met de reeds vermelde naam van de plugletklasse. Er kunnen immers verschillende pluglets van dezelfde klasse onder een andere naam in de kernel geplugd worden.
- De initialisatie parameters. Dit zijn de parameters die gebruikt kunnen worden in de initialisatie methode van een pluglet. Deze methode wordt steeds na de instantiëring aangeroepen en moet door de pluglet developer zelf geïmplementeerd worden.
- Meta-informatie die gebruikt kan worden bij het opzoeken van de pluglet door de clients.

Tot slot is het evident dat het voor een client ook mogelijk is een pluglet op te zoeken en te gebruiken. Dit aan de hand van een query mechanisme dat een match zoekt op basis van de pluglet naam, van het type en van de reeds vernoemde meta-informatie. Het inpluggen of opvragen van een pluglet resulteert in het verkrijgen van een pluglet context (`PlugletContext` interface). Deze interface abstraheert de communicatie van de client naar de pluglet toe.

[16] Een communicatieabstractie (zie sectie A.4.3) die de client-kernel communicatie abstraheert.

Naast het feit dat deze PlugletContext een interface van de client "verbindt" met de functionaliteit van de pluglet via de `Bind(...)` methode stelt het de client ook in de mogelijkheid meta-operaties uit te voeren[17] en meta-informatie[18] op te vragen. Deze abstractietechniek wordt verder besproken in sectie A.4.3.



(a) login (b) load pluglets (c) find and access pluglets

Figuur A.8: Client - Kernel interactie [112]

Er kunnen drie types interfaces onderscheiden worden die door een pluglet implementatie, geimplementeerd kunnen worden (zie figuur A.9):

- De `Pluglet` interface: Deze interface bevat tal van methodes (`init`, `start`, `stop`, `destroy`) die door de client moeten geïmplementeerd worden om statusveranderingen van de pluglet binnen de kernel implementatiespecifiek af te handelen. Zo zal bijvoorbeeld een pluglet, die instaat voor het veilig openen en sluiten van een kluisdeur, er steeds voor zorgen dat de deur van de kluis gesloten wordt wanneer een pluglet gestopt wordt zodat dieven geen misbruik kunnen maken. Het is belangrijk hierbij op te merken dat deze `Pluglet` interface, zoals uit figuur A.9 blijkt, als het ware een contract is tussen de kernel en de pluglet. Deze interface kan bijgevolg ook enkel aangesproken worden door de kernel en is onbereikbaar voor de clients in het H2O netwerk.

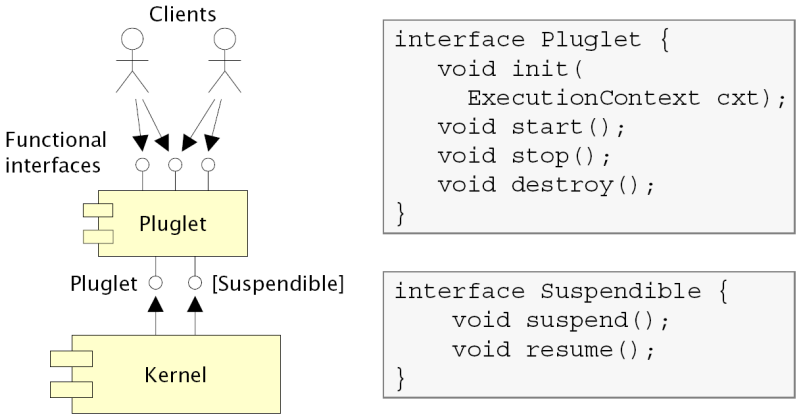
- De `Suspendible` interface: Deze interface is optioneel en stelt de developer van de pluglet in de mogelijkheid implementatiespecifiek te reageren op het *suspenden* respectievelijk *resumen* van een pluglet. Analoog aan het vorige voorbeeld zal een pluglet, die instaat voor het veilig openen en sluiten van een kluisdeur, er steeds voor zorgen dat de deur gesloten is op de momenten dat

[17] Dit zijn operaties die de toestand van de pluglet binnen de container wijzigen(start, stop, suspend, ...)

[18] Dit is de informatie die gebruikt wordt om via queries de gewenste pluglets te selecteren

de pluglet niet gebruikt kan worden. Ook deze interface is een contract tussen de pluglet en de kernel en is bijgevolg ook enkel aanspreekbaar door de Kernel.

- De interfaces die de functionaliteit van de pluglet aanduiden. Naar analogie met het vorige voorbeeld zal een pluglet, die instaat voor het veilig openen en sluiten van een een kluisdeur, `Open(pwd:Password);` en een `Close();` methode bevatten die de kluis opent en respectievelijk sluit. Het spreekt voor zich dat dit laatste type interface toegankelijk is voor de verschillende clients in het H2O netwerk om de functionaliteit van de pluglet te gebruiken.



Figuur A.9: Pluglet Interfaces [112]

A.4.3 Communicatie abstractie

In deze sectie wordt een communicatie abstractie bekeken die binnen H2O op twee plaatsen gebruikt wordt. Zoals reeds is aangehaald, wordt bij het opvragen van een pluglet steeds een `PlugletContext` meegegeven en geen object dat de `Pluglet` interface implementeert. Op een analoge manier zal een gebruiker steeds werken met een `KernelContext` en niet rechtstreeks op de `Kernel`. Om deze verschillen te verduidelijken, wordt in het verloop van deze sectie het verschil tussen een `Pluglet` en een `PlugletContext` verder uitgediept. De `Pluglet` interface is een contract tussen een pluglet en de kernel waarop de pluglet geplugd is. Het stelt de kernel in staat de status van een pluglet te wijzigen. Deze interface is onbereikbaar voor een client van de kernel. De `PlugletContext` zorgt ervoor dat de clients alsnog in de mogelijkheid zijn de status van de pluglet te controleren. Wanneer de `PlugletContext` van naderbij bekeken wordt, is het meteen duidelijk dat die dezelfde methodes (van naam) voorziet als de `Pluglet`. Het is belangrijk op de juiste manier om te gaan met deze vaststelling. Er wordt immers vaak, als gevolg van deze gelijkenissen, onterecht aangenomen dat een `PlugletContext` de `Pluglet` interface implementeert. Dit is echter **NIET** het geval.

Het verschil tussen de gelijknamige methodes wordt al snel duidelijk aan de hand van een voorbeeld. Stel bijvoorbeeld dat de `destroy()` methode van de `PlugletContext` wordt aangeroepen. Als eerste stap wordt de status van de pluglet gewijzigd

naar "*Destroying*". Vervolgens zal deze status, nadat de kernel de `destroy()` methode van de pluglet heeft aangeroepen, of na het verstrijken van de timeout periode, gewijzigd worden in "*Destroyed*".

Na het uiteenzetten van de werking van het principe blijft nog de vraag naar het waarom. Zou het niet eenvoudiger zijn de `init()`, `start()`, `destroy()`, ... rechtstreeks via de `Pluglet` interface op te roepen[19] ? Het antwoord op deze vraag is NEE en wordt gestaafd door de volgende argumenten:

Er moet een duidelijk onderscheid gemaakt worden tussen de functionaliteit van de eigenlijke pluglet enerzijds en die van de `PlugletContext` anderzijds. De `PlugletContext` is in essentie een meta-interface die instaat voor de status (levensverloop) van een pluglet. De pluglet op zich is in principe een RMIX reference (zie sectie A.4.1) die via een thin proxy de functionele interface verbindt met de functionaliteit op de resource. Deze RMIX verbinding komt tot stand via de `bind(...)` methode van de `PlugletContext`. Het is dus duidelijk dat er hier sprake is van een thin proxy enerzijds en van een meta-handle anderzijds. Op deze manier ontstaat er een duidelijke opsplitsing tussen de functionele (interfaces die de functionaliteit van een pluglet beschrijven) en de niet-functionele interfaces (`Pluglet` en `Suspendable` interface). Deze manier van werken zorgt ervoor dat bij het uitvoeren van meta-operaties enkel de `KernelContext` gebruikt moet worden, in het andere geval wordt er met de `bind(...)` methode een verbinding gemaakt met de pluglet. Het voordeel van dit design is niet enkel praktisch en esthetisch maar zorgt er ook voor dat:

- Naamconflicten tussen de functionele en de niet-functionele interfaces niet kunnen voorkomen.
- Een referentie, verkregen van de `bind(...)` methode, zonder security issues doorgegeven kan worden naar andere machines. Deze interface is immers niet in de mogelijkheid de pluglet te destroyen.

A.4.4 Security

Zoals reeds eerder besproken, is één van de basisprincipes van H2O het onafhankelijk handelen van de verschillende aanwezige componenten (Resources en Clients). Deze onafhankelijkheid brengt met zich mee dat er een duidelijke nood is aan robuuste interoperability en aan security mechanismen. Deze noodzaak, in combinatie met de provider georiënteerde visie van H2O, maakt dat het default security mechanisme terug te vinden is in de H2O-Kernel. De component die verantwoordelijk zal zijn voor de handhaving van de security constraints wordt de *gatekeeper* genoemd. Zoals zijn naam al laat vermoeden, zal die, bij het aanmelden van de client, de vereiste contraint checks uitvoeren en de desbetreffende client de resulterende permissies geven. Net zoals bij, bijvoorbeeld een rsh daemon, verloopt de communicatie na deze verificatie steeds via een opaque security proxy die de client identificatie informatie bevat zoals die verkregen is bij de gatekeeper. De autorisatie beslissingen, die genomen

[19] Technisch zou dit willen zeggen dat de pluglet referentie die verkregen wordt via de *bind(...)* methode de *Pluglet* interface zou implementeren.

worden door deze security proxies, zijn gebaseerd op de policies die door de kernel eigenaar en door de pluglet loading entity (de client die de pluglet heeft geladen) gespecificeerd zijn. De volledige pluglet code wordt uitgevoerd binnen de thread group die onder controle staat van de kernel. Code die vanuit de kernel zelf wordt aangeroepen, wordt binnen de loading entity's security context afgehandeld en code die aan de hand van remote call's wordt aangeroepen, wordt uitgevoerd binnen de discovery entry's security context. Hierbij dient opgemerkt te worden dat een pluglet in de mogelijkheid is security semantics te wijzigen. Het spreekt voor zich dat deze eventuele aanpassingen niet mogen resulteren in bredere permissies dan die van de loading en van dediscovering entities samen. De security aspecten, die door de gatekeeper dienen afgehandeld te worden, bestaan zoals in de meeste standaard security mechanismen uit twee belangrijke aspecten:

- *Authentication*: Dit principe is gebaseerd op JAAS (Java Authentication and Authorization Service beschreven in hoofdstuk 15 van [90]). Meer bepaald op de Pluggable Authentication Modules (PAM) van deze autorization service. Hoewel kernels vrij zijn in de manier waarop de authenticatie modules geconfigureerd worden, raadt H2O aan de X.509 certificaten[50][51] te gebruiken. Deze X.509 ITU-T (International Telecommunication Union - Telecommunication standardization sector) standaard specificeert standaardformaten voor *public key* certificaten en *certification path validation* algoritmes.

- *Autorization*: Autorisatie kan gedefinieerd worden als het toekennen van permissies aan een geïdentificeerde entiteit. Binnen H2O kan dit opgesplitst worden in twee aspecten:

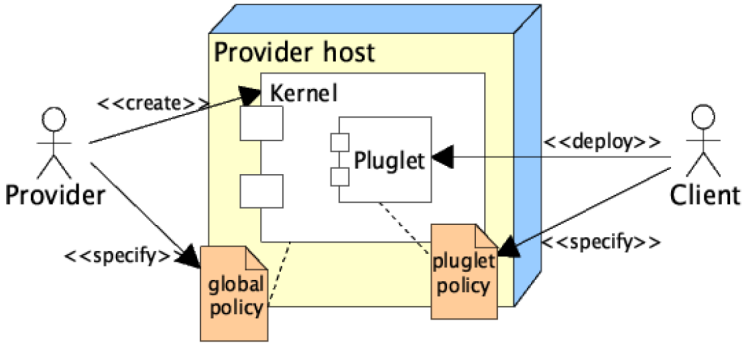
  1. Het management en de beveiliging van de resources die door de kernel provider ter beschikking worden gesteld. Deze taak wordt afgehandeld door gebruik te maken van het Java security model en van JAAS. De combinatie van deze twee concepten maakt het mogelijk aan de hand van de identiteit van de gebruiker de toegang tot de resources op een zeer nauwkeurige manier te controleren.
  2. Pluglet toegangscontrole aan de hand van de policies van de loading entities (developer, provider,...). Dit security aspect wordt afgehandeld op het niveau van de remote interfaces. De pluglet loading entity bepaalt welke client er toegang krijgt tot welk van de remote interfaces die door de pluglet geïmplementeerd worden. Dit heeft tot gevolg dat een client enkel weet heeft van die interfaces die hij ter beschikking krijgt. Van het bestaan van andere interfaces zal hij geen besef hebben en deze zullen aldus voor hem onbestaande zijn.

  Deze dualiteit in de autorisatie aspecten brengt met zich mee dat er binnen een execution context ook twee policy types bestaan (Zie figuur A.10):

  1. De *globale*, low level policy (Meer info omtrent de Policy.xml file is terug te vinden in sectie A.4.5) waarin de permissies tot de beschikbare resources

expliciet door kernel provider worden toegekend. Dit geeft de kernel provider volledige controle over de acties die een geladen pluglet mag uitvoeren binnen deze kernel.

2. De *pluglet-related* policy bepaalt welke client er toegang heeft tot welke interfaces. Het spreekt voor zich dat het opstellen van deze policy de verantwoordelijkheid is van de pluglet deployer en dat, bij afwezigheid, elke client toegang zal krijgen tot elke interface.



Figuur A.10: Global en pluglet-related policies

A.4.5 Config Files

De Configuratie van de verschillende H2O componenten is volledig gebaseerd op configuratie files. Deze configuratie files kunnen grofweg onderverdeeld worden in twee groepen:

- *De H2O ondersteunende configfiles*: Deze configfiles configureren de software die van andere projecten geadopteerd wordt om het H2O project te ondersteunen. Ze zijn gedefinieerd in het standaard Java properties formaat[20]. De twee belangrijkste concepten die met deze achtergrond geconfigureerd moeten worden, zijn:

  – *De logging functionaliteit*: Dit principe wordt geconfigureerd aan de hand van twee configuratie files:

    1. `log4j.properties`: Dit configuratiebestand dient ertoe het universeel gebruikte *log4j*[38] logging mechanisme te configureren.
    2. `logging.properties`: Deze configuratiefile geeft aan op welk niveau er informatie gelogd moet worden binnen de verschillende componenten van de H2O infrastructuur.

[20]Meer informatie is terug te vinden bij de properties klasse in de Java api: http://java.sun.com/j2se/1.4.2/docs/api/

  - *De RMIX communication layer*: De RMIX communicatie abstractie laag wordt volledig geconfigureerd aan de hand van de `rmix.provider` configuratiefile. Deze file omvat de verschillende, essentieel instelbare, parameters die nodig zijn om een degelijke configuratie tot stand te brengen. Enkele van de belangrijkste instelbare opties zijn onder meer:
    * Het RMIX provider path: Zoals in sectie A.4.1 aangehaald, moeten de JAR files, die vereist zijn om een bepaald protocol te ondersteunen, aanwezig zijn in deze directory alvorens het RMIX mechanisme communicatie via deze protocolprovider ondersteunt.
    * Verschillende timeouts die weergeven na hoeveel tijd zonder enige reactie ervan uitgegaan wordt dat een connectie niet langer bestaat.
    * Verschillende transport poorten: Deze poort identifiers geven bijvoorbeeld weer via welke poort rmix data via tcp of http worden verstuurd.
    * ...

- *De H2O specifieke configfiles*: Configfiles die de componenten configureren die tot de H2O codebase behoren. Deze configfiles worden gedefinieerd in het standaard XML formaat [75][44][96] volgens bijgeleverde DTD[21] (Document Type Definition) specificaties en kunnen eveneens in twee groepen opgesplitst worden:

  - *Security gerelateerde configfiles*: Onder deze configuratiefiles vinden we één file terug die gebruikt wordt bij de authenticatie (zie sectie A.4.4) van de gebruikers:
    * De `Users.xml` file beschrijft de verschillende gebruikers die toegang krijgen tot de H2O kernel en definieert tot welke groep de gebruiker behoort. Aan de hand van deze groep zal dan, in de autorisatie fase, bepaald worden welke privileges elke deelnemende gebruiker krijgt.

    Tevens vinden we twee configuratiefiles terug die instaan voor de definitie van het Autorisatie mechanisme zoals beschreven in sectie A.4.4.

    * De `SystemPolicy.xml` wordt gebruikt om de systeem specifieke permissies te definiëren. Concreet betekent dit dat de privileges, die betrekking hebben op de toegang tot de verschillende resources van de resource provider, hier bepaald worden. Dit komt er op neer dat de codebase van H2O en van de onderliggende componenten (RMIX, protocol providers, ...) de benodigde privileges krijgen om gebruikers, die over voldoende privileges beschikken, het gewenste resource gebruik toe te staan. Enkele voorbeelden zijn:
      · De H2O codebase ($h2o.home/lib/*) krijgt toestemming sockets te gebruiken op poort 1024.
      · De Rmix codebase krijgt de toestemming te lezen in de protocol provider directory.

[21] De bijgeleverde DTD definitie bestaat uit een .dtd file (bv kernelConfig-1.0.dtd).

        - Protocol providers krijgen de permissie locale endpoints te creeren op poort 1024.
        - ...
    * De `Policy.xml` wordt gebruikt om H2O specifieke permissies te definiëren. Concreet betekent dit dat de privileges die betrekking hebben op de toegang tot en op het gebruik van de verschillende H2O componenten hier bepaald worden. Het komt erop neer dat, afhankelijk van de authenticatie, aan de hand van deze configuratiefile bepaald wordt welke permissies een gebruiker krijgt binnen deze H2O kernel. Elke gebruiker zal, na authenticatie, met een groep verbonden zijn waarvan de premissies in de `Policy.xml` file beschreven staan. Enkele voorbeelden zijn:
        - Een Administrator krijgt alle Kernel, alle Pluglet en alle Session permissies.
        - Een deployer krijgt bijvoorbeeld slechts één Pluglet permissie namelijk de Deploy permissie.
        - ...

- *Kernel gerelateerde configfiles*: Er zijn twee Kernel gerelateerde configfiles die beide bijdragen tot de configuratie van de kernel maar alsnog een volledig verschillende doelstelling hebben:
    * De `KernelConfig.xml` valt het best te vergelijken met de `rmix.provider` configuratie file en omvat de verschillende, essentieel instelbare, parameters die nodig zijn om een degelijke kernel configuratie tot stand te brengen. Enkele van de belangrijkste instelbare opties zijn onder meer:
        - De working directory van de kernel. Hierin worden de tijdelijke kernel bestanden opgeslagen.
        - De plaats waar de loggingconfiguratiefile (logging.properties) terug te vinden is.
        - De poorten waarlangs standaard en SSL communicatie gebeurt (standaard 7799 en 7800)
        - De plaats van de User.xml file.
        - De plaats van de Autodeploy.xml file.
        - ...
    * De `Autodeploy.xml` zal, zoals in sectie A.4.6 besproken wordt, bepalen welke Pluglets er bij het starten van de H2O-kernel automatisch in de kernel geplugd worden. Deze file bevat voor elk van deze pluglets de vereiste informatie om de deployment van de pluglet tot een goed einde te brengen. Concreet komt dit neer op het volgende:
        - De naam waaronder de pluglet in de Kernel geplugd wordt.
        - De naam van de klasse die de pluglet definieert.

- De retypolicy die weergeeft wat er moet gebeuren wanneer de pluglet die geladen wordt reeds bestaat (bijvoorbeeld de pluglet vervangen of de huidige pluglet laden onder een anderen naam).
- Het classpath waarin alle codebases staan die nodig zijn om de pluglet te laden.
- Een lijst van initialisatie parameters die in de `initParams` van de door de user te implementeren `init(Pluglet.RuntimeContext context, java.util.Properties initParams)` methode van de `Pluglet` klasse terechtkomen.

A.4.6 AutoDeploy

Naast de standaard functionaliteit die tot dusver besproken is, biedt het H2O framework tevens een eenvoudige methode om pluglets tijdens de kernel-startup in de H2O kernel te pluggen. Deze methodiek heeft meerdere doelstellingen:

- Vereenvoudiging van de deployment van standaard pluglets.
- Componenten die instaan voor het vormen van een gedistribueerde virtuele machine kunnen bij startup onmiddellijk worden geladen.
- Eventuele uitbreidingen van het H2O framework, waarbij enkele pluglets de toegevoegde functionaliteit verzorgen, kunnen zonder het toevoegen van een startup mechanisme eenvoudigweg gestart worden door het standaard kernel deployment mechanisme. De extra pluglets kunnen immers tijdens de autodeployment geladen worden.

De werking van dit mechanisme is eenvoudig. Zoals in sectie A.4.5 reeds werd aangehaald, wordt de autodeployment methodiek geconfigureerd, gebruik makend van de `Autodeploy.xml` file. Deze file is terug te vinden op de plaats zoals die gespecificeerd is in de `KernelConfig.xml` file (Zie sectie A.4.5). Deze file is een XML file opgesteld volgens de DeploymentDescriptorList-1.0.dtd specificaties. Concreet bestaat de file uit een deploymentdescriptorlijst. Dit is een lijst die, zoals zijn naam al doet vermoeden, verschillende deploymentdescriptors bevat. Een deploymentdescriptor ziet er als volgt uit:

```
1 <DeploymentDescriptor name="PlugletNaam">
2   <Classpath path="http://www.de.url/path/CodeBase.jar"/>
3   <Param name="className" value="de.namespace.KlasseNaam"/>
4   <Param name="retryPolicy" value="Replace"/>
5   <InitParams>
6     <Param name="EenInitParameter" value="Een Waarde"/>
7     <Param name="EenAndereInitParameter" value="Een Waarde"/>
8   </InitParams>
9 </DeploymentDescriptor>
```

In concreto bestaat een deploymentdescriptor dus uit:

- *De Pluglet Naam*: De naam die een pluglet krijgt wanneer hij in een kernel geladen wordt. Neem echter wel nota van het feit dat een pluglet klasse meerdere malen in een Kernel geladen kan worden onder een andere Pluglet naam.

- *Het Classpath*: Dit veld bevat één of meerdere JAR bestanden waarin de codebase voor deze pluglet omvat zit. Het gebruik van URI classloaders in H2O maakt dat het naast lokale JAR files ook mogelijk is remote JAR files te specificeren zolang deze via het *http* protocol te bereiken zijn. H2O zal deze JAR files dan eerst downloaden alvorens ze te gebruiken.

- *De Class Name*: Dit is de naam van de klasse die de pluglet definieert. Hij bestaat concreet uit de package naam gevolgd door de naam van de klasse. Het is belangrijk hierbij op te merken dat deze naam snel verwisseld wordt met de *Pluglet Naam*. Deze namen zijn echter twee totaal verschillende dingen. Een klasse met een bepaalde klasse naam kan immers verschillende keren onder een andere pluglet naam in een kernel geplugd worden.

- *De retryPolicy*: Dit veld geeft weer wat er moet gebeuren wanneer de te laden pluglet reeds bestaat (bijvoorbeeld de pluglet vervangen of de huidige pluglet laden onder een andere naam).

- *InitParams*: Een lijst van initialisatie parameters die in de `initParams` van de door de user te implementeren `init(Pluglet.RuntimeContext context, java.util.Properties initParams)` `Pluglet` methode terecht komen. In deze methode kunnen ze dan gebruikt worden om de pluglet te initialiseren volgens de wensen van de developer. Zo zou bijvoorbeeld aan een pluglet, die de toegang tot een deur verzorgt, de url van de toegangscodefile via de initparameters meegegeven kunnen worden. Tijdens de initialisatiefase wordt er dan voor gezorgd dat de ingegeven codes steeds vergeleken worden met die van de opgegeven file. Op deze manier kan elke deur werken met verschillende toegangscodes.

In de praktijk zal bij het starten van een Kernel de gespecificeerde Autodeploy file ingeladen worden en de pluglets, die in de verschillende pluglet descriptors staan gedeployed worden. Hier dient nog bij opgemerkt te worden dat deze functionaliteit slechts terdege werkt vanaf H2O versie 2.1.

A.5 Operationeel Model

In deze sectie wordt het gebruik van H2O geschetst vanuit het oogpunt van de verschillende participerende entiteiten. Er zijn vier entiteittypes te onderscheiden:

- *Provider*: Deze entiteit stelt, zoals zijn naam al laat vermoeden, een resource ter beschikking. Concreet gebeurt dit door een H2O kernel te starten op de beschikbare resource.

- *Developer*: Deze entiteit ontwikkelt services (pluglets) die in een H2O kernel geplugd kunnen worden. De pluglet zal dan op deze beschikbare resource geserved worden.

- *Reseller*: Een entiteit die enkele pluglets in zijn repository heeft en deze, door ze in een kernel te laden, ter beschikking kan stellen van eventuele gebruikers.
- *Client*: De client is de gebruiker van het systeem. Hij is in de mogelijkheid:
  - Een pluglet die aanwezig is op een kernel te gebruiken.
  - Een pluglet in een kernel te pluggen en al dan niet te gebruiken.

Het opzetten van een H2O netwerk zal in verschillende opeenvolgende stappen gebeuren. De stappen die hieronder beschreven worden, worden op figuur A.11 aangeduid met de overeenkomende nummers. Alvorens de verschillende stappen te overlopen, is het voor de volledigheid belangrijk op te merken dat, zoals in sectie A.4.2 en A.4.4 wordt besproken en op figuur A.8 wordt getoond, steeds eerst een autorisatie en een authenticatie fase moeten doorlopen worden alvorens er bewerkingen[22] op de kernel kunnen gebeuren:

1. *Pluglet Developement*: Als eerste stap wordt de beschrijving gegeven van een stap die niet per definitie als eerste moet gebeuren maar die in principe altijd kan plaats vinden. Hiermee wordt zowel een periode ver voor de start van het H2O netwerk bedoeld als eender welke periode tijdens het bestaan van het netwerk. In deze stap worden pluglets ontwikkeld die een bepaalde service moet bieden aan de verschillende gebruikers. Concreet houdt dit in dat, zoals reeds eerder is vermeld en op figuur A.9 wordt aangeduid, de `Pluglet` interface en eventueel de `Suspendible` interface alsook de verschillende Functionele interfaces worden geïmplementeerd om de gewenste service aan te bieden.

2. *Kernel Deployment*: De eerste stap die moet gebeuren alvorens er van een H2O netwerk kan gesproken worden, is het deployen van één of meerdere kernels door de Providers op hun beschikbare resources. Er dient hierbij vermeld te worden dat deze stap in de meeste gevallen gecombineerd wordt met een Pluglet Deployment stap zoals wordt besproken in sectie A.4.6 en duidelijk getoond wordt aan de linker zijde van figuur A.11. Het zogenaamde *AutoDeployment*.

3. *Pluglet Deployment*: In deze fase worden de ontwikkelde pluglets op een H2O kernel geplugd door de providers (zie AutoDeployment in sectie A.4.6), de resellers of de clients.

4. *Pluglet Publishing*: Na de deployment van een pluglet zal die geregistreerd worden in één van de beschikbare registratie mechanismen die terug te vinden zijn bovenaan figuur A.11. Zoals reeds in de sectie omtrent RMIX werd aangehaald, ondersteunt RMIX en bijgevolg ook H2O standaard enkel de RMI registry[23]. Dit omdat de H2O eenvoudig mapt op andere specification methodes (bijvoorbeeld IDL of WSDL) en op andere protocols (bijvoorbeeld IIOP

[22] Met bewerkingen worden opdrachten zoals het deployen van pluglets, het verkrijgen van pluglet interfaces en vele andere bedoeld.

[23] De RMI registry werd ondersteund om backward compatibility te garanderen.

en SOAP) en omdat de gebruiker vrij gelaten moet worden in zijn keuze uit deze methodes. Het doorsnee gebruik verbindt de user ertoe de beschikbare pluglet classes te vergezellen van een WSDL description. Gebruik makend van deze specificatie en van enkele pluglet-specifieke karakteristieken is het mogelijk een pluglet te registreren via verschillende discovery schema's (bijvoorbeeld UDDI).

5. *Pluglet Discovery*: Alvorens een pluglet kan gebruikt worden, moet deze eerst ontdekt worden. Dit gebeurt aan de hand van een discovery mechanisme dat gebruik maakt van de registraties uit de voorgaande stap.

6. *Pluglet Binding*: Na de deployment van een pluglet in een kernel kan de service die hij aanbiedt, gebruikt worden door andere componenten of door clients, gebruik makend van de functionele interfaces van de pluglet. De onderliggende RMIX communicatieabstractie zorgt ervoor dat de gebruiker van de service niet gebonden is aan een vast communicatie protocol. De client kan vrij kiezen tussen de verschillende endpoints die aangeboden worden door het remote object en hij kan at runtime het protocol, waarover de communicatie gebeurt, alsnog aanpassen[24]. Bovenop deze vrije keuze tussen protocols maakt H2O het ook mogelijk een onderscheid te maken tussen verschillende gebruikers. Zoals in sectie A.4.4 wordt besproken, bevatten kernels zogenaamde *gatekeepers*. Deze maken een onderscheid tussen de verschillende gebruikers en de privileges die zij al dan niet hebben. Dit alles wordt geïllustreerd door het volgende voorbeeld gebaseerd op figuur A.6. Dit voorbeeld bestaat uit een manager(M) en een werknemer(W). Beiden hebben interesse in het gebruik van service(S) welke gedeployed is in een H2O kernel op een welbepaalde resource. Naast deze interesse spreekt het voor zich dat de manager andere privileges heeft dan een werknemer. De eerste stap verloopt voor beide gebruikers analoog. Er moet immers een connectie gemaakt worden met de H2O kernel. Dit gebeurt steeds gebruik makend van het SOAP protocol[25]. In de volgende stap wordt er, gebruik makend van de `bind(...)` methode, een remote reference naar de pluglet opgevraagd. De code die hiervoor gebruikt wordt, ziet er als volgt uit:

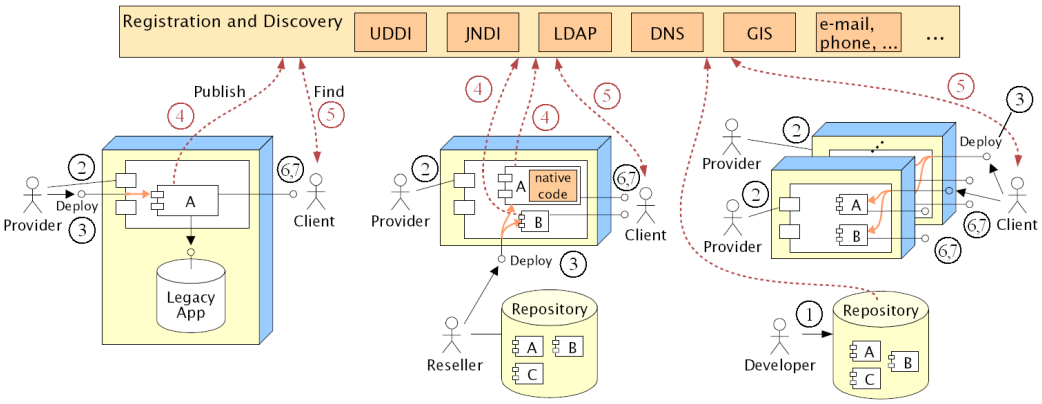
```
1 String protocol = "RMIX-JRMPX 1.0";
2 ExportProperties.Messaging transProvider =
3   new ExportProperties.Messaging(protocol);
4 ExportProperties.Transport transportType =
5   ExportProperties.Transport.CURRENT_UNENCRYPTED_COMPRESSED;
6 ExportProperties properties =
7   new ExportProperties(transProvieder,transportType);
8 S obj = (S)plugletCtxt.bind(properties);
```

[24]Het spreekt voor zich dat een ander protocol enkel kan gekozen worden als dit ook door de service wordt ondersteund. Dit wil zeggen dat de benodigde JAR files in een vooraf gespecificeerde protocol-provider-directory beschikbaar moeten zijn.

[25]De keuze voor het SOAP protocol als basis communicatie middel was, zoals in hoofdstuk A.4.1 omtrent RMIX besproken, gebaseerd op de universele ondersteuning van dit protocol.

De bind methode krijgt als parameters het messaging protocol (JRMPX 1.0) en het transport type (compressed TCP) mee. Het spreekt dan ook voor zich dat aan de hand van deze `bind(...)` methode dynamisch van protocol gewisseld kan worden. Concreet is de oproep van deze methode voor de kernel een verzoek om het pluglet object te exporteren volgens het meegegeven protocol. Het onderscheid tussen de verschillende client types gebeurt ook in deze methode. Tijdens de verwerking van de `bind(...)` methode detecteert de kernel welke gebruiker de request doet. In het geval van M zal de `bind(...)` methode eveneens een `Manager` interface exporteren, in het geval van W zal deze interface niet mee geexporteerd worden. Tot slot dient nog vermeld te worden dat omwille van het feit dat de `bind(...)` methode verschillende malen kan aangeroepen worden er verschillende endpoints naar eenzelfde pluglet kunnen ontstaan die elk gebaseerd zijn op een al dan niet verschillend protocol.

7. Tot slot kunnen de beschikbare pluglets via de door de `bind(...)` methode bezorgde functionele interfaces gebruikt worden.



Figuur A.11: H2O gebruik scenario's

BIJLAGE B

Jini

B.1 Inleiding

In dit hoofdstuk wordt op een zo bondige mogelijke manier een diepgaand overzicht gegeven van het concept Jini en de technologie die dit geheel ondersteunt. Deze inleiding bestaat uit een korte historische schets en uit een overzicht van de doelstellingen van Jini. In de tweede sectie worden de basis principes besproken. De derde sectie bestaat uit de ondersteunende componenten die nodig zijn om de beoogde doelstellingen van deze basis principes te realiseren. Tot slot wordt in de laatste sectie het geheel dan verder uitgeklaard aan de hand van een duidelijk voorbeeld.

Hoewel de eerste versie van Jini op 25 januari 1999 aan het grote publiek werd voorgesteld, start het Jini verhaal veel vroeger. Om dit terdege te begrijpen, is het belangrijk het initiële opzet van Java te kennen. De developers van wat toen

nog de naam Oak droeg, hadden als doelstelling een taal te creëren die een geheel van elektronische componenten door middel van het uitwisselen van data kon laten samenwerken. Java boekte zijn eerste grote succes in 1994 bij de release van WebRunner. Deze webbrowser maakte het mogelijk om code onder de vorm van applets van een webserver te downloaden en uit te voeren. De aard van dit en de daaropvolgende successen dreef Java af van zijn vooropgestelde doelstellingen naar de taal zoals ze nu gekend is.

Onder begeleiding van een groot gedeelte van het originele Java team werd in alle discretie het Jini project op poten gezet. Het opzet was simpel: de originele doelstellingen van Java realiseren. Dit project bleef geheim tot het in 1998 in een artikel op de voorpagina van de New York Times verscheen. Verdere publicaties in o.a. Wired magazine, zorgden ervoor dat, zoals reeds vermeld, de eerste versie op 25 januari 1999 aan het grote publiek werd voorgesteld.

De doelstellingen van Jini waren eenvoudig:

- *Plug and play:* Een toestel dat een service levert, moet enkel voorzien worden van stroom en netwerk toegang om zijn service ter beschikking te stellen.
- *Driver onafhankelijk*: Gebruikers hebben geen drivers nodig om een service te kunnen gebruiken.
- *Administratie vrij*: Elke service wordt onderhouden door diegene die hem ter beschikking stelt. Zo is het systeem vrij van enige globale administratie.
- *Combinatie mogelijkheden*: Services kunnen gebruik maken van andere services om zo een meer uitgebreide service aan te bieden.
- *Failure onafhankelijkheid*: Elke service werkt onafhankelijk en blijft dus ter beschikking bij het eventuele falen van delen van het netwerk of andere van machines.
- *Automatisch leren*: Software en toestellen leren automatisch van elkaar.

De vereisten waren eenvoudig van opzet maar vormden een hele uitdaging. Het geheel moest kunnen omgaan met de acht denkfouten in het gedistribueerde programmeren. Deze denkfouten zijn elk gebaseerd op een fout uitgangspunt. Ze veronderstellen dat de volgende acht stellingen correct blijven gedurende het bestaan van het systeem, iets wat vanzelfsprekend niet zo is:

- Het netwerk is betrouwbaar.
- Er treden geen vertragingen op binnen het netwerk.
- Er is een onbeperkte bandbreedte.
- Het netwerk is volledig veilig.
- De opbouw van het netwerk verandert niet.
- Er is slechts één netwerk beheerder.

Figuur B.1: Het Jini Concept

- Er zijn geen transportkosten.
- Het netwerk is homogeen op elk gebied (machines, netwerkverbindingen, ...)

Hoewel het Jini project in zijn doelstellingen slaagde en er aan het merendeel van de vooropgestelde vereisten voldaan werd, lanceerde het Sun Davis team op 5 juni 2003 een nieuwe Jini release (Jini 2.0). In tegenstelling tot de voorgaande nieuwe releases werd de feature set in deze versie drastisch uitgebreid. Het waren in essentie de beperkingen en wensen van de gebruikers die aan de basis lagen van deze nieuwe versie. Dit alles werd vanzelfsprekend gerealiseerd zonder van de initiële doelstellingen en vereisten af te wijken.

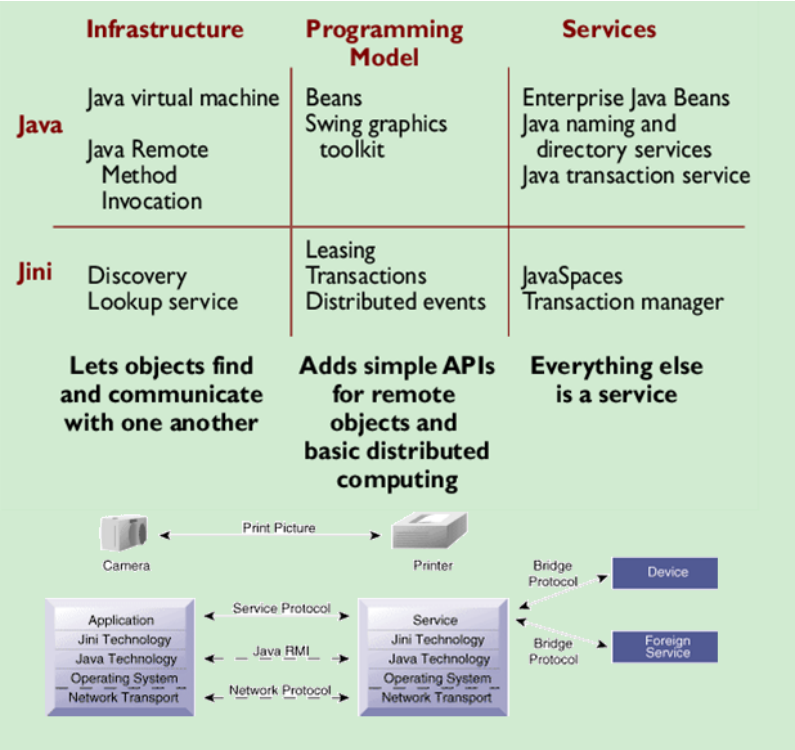
De wijzigingen spitsten zich toe op drie domeinen:

1. Programmatorische Modellen:
   - *Configuratie*: Een nieuw configuratiesysteem werd geïntegreerd. Aan de hand van deze configuratiefiles met Java uitziende code werd het mogelijk elk onderdeel (tot de services toe) te configureren.
   - *Exporter*: Er werden abstracties toegevoegd voor het exporteren van remote objecten en voor het verkrijgen van server-side informatie van een remote call in uitvoering.
   - *ProxyPreparer*: Het werd mogelijk om een service proxy een voorbereidingsfase te laten doorlopen alvorens de uitvoeringsmogelijkheden ter beschikking werden gesteld. Dit is noodzakelijk om de verworven proxy client side security constraints op te leggen alvorens deze gebruikt kan worden (Zie ook sectie B.3.7).

2. Nieuwe Infrastructuur:
    - *Security*: Er werd een security mechanisme toegevoegd dat de betrouwbaarheid en de veiligheid van informatie garandeert. Dit was vooral toegespitst op het dynamisch downloaden van code.
    - *Invocation Constraints*: Het werd mogelijk gemaakt voor de gebruiker om, aan de hand van constraints, de method calls beter te controleren en te beheren.
    - *JERI*: Jini Extensible Remote Invocation is een implementatie van het RMI programmeer model en zorgt ervoor dat elk geëxporteerd remote object aanpassingen kan doorvoeren in om het even welke laag van het gebruikte protocol.
    - *Dynamic Policies*: Zoals besproken wordt in sectie B.3.7 omtrent security, worden er, in de authenticatiefase tijdens de discovery van een service, de proxy enkele constraints en permissies opgelegd. Dit gebeurt zowel service side aan de hand van de service constraints als client side aan de hand van de client constraints. Aangezien de permissies en constraints aan client zijde door de client bepaald worden tijdens het verwerven van de proxy, spreekt het voor zich dat de client deze policies dynamisch kan aanpassen. Server side is dit minder evident aangezien de policies in principe bepaald moeten worden op het moment dat de service geregistreerd wordt. Deze nieuwe Dynamic Policy techniek zal er echter voor zorgen dat bij het instantiëren van een proxy voor een client deze policies alsnog dynamisch kunnen gewijzigd worden. Dit door het mogelijk te maken policies dynamisch te zetten bij een Classloader. Zoals ook uit het volgende puntje zal blijken en uitgebreid besproken zal worden in sectie B.3.7 zal de Classloader een zeer belangrijke rol spelen in het Jini security mechanisme.
    - *Preferred Classloading*: Traditionele Classloading zoekt steeds naar een parent Classloader die deze klasse reeds voorheen heeft geladen en delegeert de verdere afhandeling aan deze parent Classloader. In preferred classloading zal dit niet langer het geval zijn. Dit maakt dat elke Classloader zijn geprefereerde versie van een klasse kan laden los van zijn parents. Dit wordt verwezenlijkt door de eerste URL locatie van de Classloader te voorzien van een lijst met geprefereerde klassen.
    - *Discovery Protocol v2*: Deze nieuwe versie zorgt er enerzijds voor dat de verschillende reeds bestaande discovery formaten ondersteund blijven en hier bovenop nieuwe formaten worden toegevoegd. Dit voor elk van de discovery protocols. Tevens worden de nieuwe Jini2 security mechanismen, zoals class trust verification en security constraints, in deze implementatie opgenomen.
3. Update van de bestaande services: Met deze aanpassingen werd het reeds eerder aangehaalde configuratiesysteem geïntegreerd in de reeds bestaande services.

Het spreekt dan ook voor zich dat in het verdere verloop van dit werk met het concept JINI steeds naar deze Sun Davis versie zal gerefereerd worden.

Tot slot geeft figuur B.2 een duidelijk overzicht van de plaats die Jini en Java innemen in het totaal concept. Wat hier duidelijk opvalt is dat Jini een extra aan functionaliteit voorziet bovenop de Java infrastructuur. Dit lagen model wordt vooral duidelijk in het onderste gedeelte van de figuur. De onderliggende code en communicatie functionaliteit die nodig zijn om de Jini Technologie te ondersteunen, worden aangerijkt door de reeds bestaande Java Technologie. Deze structuur zorgt ervoor dat de originele Java doelstellingen bereikt worden door de Java Technologie te combineren met de Jini Technologie.

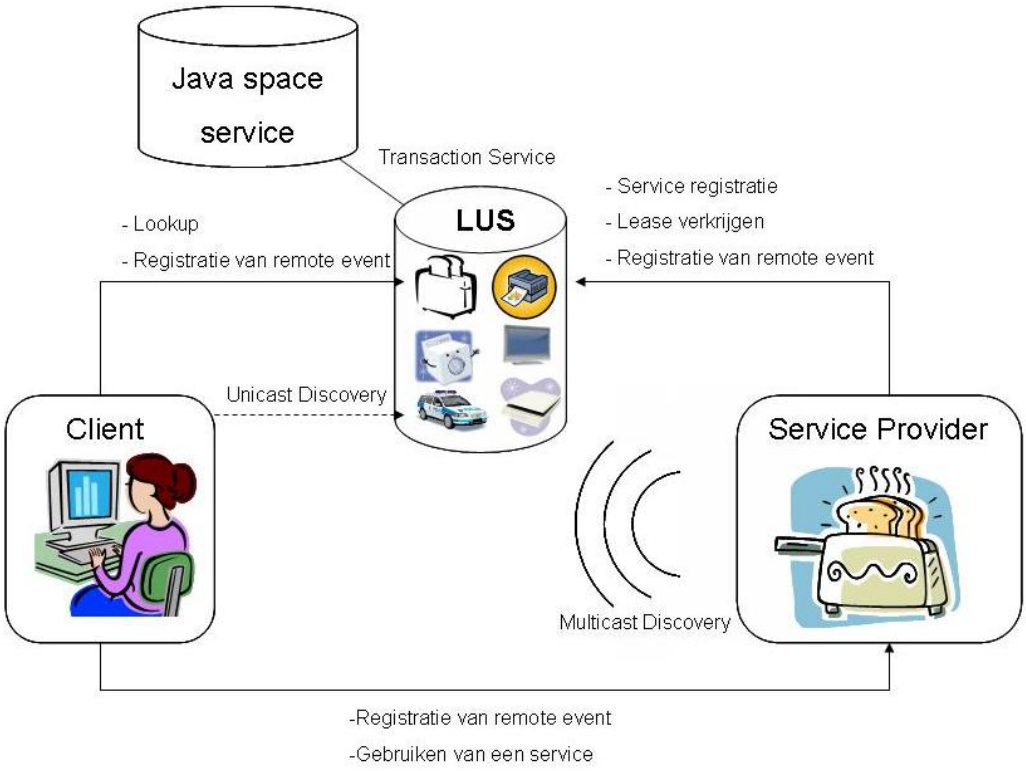


Figuur B.2: Java vs. Jini

B.2 Basis Principes

In deze sectie wordt het principe waarop Jini gebaseerd is verder uitgewerkt. Alvorens elk onderdeel individueel te bespreken, is het belangrijk een globaal overzicht te hebben van het Jini concept. Het principe van Jini draait volledig rond het ter beschikking stellen van services en het gebruiken van deze services. Zoals uit figuur B.3 meteen blijkt, staat de LUS of LookUp Service hierin centraal. Het aanbieden

van of het gebruik van een service, wordt steeds voorafgegaan door het registreren en respectievelijk het opzoeken van deze service op de LUS. Alvorens deze opzoeking of registratie kan gebeuren, moet er een LUS ter beschikking zijn. Het ontdekken van zo een LUS en de principes die hiermee gepaard gaan, zullen de Discovery genoemd worden.



Figuur B.3: Jini Concepten

Het hele JINI concept is gebaseerd op zes principes:

1. Lookup: subsectie B.2.1
2. Discovery: subsectie B.2.2
3. Leasing: subsectie B.2.3
4. Events: subsectie B.2.4
5. Transacties: subsectie B.2.5
6. Services: subsectie B.2.6

B.2.1 Lookup

Een Jini netwerk omgeving bestaat uit een aantal communities. Elk van deze communities bestaat uit een reeks van services die ter beschikking worden gesteld en

een groep clients die deze services willen gebruiken. Zoals reeds eerder gesuggereerd werd, staat de LUS centraal in zo een community. In essentie is zijn taak drieledig:

1. *Registration*: Een service provider kan zijn deployed service ter beschikking stellen aan de community door deze te registreren bij de LUS.

2. *Service lookup*: Een client kan op zoek gaan naar een service die voldoet aan zijn vooropgestelde eisen.

3. *Notification*: Een client kan zich registreren bij de LUS om via notificatie op de hoogte gehouden te worden van veranderingen in bestaande services of bij het verschijnen van nieuwe services. Dit design pattern, ook wel het *Publish/subscribe* of *observer pattern*[1] genoemd, waarin een message gestuurd wordt naar een groep van geïnteresseerden zonder de specifieke ontvangers te kennen, is duidelijk loosely-coupled en garandeert scalability.

B.2.2 Discovery

De centrale functionaliteit van een LUS in een Jini community maakt dat het essentieel is dat elke provider (degene die een service wil aanbieden) en elke client (degene die de service wil gebruiken) een dergelijke LUS ter beschikking hebben. Aangezien er initieel geen connectie met een LUS aanwezig is, werd er een techniek ontwikkeld die instaat voor het ontdekken van zo een LUS.

Deze techniek omvat de drie Jini Discovery protocols:

1. *Multicast Request Protocol*:

   Dit protocol wordt gebruikt door client en provider om binnen de multicastradius op zoek te gaan naar een lookup service.

2. *Multicast Announcement Protocol*:

   Dit protocol werkt omgekeerd en wordt gebruikt door de lookup services om de providers en de clients binnen de multicast radius te informeren over hun bestaan.

3. *Unicast Discovery Protocol*:

   Dit protocol wordt gebruikt door clients en providers wanneer de netwerklocatie van de lookup service gekend is. Deze plaats wordt gespecificeerd aan de hand van een URL. Deze URL bevat "jini" als protocol specifier en ziet er als volgt uit: *jini://raiden.ua.ac.be*. Met dit protocol wordt het mogelijk gemaakt om lookup services te bereiken buiten de multicast radius. Dit is essentieel in een tijdperk waar switchen en firewalls multicast messages in vele gevallen tegenhouden.

[1]Het observer design pattern wordt gedefinieerd als: *Define a one-to-many dependency between objects so that when one object changes state, all its dependent are notified and update automatically.*

B.2.3 Leasing

Een dynamisch netwerk wordt gekarakteriseerd door de onvoorspelbare veranderingen die erin optreden. Services kunnen even plots verschijnen als verdwijnen, client machines kunnen down gaan tijdens het gebruik van een service, ... Dit alles kan gebeuren zonder enige vorm van notificatie (Denk maar aan wat er gebeurt wanneer het netwerk faalt of er een machine crasht). Indien dit probleem niet zou aangepakt worden, zouden de gevolgen niet te overzien zijn:

- Gecrashte clients zouden eeuwig een bepaalde service kunnen claimen.
- Gecrashte services zouden blijven bestaan in de LUS. Dit is enerzijds een memory leak aangezien er steeds meer services zouden geregistreerd worden en de gecrashte services nooit zouden verdwijnen. Anderzijds heeft dit als gevolg dat de LUS incorrecte informatie zou verspreiden. Clients zouden services willen gebruiken die reeds lang niet meer bestaan.
- Services zouden na een crash twee maal kunnen geregistreerd zijn.

Om dit probleem op te lossen, werkt Jini met een Leasing Model. Wanneer een client een service wil gebruiken, krijgt hij een lease van de service provider. Zo een lease is geldig voor een bepaalde tijdspanne die wordt bepaald door de lease grantor. Indien een client de service wenst te blijven gebruiken, moet hij de provider hiervan op de hoogte stellen alvorens de leasetijd verstreken is. Dit gebeurt eenvoudigweg door een nieuwe lease te vragen. Wanneer een lease verstrijkt, kan de provider ervan uitgaan dat de client hetzij door netwerk problemen hetzij door hardware problemen niet langer in staat is de service te gebruiken en deze vervolgens terug ter beschikking stellen voor de andere clients. Tevens weet een client wanneer zijn lease niet wordt verlengd dat er waarschijnlijk iets mis is met de provider en kan hij hierop adequaat reageren. Hier komt nog bij dat de Lookup service op zich ook een service is waarvan al de andere services client zijn. Dit brengt met zich mee dat de LUS op de hoogte wordt gehouden van de status van de geregistreerde services aangezien deze telkens hun lease moeten verlengen. Wanneer dit niet gebeurt, beschouwt de LUS deze service als verdwenen en is deze niet langer geregistreerd.

B.2.4 Events

De Jini Remote Events vertonen enerzijds grote gelijkenissen met het gekende Java event model maar zijn anderzijds ook sterk verschillend. Net zoals bij Java zal de ontvanger van de events eerst een event listener moeten registreren bij de event generator. Wanneer er zich een event voordoet, zal deze event generator een notificatie doorgeven aan de geregistreerde clients. Dit gebeurt via de event listeners en enkel bij diegenen die interesse hebben in het specifieke event dat zich heeft voorgedaan. Er is echter ook een groot verschil met de Java events. Waar in Java listeners enkel specifieke events kunnen ontvangen (MouseListener kan enkel events van de muis ontvangen) moeten Jini event listeners in de mogelijkheid zijn elke mogelijke event te ontvangen. Dit wordt gerealiseerd door elk event object en elke listener te voorzien van dezelfde interface.

B.2.5 Transacties

Transacties vinden we terug in het Jini framework om het hoofd te bieden aan partial failures. Wanneer het uitvoeren van een taak gedeeltelijk faalt, is het heel belangrijk te weten welke opdrachten er wel doorgevoerd zijn en welke de gevolgen ervan zijn. Het is immers belangrijk ervoor te zorgen dat het systeem niet inconsistent wordt. De oplossing hiervoor, binnen Jini, wordt aangeboden door een transaction based mechanisme met ACID [13] (Atomicity, Consistency, Isolation, Durability) eigenschappen, gebaseerd op het 2PC (two phase commit). Dit mechanisme is geadopteerd uit de database wereld waar het gebruikt wordt om de accuraatheid en consistentie van gedistribueerde data te garanderen [92] [71]. Hoewel deze transacties later uitgebreider behandeld worden is het toch belangrijk op te merken dat de Jini approach hierin heel basic is en de implementatie van de verschillende methodes open laat voor de gebruiker.

B.2.6 Services

Een service is een dienst die aangeboden wordt door de provider. Er rest hem enkel nog de registratie bij de LUS om hem ter beschikking te stellen. Men zou het kunnen vergelijken met een leerkracht die bijles geeft en een advertentie in de lokale streekkrant zet of zoals Jan Newmark het verwoordt:

> *"A Service is a logical concept such as a blender, a chat service or a disk. It will usually turn out to be defined by a Java interface, and commonly the service itself will be identified by this interface. Each service can be implemented in many ways, by many different vendors. For example, there may be Joe's dating service, Mary's dating service or many others. What makes them the "same" service is that they implement the same interface."*

Dit heeft als gevolg dat elke gebruiker van Jini de neiging kan hebben om een eigen interface te ontwikkelen. Het nadeel hiervan is enerzijds dat bij toenemende complexiteit van de aangeboden diensten (iets wat vaak voorkomt bij diensten die aan populariteit toenemen) de interface recht evenredig in complexiteit zal stijgen. Anderzijds zullen veel providers met dezelfde diensten elk hun eigen interface ontwikkelen zodat er aan het originele opzet van Jini wordt voorbijgegaan. Interface standaardisatie zal daarom één van de key factoren zijn die, het al dan niet slagen van een wereldwijd Jini netwerk, in de hand zullen werken. Het blijft daarom tot op heden een grote verrassing en tegelijkertijd teleurstelling dat er in dit onderzoeksdomein nog steeds geen successen geboekt zijn.

B.3 Ondersteunende Concepten

In deze sectie worden de verschillende concepten en componenten die nodig zijn om het opzet van deze basisprincipes te bereiken meer in detail uitgediept. Dit zowel

op het niveau van gebruik en implementatie als op het niveau van de low level ondersteuning.

B.3.1 Lookup Service

Wanneer er dieper ingegaan wordt op Discovery, blijkt dat er bij de Lookup Service heel wat meer functionaliteit komt kijken dan deze besproken in sectie B.2.1. De basis functionaliteit moet uitgebreid worden naar de volgende acht punten:

1. *Registration*:

   Wanneer een provider zijn service ter beschikking wil stellen, moet deze geregistreerd worden bij de LUS. Deze registratie gebeurt door de provider aan de hand van een `ServiceItem` object. Dit object bestaat uit de volgende velden:

   - Proxy object.
   - Service ID.
   - Set van service beschrijvende attributen.

   Dit service item zal tijdens de registratie naar de Lookup Service worden gestuurd. De objecten worden zorgvuldig opgeslagen om na het eventuele falen van het systeem de reeds geregistreerde services opnieuw te registeren. Na de registratie wordt een `ServiceRegistration` interface object teruggegeven aan de provider. Deze interface kan gebruikt worden om toegang te krijgen tot de lease die de service van de LUS heeft gekregen. De interface heeft tevens de mogelijkheid de beschrijvende attributen van de service te wijzigen. De lease moet telkens, voor hij afloopt, verlengd worden om de registratie van de service te behouden. Gebeurt dit niet dan zal de service niet langer geregistreerd en bijgevolg niet meer toegankelijk zijn.

   De beschrijvende attributen implementeren de `EntryMarker` interface. Deze interface is afgeleid van `java.io.serializable`. Dit maakt dat de geïmplementeerde attributen op triviale wijze kunnen doorgegeven worden aan een remote machine. Deze attributen kunnen de meest complexe serializeerbare Java objecten bevatten. Het is dus mogelijk een volledige interface via zo een attribuut door te geven. Sterker nog, het is mogelijk verschillende user interfaces door te geven en aan de hand van het type client een keuze te maken uit deze interfaces. Er moet hier opgemerkt worden dat de gebruiker niet de volledige interface code moet downloaden. Het is perfect mogelijk dat enkel de interface die nuttig is voor die specifieke gebruiker wordt doorgegeven.

2. *Service lookup*:

   Het opzoeken van services in de LUS is één van de sterke punten van Jini. Er zal niet langer geselecteerd worden aan de hand van de service naam zoals dat in CORBA of RMI het geval is. De selectie van de service gebeurt aan de hand van een match making mechanisme. Dit mechanisme matcht een meegegeven template met de informatie die na registratie bij de LUS is opgeslagen. De template bestaat uit de volgende onderdelen:

- *Service ID*: Deze ID wordt gebruikt wanneer een specifieke entiteit van een service opgevraagd wordt.
- *Service Types*: Dit type wordt beschreven aan de hand van een lijst van classes en/of interfaces die de servicy proxy moet uitbreiden of implementeren.
- *Attributen*: De set van attributen die moeten matchen met de attributen van de proxy zoals die bij registratie worden doorgegeven (meer info is terug te vinden bij het toppic omtrent registratie).

Zoals reeds eerder vermeld, worden attributen gemarshalled door het netwerk getransporteerd. Dit wil zeggen dat de LUS deze bitsequenties bij registratie kan opslaan en vergelijken met de bitsequenties die via de templates worden doorgestuurd. Het is bijgevolg logisch dat attributen matchen wanneer hun binaire sequenties identiek zijn.

De lookup procedure vergelijkt elk onderdeel van de template met de informatie van elke service die beschikbaar is op de LUS. Dit resulteert in een verzameling services die voldoet aan de vereisten van de client. Men kan dus stellen dat de lus een soort marriage broker is. Dit design pattern wordt de "*Broker as Matchmaker*" genoemd [99].

Dit principe heeft tal van voordelen en kan bijdragen tot een betere performantie en robuustheid. Het is immers mogelijk om dezelfde service door verschillende providers te laten aanbieden. Sterker nog, verschillende services kunnen dezelfde dienst aanbieden (=zelfde interface hebben) maar hem op een verschillende manier uitvoeren. De aanwezigheid van verschillende services met eenzelfde interface brengt met zich mee dat het falen van één van de services niet resulteert in het falen van het gehele systeem. Zijn taak zal overgenomen worden door een andere service met dezelfde interface. Op dezelfde manier kan een eventueel scalability probleem opgelost worden. Het is immers mogelijk de load van één service te spreiden over verschillende services met eenzelfde interface.

Tot slot is het wel belangrijk op te merken dat dit lookup mechanisme één grote beperking heeft. Het is immers niet mogelijk om gedeeltelijke matches te maken. Op het eerste gezicht lijkt de impact hiervan beperkt maar verdere analyse leert ons echter dat de gevolgen groter zijn dan verwacht. Deze beperking impliceert immers dat het niet mogelijk is de beste benaderende service te selecteren en te gebruiken. Het gevolg hiervan is snel duidelijk. Stel: we willen een kleurenprinter gebruiken binnen de *Jini community*[2]. Er zijn echter verschillende kleurenprinters verspreid wereldwijd. Het is voor iedere gebruiker evident dat het printen uitgevoerd zou worden op de dichtst bijzijnde printer. Dit is echter onmogelijk te implementeren binnen Jini. Projecten zoals de Location service hebben hieraan het hoofd willen bieden maar zijn steeds

[2]Een Jini Community is de verzameling participerende entiteiten die met een geconnecteerde graph met elkaar verbonden kunnen worden. Hieronder verstaan we zowel de resources, de service providers, de service gebruikers en de verschillende lookup services.

gestrand op een beperkte realisatie van hun doelstellingen. De enige oplossing tot dusver is op application level via user gespecificeerde algoritmiek een juiste keuze te maken.

3. *Notification*:

   Clients kunnen ervoor zorgen dat ze op de hoogte worden gehouden van nieuwe services die ter beschikking worden gesteld, van services die niet langer bestaan en van services waarbij veranderingen zijn doorgevoerd. Het principe waarmee Jini dit verwezenlijkt, wordt Remote Events genoemd en wordt later besproken in sectie B.3.5.

4. *ID generatie*:

   Om een onderscheid te maken tussen verschillende instanties van services gebruikt Jini een Service ID. Het is de verantwoordelijkheid van de lookupservice om bij registratie na te gaan of de service al over zo een ID beschikt en bij afwezigheid hiervan zo een unieke ID te genereren. De ID is 128 bit en bestaat uit 5 velden die tijd en host adres afhankelijk gegenereerd worden. Bij het herstarten van een service zal hij zich met dezelfde ID bij de lookup registreren zodat er geen duplicaten kunnen ontstaan en zodat de gebruiker, via ID matchmaking, kan terugvallen op een welbepaalde geprefereerde service.

5. *Bootstrap function*:

   De Jini lookup service zal ook een belangrijke rol spelen in het discovery proces. Aan de hand van het Multicast Announcement Protocol zal hij nieuwe services binnen de multicast radius attent maken op zijn bestaan. Deze services kunnen zich dan via de Proxy van deze lookup service registreren en deelnemen aan het Jini netwerk. Meer diepgang omtrent dit Multicast Announcement Protocol is terug te vinden in sectie B.3.2.

6. *Inter federation link*:

   Aangezien de Jini lookup service zelf ook een service is, heeft hij de mogelijkheid zichzelf te registreren bij een andere lookup service. Dit stelt een lookup service in staat om zich via unicast discovery te registreren bij een community waarvan de lookup service buiten de multicast radius valt. Services van deze community hebben dan toegang tot de verschillende services van deze geregistreerde lookup service. Dit principe, waarin eerst een lookup service wordt opgevraagd bij een andere lookupservice, wordt hierarchical lookup genoemd.

7. *Integratie van discovery schema's*:

   Naast de links tussen verschillende Jini federations is het ook mogelijk een connectie te verzorgen tussen verschillende discovery architecturen. Dit gaat in beide richtingen. Enerzijds kan een andere archtitectuur zich registreren als een lookup service in een bestaande Jini lookup service. Anderzijds is het ook mogelijk een Jini lookup service te registreren in een lookup service die werkt volgens een andere discovery architectuur.

8. *Netwerkonderhoud*:

   Het contacteren van niet-bestaande services of het sturen van messages naar niet-bestaande gebruikers brengt belangrijke netwerkoverhead met zich mee. Aangezien deze twee fenomenen rechtstreeks in verband staan met de lookup service kan men stellen dat de optimale werking van het netwerk vooral bepaald wordt door deze lookup service. Om deze problematiek aan te pakken, wordt er gebruik gemaakt van leases. Het concept lease komt in Jini overeen met de verwachte bestaansduur van een entiteit. Wordt een lease verlengd dan gaat de lookup service ervan uit dat deze entiteit gedurende de tijd van de lease zal blijven bestaan, indien hij niet wordt verlengd, beschouwt de lookup service deze entiteit als niet langer bestaande. Op deze manier kunnen resources niet langer dan de lease tijd door een verdwenen client bezet worden of kunnen serviceproxies niet langer dan de lease tijd zonder de serviceondersteuning aangeboden worden.

B.3.2 Discovery protocols

Zoals sectie B.3.1 reeds beschreef,vormt de lookup van services de basis van heel het Jini concept. Het spreekt dan ook voor zich dat het hele systeem staat of valt met het vinden van een lookupservice. Het vinden (discoveren) van een lookup service gebeurt aan de hand van drie protocols: Het Multicast Request Protocol, het Multicast Announcement Protocol en het Unicast Discovery Protocol. Het vervolg van deze sectie bespreekt de werking en het toepassingsgebied van elk van deze protocols.

1. *Multicast Request Protocol*

   Indien een service gestart wordt en niet beschikt over een adres van een lookup service is het essentieel om zo snel mogelijk contact te maken met een lookup service ergens in het netwerk. Om dit te bewerkstelligen, wordt er gebruik gemaakt van het Multicast Discovery Protocol. Er wordt dan een Multicast message verzonden via een UDP gebaseerde socket naar het vaste multicast adres 224.0.1.85 op poort 4160. Alle lookupservices zullen luisteren op dit adres en de messages ontvangen als ze zich binnen de multicast radius bevinden. De boodschap die verzonden wordt, bestaat uit de volgende velden:

   - *Protocol versie*: Deze integer bevat de versie van het Multicast Request Protocol
   - *Poort*: Dit veld bevat de poortnummer waarop de discoverer(diegene die de multicast messages verstruurd heeft) zal luisteren naar een antwoord van de lookup service (een *discovery responce message*).
   - *Groep lengte*: Elke lookupservice behoort tot één of meerdere groepen. Elk van deze groepen bepaalt het type van services die geregistreerd zijn op de lookupservices die tot deze groep behoren. De services kunnen bij het discoveren van lookupservices ter registratie van de service verkiezen enkel de lookup services te ontdekken die tot de groepen behoren

waarbinnen de specificatie van de service valt. Ook de clients kunnen bij het ontdekken verkiezen enkel deze lookupservices op te vragen waarop services geregistreerd zijn die binnen bepaalde groepen vallen. Om dit te bewerkstelligen wordt een lijst van groepsnamen meegegeven waarin de discoverer(de service of client die de multicast message verstuurd) interesse vertoont. Dit veld geeft het aantal groepsnamen weer die terug te vinden zijn in het groepen veld.

- *Groepen*: Dit veld bevat de set van groepsnamen waarin de discoverer geïntereseerd is.
- *Heard Lengte*: Dit veld bevat het aantal Service ID's die terug te vinden zijn in het Heard veld.
- *Heard*: Dit veld bevat een set van service ID's van de verschillende lookup services waarvan hij reeds een antwoord heeft gekregen. Deze lookup services zullen dan niet reageren zodat overbodig netwerk gebruik vermeden wordt.

Wanneer een lookup service de hierboven beschreven boodschap ontvangt en zijn ID niet voorkomt in het Heard veld zal hij antwoorden met een discovery response message. Deze message wordt verder in deze sectie beschreven bij het topic omtrent het Unicast Discovery Protocol.

Bij het gebruik van dit protocol is het belangrijk te weten dat deze UDP multicast messages in lengte begrensd zijn op 512 bytes. Dit omdat, volgens IP standaarden, messages van deze lengte steeds volledig of niet aankomen. Dit draagt bij tot de eenvoud van Jini en vermijdt het toevoegen van een extra abstractielaag die zou moeten instaan voor de intacte en geordende ontvangst van UDP berichten. Deze beperking heeft echter wel enkele gevolgen. Wanneer het *Groepen* veld te groot wordt, is er voor gekozen de discoverer te verplichten zijn discovery request op te splitsen in verschillende requests. Dit houd in dat er een lookup request zal gemulticast worden voor elke subset van groepen die binnen één instantie van het groepvenveld gedefineerd kan worden. Wanneer het Heard veld te groot dreigt te worden is het toegelaten het aantal ID's binnen dit veld te beperken tot de maximale hoeveelheid die het veld kan omvatten. Met de overige ID's moet er geen rekening gehouden worden aangezien dit veld niet bijdraagt tot de correcte werking van het protocol en enkel dient ter optimalisatie van het protocol.

2. *Multicast Announcement Protocol*

   Zoals reeds eerder vermeld, zal ook de lookup service proberen het netwerk attent te maken op zijn aanwezigheid. Dit alles gebeurt aan de hand van het Multicast Announcement Protocol. Vanaf de start van de lookupservice zal hij op regelmatige tijdstippen (default om de 2 minuten maar is aanpasbaar via de `net.jini.discovery.announce system property`) een multicast announcement message sturen. Op deze manier zorgt de lookup service ervoor dat er op een dynamische manier Jini communities worden gevormd. Elke

service provider op zich kan dan beslissen of hij al dan niet deelneemt aan een bepaalde community. Ook dit protocol werkt, zoals zijn naam reeds doet vermoeden, aan de hand van multicast boodschappen. Deze boodschappen, die ook een maximale grootte hebben van 512 bytes, worden verzonden via UDP gebaseerde sockets naar het vaste multicast adres 224.0.1.84 op poort 4160. De boodschap ziet er echter iets anders uit dan de Multicast Request boodschap en heeft volgende velden:

- *Protocol versie*: Deze integer bevat de versie van het Multicast Request Protocol.
- *Host*: Deze string bevat de naam van de host waarop de lookup service draait.
- *Poort*: Dit veld bevat de poortnummer waarop de lookup service luistert naar unicast discovery requests (Hierover meer in het volgende topic).
- *Service ID*: Deze ID wordt gebruikt door de ontvangende services om na te gaan of een lookup service reeds gekend is. Indien dit het geval is, wordt de boodschap genegeerd, is dit niet zo dan kan hij de gewenste acties ondernemen.
- *Groep Lengte*: Dit veld bevat het aantal groepen die terug te vinden zijn in het groepen veld.
- *Groepen*: Dit veld bevat de verschillende groepen die ondersteund worden door de lookupservice. Een service kan dan beslissen of hij binnen zo een groep valt en zich daar dan al dan niet registreert.

Wanneer een discoverer een bovenstaande message ontvangt, onderzoekt hij of hij de lookupservice reeds kent aan de hand van de Service ID. Indien dit niet het geval is, kijkt hij na of er bij de groepen een groep zit waarin hij geïnteresseerd is. Indien dit het geval is, kan hij zich registreren bij die lookupservice onder die bepaalde groep aan de hand van het Unicast Discovery Protocol (zie volgende topic).

3. *Unicast Discovery Protocol*

   Het Unicast Discovery Protocol wordt, zoals reeds eerder vermeld, gebruikt door de discoverer om een proxy op te vragen van een lookup service. In dit protocol wordt het TCP protocol gebruikt in plaats van het UDP protocol. Dit maakt het mogelijk voor discoverers om ook lookup services, die buiten de multicast radius liggen, te contacteren.

   Het protocol kan dus gebruikt worden wanneer de discoverer beschikt over het netwerkadres en de poortnummer van de lookup service die hij wil contacteren. Indien dit het geval is, wordt er een Tcp unicast discovery request gestuurd naar de lookup service. Deze request bestaat uit één enkel veld namelijk een integer die de protocol versie beschrijft. Wanneer een lookupservice een dergelijk bericht ontvangt, zal hij hierop antwoorden met een unicast response message. Dit bericht bestaat uit een marshalled object dat de lookupservice

proxy omvat. Indien dit bericht een antwoord is op een unicast discovery message wordt het steeds terug gestuurd op poort 4160. Is het echter een antwoord op een multicast request message dan zal er geantwoord worden via de poortnummer die vermeld staat in de multicast request message (zie hierboven).

Wanneer de discoverer de message ontvangt, zal hij het gemarshalde object unmarshallen en beschikt hij over een lookup service proxy die hij kan gebruiken voor de registratie van zijn service of voor het opvragen van andere services.

B.3.3 Communcatie

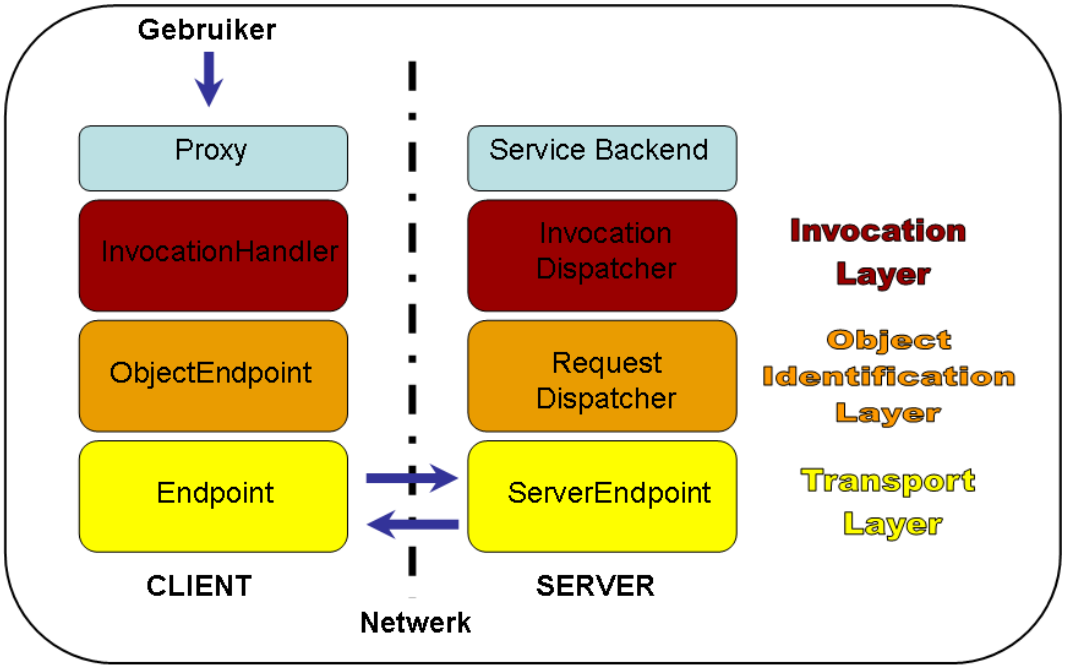
In dit hoofdstuk wordt de communicatie tussen de verschillende services in Jini besproken. Op het bovenste niveau wordt er gekeken naar de verschillende design opties die gevolgd kunnen worden om de communicatie tussen proxy en service backend te verzorgen. Na deze uiteenzetting volgt dan de concrete implementatie van deze concepten. Maar alvorens deze design methodiek besproken kan worden, is het belangrijk afdoende kennis te hebben van het communicatie protocol (JERI: Jini Extensible Remote Invocation). Dit zorgt onafhankelijk van deze design mogelijkheden voor de communicatie tussen proxy en backend.

JERI

De gemeenschappelijke Java roots en de overeenstemmende concepten van RMI[41] [100] en Jini maken dat RMI een voor de hand liggende keuze zou zijn om de proxy backend communicatie te verzorgen. De oorspronkelijke implementatie van RMI gebruikte JRMP, wat heel dicht bij de Java programmeertaal staat. De combinatie van toenemende populariteit en van een gebrek aan interoperabiliteit zorgde ervoor dat er naast de standaard RMI versies tal van andere versies verschenen zoals RMI over HTTP, RMI over IIOP (the CORBA transport protocol)[101][81], RMI over SSL/TLS[12], RMI over Firewire (IEEE 1394)[86], ... Het ontstaan van deze nieuwe versies bracht enerzijds nieuwe mogelijkheden met zich mee maar zorgde er anderzijds voor dat al deze protocols naast elkaar bestonden. De ultieme oplossing zou zijn dat al deze communicatiemethoden geabstraheerd zouden worden in één mechanisme dat het meest geschikte protocol zou kiezen. Dit gebeurde met de ontwikkeling van JERI (Jini Extensible Remote Invocation). Deze nieuwe RMI implementatie had eenzelfde semantiek als JRMP maar een verschillende protocol stack. De bedoeling was deze protocol stack publiek te maken om zo de flexibiliteit te vergroten.

Essentieel bestaat JERI, zoals terug te vinden is in figuur B.4, uit het volgende drie lagen model dat de communicatie tussen de proxy en de backend verzorgt:

1. *Transport Layer*: De transport layer verzorgt de abstractielaag voor request/-respons gebaseerde communicatie. Deze abstractielaag is zeer pluggable. Hiermee wordt bedoeld dat iedereen gebruik makend van de *net.jini.jeri.connection* package een implementatie van deze abstractielaag kan construeren. Zo een implementatie vereist enkel dat de interface geïmplementeerd is en dat de binaire



Figuur B.4: Het Jeri Lagen Model

byte sequenties (requests en responses) van client naar server en omgekeerd getransporteerd worden. Om dit te verwezenlijken, moeten er twee belangrijke klassen geïmplementeerd worden namelijk:

- *De Endpoint*: Dit is één remote endpoint dat gebruikt wordt om requests te versturen.
- *De ServerEndpoint* Dit zijn één of meerdere endpoints die op de servermachine luisteren of er requests binnen komen.

2. *Object Identification Layer*: De Object identificatation layer verzorgt de functionaliteit van remote objecten die los staat van de implementatie. Hiermee wordt bedoeld dat deze laag de communicatie zal verzorgen van alles wat los staat van de oproep van object methodes. Bijvoorbeeld de Garbage collection.

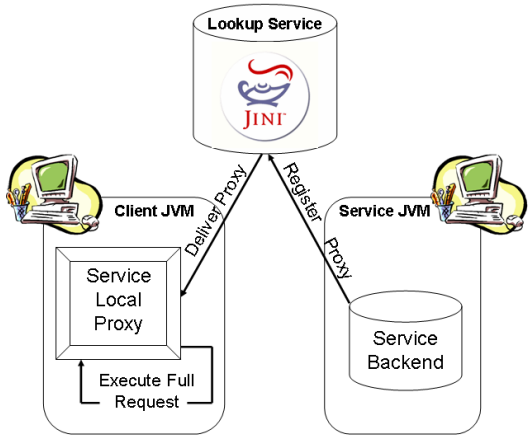
3. *Invocation Layer*: De invocation layer behandelt de concrete oproep van methodes en hetgeen hiermee samenhangt (argumenten, returnvalues, exceptions, marshalling/unmarshalling,...). De service proxy beschikt hiervoor over een invocation handler die ervoor zorgt dat op de server de Request Dispatcher de remote calls forward naar de Invocation Dispatcher.

Proxy Design

Naast de low level JERI, die de communicatie tussen proxy en service verzorgt, bestaan er in Jini verschillende design mogelijkheden die de samenwerking tussen de proxy en de service backend verzorgen. Er kunnen hierin drie design mogelijkheden onderscheiden worden:

1. *Local Proxy (ook 'Fat' Proxy genoemd)*:

   Het local proxy design zorgt ervoor dat, bij registratie van een service, de proxy de volledige codebase bevat. Dit wil zeggen dat bij het gebruik van de service de proxy wordt doorgestuurd naar de gebruiker en dat al de functionaliteit van deze service in de proxy vervat zit. Elke methode zal uitgevoerd worden op de machine van de service gebruiker. Figuur B.5 verduidelijkt dit.



Figuur B.5: Local proxy design

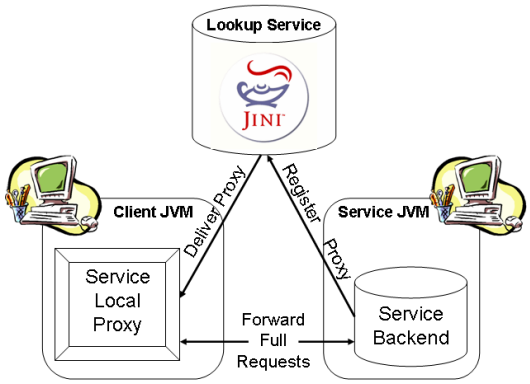
2. *Forwarding Proxy (ook 'Thin' Proxy genoemd)*:

   Het forwarding proxy design koppelt de oproep van een methode van de interface direct aan de oproep van een remote methode van de service backend. Zoals reeds eerder besproken werd, zal dit geïmplementeerd worden via het JERI design zodat de onderliggende communicatie protocols weg geabstraheerd worden voor de designer en voor de gebruiker van de service. Figuur B.6 verduidelijkt dit.

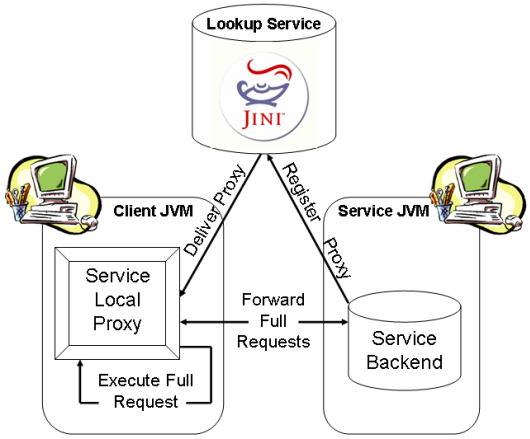
3. *Smart Proxy*:

   Het smart proxy design is, zoals zijn naam al laat vermoeden, slimmer dan het Forwarding en Local Proxy design. In dit design worden beide methodieken gecombineerd. De designer van de Proxy wordt in de mogelijkheid gesteld een proxy te maken die een gedeelte van zijn methodes in de JVM van de client zal uitvoeren en een ander gedeelte zal forwarden naar de service backend. Figuur B.7 verduidelijkt dit

De belangrijkste eigenschap van deze manier van werken is dat het proxy design volledig los staat van de gebruiker. Dynamic classloading en het gebruik van interfaces zorgen ervoor dat de functionaliteit van de service losgekoppeld is van het gebruikte proxy design en het gebruikte netwerk protocol. Dit heeft enkele belangrijke gevolgen:



Figuur B.6: Forwarding proxy design



Figuur B.7: Smart proxy design

- Zoals reeds besproken, is de grootste uitdaging bij het ontwikkelen van een gedistribueerd systeem te voldoen aan de 8 fallacies van distributed computing. Één van de grootste problemen is hier dat de developers de remoteness van objecten volledig willen abstraheren. Jini zorgt er ondanks deze abstractielaag voor dat gebruikers alsnog eventuele problemen in verband met deze remoteness moeten aanpakken door hen een Java interface op te leggen die bij eventuele remote problemen de nodige exceptions teruggeeft.

- De dynamic classloading maakt het mogelijk voor een gebruiker de polymorfe eigenschappen van de Java programmeertaal te behouden op het Jini netwerk. Op deze manier is het immers mogenlijk om: `Remote` interfaces uit te breiden van andere interfaces, subklassen van een parameter type mee te geven aan een methode, klassen een aantal interfaces te laten implementeren en de verantwoordelijkheden hiervan op zich te nemen.

Implementatie

Tot slot is het belangrijk om te weten dat, na de implementatie van dergelijke objecten in de praktijk, nog enkele stappen moeten ondernomen worden alvorens deze objecten remote aanspreekbaar zijn. Dit concept wordt *Exporting* genoemd. Om dit beter te begrijpen wordt er, in wat nu volgt, eerst kort besproken hoe dit principe standaard in Java (RMI) werkte alvorens er wordt overgegaan op het Jini principe.

In Java RMI werkt het systeem in de volgende stappen:

- Implementeer een klasse die de gewenste te exporteren functionaliteit bevat, die een uitbreiding is van `UnicastRemoteObject` en die de `Remote` interface implementeert. bv:

```
1  package be.ac.ua.comp.rmitest
2  import java.rmi.*;
3  import java.rmi.server.*;
4  public class RmiExportTest extends UnicastRemoteObject implements
       Remote{
5    public RmiExportTest(){
6    ...
7    }
8
9    public EenMethode(){
10     ...
11   }
12 }
```

- Tot Java versie 5.0 was het nodig om na de standaard compilatie van de code het `rmic` compilatie commando uit te voeren om op deze manier de creatie van de proxy's mogelijk te maken (De stub files worden gecompileerd in deze compilatie fase). vb:

```
1  javac /be/ac/ua/comp/rmitest/RmiExportTest.java
2  rmic -v1.2 be.ac.ua.com.rmitest.RmiExportTest
```

- De extensie van de `UnicastRemoteObject` klasse zal er voor zorgen dat, bij het aanmaken van een instantie van deze klasse, de proxy geregistreerd zal worden bij de VM en dat, bij aanroep van een methode van de Proxy, de methode rechtstreeks uit het originele object wordt aangeroepen.

In Jini werkt dit mechanisme enigszins anders. Vanaf Jini2.0 is er een klasse met verschillende methodes voorzien die het exporteren en unexporteren voor hun rekening nemen. Meer info omtrent deze Exporter klasse kan terug gevonden worden in de Jini2.0 API[3] Het exporteren wordt geïllustreerd aan de hand van onderstaand vrijblijvend voorbeeld. In dit voorbeeld wordt in de mainfunctie de JiniExportTest klasse die de main bevat opeenvolgens gecreëerd, geëxporteerd en tot slot voor ons programma afloopt terug geunexport.

[3]De Jini2.0 API is terug te vinden op: *http://java.sun.com/products/jini/2.0.2/doc/api/*.

```
1  package be.ac.ua.comp.jinitest
2  import java.rmi.*;
3  import net.jini.export.*;
4  import net.jini.jrmp.JrmpExporter;
5  public class JiniExportTest extends UnicastRemoteObject implements
       Remote{
6    public static void main(String[]args) throws Exception{
7      // Er wordt een instantie van een exporterimplementatie aangemaakt
8      Exporter exporter = new JrmpExporter();
9      // Een instantie van het huidige object wordt ge"exporteerd.
10     // Dit resulteert in een remote proxy object.
11     Remote proxy = exporter.export(new JiniExportTest());
12     // de JiniExportTest klasse is niet langer ge"exporteerd.
13     exporter.unexport(true);
14   }
15
16   // een methode die remote gebruikt kan worden
17   public EenMethode(){
18     ...
19   }
20 }
```

Deze methode geeft de gebruiker de volledige keuze in welk type van `Exporter` (vb: `ActivationExporter`, `BasicJeriExporter`, `IiopExporter`, `InstantiatorAccessExporter`, `JrmpExporter`, `MonitorAccessExporter`, `ProxyTrustExporter`, `SunJrmpExporter`, `SystemAccessExporter`) hij wilt gebruiken. Hij kan er zelfs voor kiezen een eigen type van exporter te gebruiken. Deze werkwijze zorgt echter wel voor de beperking dat dit type reeds at compile time moet vastliggen. Om dit probleem te verhelpen, gebruikt Jini ook hier zijn configuratie mechanisme. Dit mechanisme wordt diepgaander besproken in sectie B.3.8. Het principe geeft de mogelijkheid het type van Exporter te specificeren in een configfile. Deze configfile wordt dan in de code ingelezen en de gewenste exporter wordt aangemaakt. Het volgende voorbeeld verduidelijkt dit.

```
1  // Dit configuratiebestand geeft aan dat er een JeriExporter moet
       aangemaakt worden
2  // die via tcp zal communiceren
3
4  import net.jini.jeri.BasicILFactory;
5  import net.jini.jeri.BasicJeriExporter;
6  import net.jini.jeri.tcp.TcpServerEndpoint;
7  JeriExporterTest{
8    exporter = new BasicJeriExporter(TcpServerEndpoint.getInstance(0),
9          new BasicILFactory());
10 }
11
12
13
14 // Deze code geeft aan hoe een dergelijk configuratiebestand wordt
       gebruikt.
15
16 package be.ac.ua.comp.jinitest;
17 import java.rmi.*;
```

```
18 import net.jini.config.*;
19 import net.jini.export.*;
20 public class JiniConfigExportTest implements Remote{
21   public static void main(String[] args) throws Exception{
22     // het path van de configfile uit het eerste argument halen
23     String[] configArgumenten = new String[] {args[0]};
24     // het configuratie bestand ophalen
25     Configuration config
26       =ConfigurationProvider.getInstance(configArgumenten);
27     // gebruik de configfile om een exporter instantie aan te maken
28     Exporter exporter = (Exporter) config.getEntry( "JeriExporterTest"
         ,
29           "exporter",Exporter.class);
30     // een instantie van de huidige klasse wordt geexporteerd.
31     Remote proxy = exporter.export(new JiniCOnfigExportTest());
32
33
34     // de export wordt gestopt
35     exporter.unexport(true);
36   }
37
38   // een methode die remote gebruikt kan worden
39   public EenMethode(){
40     ...
41   }
42 }
```

Tot slot dient nog opgemerkt te worden dat elk geëxporteerd object een individuele instantie nodig heeft van een exporter. Dit brengt met zich mee dat bij het exporteren van een nieuw object er telkens ook een nieuwe instantie van een exporter implementatie moet aangemaakt worden. Tevens is het belangrijk op te merken dat de te exporteren klassen steeds rechtstreeks of onrechtstreeks de `Remote` interface moeten implementeren.

B.3.4 Leasing

De structuur van een Jini netwerk brengt met zich mee dat gebruikers, naast het aanvragen of een service gebruikt mag worden, ook specifiek moeten meedelen wanneer ze een service niet langer willen gebruiken. Dit brengt met zich mee dat er een onnatuurlijke machtsverhouding ontstaat tussen gebruiker en provider. Hierin zal, eens een resource is toegekend aan een gebruiker, een provider niet langer kunnen beslissen wanneer hij zijn diensten wil stopzetten of aan een andere gebruiker wil toekennen. Erger nog, een resource kan eeuwig ter beschikking blijven van een gebruiker zonder dat deze die ooit nog gebruikt.

Dit probleem zou eventueel opgelost kunnen worden door op constante basis het gebruikspatroon van service en client na te gaan. Deze constante monitoring van resources en gebruikers zou echter weinig schaalbaar zijn en biedt dus geen realistische oplossing. In een meer realistische oplossing zouden we de gebruikers van een resource kunnen opleggen dat deze op regelmatige tijdstippen de resources moeten overtuigen van het belang van de service voor de gebruiker. Dit concept wordt “proof of interest” genoemd. Het Jini leasing paradigma implementeert dit

concept tijdsafhankelijk. Bij het ter beschikking stellen van een resource aan een gebruiker wordt er onderhandeld over een tijdspanne waarin de gebruiker het recht heeft de resource te gebruiken. Wil een gebruiker na deze tijdspanne de service nog verder gebruiken dan is het zijn verantwoordelijkheid opnieuw voor een tijdspanne te onderhandelen (de lease vernieuwen). Gebeurt dit niet, dan wordt de resource terug ter beschikking gesteld in het Jini netwerk. Alvorens er dieper wordt ingegaan op de verschillende stappen die het lease process onderneemt is het, om een beter inzicht in de materie te verkrijgen, essentieel om op te merken dat de LUS op zich ook een service is en dat de verschillende services die zich bij de LUS registreren bij deze registratie de rol vertolken van een client bij de LUS service.

De hierboven vermelde onderhandelingen gebeuren in één ronde. De gebruiker deelt de gewenste tijd mee aan de resource waarop deze de leasetijd, die tussen 0 en de gevraagde tijdspanne ligt, teruggeeft. De gebruiker heeft geen andere keuze dan hiermee akkoord te gaan en kan enkel een andere tijdspanne aanvragen wanneer de lease bijna verlopen is en verlengd moet worden.

Bij het onderhandelen van de leasetijd moet er met een aantal zaken rekening gehouden worden:

1. Aangezien de klokken van de verschillende deelnemende machines in de meeste gevallen niet op elkaar staan afgestemd, is het onmogelijk om met absolute tijdseenheden te werken. Er is daarom ook voor relatieve tijdseenheden gekozen.

2. Wanneer een client down gaat, zal de resource in het slechtste geval terug vrijgegeven worden na de volledige lease tijd. Het is daarom belangrijk om bij het onderhandelen van de leasetijden rekening te houden met de uptime geschiedenis van een systeem. Hier moet de kans dat een systeem down gaat over een leaseperiode gereduceerd worden onder een bepaalde minimum waarde.

3. Wanneer een service provider down gaat, staat zijn services nog steeds in de LUS registers. De kans bestaat dan dat gebruikers de onbereikbare resources alsnog proberen te contacteren. Het spreekt voor zich dat deze situatie tal van problemen met zich mee brengt . Dit probleem is niet te verhelpen maar kan wel gereduceerd worden. Dit door de resources op te leggen, na registratie, hun leases te vernieuwen binnen een tijdsduur waarin de kans op falen onder een minimum waarde ligt.

4. Een laatste probleem is analoog aan het voorgaande. Wanneer een gebruiker een service gebruikt op een resource die faalt, zal hij hiervan slechts op de hoogte gesteld worden nadat de lease van de service bij de lookup service verlopen is.

We kunnen hieruit concluderen dat dit selfhealing systeem enerzijds nodig is en anderzijds voor een gezond systeem zorgt. Het brengt echter wel een toegenomen administratieve complexiteit met zich mee. Er mag vooral niet vergeten worden dat het zeer belangrijk is zorgvuldig leasetijden te kiezen die stroken met de voorspelde

uptime van zowel de gebruiker als van de provider.

Naast deze standaard implementatie is het in Jini ook mogelijk een eigen/meer geavanceerd leasing mechanisme op te zetten. De Jini package die hiertoe dient, wordt de *Landlord package* genoemd. Het principe is zoals op figuur B.8 wordt weergegeven gebaseerd op vijf basiscomponenten:

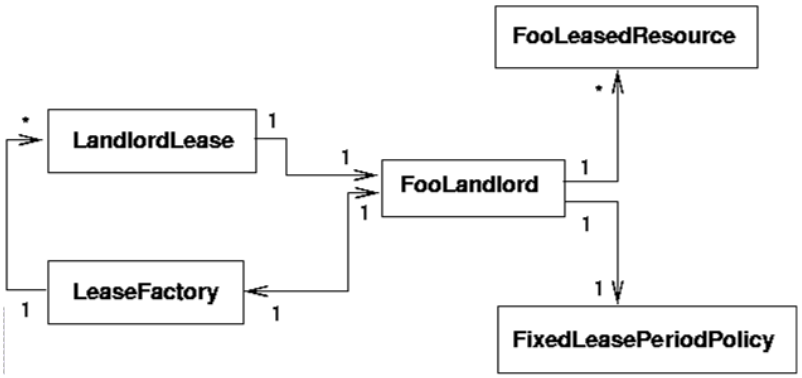
1. *De Landlord instantie* implementeert de Landlord interface en wordt op figuur B.8 FooLandlord genoemd. De Landlord is als het ware een tussenpersoon tussen diegene die een lease vragen en de resources die voor een bepaalde lease tijd ter beschikking worden gesteld.

2. *De Lease Factory* maakt op aanvraag van de Landlord de lease objecten aan.

3. *De LandlordLease* is een object dat een bepaalde gebruiker de mogelijkheid biedt om gedurende een vastgelegde periode een specifieke resource te gebruiken.

4. *De Leased Resource* op figuur B.8 FooLeasedResource genoemd, is de resource die gedurende een bepaalde periode ter beschikking van een gebruiker wordt gesteld.

5. *De LeasePeriodPolicy* bepaalt aan de hand van zijn policy regels de leasetijd die wordt toegekend aan een gebruiker. Op figuur B.8 wordt de concrete FixedLeasePeriodPolicy implementatie gebruikt. Deze zal leases toekennen rekening houdend met vaste default lease tijden en maximale lease tijden.



Figuur B.8: Het landlord leasing principe

Het principe werkt dan als volgt:

- De `landlord` instantiëert een lease factory en vraagt deze om de gewenste leases te creëren.

- Wanneer een lease een request krijgt wordt deze request in combinatie met de UUID[4] doorgegeven aan de `landlord`. Een dergelijke request, zoals bijvoorbeeld *een lease renew request*, wordt afgehandeld door het oproepen van de remote `renew();` methode uit de `landlord` klasse.

- Op de `landlord` worden geen volledig lease objecten bijgehouden maar dient een cookie met de vereiste informatie ter administratie van elke lease.

- Ook de services hebben een representatie op de `landlord`. Zo zal bijvoorbeeld de lookup service een gemarshalde vorm van elke geregistreerde service proxy bevatten.

- Naast de informatie omtrent de resource en de cookie van de lease, bevat de `landlord` tevens aanvullende essentiële informatie zoals de duration en expiration van de lease.

- De manier waarop deze informatie wordt bijgehouden, wordt gedefiniëerd in de implementatie van de `LeasedResource` interface.

- Het toekennen van de leasetijden zal niet door de `landlord` zelf gebeuren. Het is immers netter om voor deze taak een onafhankelijk object te voorzien die verantwoordelijk is voor de policy. Deze taak wordt in het Jini lease mechanisme vervuld door een implementatie van LeasePeriodPolicy zoals bijvoorbeeld de FixedLeasePeriodPolicy uit figuur B.8.

B.3.5 Events

In het hele Jini concept is het belangrijk dat de verschillende onderdelen op de hoogte blijven van belangrijke gebeurtenissen (events) binnen het Jini netwerk. Denk hierbij maar aan het verdwijnen of het registreren van een bepaalde service. Een mogelijke manier om hieraan tegemoet te komen, is een polling mechanisme. Hierbij zouden de verschillende onderdelen die informatie nodig hebben deze informatie volgens welbepaalde tijdsintervallen kunnen opvragen aan de betrokken componenten. Deze manier van werken is echter niet erg economisch en brengt de schaalbaarheid in gevaar. Hierbij komt nog dat de implementatie van zo een systeem niet binnen de structuur van Jini past. Om deze problemen te verhelpen, werd er in Jini geopteerd voor een asynchroon event notification mechanisme ook wel *publish subscribe* mechanisme genoemd. In de concrete Jini implementatie zal de propagatie van events door het netwerk steeds gebeuren aan de hand van messages. Het is om deze reden belangrijk op te merken dat wanneer de term message gebruikt wordt in deze sectie er steeds zal gedoeld worden op een Event message.

In wat nu volgt, wordt het Java event model besproken, vervolgens worden de eventuele problemen blootgelegd die ontstaan wanneer dit model ook remote zou toegepast worden. Tot slot wordt het Jini event model behandeld.

Het Java event model werkt als volgt:

[4]De *net.jini.id.Uuid* klasse implementeert een 128-bit grote uique identifier die bijvoorbeeld gebruikt wordt om verschillende instanties van een zelfde service te onderscheiden.

1. De event consumer (diegene die interesse heeft in bepaalde events), ook wel subscriber genoemd, registreert de eventtypes waarin hij interesse heeft bij de event producer (de component waarin eventuele gebeurtenissen plaats vinden) ook wel publisher genoemd. Deze registratie gaat gepaard met het doorgeven van een event listener object.

2. De event producer onderhoudt de lijst van interessegebieden van de verschillende consumers samen met de event listener objecten.

3. De event producer monitort het systeem en checkt of de gebeurtenissen die optreden binnen het interessegebied van bepaalde consumers liggen.

4. Wanneer zo een event optreedt, overloopt de producer de lijst van registraties en brengt hij de verschillende consumers die interesse hadden in de huidige event sequentieel op de hoogte. Dit door sequentieel de `notify()` methode van de verschillende listener objecten aan te roepen. Deze methode aanvaardt een event object als parameter. Dit object zal enerzijds de data omvatten die de huidige gebeurtenis omschrijft, anderzijds bevat het ook een referentie naar de event producer zodat de consumer eenvoudig de oorsprong van de event kan bepalen.

Wanneer dit systeem gemapt wordt op een Jini netwerk zal er niet langer gewerkt worden met enkel local events maar moet er ook gebruik gemaakt worden van remote events. De afhandeling van deze remote events brengt tal van verschillen ten opzichte van de local events met zich mee:

- *De aflevering van events is niet gegarandeerd*:

  Een essentieel verschil tussen lokale en remote events is de onzekerheid omtrent de aflevering van events. Het is immers mogelijk dat een gedeelte van het netwerk faalt, of erger nog, dat de event consumer zelf faalt. In dit geval zullen de events nooit afgeleverd worden. In het lokale geval is het ook mogelijk dat de event consumer faalt maar dit gaat in de meeste gevallen samen met het totale falen van het systeem. Dit onderscheid tussen gedeeltelijk falen bij remote events en totaal falen bij lokale events is een factor waar terdege rekening mee gehouden moet worden.

- *De aflevering van events in volgorde is niet gegarandeerd*:

  Het op volgorde afleveren van events of *totally ordered message delivery* in een lokale opstelling is niet gegarandeerd maar kan wel ondersteund worden zonder een grote extra computationele kost. In het remote geval ligt dit anders. Het afleveren in volgorde van de messages hangt af van het protocol. De computationele kost van protocols die dit ondersteunen, ligt in de meeste gevallen veel hoger. Het is dus belangrijk bij de opbouw van software na te gaan voor welke events de volgorde van aflevering essentieel is en voor welke niet en gebruik makend van deze info een geschikt event protocol te kiezen. Een beperking in deze ondersteuning kan immers een aanzienlijke performantiewinst opleveren.

- *Afleveringstijd vs. afhandelingstijd*:

  Het afleveren van events in het lokale geval zorgt voor weinig overhead. De latency[5] zal er echter in het remote geval voor zorgen dat de tijd nodig voor het afleveren van een event zal oplopen. Deze tijdskost maakt het noodzakelijk na te denken over het belang en de frequentie van bepaalde events. Er kan zelfs gekozen worden om events in batches door te sturen om zo de netwerk overhead te beperken.

Het zal hier dus zondermeer duidelijk zijn dat developers zich een beeld moeten vormen van het belang van de vorige drie punten en naar dit beeld moeten handelen. Ze moeten een afweging maken tussen performantie en betrouwbaarheid, nadenken over een evenwicht waarin falende events geen weerslag hebben op de programmatuur zonder dat een performantie eis in het gedrang komt.

Jini breidt het Java event model uit naar remote events via het RMI paradigma[100] [41], meer bepaald de JERI implementatie. Elk van de remote events wordt uitgebreid van de `net.jini.core.event.RemoteEvent` klasse en al de event listeners worden uitgebreid van de `net.jini.core.event.RemoteEventListener` interface. De remote event klasse bevat de volgende attributen:

- *Event ID*: De ID wordt gebruikt om een onderscheid te kunnen maken tussen verschillende soorten events. In java werd voor elk type event een nieuw object type aangemaakt. In Jini is dit, zoals verder beschreven wordt, om flexibiliteitsredenen niet het geval. Differentiatie tussen events is echter wel essentieel en wordt ondersteund door deze ID.

- *Sequence number*: Dit volgnummer nummer wordt gebruikt om te bepalen in welke volgorde de events gegenereerd zijn.

- *Source*: Bevat een verwijzing naar het object dat het event gegenereerd heeft.

- *Handback object*: Dit object wordt bij de registratie van een eventlistener aan de producer doorgegeven. Dit object bevat attributen die de context waarin de listener geregistreerd is beschrijven. Deze attributen kunnen bij de afhandeling van een event gebruikt worden om dit event in zijn beoogde context te plaatsen.

Zoals in het punt hierboven aangehaald, is er dus een essentieel verschil tussen de Java API's en deze van Jini. In Java heeft elk soort event een ander type en heeft elk van deze types een specifieke listener interface die gebaseerd is op het type event. Binnen Jini zal er slechts één event type zijn en één soort listener interface die deze events zal ontvangen. Het onderscheid wordt hier dan gemaakt door de event ID. Deze manier van werken zorgt ervoor dat een algemene afwikkeling van de events mogelijk gemaakt wordt. De kracht hiervan zit in de vrijheid die een receiver krijgt bij de afwikkeling van de events. Elke applicatie op zich heeft immers andere vereisten en wil het eventmodel benaderen volgens zijn doelstellingen. Een goed voorbeeld hiervan zijn de zogenaamde mailboxes. Sommige services verkiezen

[5]Latency is de tijd nodig om de event data over het netwerk te versturen.

slechts een beperkte tijd actief te zijn. Dit maakt dat ze in een inactieve periode niet in staat zijn messages af te handelen. De oplossing hiervoor is een message handler die berichten bij ontvangst in een wachtlijst plaatst (de mailbox). Bij het ontwaken van de service kunnen deze messages dan sequentieel behandeld worden. Een concrete implementatie hiervan is de mercury service van SUN.

Een feature die het geheel nog krachtiger maakt, is de mogelijkheid om third party handlers te gebruiken. De message handlers van een service zullen zich dan op een andere machine bevinden zodat de afhandeling van de messages geen extra resources van de service vraagt. Dit is enerzijds belangrijk voor services die geen extra resources willen of kunnen afstaan. Anderzijds kan zo een third party handler ook gebruikt worden voor services die zich in een omgeving bevinden die geen RMI ondersteunt. De events worden dan naar de third party handler gestuurd die de messages dan via een ondersteund communicatiemechanisme doorstuurt naar de betrokken service.

De algemeenheid van het messaging mechanisme en de uniformiteit van de listener interface zorgen er tevens voor dat verschillende event handlers in serie kunnen geschakeld worden, de zogenaamde event handling pipeline. Het voordeel hiervan is dat de inkomende messages in elke event handler gefilterd kunnen worden en enkel de interessante events over blijven.

Tot slot is het belangrijk op te merken dat ook het messaging mechanisme onderworpen is aan leases. De message generator stelt een methode ter beschikking van de services die het mogelijk maakt zich te registreren voor event notificatie. Hoewel Jini geen syntactische signature voor deze methode oplegt, ontvangt deze methode in de meeste gevallen een listener object en stuurt ze een EventRegistration object terug. Dit object bevat een lease die bepaalt gedurende welke periode een producer events zal sturen naar deze consumer. Wanneer de consumer events wil blijven ontvangen, moet deze lease vernieuwd worden voor de lease verstreken is (zie sectie B.3.4). Dit systeem zorgt ervoor dat het versturen van events naar onbestaande of onbereikbare services geminimaliseerd wordt.

B.3.6 Transacties

Om de mogelijkheid te bieden partial failure tegen te gaan, ondersteunt het Jini framework transacties. De werking van deze transacties is gebaseerd op het 2PC (two phase commit) protocol. Dit protocol wordt reeds sinds 1980 gebruikt om automatisch de commit en rollback uitvoeringsstappen binnen een transactie te controlleren en te monitoren. Het 2PC mechanisme stond hier in voor de simultane updates, op verschillende databases, binnen een gedistribueerd systeem. Aan de hand van synchronized locking van de verschillende onderdelen van de transacties garandeert dit mechanisme het behoud van integrity en accuracy. Enkele bekende toepassingen zijn hotel/vlucht reservatie systemen, beurs transactie systemen, bankieringssystemen, credit card systemen, ... Meer info omtrent de database achtregrond van dit protocol is terug te vinden in volgende referenties: [92], [71]. Om de werking van de Jini transacties beter te begrijpen, volgt eerst een uiteenzetting van het 2PC protocol alvorens over te gaan tot een uiteenzetting van de Jini transacties.

Two phase commit protocol

In het 2PC protocol worden de *transaction participants* onder begeleiding van de *transaction manager* door de volgende stappen begeleid.

1. De participants lichten de transaction manager in over hun deelname aan de transactie.
2. De transaction manager geeft al de participants (De verschillende services die samen een atomaire actie willen ondernemen) de opdracht om naar de *prepare state* over te gaan.
3. De participants maken deze overgang gebruik makend van instructies die *rollback* ondersteunen.
4. Wanneer één van de participants niet in de mogelijkheid is om tot de prepare state te komen, meldt hij dit aan de transaction manager aan de hand van een *abord* message.
5. Wanneer één of meerdere participants de prepare state niet bereiken, geeft de transaction manager alle participants de opdracht de instructies uit stap 3 ongedaan te maken aan de hand van hun *rollback* functies. Vervolgens wordt de transactie beëindigd.
6. In het geval elke participant de prepare state succesvol heeft bereikt, geeft de transaction manager elke participant de *commit* opdracht. Elke participant voert dan de gevolgen van stap 3 door en rondt de transactie verder af. Het spreekt voor zich dat ervan uit wordt gegaan dat deze commit opdracht steeds doorgevoerd kan worden onafhankelijk van het eventuele falen van een component.
7. Na de succesvolle commits van alle participants worden als laatste stap de gebruikte resources terug vrijgegeven en wordt de tijdelijk 2PC gerelateerde data verwijderd.

Two phase commit in Jini

In tegenstelling tot de tot op implementatie niveau verfijnd uitgewerkte transaction protocols in database systemen, gebruikt Jini een heel andere aanpak. De verschillende programmatuurvormen die op een gedistribueerd systeem draaien, maken het immers moeilijk of zelfs onmogelijk een statische implementatie op te zetten die de transacties in elk van deze vormen ondersteunt. Om hieraan tegemoet te komen, werkt Jini met een interface die de werking van het 2PC protocol beschrijft. De transaction manager zal dan volledig vertrouwen op de implementatie die de gebruiker aan deze interface heeft gekoppeld. Dit principe heeft twee grote voordelen:

- *Implementatie vrijheid*: De verschillende services kunnen de transacties implementeren volgens hun eigen noden. Concreet wil dit zeggen dat software waarbij de persistentie primeert op de performantie (bijvoorbeeld bij databases), zeer complexe en computationeel intensieve algoritmes kunnen gebruiken. Developers van software waarbij de performantie primeert kunnen dan weer kiezen voor een snelle eenvoudige afwikkeling van het transactie protocol.

- *Interface uniformiteit*: Zoals reeds eerder vermeld, kunnen de verschillende services een verschillende implementatie gebruiken voor de afhandeling van het transactie protocol. Het gebruiken van dezelfde interface zorgt voor een uniform uitzicht van de participants naar de transaction manager toe. Deze is dan ook in staat elk van de services te laten participeren (participant) bij de afhandeling van een transactie. Dit heeft als gevolg dat het toevoegen van meer transaction managers de mogelijkheid biedt om de participants te verdelen over de verschillende managers om zo de schaalbaarheid te vergroten.

Het principe heeft ook één niet ter verwaarlozen nadeel:

- *Implementatie vrijheid*: Zoals reeds besproken is de implementatievrijdheid op het vlak van keuze van algoritmische complexiteit een groot voordeel. De vrijheid veroorzaakt echter ook een groot nadeel. Aangezien de transaction manager enkel communiceert met de participants via de standaard interface heeft hij geen controle over de eigenlijke implementatie van de participant. Een beperkte ondersteuning van de *roll-back* of een onvolledige ondersteuning van een feilloze commit kunnen voor ernstige problemen zorgen.

Jini's two phase commit in de praktijk

:

In de praktijk werkt het principe als volgt. De transaction manager is, zoals reeds aangehaald, de verantwoordelijke voor de coördinatie van een transaction. Bij het opzetten van een transaction wordt een nieuw transaction object opgevraagd bij deze transaction manager. De ID van dit object wordt samen met de transaction manager interface doorgegeven aan al de deelnemers van de gewenste transaction. Deze deelnemers bestaan doorgaans uit method calls die de gewenste *rollback* functionaliteit ondersteunen. Services die transaction management ondersteunen, beschikken bijgevolg over methodes waaraan transaction ID's en transaction management proxy kunnen meegegeven worden. De aanroep van deze methodes door een gebruiker zal bijgevolg zorgen voor een groepering van verschillende deelnemers binnen eenzelfde transactie. De deelnemers zullen gevolg geven aan de methode oproep door de `join()` methode via de transaction manager proxy aan te roepen. De ID die bij deze operatie wordt meegegeven, zorgt voor de groepering van de deelnemers per transaction. Aan de hand van de `TransactionParticipant` interface, die eveneens door de deelnemers aan de transaction manager wordt meegegeven, handelt deze laatste, zoals reeds eerder beschreven, opeenvolgens de prepare, commit of abord methodes af.

B.3.7 Security

Aangezien security een belangrijk onderdeel is van het Jini framework doch de relevantie ervan ten aanzien van deze thesis slechts beperkt is, wordt ervoor geopteerd om slechts een beknopte basisuiteenzetting te geven van de verschillende technieken die hierin thuishoren.

Het basisprincipe van de Jini security is reeds sinds de aanvang van het project gebaseerd op het standaard Java security principe. Dit principe behandelt de permissies toegekend aan gedownloade code. De *SecurityManager* fungeert als centrale

controle eenheid, die op basis van de aard van de operatie en de context waarin deze gebeurt, beslist of de operatie al dan niet toegelaten is.

De effecten van dit principe zijn het duidelijkst zichtbaar bij de RMI classloading. Het uitvoeren van RMI calls gaat vaak gepaard met het downoaden van gerelateerde code. Alvorens de RMI Classloader mag overgaan tot het downloaden van code moet hij een veiligheidsgarantie krijgen van de `RMISecurityManager`. De toelatingsvoorwaarden zijn gebaseerd op de informatie die terug te vinden is in de security policy. Bij afwezigheid van deze policy zal het niet toegelaten zijn ergens code te downloaden, in het andere geval bepalen de policy regels welke bronnen al dan niet te vertrouwen zijn. Verder is het belangrijk op te merken dat de policies in de meeste gevallen terug te vinden zijn in de zogenaamde policy files en dat de permissies niet enkel source gebaseerd zijn maar kunnen varieren van filespecifieke constraints tot Socketconstraints en nog veel meer.

Aangezien het grootste gevaar bij het gebruik van remote code schuilgaat achter het downloaden van onbetrouwbare code zal het noodzakelijk zijn dat in elke policy file gespecificeerd staat welke code op welke machines betrouwbaar is. Dit alles is perfect implementeerbaar op een statische single purpose cluster maar past niet binnen de dynamische fylosofie van het Jini framework. Om hieraan tegemoet te komen, zijn er tal van technieken ontworpen die zonder een statische specificatie van betrouwbare code toch afdoende veiligheid kunnen garanderen binnen de dynamische Jini omgeving:

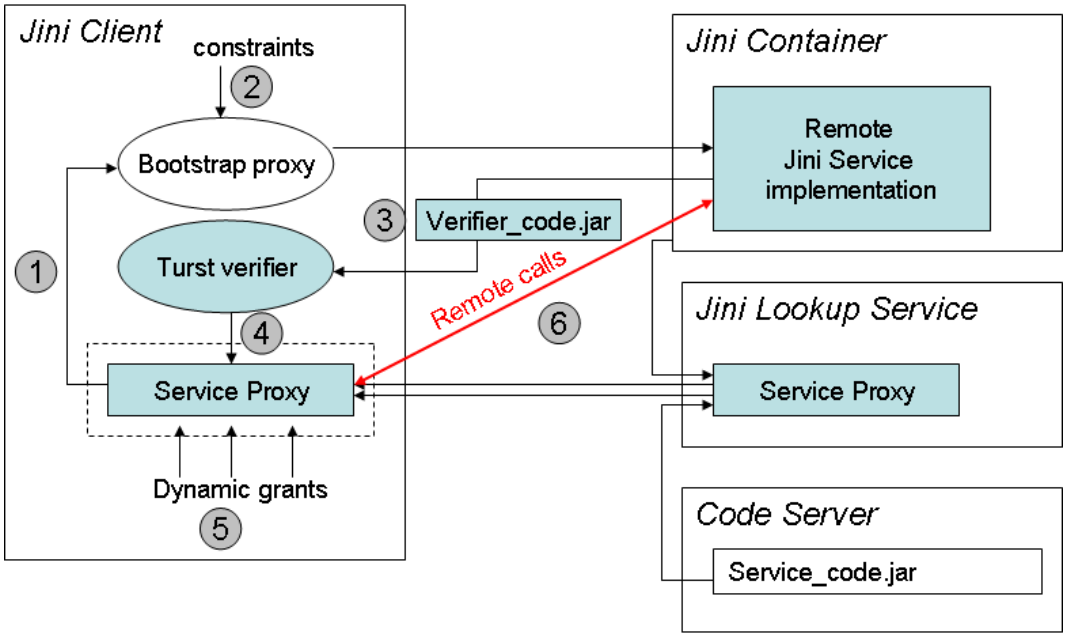
Authentication

Een belangrijk topic binnen beveiliging is de *indentificatie* en *verificatie* van een gebruiker. In Java kan een gebruiker meerdere identiteiten hebben. Een gebruiker wordt een *subject* genoemd en zijn identiteiten de *principals*. Een gebruiker identificeert zich aan de hand van een principal. Deze principal wordt geverifieerd aan de hand van meegeleverde informatie. Zo kan een gebruiker zich bijvoorbeeld identificeren aan de hand van zijn username in combinatie met een wachtwoord dat gebruikt zal worden voor de verificatie. De ondersteuning voor principals zal binnen Jini verzorgd worden door de JAAS(Java Authentication and Authorization Service). Aangezien dit topic buiten deze thesis valt zal hierop niet verder ingegaan worden. De nodige informatie is terug te vinden in [91].

Proxy trust

Bij het gebruik van een service zal de client in de eerste stap een service proxy verkrijgen. Deze proxy manifesteert zich onder de vorm van een gemarshaled object vergezeld van een URL. Deze URL wordt gebruikt om de noodzakelijke class files te bekomen. Dit kan gebeuren zonder verificatie via standaard HTTP of met class file verificatie via HTTPS of HTTPMD. Bij de ontvangst van een object gaat een gebruiker ervan uit dat dit het gewenste object is. Het is echter perfect mogelijk dat binnen het gehele process dit object vervangen is door een onbetrouwbaar object. Om dit misbruik op te sporen, is er binnen Jini nood aan een verificatiemechanisme. Een depth-first verificatie van het proxy object zou een oplossing kunnen bieden voor dit probleem. Elke proxy klasse wordt onderzocht. De lokale klassen kunnen per definitie vertrouwd worden en worden niet verder onderzocht. De gedownloade klassen

daarentegen zullen onderworpen moeten worden aan een verificatieprocedure. De enige mogelijke instantie die in staat is een proxy te verifiëren, is de originele service. Deze proxy kan echter niet gebruikt worden als communicatiemechanisme naar de service aangezien de proxy nog niet geverifieerd is. Om dit te omzeilen, worden de volgende stappen ondernomen (Zie ook Figuur B.9):



Figuur B.9: Proxy betrouwbaarheidsverificatie

1. De client verkrijgt een bootstrap proxy naar de service van de ongeverifiëerde proxy. Deze bootstrap proxy implementeert enkel de (`net.jini.security.ProxyTrust`) interface die slechts één methode, namelijk de `TrustVerifier getProxyVerifier()` methode, omvat. Aan de hand van een object graph wordt nagegaan of de proxy, zoals vereist, enkel lokale code bevat. Na deze verificatie is het mogelijk deze bootstrap proxy te gebruiken om op een veilige manier de betrouwbaarheid van de originele proxy te verifiëren bij de service.

2. Eens deze betrouwbare proxy naar deze service is opgezet, wordt het constraint mechanisme van de bootstrap proxy gebruikt om de authenticiteit van de server en de betrouwbaarheid van de data te bepalen. Deze stap wordt genomen omdat het van het grootste belang is dat enerzijds de service een betrouwbare TrustVerifier verstuurt en anderzijds deze TrustVerifier betrouwbaar blijft tot hij bij de client aankomt. Om dit te verwezenlijken, gebeurt er eerst een authenticatie van de server en wordt vervolgens het integrity mechanisme (zie volgende sectie omtrent Integrity) gebruikt om de betrouwbaarheid van de data en van de verstuurde code te garanderen.

3. Na de vorige twee stappen is de client nu zeker van de integrity van de TrustVerifier die hij verkrijgt bij het aanroepen van de `getProxyVerifier()` methode.

4. De code die gebruikt wordt door het TrustVerifier object wordt bepaald door de service. In de meeste gevallen wordt echter de standaard implementatie gebruikt die binnen de Jini security environment terug te vinden is. Dit algoritme is gebaseerd op de semantische vergelijking van objecten. Het TrustVerifier object omvat een instantie van de service proxy waarvan de correctheid gegarandeerd is. De vergelijking tussen deze correcte versie en de originele proxy zal de betrouwbaarheid bepalen. Om dit standaard algoritme te gebruiken, moet de proxy de TrustEquivalence interface en zijn enige methode, de CheckTrustEquivalence(Object obj), implementeren.

5. Wanneer de bovenstaande stappen succesvol doorlopen zijn, is de betrouwbaarheid van de proxy gegarandeerd. Dit stelt de client in de mogelijkheid de gewenste permissies toe te kennen aan de job.

6. Er is vrije communicatie tussen server en client binnen de afgesproken constraints.

Integrity

Net zoals de onbewaakte geldtransporten op de weg is het onbewaakt transporteren van data over het netwerk gevoelig voor malafide praktijken. Het is immers perfect mogelijk data en objecten die via HTTP getransporteerd worden over het netwerk te onderscheppen en naar believen te wijzigen. Om deze praktijken onmogelijk te maken, is het Integrity mechanisme toegevoegd aan het Jini security framework.

Class en Jar files zijn meestal terug te vinden op een webserver die aanspreekbaar is via het HTTP protocol. HTTP kan echter geen enkele garantie bieden omtrent de integrity van de verstuurde data. Het spreekt dan ook voor zich dat de vervanging van HTTP door een protocol dat wel integrity ondersteunt een absolute noodzaak is.

Een eerste mogelijke oplossing is het gebruik van HTTPS URL's om de locatie van de class files te bepalen. Het gebruik van zowel een authenticatie als van een encryptie mechanisme maakt dat dit protocol zowel integrity als discretie verzekert. Het protocol heeft echter ook enkele nadelen:

- Aangezien de HTTP server zich als host zal moeten authenticeren, zal hij verplicht zijn om over een private host key te beschikken. Dit brengt problemen met zich mee voor vanilla users. Ze hebben immers toegang nodig tot de private key om de gewenste communicatie mogelijk te maken.

- Hoewel het encryptie mechanisme een essentieel onderdeel is dat bijdraagt tot de integrity, is het voor de client volstrekt onbelangrijk dat de data geencripteerd wordt doorgestuurd. In de meeste gevallen heeft immers elke client toegang tot deze informatie en draagt deze encryptie weinig tot niets bij tot de confidentiality. Een systeem waarin integrity gegarandeerd wordt zonder dat encryptie vereist is, zou een betere oplossing zijn voor het probleem.

- De performantie daalt drastisch door het toepassen van de verschillende HTTPS mechanismen.

Om ook aan deze problematiek tegemoet te komen, introduceerde Jini HTTPMD wat staat voor Hyper Text Transfer Protocol Message Digest. Deze methode berekent voor elke JAR file een *Message Digest* en voegt deze als volgt toe aan de URL:

```
httpmd://jarfile.jar;md5=40c8812dce7b9f8fb0a3b364af62b974
```

Na het downloaden van de JAR file op de remote machine zal deze *message digest* eveneens berekend worden. Het identiek zijn van de berekende MD en van de MD uit de URL staat garant voor de betrouwbaarheid van de gedownloade code. Indien dit niet het geval is, resulteert deze vergelijking in een `jini.url.httpmd.WrongMessageDigestException`. Het is belangrijk om op te merken dat deze methode enkel werkt met jar files en niet met directory georieënteerde klassenhiërarchieen.

De gebruiker is volledig vrij in de keuze van het MD constructie algoritme. De implementatie van een dergelijk betrouwbaarheidsprotocol gebeurt in drie stappen:

1. De gebruiker ontwikkelt een nieuwe protocolhandler door de `java.net.URLStreamHandler` klasse uit te breiden.

2. De gebruiker ontwikkelt een nieuwe betrouwbaarheidscontrole klasse. Deze klasse, die afgeleid is van de `net.jini.security.IntegrityVerifier` klasse, bepaalt of een URL conform is met het gebruikte MD digest protocol. Het is belangrijk hierbij op te merken dat de naam van deze controle klasse moet toegevoegd worden aan de `META-INF/services/net.jini.security.IntegrityVerifier` (terug te vinden in de jsk-resource.jar file)

3. De volgende System property moet gezet worden: “java.protocol.handler.pikgs = path.to.custom.protocol.handler”. Op deze manier is de VM op de hoogte van de niewe URL handler. Dit omdat de mapping tussen URL schema’s en URL handlers globaal is binnen de virtuele machine.

Tot slot is het belangrijk op te merken dat de `net.jini.core.constraint.Integrity` constraint gezet moet zijn op een proxy die integrity controle wil ondersteunen (Zie sectie omtrent Security Constraints). Hierbij komt tevens dat `java.protocol.handler.pkgs` system property op *net.jini.url* gezet moet worden.

Confidentiality

Confidentiality betekent dat de communicatie tussen client en service niet door een derde partij gelezen kan worden. Dit wordt bewerkstelligd door het gebruik van een JERI(Jini Extensible Remote Invocation) transport provider die `net.jini.core.constraint.confidentiality` constraint ondersteunt (Zie volgende sectie omtrent Security constraints).

Security Constraints

De Security Constraints zijn een mechanisme om de gewenste security vereisten te specificeren. Enerzijds dienen ze als ondersteuning voor de authenticatie, de confidentiality en de integrity controle zoals reeds eerder beschreven. Anderzijds verzorgt

dit systeem de client en server geöriënteerde security vereisten. De constraints kunnen opgesplitst worden in twee groepen:

1. *De Eisen*: Deze constraints beschrijven aan welke voorwaarden een client en een server moeten voldoen om een proxy in gebruik te nemen. Een mooi voorbeeld hiervan is een service die instaat voor het openen van een deur. De vereiste die hier zou kunnen gesteld worden, is dat de client zich geïdentificeerd heeft als een gebruiker die toegang heeft tot de ruimte waar de deur op uitgeeft. Indien aan deze voorwaarde niet voldaan is, mag het niet mogelijk zijn om de service te gebruiken (De deur te openen).

2. *De Wensen*: Deze constraints beschrijven het onderscheid tussen de stappen die de client en de server zonder enige tegenzin willen ondernemen en aan dewelke ze enkel willen tegemoet komen als het echt vereist is. Ook in dit geval is de deur een goed voorbeeld. Een client zal, om privacy redenen, niet aan elke deur al zijn persoonlijke gegevens willen meegeven alvorens hij ze gaat open maken. De constraint zorgt er dan voor dat deze informatie enkel wordt verschaft wanneer dit een expliciete vereiste is van de service (de deur).

Het gebruik van deze twee soorten constraints zorgt ervoor dat er zich vier verschillende situaties kunnen voordoen:

1. *Conflictloze situatie*: De eisen van de service zijn voldaan zonder een wens van de client te schaden. Het spreekt voor zich dat de aangeroepen proxy methode zonder problemen gebruikt kan worden.

2. *Wensconflicten*: In deze situatie bestaat er een conflict tussen de wensen van de client en van de server. De server zal bijvoorbeeld authenticatie wensen en de client zal zich liever niet authenticeren. Het conflict wordt opgelost door geen rekening te houden met de wensen. De proxy methode kan alsnog zonder problemen gebruikt worden.

3. *Wens - Eis conflict*: In dit geval bestaat er een conflict tussen de wensen en de eisen. Om aan de eisen te kunnen voldoen, zullen niet alle wensen ingewilligd kunnen worden. De server zal bijvoorbeeld authenticatie eisen en de client wil zich liever niet authenticeren. Het conflict wordt opgelost door aan alle vereisten te voldoen en enkel aan de wensen te voldoen waar het mogelijk is. In dit voorbeeld zal de client zich authenticeren.

4. *Eisconflict*: De eisen van de client en van de service zijn zo opgesteld dat het onmogelijk is om aan alle eisen te voldoen. De gewenste stappen kunnen bijgevolg niet uitgevoerd worden en de methodeoproep zal resulteren in een java.rmi.ConnectIOException. Deze situatie treedt bijvoorbeeld op wanneer een service authenticatie vereist en een client anoniem eist te blijven.

De concrete implementatie van dit security mechanisme gebeurt aan de hand van de implementatie van twee interfaces:

1. Proxy's die security constraints ondersteunen, zijn verplicht `net.jini.core.constraint.RemoteMethodControl` interface te implementeren. Deze interface bestaat uit twee belangrijke methodes namelijk:

   (a) `RemoteMethodControl setConstraints(MethodConstraints const);`

   (b) `MethodConstraints getConstraints();`

   In deze eerste methode implementeert de service developer de manier waarop de service moet omgaan met de verschillende constraints. Via de tweede methode heeft de developer de mogelijkheid op een selectieve manier die opgelegde constraints mee te delen aan de Proxy gebruikers.

2. Constraints worden ontwikkeld door de implementatie van de `net.jini.core.constraint.Invocationconstraint` interface. Deze interface bevat geen methodes en dient enkel ter indicatie van het constraint type.

Het principe werkt dan als volgt. Alvorens de server de proxy exporteert, wordt de `setConstraints` methode aangeroepen. Deze methode resulteert in een proxy waarop de gewenste server constraints van toepassing zijn. Na het exporteren en transporteren van de proxy zal de client, bij het ontvangen van de proxy, op zijn beurt de `setConstraints` methode aanroepen. Deze finale proxy bevat dan naast de server constraints eveneneens de client constraints. Het gebruik van deze proxy garandeert de naleving van de constraints bij het gebruik van de desbetreffende service.

Zoals reeds besproken, zijn enkele Jini constraints essentieel ter ondersteuning van de verschillende reeds eerder vernoemde security mechanismen. Meer specifiek zullen de volgende vier constraints noodzakelijk zijn ter ondersteuning van het *Proxy Trust* mechanisme:

- *Integrity*: Deze constraint wordt gezet ter indicatie dat zowel class file als data integrity vereist zijn alvorens een proxy gebruikt kan worden.
- *Server Authentication*: Deze constraint duidt aan dat de proxy enkel kan gebruikt worden wanneer de server zich geauthenticeerd heeft bij de client.
- *Client Authentication*: Deze constraint is gelijkaardig aan de voorgaande en verplicht de client zich te authenticeren bij de server alvorens de proxy gebruikt kan worden.
- *Delegation*: Deze constraint stelt de server in de mogelijkheid zich te authenticeren als client bij een andere server wanneer de aanroep van een methode services op deze andere server vereist.

Enkel authenticatie is in vele gevallen niet genoeg. Het is doorgaans essentieel te definiëren op welke manier een client geauthenticeerd moet zijn alvorens hij bepaalde proxy methodes mag aanroepen. Denk maar aan het voorbeeld van de deur. Wanneer een gebruiker in een bank zich als klant registreert, is het niet de bedoeling dat hij de kluis van de bank open krijgt. Indien hij dit doet als bankdirecteur moet

hij echter wel toegang krijgen. Om dit principe te bewerkstelligen bevat Jini reeds de volgende constraints:

- *ClientMinPrincipal*: Zoals reeds in het gedeelte over authenticatie werd besproken, zal een gebruiker zich registreren aan de hand van een *principal* (identiteit). Elk van de bestaande principals heeft een bepaalde graad van mogelijkheden. De ClientMinPrincipal definiëert de minimale principal die vereist zal zijn om de proxy te gebruiken.

- *ClientMaxPrincipal*: Omgekeerd is de client in de mogelijkheid de hoeveelheid vrijgegeven informatie te beperken. Zo kan hij definiëren dat de server slechts tot op een bepaalde hoogte mag weten over welke principals een client beschikt. Denk maar aan een CIA agent. Deze zal binnen de ambassade waar hij undercover werkt in de mogelijkheid willen zijn alle deuren te openen. De deuren mogen echter enkel als informatie meekrijgen dat hij een werknemer van de ambassade is en niet dat hij tevens een CIA agent is.

- *ClientMinPrincipalType*: In vele gevallen wil een service toegang geven tot een groep van gebruikers. In dit geval zal de ClientMinPrincipalType constraint eisen dat de client zich authenticeert als een principal die binnen het gewenste principaltype valt. De voordeur van een bedrijf zal bijvoorbeeld door alle personeelsleden geopend moeten kunnen worden.

- *ClientMaxPrincipalType*: Net zoals in het vorige geval kan een client verschillende principals hebben die tot verschillende types behoren. De client kan er dan voor kiezen om slechts gegevens tot op een bepaald niveau vrij te geven. Hij specificeert dan het maximale type niveau waaruit informatie mag vrijgegeven worden. Concreet betekent dit dat enkel principals van dit type of lager als informatie worden doorgegeven aan de server.

- *ServerMinPrincipal*: De ServerMinPrincipal wordt door de client gedefiniëerd en bepaalt de minimale serverauthenticatie die de client vereist alvorens hij tot het uitvoeren van Service Methodes overgaat.

Tot slot bevat de huidige Jini implementatie nog enkele extra constraints die betrekking hebben op de quality of service:

- *ConnectionRelaltiveTime*: Deze constraint bepaalt de maximale hoeveelheid tijd die er mag gespendeerd worden aan het opzetten van een connectie tussen de proxy en de service. Deze constraint kan eventueel gebruikt worden om aan load balancing te doen. Wanneer een service te sterk belast wordt, kan de afhandeling van deze constraint ervoor zorgen dat er geconnecteerd wordt met een analoge service op een minder druk bezette server.

- *ConnectionAbsoluteTime*: Geeft exact weer op welk tijdstip de connectie opgezet moet zijn.

- *Confidentiality*: Deze constraint eist dat de data, die door het netwerk wordt gezonden, niet door derden gelezen of aangepast kan worden.

Naast deze meer algemene en ondersteunende constraints die reeds in Jini omvat zitten, beoogt het constraint mechanisme de individuele implementatie van software specifieke constraints. De developer wordt dan niet enkel verantwoordelijk gesteld voor de afhandeling van de constraints maar tevens voor de implementatie ervan. Let er wel op dat deze constraints zowel door de server als door de client gekend moeten zijn. Indien dit niet het geval is, zal er nooit aan de constraint voldaan kunnen worden en zullen de gewenste instructies niet uitgevoerd kunnen worden.

Dynamic permission grants

Tot op heden stonden de verschillende security issues in het teken van het opzetten van een veilige connectie tussen de gebruiker en de service. Eens de veiligheid van de proxy gegarandeerd is, rest ons nog de implementatie van de verschillende proxy permissies. In de context van het hierboven aangehaalde voorbeeld spreekt het voor zich dat de proxy van een klant van een bank de bankkluis niet zo maar zal mogen open maken dit in tegenstelling tot de proxy van de bankdirecteur. Het proces dat hier in werking treed, is het autorisatie proces. Het zal er voor zorgen dat elk type gebruiker aan de hand van zijn principals toegang krijgt tot de tot zijn bevoegdheid behorende resources. Het is belangrijk dat dit autorisatieproces pas plaats vindt nadat de proxy trust is geverifiëerd, dit om misbruik te vermijden. Enkele voorbeelden van resource permissies zijn `FilePermission` , `PrivateCredentialPermission` , `SocketPermission` , `DownloadPermission` , etc.. In het standaard model zijn de code permissies terug te vinden in een statische policy file. Deze code permissies worden toegekend aan de hand van het protection domein waartoe de code behoort. Zo een protection domein wordt gedefinieerd aan de hand van drie punten:

- *De source code*: Het is hier belangrijk te weten wie de code gesigned heeft en de plaats te kennen waarvan de code wordt gedownload.

- *De uitvoerder van de code*: Het is hier belangrijk te weten over welke principals de uitvoerder beschikt.

- *The Classloader*: Aangezien er een one-to-one mapping bestaat tussen de Classloader en de codebase, zal de Classloader instaan voor de specificatie van de codebase.

Zoals het laatste punt omtrent de Classloader al laat vermoeden, is het belangrijk permissies toe te kennen aan de hand van deze Classloader. Het is immers niet de bedoeling voorkennis te hebben over de origine van de gebruikte code. Om hieraan tegemoet te komen zonder de authenticatie restricties te verloochenen, wordt de opdracht tot het toekennen van permissies doorgeschoven naar de Classloader. Hij zal er voor zorgen dat er dynamisch permissies worden toegekend aan de verschillende protection domains van de proxy klassen. Zoals reeds hierboven is aangehaald zal er een one-to-one mapping bestaan tussen de Classloader en de codebase en kunnen we dus stellen dat het toekennen van permissies op het niveau van de Classloader gelijk staat aan het toekennen van permissies op het codebase niveau.

Standaard gebeurt het toekennen van permissies aan de hand van statische policy config files. Wanneer een applicatie het gebruik van bepaalde resources vereist, zal de JVM's policy provider, aan de hand van deze statische policy files, het protection domein van de gewenste code en de principal van de user die de applicatie uitvoert, beslissen of de applicatie het recht krijgt de resources te gebruiken.

In het Jini security framework zal dit echter niet altijd volstaan. Het framework beschikt dan ook over een policy provider die at runtime gespecificeerde dynamische permissietoekenning ondersteunt. De `net.jini.security.policy.DynamicPolicy` interface definieert een policy provider die in staat is een set van permissies toe te kennen aan elk van de protection domains die met de Classloader van de klasse geassocieerd zijn.

De hieronder gedefinieerde grant methode specificeert een set van permissies voor elk van de protection domains die een minimale set van principals definieert. Tevens staat hij in voor de binding van deze pemissies met de Classloader die de klasse heeft ingeladen. De permissies worden bijgevolg, zoals hierboven reeds beschreven, geassocieerd met de Classloader.

```
void grant(Class cl, Principal[] principals, Permission[] permission)
```

De `DynamicPolicyProvider` klasse implementeert de `DynamicPolicy` en omsluit een file-gebaseerde statische policy provider (`PolicyFileProvider`). Merk op dat, om dit alles mogelijk te maken, het essentieel is om de `net.jini.security.GrantPermission` toe te kennen. Deze omvat een lijst van permissies die dynamisch mogen toegekend worden. Tot slot dient hier nog aan toegevoegd te worden dat de `Security.grantSupported()` methode gebruikt kan worden om na te gaan of een bepaalde security provider een welbepaalde dynamische permissietoekenning ondersteunt.

B.3.8 Configuratie

Een sterk minpunt van Jini is zijn reputatie om moeilijk in- en opstelbaar te zijn. Deze problematiek ligt in het feit dat de setup gepaard gaat met verschillende scripts en klasses die gegroepeerd moeten worden in verschillende jar files en verspreid moeten worden over de verschillende betrokken JVM's. De security acces problematiek die hieruit volgt, in combinatie met firewall issues, zorgt al snel voor frustratie. Hier komt nog bij dat foutieve configuratie moeilijk opspoorbaar is en dat herconfiguratie een enorme inspanning vraagt. Dit laatste vooral omdat traditioneel veel instellingen in de source code gespecificeerd diende te worden. Dit maakte dat hercompilatie en herverdeling telkens noodzakelijk was. Om dit probleem alsnog te minimaliseren, is het `net.jini.config` package toegevoegd aan de Jini 2.0 releases. Het doel van deze package is de ontkoppeling van deployment informatie en sourcecode. Het gehele systeem is gebaseerd op een klasse en een interface:

- Interface `net.jini.config.configuration` : Deze interface beschrijft hoe een configuratie moet ondervraagd worden. Essentieel bestaat de interface uit verschillende vergelijkbare methodes die elk een bepaald basistype of Objecttype teruggeven:

  - `Oject getNonNullEntry(Configuration config, String component, String name, Class type)`
  - `Object Config.getNonNullEntry(Configuration config, String component, String name, Class type, Object defaultValue)`
  - `Object Config.getNonNullEntry(Configuration config, String component, String name, Class type, Object defaultValue, Object data)`
  - `long Config.getLongEntry(Configuration config, String component, String name, long defaultValue, long min, long max)`
  - `int Config.getIntEntry(Configuration config, String component, String name, int defaultValue, int min, int max)`
  - `float Config.getFloatEntry(Configuration config, String component, String name, float defaultValue, float min, float max)`

  Hiernaast omvat Jini tevens een skeletimplementatie van deze interface onder de naam `net.jini.config.AbstractConfiguration` die de implementatie van een eigen configuratieobject aanzienlijk vereenvoudigt.

- De `net.jini.config.ConfigurationProvider` klasse is een abstracte klasse die standaardmethodes implementeert om de bovenstaande configuratieobjecten te verwerven aan de hand van configureerbare providers. Aangezien deze klasse abstract is, kan ze niet geïnstantieerd worden. Om de configuration provider alsnog te specificeren wordt er gebruik gemaakt van de naam van de provider klasse die vervat zit in de `META-INF/services/net.jini.config.-Configuration` resource. Indien deze klasse niet gevonden wordt, wordt er gebruik gemaakt van de `ConfigurationFile` klasse. Dit is de standaard provider klasse en wordt gebruikt in de meeste Jini 2.0 programmatuur. Het spreekt voor zich dat de gebruiker vrij is alsnog een andere implementatie te verkiezen. Gekende alternatieven zullen bijvoorbeeld informatie uit sockets of uit een database halen.

  De `ConfigurationFile` kan niet vergeleken worden met een configuratie bestand dat enkel data bevat. Het streeft naar een evenwichtige verhouding tussen data enerzijds en de ondersteuning van standaard programmeertaal functionaliteit anderzijds. De structuur die hierin wordt gebruikt, doet zeer sterk denken aan Java maar beperkt zich tot:

  - De opslag van waarden in variabelen
  - Het aanmaken van objecten
  - De oproep van statisch methodes
  - Een beperkte toegang tot de *calling environment*

  Dit brengt met zich mee dat een gebruiker in de mogelijkheid is een standaard implementatie te maken zonder gebruikt te maken van lusinstructies, condities

en niet statische methode oproepen. Meer informatie over de syntax is terug te vinden in de API documentatie van de `ConfigurationFile` klasse.

Om dit alles te verduidelijken volgt hier een klein voorbeeldje:

```
import net.jini.jeri.BasicILFactory;
import net.jini.jeri.BasicJeriExporter;
import net.jini.jeri.tcp.TcpServerEndpoint;
import com.sun.jini.config.ConfigUtil;
import net.jini.core.discovery.LookupLocator;
import net.jini.core.entry.Entry;
import net.jini.lookup.entry.*;
import java.io.File;

Voorbeeld{
  localhost = ConfigUtil.getHostName();
  port = "80";
  directory = "/classes";
  codebase = ConfigUtil.concat(new String[] {"http://", localhost, ":"
      , port, directory});
  exporter = new BasicJeriExporter(TcpServerEndpoint.getInstance(0),
      new BasicILFactory());
}
```

De ConfigurationFile

```
...
Configuration config = ConfiguratioProvider.getInstance("Voorbeeld.
    conf");
Exporter exp = (Exporter) config.getEntry("Voorbeeld"
    ,"exporter","Exporter.class");
String codeb = (String) config.getEntry("Voorbeeld"
    ,"codebase",String.class);
...
```

Het gebruik van de configurationfile in de code

Deze manier van werken is zeer flexibel. Wanneer men bijvooreeld niet langer JERI maar JRMP wilt gebruiken kan dit eenvoudig door:

```
  exporter = new BasicJeriExporter(TcpServerEndpoint.getInstance(0)
    ,new BasicILFactory());
```

te vervangen door:

```
  exporter = new JrmpExporter();
```

Tevens kan er op een eenvoudige manier voor een meer secure connectie gekozen worden door:

```
  TcpServerEndpoint.getInstance(0)
```

te vervangen door:

```
  SslServerEndpoint.getInstance(0)
```

Aangezien de Voorbeeld.conf een configuratie file is kunnen al deze aanpassingen na compilatie net voor de deployment eenvoudig doorgevoerd worden. Er is niet langer hercompilatie nodig. De code (jar files) kan met andere woorden stabiel op de verschillende locaties blijven staan en enkel de configfiles moeten gewijzigd worden.

B.3.9 Classloading

De omvang van het topic *Classloading* maakt dat een uiteenzetting van heel dit domein ver buiten de scope van deze thesis valt. Om alsnog een beeld te scheppen van de Jini gerelateerde classloading extensies geeft deze sectie een zeer beperkte omschrijving van de basisprincipes van classloading en beschrijft het hoe de Jini extensies hier in passen.

Standaard JVM Classloading

Classloading is gebaseerd op één basisprincipe. Dit principe stelt dat een JVM een klasse die door twee verschillende Classloaders geladen wordt beschouwd als twee verschillende types. Formeel stelt men dat elke $\langle Class, Classloader \rangle$paar uniek bepaald is. Dit brengt met zich mee dat er ten allen tijde moet voorkomen worden dat een Classloader een klasse meerdere keren laadt. Om dit te bewerkstelligen zal elke Classloader elke class load opdracht delegeren naar zijn parent Classloader. Indien een parent de klasse reeds geladen heeft zal deze klasse gebruikt worden, indien dit niet het geval is zal de parent klasse proberen de klasse zelf te laden. Enkel in het geval van failure zal de load taak terug gedelegeerd worden naar de originele child klasse. Dit principe heeft als gevolg dat elke klasse slecht één maal geladen wordt en dat de versies van een door een Classloader geladen klasse ook gebruikt zal worden door al zijn afstammelingen.

Preferred Classloading

Een prefered Classloading zal zoals zijn naam het aangeeft, de versie van de klassen laden die hij zelf prefereert. Om dit te verwezelijken zal het laden van een klasse niet langer als default gedelegeerd worden naar de parent Classloader. De Classloader zal nu zelf de keuze maken tussen het delegeren of het laden van een eigen versie van de klasse. Om dit alles te verwezelijken wordt de eerste URL locatie van de Classloader voorzien van een lijst met geprefereerde klassen. Bij het laden van deze klassen zal een Classloader een eigen versie laden. Bij de overige klassen wordt de taak doorgedelegeerd naar de parent Classloader.

Een concrete voorbeelden van deze techniek is terug te vinden bij de “Sun’s contributed services” in de JSK (Jini Software Kit) zoals: Reggiem Mahalo,Landlord,.... Deze bevatten een *preferred list* in hun “*-dl.jar” files. De lijst stelt dat de verschillende interfaces NIET geprefereerd zijn maar de verschillende implementaties WEL. Hiermee wordt de mogelijkheid gecreëerd om nieuwe, aangepaste of alternatieve versies van de service implementaties te laden zonder de interface compatibiliteit te verliezen.

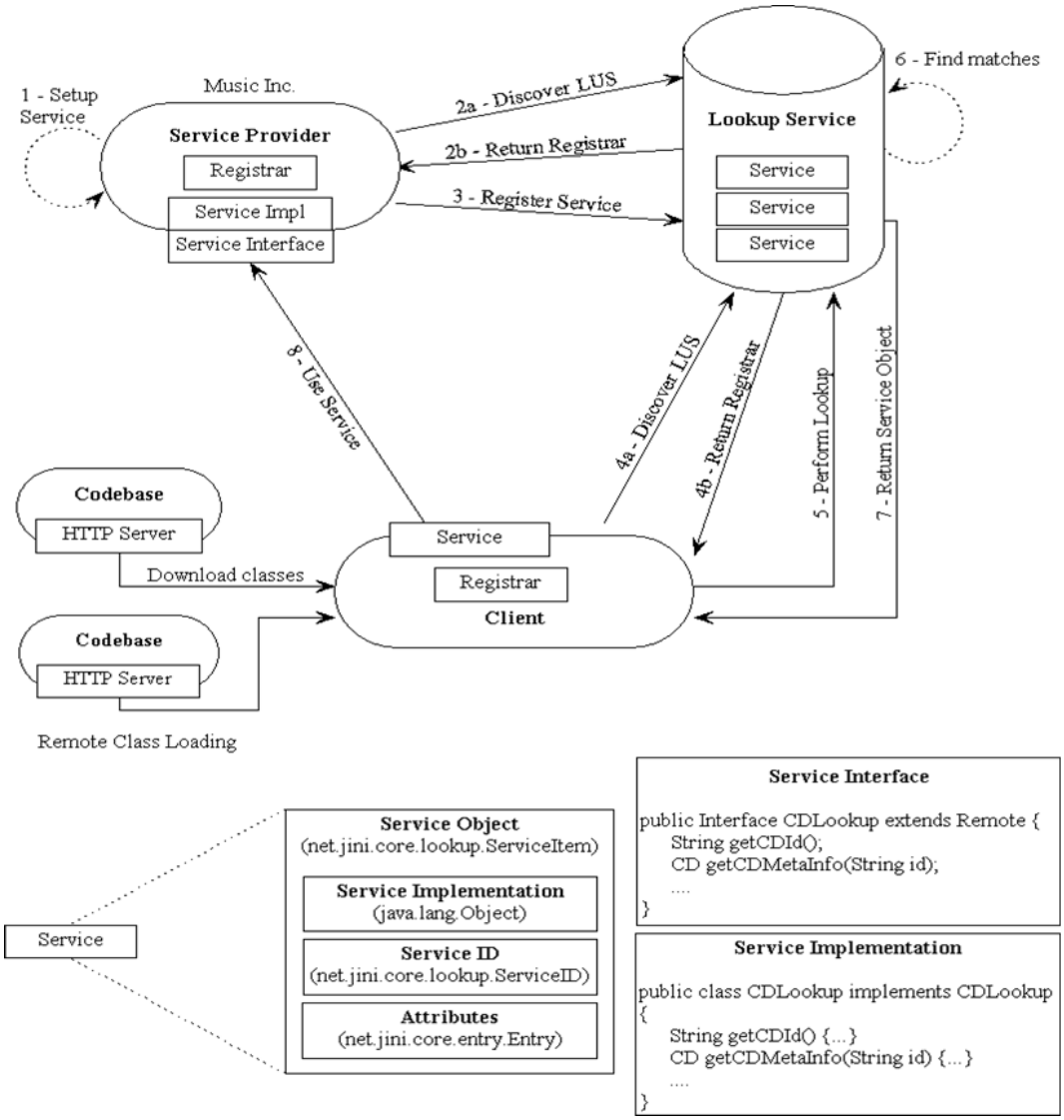
Concreet ziet het verschil van de gebruiker er als volgt uit. Stel dat hij *“een-*

Service.jar" archive file importeert. Wanneer er niet wordt gewerkt met prefered classes, worden de implementatie klassen van het classpath gebruikt in plaats van de implementatie klassen van *"eenService-dl.jar"*. Wanneer er met preferred classes wordt gewerkt daarentegen, zullen steeds de meest recente implementaties worden ingeladen van *"Codebase/eenService-dl.jar"* (en dus niet van zijn classpath).

B.4 Operationeel Model

Deze sectie geeft aan de hand van een klein voorbeeld een schematische voorstelling van de werking van Jini (zie figuur B.10). Het Opzetten van de Jini service werkt in acht stappen:

1. *Service Setup*: De Service interface (CDLookup) en de Service implementatie (CDLookupImpl) worden geconstruëerd.
2. *Server LUS Discovery*: De server ontdekt de LUS en krijgt een registrar.
3. *Server registers Service*: Aan de hand van de registrar wordt de service geregistreerd en opgeslagen. Hiermee wordt, het voor de gebruiker visuele gedeelte van de service, publiek toegankelijk gemaakt. De verschillende clients zijn dan in de mogelijkheid dit te downloaden.
4. *Client LUS Discovery*: De client ontdekt de LUS en krijgt een registrar.
5. *Client Service Lookup*: De client zoekt via de verkregen registrar de verschillende beschikbare services op aan de hand van een service template.
6. *Service Matching*: De server zoekt aan de hand van de template alle matchende services op.
7. *Service doorgave*: De server geeft kopieën van de matchende services objecten door aan de client.
8. *Service gebruik*: De Client kan aan de hand van de verkregen objecten met de verschillende services communiceren en er de gewenste methodes van aanroepen.



Figuur B.10: Het Operationeel Model

BIJLAGE C

CoBRA Voorbeelddistributie Output

Deze appendix geeft de output van de Voorbeeld CoBRA distributie uit sectie 5.6.4 weer:
drone22of1 heeft 1 processors
drone55of1 heeft 1 processors
drone23of1 heeft 1 processors
drone50of1 heeft 1 processors
drone26of1 heeft 1 processors
drone05of1 heeft 2 processors
drone48of1 heeft 1 processors
drone26of1 heeft 1 processors
drone11of1 heeft 1 processors
drone22of1 heeft 1 processors
drone08of1 heeft 1 processors
drone31of1 heeft 1 processors
drone05of1 heeft 2 processors
drone26of1 heeft 1 processors
drone52of1 heeft 1 processors
drone34of1 heeft 1 processors
drone17of1 heeft 1 processors
drone05of1 heeft 2 processors
drone27of1 heeft 1 processors
drone26of1 heeft 1 processors
drone51of1 heeft 1 processors
drone11of1 heeft 1 processors
drone48of1 heeft 1 processors
drone35of1 heeft 2 processors
drone40of1 heeft 2 processors
drone53of1 heeft 1 processors

drone22of1 heeft 1 processors
drone55of1 heeft 1 processors
drone23of1 heeft 1 processors
drone16of1 heeft 1 processors
drone39of1 heeft 1 processors
drone31of1 heeft 1 processors
drone21of1 heeft 1 processors
drone44of1 heeft 1 processors
drone10of1 heeft 1 processors
drone06of1 heeft 2 processors
drone45of1 heeft 1 processors
drone28of1 heeft 1 processors
drone32of1 heeft 1 processors
drone55of1 heeft 1 processors
drone26of1 heeft 1 processors
drone18of1 heeft 1 processors
drone16of1 heeft 1 processors
drone22of1 heeft 1 processors
drone33of1 heeft 1 processors
drone21of1 heeft 1 processors
drone41of1 heeft 1 processors
drone06of1 heeft 2 processors
drone44of1 heeft 1 processors
drone38of1 heeft 1 processors

BIJLAGE D

GIPSy Configfiles

Deze appendix bevat concrete voorbeelden van de GIPSy gerelateerde configuratie-objecten zoals deze beschreven zijn in sectie 9.4.2. Zoals reeds werd vermeld, kunnen er verschillende versies van deze objecten aangemaakt worden (bv. config file versie, database versie, ...). Omwille van de eenvoud zal er telkens een voorbeeld gegeven worden van de configuratie file versie van deze objecten.

D.1 Het Gobale configuration object

```
Model1File=/home/peter/configfiles/Kriging.xml
DataSelectorFile=/home/peter/configfiles/DistanceDataSelector.xml
StartDataSelectorFile=/home/peter/configfiles/BasicStartGridSelector.
    xml
ErrorCalculatorFile=/home/peter/configfiles/BasicErrorCalculator.xml
DataFile=/home/peter/data/total.txt
SelectionAmount=100
WorkerAmount=5
ResultFile=/home/peter/result/Result.dat
LogFileName=/home/peter/log/testje
Plot=false
```

D.2 Het ErrorCalculator configuration object

```
<ErrorCalculator name="be.ac.ua.comp.GIPSy.ErrorCalculator.
    BasicErrorCalculator">
  <Path>file:///home/peter/lib/Errorcalculator.jar</Path>
</ErrorCalculator>
```

D.3 Het SampleSelector configuration object

```
<DataSelector name="be.ac.ua.comp.GIPSy.DataSelectors.NPDataSelector">
  <Path>file:///home/peter/lib/Dataselector.jar<\Path>
  <Param name="neighbours">5</Param>
  <Param name="dataselector">
    /home/peter/configfiles/BasicErrorCalculator.xml
  </Param>
  <Param name="penalisation">100.0</Param>
  <Param name="predictorlist">
    <Item>/home/peter/configfiles/Pol5.xml</Item>
    <!-- Hier kunnen meerdere modellen staan >
  </Param>
</DataSelector>
```

D.4 Het Model configuration object

```
<PredictorDescription>
  <Path>file:///home/peter/lib/Kriging.jar<\Path>
  <Factory name="be.ac.ua.comp.GIPSy.Predictors.KrigingPredictors.
      KrigingFactory">
  </Factory>
  <Predictor name="SplittedKrigingPredictor">
    <Param name="poldegree">2</Param>
    <Param name="scale">1</Param>
    <Param name="kernel>Inverse Multiquadri<Param>
  </Predictor>
</PredictorDescription>
```

D.5 Het StartDataSelector configuration object

```
<StartDataSelector name="be.ac.ua.comp.GIPSy.StartGridSelector.
    BasicStartGridSelector">
  <Path>file:///home/peter/lib/StartDataselector.jar<\Path>
</StartDataSelector>
```

BIJLAGE E

Neurale Netwerk Software Libraries

Deze sectie bevat een korte beschrijving en een beoordeling van de verschillende Neurale Netwerk libraries die beschikbaar zijn.

E.1 Yawn 2

E.1.1 Website

http://sourceforge.net/projects/yawn2/
http://yawn2.sourceforge.net/

E.1.2 Omschrijving

Yawn (Yet Another {Weird — Wacky — Witty — Wonderful — W*} Network) is een Java framework om neurale netwerken te trainen en te testen onafhankelijk van waar ze gebruikt zullen worden. Het is ontwikkeld aan de Universiteit Carlos III in Madrid

E.1.3 Pluspunten

- De Yawn API omvat enkel netwerk-modellen, waaronder bijvoorbeeld multilayer-perceptron model. Het gaat hier echter slechts om een referentie-implementatie die volgens de developers geen goede resultaten behaalt.
- Het is mogelijk een netwerk onafhankelijk van het gebuik te testen en te trainen.

E.1.4 Documentatie

De documentatie is zeer beperkt. De developers geven zelf aan dat dit werk voor de toekomst is.

E.1.5 Beoordeling

Dit project heeft zeker potentieel, maar dient nog grondig herwerkt te worden. De ontwikkeling werd echter beëindigd in 2004. Het krijgt bijgevolg een negatieve beoordeling.

E.2 JAS

E.2.1 Website

http://sourceforge.net/projects/jaslibrary/
http://jaslibrary.sourceforge.net/

E.2.2 Omschrijving

De java-toolkit JAS is gespecialiseerd in agent-based simulaties. Het is ontwikkeld aan de Universiteit van Toronto.

E.2.3 Pluspunten

- Neurale netwerken en genetische algoritmes worden ondersteunt.
- De functionaliteit centraliseert zich rond een simulaties van agent systemen. Het geheel wordt ondersteunt met een eigen classloader, database, tijdseenheid management, ...

E.2.4 Documentatie

De API documentaties is voor een aantal specifieke voorbeelden voor handen. Het is echter jammer dat geen enkel van deze voorbeelden gebruik maakt van neurale netwerken. De documentatie is voor dit project dan ook verwaarloosbaar en aldus onbestaande.

E.2.5 Beoordeling

Dit project is meer geschikt voor agent simulaties die gebruik maken van een neurale netwerken. Het is niet specifiek gericht op het modelleren van neurale netwerken wat de relevantie in dit project nihil maakt.

E.3 NeuralJ

E.3.1 Website

http://sourceforge.net/projects/neuralj/
http://www.neuralj.org/

E.3.2 Omschrijving

NeuralJ is een java-library voor neurale netwerken en genetische algoritmen en heeft als doel het eenvoudigste en meest flexibele neurale netwerk framework te worden.

E.3.3 Pluspunten

- Biedt ondersteuning voor genetische algoritmes en neurale netwerken, meerbepaald multilayer perceptron met backpropagation.

E.3.4 Documentatie

De documentatie bestaat uit Java API documentatie en een drietal kleine voorbeelden. Deze beperkte documentatie is echter wel voldoende om met het framework te kunnen werken.

E.3.5 Beoordeling

De verschillende mogelijke netwerktopologieën zijn gepredefinieerd wat maakt dat de gebruiker slechts een beperkte vrijheid heeft. Dit in combinatie met de ondermaatse documentatie maakt het zeer moeilijk (niet onmogelijk) om met het framework te werken. Aangezien dit project nog zeer jong is kunnen we besluiten dat het in de huidige vorm niet bruikbaar is maar dat de voortgang van het project naar de toekomst toe zeker moet opgevolgd worden.

E.4 NNTK

E.4.1 Website

http://sourceforge.net/projects/nntk/

E.4.2 Omschrijving

NNTK (Neural Network ToolKit) is een Java-API die instaat voor het modelleren, implementeren en trainen van neurale netwerken. Het project werd ontwikkeld aan de Universiteit van Brazilië als een uitbreiding op het UnBMiner datamining-project.

E.4.3 Pluspunten

- Bevat een eenvoudige grafische userinterface die een netwerk kan trainen met ingelezen data.
- Biedt ondersteuning voor Hopfield, Multilayer perceptron, Radial Basis Function netwerken

E.4.4 Documentatie

Er is geen documentatie beschikbaar.

E.4.5 Beoordeling

Het framework werkt aan de hand van een GUI en valt bijgevolg niet eenvoudig te integreren in andere projecten. Het trainen van een grote groep datapunten (1000 punten) duurt enorm lang. De opbouw van de code is zowel estetisch als programatorisch ondermaats en bovendien is er sinds 2005 geen nieuwe release meer geweest.

E.5 Boone

E.5.1 Website

http://sourceforge.net/projects/boone/
http://www.cosy.sbg.ac.at/project/robolab/?projects-boone

E.5.2 Omschrijving

Het Boone (Basic Object-Oriented Neural Environment) project tracht een eenvoudig framework aan te bieden voor het modelleren van neurale netwerken. Het is ontwikkeld aan de Universiteit van Salzburg.

E.5.3 Pluspunten

- Focus ligt op flexibiliteit.
- Biedt ondersteuning voor feed-forward en Hopfield netwerken.

E.5.4 Documentatie

De Java API documentatie is beschikbaar. Er is tevens een, zij het verouderde (2003), handleiding beschikbaar op de website.

E.5.5 Beoordeling

De vooropgestelde flexibiliteit van dit framework heeft een negatieve invloed op de complexiteit. De beperkte en verouderde documentatie maakt dan ook dat dit framework niet voldoet aan de vereisten.

E.6 SNARLI

E.6.1 Website

http://sourceforge.net/projects/snarli/ http://snarli.sourceforge.net/

E.6.2 Omschrijving

SNARLI heeft als doelstelling om op een eenvoudige manier verschillende soorten neurale netwerken te modeleren. Het is ontwikkeld onder leiding van Simon D. Levy aan de universiteit van Washington.

E.6.3 Pluspunten

- Feed-forward netwerken met backpropagation en Kohonen netwerken worden ondersteund.
- De API is zeer éénvoudig te integreren in een stuk software.

E.6.4 Documentatie

De documentatie is opgebouwd uit de Java API documentatie en tal van voorbeelden.

E.6.5 Beoordeling

Deze library is vrij éénvoudig te gebruiken en op het eerste zicht een uitstekende kandidaat voor de GIPSy implementatie. Het project is echter reeds sinds 2003 in beta-stadium en blijkt nog tal van fouten te bezitten. Omwille van deze reden is er dan ook voor gekozen die framework niet te gebruiken.

E.7 JCortex

E.7.1 Website

- http://sourceforge.net/projects/jcortex/
- http://jcortex.sourceforge.net/
- http://www.jcortex.com

E.7.2 Omschrijving

JCortex is een totaalpakket dat de software ontwikkelaar de mogelijkheid geeft om de meest exotische neurale netwerk constructies te vervaardigen en te gebruiken. Het is ontwikkeld aan de Universiteit van Pontificia Comillas in Spanje.

E.7.3 Pluspunten

- Ondersteunt een breed gamma netwerken (multilayer perceptron, hopfield, ...)
- Er bestaat een tool die het mogelijk maakt via een grafische user interface netwerken op te bouwen en deze dan verder via het framework te gebruiken.

E.7.4 Documentatie

Het project is slecht gedocumenteerd. De gebruiker moet het stellen met een beperkte API documentatie en een Spaanse handleiding.

E.7.5 Beoordeling

In tegenstelling tot de veelbelovende doelstelling van dit project is de huidige situatie bedroevend. Er is enkel een alpha-versie beschikbaar en de innoverende componenten van het framerwork zoals de Grafische Netwerk builder blijken niet te werken. De beperkte documentatie en de onbetrouwbaarheid van de bestaande code maken dan ook dat dit framework niet verder gebruikt zal worden.

E.8 Joone

E.8.1 Website

http://sourceforge.net/projects/joone/ http://www.jooneworld.com/

E.8.2 Omschrijving

De Joone (Java Object Oriented Neural Engine) is ontworpen door de Joone-community om neurale netwerken te ontwikkelen, te trainen en te testen. Het opzet is een krachtige omgeving creren voor professionele gebruikers.

E.8.3 Pluspunten

- Tal van netwerk topologieën worden ondersteund: feed-forward netwerken, time delay netwerken, Kohonen self-organising-maps,....
- De modulaire opbouw maakt het geheel eenvoudig aanpasbaar en makkelijk uitbreidbaar.
- Gebruik makend van een grafische editor kunnen verschillende netwerken opgebouwd worden en vervolgens in de eigen code gebruikt worden.
- De Distributed Training Environment (DTE) maakt het mogelijk om verschillende netwerken parallel te trainen en te gebruiken.
- Joone is een levend project en wordt nog steeds actief ondersteund door de developers.

E.8.4 Documentatie

De documentatie bestaat uit Java API documentatie, een zeer uitgebreide manual en een wiki met voorbeelden en tutorials.

E.8.5 Beoordeling

De eenvoud in gebruik in combinatie met de kracht van het framework maken dat deze toolbox een uitstekende kandidaat is voor het GIPSy framework.

E.9 JavaNNS

E.9.1 Website

http://www.cis.cau.edu/675/javasnns.html

E.9.2 Omschrijving

JavaNNS is een neurale netwerk simulator ontwikkeld aan de Universiteit van Tbingen. Het is een frontend op SNNS (Stuttgart Neural Network Simulator).

E.9.3 Pluspunten

- De grafische frontent geeft een intuïtief beeld van het ontwikkelde neurale netwerk.
- JavaNNS is ontwikkeld boven op de SNNS (Stuttgart Neural Network Simulator) en ondersteunt aldus een zeer groot gamma leeralgoritmes en andere functionaliteit.

E.9.4 Documentatie

De documentatie bestaat uit een uitgebreide handleiding.

E.9.5 Beoordeling

Het gebruik van de SNNS C-library maakt enerzijds dat de aangeboden functionaliteit afdoende uitgebreid is. Anderzijds is het juist deze JNI interfacing met de SNNS C-library dat er voor zorgt dat de functionaliteit enkel in de aangeboden grafische frontent kan gebruikt worden. Dit maakt dat het geheel niet toepasbaar zal zijn in ons framework. Hierbij komt nog dat Java NNS niet langer actief ontwikkeld wordt. De laatste release dateert van 2002.

E.10 Conclusie

De complexiteit van Neurale netwerken maakt dat vele pogingen worden ondernomen om bruikbare software te schrijven die deze problematiek ondersteunt. Weinig van deze pogingen resulteren echter in bruikbare software. De problematiek zit hem dan vooral in de stabiliteit en de toepasbaarheid van de software. Dit in combinatie met slechte documentatie maakt dat slechts weinige libraries voldoen aan de vereisten van GIPSy. De Joone library scoort echter zowel op het gebied van gebruiksvriendelijkheid, plugability als functionaliteit zeer bevredigend. Er is dan ook voor gekozen om dit framework als neuraal netwerk library te gebruiken in het GIPSy framework.

Bibliografie

[1] Apache webserver http://www.apache.org.

[2] APST available: http://grail.sdsc.edu/projects/apst.

[3] CoGKit commodity grid kit available: http://www.omg.org/library/iiop4.html.

[4] EGEE enabling grids for e-sience available: http://public.eu-egee.org.

[5] Folding@home available: http://folding.stanford.edu.

[6] Globus available: http://www.globus.org.

[7] HTTP hypertext transfer protocol available: http://www.w3.org/protocols.

[8] IIOP internet inter-orb protocol available: http://www.omg.org/library/iiop4.html.

[9] Nimrod available: http://www.csse.monash.edu.au/ davida/nimrod/.

[10] Secure shell available: http://www.ssh.com.

[11] Winstone servlet container: http://winstone.sourceforge.net/.

[12] *SSL and TLS: designing and building secure systems.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2001.

[13] ISO 10026-1. Information technology - Open Systems Interconnection - Distributed Transaction Processing. Final draft, ISO/IEC, 1992.

[14] Arshad Ali, Ashiq Anjum, Julian Bunn, Richard Cavanaugh, Frank van Lingen, Richard McClatchey, Muhammad Atif Mehmood, Harvey Newman, Conrad Steenberg, Michael Thomas, and Ian Willers. Predicting the resource requirements of a job submission.

[15] David P. Anderson, Jeff Cobb, Eric Korpela, Matt Lebofsky, and Dan Werthimer. Seti@home: an experiment in public-resource computing. *Communications of the ACM*, 45(11):56–61, November 2002.

[16] F. Arickx, J. Broeckhove, P. Hellinckx, D. Dewolfs, and K. Vanmechelen. *The Encyclopedia of Grid Computing Technologies and Applications*, chapter Grid-based nuclear physics applications. 2008.

[17] Pieter Audiens. Cobras- cobra distribution of javaclasses. Master's thesis, University of Antwerp, 2008.

[18] Keith Ballinger, Peter Brittenham, Ashok Malhotra, William A. Nagy, and Stefan Pharies. Web services inspection language (ws-inspection) 1.0.

[19] Å. Björck. *Numerical Methods for Least Squares Problems*. SIAM, Philadelphia, 1996.

[20] BOINC. Berkeley open infrastructure for network computing available: http://boinc.berkeley.edu/.

[21] J. Brevik, D. Nurmi, and R. Wolski. Automatic methods for predicting machine availability in desktop grid and peer-to-peer systems. In *CCGRID '04: Proceedings of the 2004 IEEE International Symposium on Cluster Computing and the Grid*, pages 190–199, Washington, DC, USA, 2004. IEEE Computer Society.

[22] J. Broeckhove, Frans Arickx, P. Hellinckx, V. S. Vasilevsky, and A. V. Nesterov. The H-5 resonance structure studied with a three-cluster J-matrix model. *J. Phys.*, G34:1955–1970, 2007.

[23] Rajkumar Buyya and Manzur Murshed. Gridsim: A toolkit for the modeling and simulation of distributed resource management and scheduling for grid computing. In *Concurrency and Computation: Practice and Experience*, page 2002. Wiley Press, 2002.

[24] Henri Casanova, Graziano Obertelli, Francine Berman, and Rich Wolski. The apples parameter sweep template: user-level middleware for the grid. In *Supercomputing '00: Proceedings of the 2000 ACM/IEEE conference on Supercomputing (CDROM)*, page 60, Washington, DC, USA, 2000. IEEE Computer Society.

[25] Chih-Chung Chang and Chih-Jen Lin. Libsvm – a library for support vector machines. http://www.csie.ntu.edu.tw/ cjlin/libsvm/.

[26] Roberto Chinnici, Jean-Jacques Moreau, Arthur Ryman, and Sanjiva Weerawarana. Web services description language (wsdl) version 2.0 part 1: Core language.

[27] Condor. Available at http://www.cs.wisc.edu/condor/.

[28] M. V. Devarakonda and R. K. Iyer. Predictability of process resource usage: A measurement-based study on unix. *IEEE Trans. Softw. Eng.*, 15(12):1579–1586, 1989.

[29] A. Downey. Predicting queue times on space-sharing parallel computers. *ipps*, 00:209, 1997.

[30] W. Keith Edwards. *Core JINI*. Prentice Hall Professional Technical Reference, 2000. Foreword By-Bill Joy and Foreword By-Brian Murphy.

[31] V. A. Epanechnikov. Non-parametric estimation of a multivariate probability density. *Theory of Probability and its Applications*, 14(1):153–158, 1969.

[32] Randall Eubank. *Spline Smoothing and Nonparametric Regression*. 1988.

[33] B.P. Foley, D.P. Spooner, P.J. Isitt, S.A., and Jarvis G.R. Nudd. Performance prediction for a code with data-dependant runtimes. In *UK All Hands Meeting*, 2005.

[34] Ian Foster and Carl Kesselman. The globus toolkit. pages 259–278, 1999.

[35] R. F. Freund. Optimal selection theory for superconcurrency. In *Supercomputing '89: Proceedings of the 1989 ACM/IEEE conference on Supercomputing*, pages 699–703, New York, NY, USA, 1989. ACM.

[36] J.H. Friedman, F. Baskettand, and L.J. Sustek. An algorithm for finding nearest neighbors. *IEEE Transaction on computer*, 24:1000–1006, 1975.

[37] Richard Gibbons. A historical application profiler for use by parallel schedulers. In *IPPS '97: Proceedings of the Job Scheduling Strategies for Parallel Processing*, pages 58–77, London, UK, 1997. Springer-Verlag.

[38] Geki Glc. *The complete log4j manual*. QOS.ch, 2004.

[39] Dirk Gorissen. H2o metacomputing - jini lookup and discovery. Master's thesis, University of Antwerp (UA), July 2004.

[40] Palmer Grant. *Java event handling*. Prentice Hall PTR, 2001.

[41] William Grosso. *Java RMI*. 2001.

[42] H2O. http://dcl.mathcs.emory.edu/php/loadpage.php?content=/h2o/overview.php.

[43] W. Härdle. *Applied Non-Parametric Regression*. Cambridge University Press, 1990.

[44] Elliotte Rusty Harold and W. Scott Means. *XML in a Nutshell: A Desktop Quick Reference*. O'Reilly & Associates, Inc., Sebastopol, CA, USA, 2000.

[45] P. E. Hart. The condensed nearest neighbour rule. *IEEE Transaction Information Theory*, 1968.

[46] P. Hellinckx, G. Stuer, W. Hendrickx, F. Arickx, and J. Broeckhove. Grid-user driven grid research, the cobra grid. In *CCGRID '06: Proceedings of the Sixth IEEE International Symposium on Cluster Computing and the Grid (CCGRID'06)*, page 49, Washington, DC, USA, 2006. IEEE Computer Society.

[47] Peter Hellinckx, Frans Arickx, Jan Broeckhove, and Gunther Stuer. The cobra grid: a highly configurable lightweight grid. *International Journal of Web and Grid Services*, 3:267–286(20), 20 August 2007.

[48] Peter Hellinckx, Gunther Stuer, David Dewolfs, Fans Arickx, Jan Broeckhove, and Tom Dhaene. Dynamic problem-independent metacomputing characterization applied to the condor system. In *Procedings ESM2003 p.262-269*, 2003.

[49] Carl Kesselman Ian Foster and S. Tuecke. The open grid service architecture. *The Grid2: Blueprint for a New Computing Infrastructure*, pages 215–258, 2004.

[50] ISO. Itu-t rec. x.509—iso/iec 9594-8, the directory: Authentication framework. 2000.

[51] ISO. Itu-t rec. x.812—iso/iec 10181-3:1996, 1996.

[52] Michael A. Iverson, Füsun Özgüner, and Gregory J. Follen. Run-time statistical estimation of task execution times for heterogeneous distributed computing. In *HPDC '96: Proceedings of the 5th IEEE International Symposium on High Performance Distributed Computing*, page 263, Washington, DC, USA, 1996. IEEE Computer Society.

[53] Michael A. Iverson, Füsun Özgüner, and Lee C. Potter. Statistical prediction of task execution times through analytic benchmarking for scheduling in a heterogeneous environment. *IEEE Trans. Computers*, 48(12):1374–1379, 1999.

[54] W. Vanroose V.S. Vasilevsky J. Broeckhove, F. Arickx. The modified j-matrix method for short range potentials. *J. Phys. A: Gen37(2004) 1-13*, 2004.

[55] Jama. http://math.nist.gov/javanumerics/jama/.

[56] JAX-RPC. Available at https://jax-rpc.dev.java.net/.

[57] Jini. http://www.jini.org/.

[58] Joone. http://www.jooneworld.com/.

[59] Peter Kacsuk, Norbert Podhorszki, and Tamas Kiss. Scalable desktop grid system. Technical Report TR-0006, Institute on System Architecture, CoreGRID - Network of Excellence, May 2005.

[60] Ashfaq A. Khokhar, Viktor K. Prasanna, Muhammad E. Shaaban, and Cho-Li Wang. Heterogeneous supercomputing: problems and issues. In *Proc. of the 1992 Workshop on Heterogeneous Processing*, 1992.

[61] Ashfaq A. Khokhar, Viktor K. Prasanna, Muhammad E. Shaaban, and Cho-Li Wang. Heterogeneous computing: Challenges and opportunities. *Computer*, 26(6):18–27, 1993.

[62] Derrick Kondo, Michela Taufer, Charles L. Brooks III, Henri Casanova, and Andrew A. Chien. Characterizing and evaluating desktop grids: An empirical study. In *IPDPS*, 2004.

[63] Vijaykumar Krishnaswamy, Dan Walther, Sumeer Bhola, Ethendranath Bommaiah, George Riley, Brad Topol, and Mustaque Ahamad. Efficient implementations of java remote method invocation (rmi). In *COOTS'98: Proceedings of the 4th conference on USENIX Conference on Object-Oriented Technologies and Systems (COOTS)*, pages 2–2, Berkeley, CA, USA, 1998. USENIX Association.

[64] Dawid Kurzyniec, Vaidy Sunderam, and Mauro Migliardi. On the viability of component frameworks for high performance distributed computing: A case study. In *HPDC '02: Proceedings of the 11th IEEE International Symposium on High Performance Distributed Computing*, page 275, Washington, DC, USA, 2002. IEEE Computer Society.

[65] Dawid Kurzyniec, Tomasz Wrzosek, and Vaidy Sunderam. Heterogeneous access to service-based distributed computing: The rmix approach. In *IPDPS '03: Proceedings of the 17th International Symposium on Parallel and Distributed Processing*, page 100.1, Washington, DC, USA, 2003. IEEE Computer Society.

[66] Dawid Kurzyniec, Tomasz Wrzosek, Vaidy Sunderam, and Aleksander Sominski. Rmix: A multiprotocol rmi framework for java. In *IPDPS '03: Proceedings of the 17th International Symposium on Parallel and Distributed Processing*, page 140, Washington, DC, USA, 2003. IEEE Computer Society.

[67] Arnaud Legrand, Loris Marchal, and Henri Casanova. Scheduling distributed applications: the simgrid simulation framework. In *CCGRID '03: Proceedings of the 3st International Symposium on Cluster Computing and the Grid*, page 138, Washington, DC, USA, 2003. IEEE Computer Society.

[68] Minglu Li, Hui Liu, Feilong Tang, Feng Hong, Changjun Jiangand Weiqin Tong, Aoying Zhou, Yadong Gui, Hao Zhu, and Shui Jiang. Shanghaigrid in action: the first stage projects towards digital city and city grid. *International Journal of Grid and Utility Computing*, Vol. 1, No.1:pp. 22 – 31, 2005.

[69] Tim Lindholm and Frank Yellin. *Java Virtual Machine Specification*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.

[70] Michael Litzkow, Todd Tannenbaum, Jim Basney, and Miron Livny. Checkpoint and migration of UNIX processes in the Condor distributed processing system. Technical Report UW-CS-TR-1346, University of Wisconsin - Madison Computer Sciences Department, April 1997.

[71] M. L. Liu, D. Agrawal, and A. El Abbadi. The performance of two-phase commit protocols in the presence of site failures. In *FTCS-24: 24th International Symposium on Fault Tolerant Computing*, pages 234–243, Austin, Texas, 1994. IEEE Computer Society Press.

[72] Ram Kumar Luc Clement, Tony Rogers. Oasis uddi specification tc.

[73] Jason Maassen, Rob Van Nieuwpoort, Ronald Veldema, Henri E. Bal, Thilo Kielmann, Ceriel J. H. Jacobs, and Rutger F. H. Hofman. Efficient java rmi for parallel programming. *ACM Trans. Program. Lang. Syst.*, 23(6):747–775, 2001.

[74] Jason Maassen, Rob van Nieuwpoort, Ronald Veldema, Henri E. Bal, and Aske Plaat. An efficient implementation of java's remote method invocation. In *Principles Practice of Parallel Programming*, pages 173–182, 1999.

[75] Brett McLaughlin. *Java & XML*. O'Reilly & Associates, Inc., Sebastopol, CA, USA, 2001.

[76] James W. Mickens and Brian D. Noble. Exploiting availability prediction in distributed systems. In *NSDI'06: Proceedings of the 3rd conference on Networked Systems Design & Implementation*, pages 6–6, Berkeley, CA, USA, 2006. USENIX Association.

[77] Sun Microsystems. Dynamic proxy classes: Available at http://java.sun.com/j2se/1.5.0/docs/guide/reflection/proxy.html.

[78] Sun Microsystems. Getting started using rmi: Available at http://java.sun.com/j2se/1.4.2/docs/guide/rmi/getstart.doc.html.

[79] Sun Microsystems. Java bean specification : Available at http://java.sun.com/products/javabeans/docs/spec.html.

[80] Sun Microsystems. Java remote method invocation specification: Available at http://java.sun.com/j2se/1.5.0/docs/guide/rmi/spec/rmitoc.html.

[81] Sun Microsystems. Java rmi over iiop : Available at http://java.sun.com/products/rmi-iiop/.

[82] Sun Microsystems. Onc-rpc specification : Available at http://www.ietf.org/rfc/rfc1831.txt.

[83] M. Migliardi and V. Sunderam. The harness metacomputing framework. 1999.

[84] Svetozar Misljencevic. Jini service container. Master's thesis, University of Antwerp, 2006.

[85] Douglas C. Montgomery. *Design and Analysis of Experiments*. John Wiley & Sons, 2006.

[86] Tai-Yeon Ku; Dong-Hwan Park; Keong-Deok Moon. A java-based architecture supporting ieee 1394 for home entertainment network. *Consumer Electronics, 2002. ICCE. 2002 Digest of Technical Papers. International Conference on*, pages 108–109, 2002.

[87] Silas De Munck. Parameter sweep predictie. Master's thesis, Universiteit Antwerpen, 2007.

[88] Farrukh Nadeem, Muhammad Murtaza Yousaf, Radu Prodan, and Thomas Fahringer. Soft benchmarks-based application performance prediction using a minimum training set. In *E-SCIENCE '06: Proceedings of the Second IEEE International Conference on e-Science and Grid Computing*, page 71, Washington, DC, USA, 2006. IEEE Computer Society.

[89] Christian Nester, Michael Philippsen, and Bernhard Haumacher. A more efficient rmi for java. In *JAVA '99: Proceedings of the ACM 1999 conference on Java Grande*, pages 152–159, New York, NY, USA, 1999. ACM.

[90] Scott Oaks. *Java security.* O'Reilly & Associates, Inc., Sebastopol, CA, USA, 1998.

[91] Scott Oaks. *Java security.* Second edition, 2001. Covers JAAS and JSEE; Writing and deploying secure applications; Covers Java 1.1, Java 2, and JCE 1.2.1.

[92] ORACLE, Redwood City, CA:Oracle. *ORACLE7 Server Concept Manual(6693-70-1292)*, 1992.

[93] Daniel Pease, Arif Ghafoor, Ishfaq Ahmad, David L. Andrews, Kamal Foudil-Bey, Thomas E. Karpinski, Mohammad A. Mikki, and Mohamed Zerrouki. Paws: A performance evaluation tool for parallel computing systems. *Computer*, 24(1):18–29, 1991.

[94] Aske Plaat, Henri E. Bal, and Rutger F. H. Hofman. Sensitivity of parallel applications to large differences in bandwidth and latency in two-layer interconnects. *Future Gener. Comput. Syst.*, 17(6):769–782, 2001.

[95] PrimeGrid. Available at http://www.primegrid.com/.

[96] Erik T. Ray and Christopher R. Maden. *Learning XML.* O'Reilly & Associates, Inc., Sebastopol, CA, USA, 2001.

[97] Brian Reistad and David K. Gifford. Static dependent costs for estimating execution time. *SIGPLAN Lisp Pointers*, VII(3):65–78, 1994.

[98] X. Ren, S. Lee, R. Eigenmann, and S. Bagchi. Resource availability prediction in fine-grained cycle sharing systems. In *Proceedings of the Conference on High Performance Distributed Computing*, 2006.

[99] Linda Rising. *Patterns Handbook: Best Practices*, chapter A Pocket-Sized Broker, pages 171–180. Cambridge University Press, New York, NY, USA, 1997.

[100] RMI. http://java.sun.com/javase/technologies/core/basic/rmi/index.jsp.

[101] William Ruh, Thomas Herron, and Paul Klinker. *IIOP Complete: Understanding CORBA and Middleware Interoperability*. 2000.

[102] Seti@Home. Available at http://setiathome.ssl.berkeley.edu.

[103] Inderjeet Singh, Sean Brydon, Greg Murray, Vijay Ramachandran, Thierry Violleau, and Beth Stearns. *Designing Web Services with the J2EE 1.4 Platform: JAX-RPC, XML Services, and Clients*. Pearson Education, 2004.

[104] A. Slominski, M. Govindaraju, D. Gannon, and R. Bramley. Design of an XML based Interoperable RMI System : SoapRMI C++/Java 1.1. In *Proceedings of PDPTA*, pages 1661–1667, June 25-28, 2001.

[105] L. Smarr and C.E. Catlett. Metacomputing. *Communications of the ACM*, 35(6):45–52, 1992.

[106] Warren Smith. Prediction services for grid computing. In *Proceedings of the 22nd IEEE International Parallel and Distributed Processing Symposium*, 2007.

[107] Warren Smith, Ian T. Foster, and Valerie E. Taylor. Predicting application run times using historical information. In *IPPS/SPDP '98: Proceedings of the Workshop on Job Scheduling Strategies for Parallel Processing*, pages 122–142, London, UK, 1998. Springer-Verlag.

[108] Warren Smith and Parkson Wong. Resource selection using execution and queue wait time predictions. Technical report, NASA Ames Research Center, 2002.

[109] James Snell, Doug Tidwell, and Pavel Kulchenko. *Programming Web services with SOAP*. O'Reilly & Associates, Inc., Sebastopol, CA, USA, 2002.

[110] Marc Snir and Steve Otto. *MPI-The Complete Reference: The MPI Core*. MIT Press, Cambridge, MA, USA, 1998.

[111] G. Stuer, K. Vanmechelen, and J. Broeckhove. A commodity market algorithm for pricing substitutable grid resources. *Future Generation Computer Systems*, 23(5):688–701, 2007.

[112] Vaidy S. Sunderam and Dawid Kurzyniec. Lightweight self-organizing frameworks for metacomputing. In *HPDC*, pages 113–, 2002.

[113] Douglas Thain, Todd Tannenbaum, and Miron Livny. Condor and the grid. In Fran Berman, Geoffrey Fox, and Tony Hey, editors, *Grid Computing: Making the Global Infrastructure a Reality*. John Wiley & Sons Inc., December 2002.

[114] Matlab Neural Network Toolbox. http://www.mathworks.com/products/neuralnet/.

[115] Kim Topley. *Java Web Services in a Nutshell*. O'Reilly & Associates, Inc., Sebastopol, CA, USA, 2003.

[116] K. Vanmechelen and J. Broeckhove. A comparative analysis of single-unit vickrey auctions and commodity markets for realizing grid economies with dynamic pricing. In J. Altmann and D.J. Veit, editors, *Proceedings of the 4th International Workshop on Grid Economics and Business Models (GECON 2007)*, volume 4685 of *Lecture Notes in Computer Science*, pages 98–111, Heidelberg, 2007. Springer-Verlag.

[117] Kurt Vanmechelen, Wim Depoorter, and Jan Broeckhove. Economic grid resource management for CPU bound applications with hard deadlines. In *Proceedings of CCGrid 2008*, pages 258–266. IEEE Computer Society, 2008.

[118] Kurt Vanmechelen, Wim Depoorter, and Jan Broeckhove. A simulation framework for studying economic resource management in grids. In Marian Bubak, G. Dick van Albada, Jack Dongarra, and Peter M. A. Sloot, editors, *ICCS (1)*, volume 5101 of *Lecture Notes in Computer Science*, pages 226–235. Springer, 2008.

[119] V. Vasilevsky, A. V. Nesterov, F. Arickx, and J. Broeckhove. Algebraic model for scattering in three-s-cluster systems. i. theoretical background. *Physical Review C*, 63:034606, 2001.

[120] V. Vasilevsky, A. V. Nesterov, F. Arickx, and J. Broeckhove. Algebraic model for scattering of three-s-cluster systems. ii. resonances in the three-cluster continuum of 6he and 6be. *Physical Review C*, 63:034607, 2001.

[121] Sam Verboven. Commodity market-based resource allocation in cobra. Master's thesis, University of Antwerp, 2007.

[122] Sam Verboven, Peter Hellinckx, Frans Arickx, and Jan Broeckhove. Dynamic grid scheduling using job runtime requirements and variable resource availability. In *Proceedings of EuroPar 2008*, 2008.

[123] Aaron E. Walsh, editor. *Uddi, Soap, and Wsdl: The Web Services Specification Reference Book*. Prentice Hall Professional Technical Reference, 2002.

[124] Richard Wolski. Dynamically forecasting network performance using the network weather service. *Cluster Computing*, 1(1):119–132, 1998.

[125] Jaehyung Yang, Ishfaq Ahmad, and Arif Ghafoor. Estimation of execution times on heterogeneous supercomputer architectures. In *ICPP '93: Proceedings of the 1993 International Conference on Parallel Processing*, pages 219–226, Washington, DC, USA, 1993. IEEE Computer Society.

[126] T. Yang and A. Gerasoulis. DSC: Scheduling parallel tasks on an unbounded number of processors. Technical Report TRCS94-12, University of California at Santa Barbara, 20, 1994.

[127] Yanyong Zhang, Mark S. Squillante, Anand Sivasubramaniam, and Ramendra K. Sahoo. Performance implications of failures in large-scale cluster scheduling. In *Lecture Notes in Computer Science*, 2005.