Agentbaseret ressourceallokering

"Agent-based Resource Allocation"

Nicolai Graff Andersen
Michael Bigom Herold

LYNGBY 2002
EKSAMENSPROJEKT
NR. 56/02

IMM

Forord

Denne rapport præsenterer resultaterne af vores eksamensprojekt ved Institut for Informatik og Matematisk Modellering (IMM), Danmarks Tekniske Universitet (DTU).

Projektet indgår som den afsluttende opgave af civilingeniørstudiet. Projektet er lavet i perioden 1. februar til 30. august 2002.

Vejleder ved projektet har været Robin Sharp.

Vi vil gerne benytte lejligheden til at takke Robin Sharp for hans tålmodighed og gode vejledning gennem hele projektforløbet.

Lyngby d. 30. august 2002

Nicolai Graff Andersen

Michael Bigom Herold

Abstract

I et globalt computernet er der et stort potentiale for, at man kan dele ubenyttede ressourcer med hinanden. Med et system, som kan håndtere ressourceallokering i et globalt computernet, vil man have muligheden for, at man fra en ganske almindelig arbejdsstation vil have adgang til enorme mængder ressourcer.

Med denne afhandling undersøger vi mulighederne for, at lave et globalt ressourceallokeringssystem. Ud fra forskellige analyser, udvikler vi en model, hvor der laves en hierarkisk loadbalancering mellem eksisterende lokale ressourceallokeringssystemer. Modellen afviger ikke bare fra traditionelle centraliserede ressourceallokeringssystemer, men er også meget forskellig fra andre hierarkiske modeller.

I vores model tages der højde for, at opgaver har forskellige ressourcekrav, og at computere har forskellige ressourceudbud. Vi begrænser modellen ved at betragte alle opgaver som uafhængige af hinanden. I modellen laves der multiple loadbalanceringer. Dette gøres, fordi der indenfor opgaver med bestemte ressourcebehov kan være en overbelastning, mens der for opgaver med andre ressourcebehov ikke nødvendigvis er en overbelastning.

Vi implementerer en prototype af modellen og viser gennem forsøg at modellen opfører sig som forventet. Desuden beskriver vi, hvordan vores model vil kunne anvendes i *The Globus project*.

English version:

In a global computer network there is great potential of sharing unused computer resources. Having a system which can handle the resource allocation in a global computer network will give the opportunity of gaining access to an enourmeous amount of resources from your workstation.

In this thesis we will investigate the opportunities of making such a global resource allocation system. Through analysis we develop a model in which a hierarchical load balancing between existing local resource allocation systems come into existence. This is not only different from traditional resource allocation systems but also different from other hierarchical approaches of models.

In our model we take into account that tasks have individual resource demands tasks, and that computers have different resource supplies. We limit our model by looking at all tasks as independent ones. In our model multiple load balancing is necessary beacuse tasks with specific resource demands may be in an overloaded state while tasks with other demands are not.

We implement a prototype of the model and show through tests that the model acts as expected. We also describe how our model can be used in *The Globus project*.

Keywords: Resource allocation, hierarchical loadbalancing, grid computing, Linda

Indhold

1 Indledning **1**
- 1.1 Begreber . . . 1
- 1.2 Relateret arbejde . . . 2
- 1.3 Problemformulering . . . 3
- 1.4 Rapport struktur . . . 3

2 Ressourceallokering **5**
- 2.1 Ressourcer, arbejdere og opgaver . . . 5
- 2.2 Optimeringsstrategi . . . 6
- 2.3 Schedulering . . . 9
- 2.4 Global optimering . . . 10
- 2.5 Vores model . . . 13

3 Hierarkisk loadbalancering **15**
- 3.1 Generel model . . . 16
- 3.2 Information i hierarkiet . . . 17
  - 3.2.1 Tidsforsinkelse . . . 18
  - 3.2.2 Kompleksitet . . . 18
- 3.3 Belastningstilstande . . . 18
- 3.4 Opgaver stilles i bunden . . . 19
- 3.5 Model 1 - Simpel loadbalancering . . . 20
  - 3.5.1 Model eksempel . . . 22
  - 3.5.2 Vurdering af Model 1 . . . 24
- 3.6 Model 2 - Opgaver og arbejdere med forskellige størrelser . . . 24
  - 3.6.1 Belastningstilstande og skemalægning . . . 26
  - 3.6.2 Informationsniveau . . . 27
  - 3.6.3 Eksempel på skemalægning . . . 29
  - 3.6.4 Håndtering af tiden . . . 31
  - 3.6.5 Modellen . . . 31
  - 3.6.6 Vurdering af model 2 . . . 35
- 3.7 Model 3 - Opgaver og arbejdere med forskellige krav/udbud . . . 35
  - 3.7.1 Krav og udbud . . . 36
  - 3.7.2 Gruppering af opgaver . . . 37
  - 3.7.3 Flere loadbalanceringer . . . 38
  - 3.7.4 Aktive opgavegrupper . . . 39
  - 3.7.5 Åbning og lukning af aktive opgavegrupper . . . 41
  - 3.7.6 Arbejdere kan løse flere forskellige opgaver . . . 42
  - 3.7.7 Belastningstilstande og skemalægning . . . 43

3.7.8 Informationsniveau . . . 44
3.7.9 Modellen . . . 45
3.7.10 Vurdering af model 3 . . . 49
3.8 Udvidet model 3 . . . 50
3.8.1 Kontrolleret flytning af opgaver . . . 50
3.8.2 Forbedret fordeling af opgaver . . . 52

4 Implementering 55
4.1 Teknologi . . . 55
4.2 Linda . . . 56
4.2.1 Linda varianter . . . 57
4.2.2 Ressourcer, opgaver og Linda . . . 58
4.2.3 Krav til tupelspace . . . 59
4.2.4 Linda implementering . . . 60
4.3 Opgaver og arbejdere . . . 62
4.4 Kommunikation . . . 63
4.5 Testmiljø . . . 63
4.5.1 Simulator . . . 64
4.5.2 Dataindsamling og -behandling . . . 65
4.6 Implementering af modeller . . . 65
4.6.1 Generel model . . . 67
4.6.2 Model 1 - simpel load balancering . . . 71
4.6.3 Model 2 - udvidet load balancering . . . 71
4.6.4 Model 3 - kompleks load balancering . . . 72
4.7 Afrunding . . . 72

5 Resultater 73
5.1 Fremgangsmåde . . . 73
5.2 Model 1 . . . 76
5.3 Model 2 . . . 77
5.4 Model 3 . . . 80
5.5 Vurdering af resultater . . . 85

6 Globus integration 87

7 Konklusion 91
7.1 Vurdering af model . . . 92
7.2 Fremtidigt arbejde . . . 92

A Formalisering af tupelspace i RAISE 95
A.1 Base Linda . . . 95
A.2 Datastorage . . . 98
A.3 Linda . . . 100

B Implementering 105
B.1 Model 1 - simpel load balancering . . . 105
B.2 Model 2 - udvidet load balancering . . . 107
B.3 Model 3 - kompleks loadbalancering . . . 112

C Forsøgsresultater **119**
C.1 Forsøg 1-1 119
C.2 Forsøg 1-2 123
C.3 Forsøg 2-1 126
C.4 Forsøg 2-2 128
C.5 Forsøg 2-3 129
C.6 Forsøg 3-1 133
C.7 Forsøg 3-2 138
C.8 Forsøg 3-3 140
C.9 Forsøg 3-4 141
C.10 Forsøg 3-5 146

D Indhold af medfølgende CD-rom **149**

Kapitel 1

Indledning

Nutidens store udbredelse af informationsteknologi betyder, at næsten alle computere er forbundet med hinanden via højhastighedsdatanet. Herigennem er der skabt stor mulighed for, at dele computernes ressourcer på tværs af sågar store fysiske afstande. I dag er de fleste computere ret kraftige og står faktisk ubenyttede hen en stor del af tiden, som f.eks. almindelige arbejdsstationer. Hvis man opsummerer al den ubenyttede computerkraft, så er der tale om en enorm mængde. Systemer, som kan udnytte disse ubenyttede ressourcer, er derfor meget interessante.

Et af de grundlæggende problemer i sådanne systemer er, at udtænke hvordan man får distribueret opgaver ud til ubenyttede maskiner. Herunder også, hvor og hvornår opgaverne udføres optimalt. Det skal tages med i betragtning, at ikke alle opgaver kan løses på alle maskiner. Opgaver stiller krav til hvilke ressourcer de arbejdende maskiner skal stille til rådighed. Disse problemer falder ind under begrebet *ressourceallokering*.

Man betegner systemer, hvor der kan distribueres opgaver ud på et stort (globalt) netværk af computere, for *grid computing*. Grid computing dækker over mange emner og indeholder aspekter som sikkerhed, ressourceallokering, datadistribution osv. Ordet *grid* stammer oprindeligt fra det elektriske forsyningsnet, hvor elektricitetsværkerne udveksler strøm med hinanden, når der i et område er en overproduktion af strøm, mens der et andet sted er et underskud. Set fra en brugers synspunkt, så er han/hun ligeglad med, hvor elektriciteten kommer fra. Brugeren forventer bare, at når han/hun sætter en lampe i stikkontakten, så kommer der lys og så er det ligemeget om strømmen kommer fra det ene kraftværk eller det andet. Grid computing har mange paralleller til det elektriske forsyningsnet[8]. Her gælder det også om, at når man tilkobler sin computer til et datanet, så får man adgang til de mængder ressourcer, man måtte have brug for. Ligesom med elektriciteten er man ligeglad med, hvor ressourcerne fysisk er placeret.

Dette projekt fokuserer på ressourceallokering i meget store heterogene computernet.

1.1 Begreber

Vi vil nu kort beskrive nogle grundlæggende begreber. Begreberne vil blive forklaret yderligere i kapitel 2.

Ressourceallokering: Begrebet dækker over det, at *allokere* opgaver til maskiner. *Allokere* betyder, at man skal finde en måde, hvorpå man skal fordele en mængde opgaver på en mængde maskiner. Der skal specielt tages højde for, at en opgave ikke kan udføres af alle maskiner, da opgaver har *ressourcekrav* og maskiner har *ressourceudbud*.

Ressourcemanager: En *ressourcemanager* er en entitet, som sørger for ressourceallokering i et afgrænset område.

Loadbalancering: Loadbalancering er en teknik, hvor man prøver, at fordele opgaver mellem computere således, at der ikke er computere, der er frie, mens andre er overbelastede. Typisk vil en loadbalancering virke ved, at man flytter opgaver fra overbelastede computere, til computere, som er har frie ressourcer.

1.2 Relateret arbejde

I det følgende gennemgås teori og praksis ved nogle eksisterende arbejder indenfor ressourceallokering i globale systemer. Herefter opsummeres og der følger en kort beskrivelse af, hvad vi bruger og hvordan vi afviger fra de beskrevne eksisterende arbejder.

Rainer Pollak[18] foreslår en hierarkisk loadbalanceringsmodel kaldt *PaLaBer*. Hierarkiets blad-komponenter repræsenterer hver især en *maskine*. De resterende knudepunkter i hierarkiet har til opgave at sende information om *overbelastede* blade, enten til et af dets egne *underbelastede* børn eller til forælderknudepunktet. Når en underbelastet bladkomponent modtager information om en overbelastet, så kommunikerer de to for at lave en migræring af en opgave. Nye opgaver ligger i en kø i roden af hierarkiet, og når roden kan se at den generelle belastningstilstand i træet kommer under et vist niveau, så sendes opgaverne afsted til de mindst belastede blade. Roden af træet sørger for, ud fra belastningsinformation sendt op gennem træet, at bestemme tærskelværdier for hvornår et knudepunkt er underbelastet og hvornår det er overbelastet. Disse værdier distribueres hele tiden ud til alle knudepunkterne i træet. I modellen tages ikke højde for, at opgaver stiller krav til de maskiner de skal udføres på, idet det tages for givet, at alle opgaver kan udføres på alle maskiner. Et andet problem ved modellen er, at alle nye opgaver ankommer og ligger i kø samme sted, hvilket gør knudepunktet til en flaskehals i store systemer.

The Globus Project[6][7] er et projekt, hvor man har defineret nogle rammer for hvordan et grid bygges op, samt hvilke komponenter, det skal indeholde. Globus indeholder et meget bredt aspekt af grid computing, f.eks. sikkerhed og ressourceallokering. I et globus grid inddeles maskiner i grupper. I hver enkelt gruppe bruges en lokal ressourcemanager til ressourceallokering i gruppen. Alle grupperne registrerer information om deres maskiner i en distribueret global informationsservice kaldt MDS[5]. Informationen indeholder oplysninger om hvilket udbud de forskellige maskiner har af ressourcer. Når nye opgaver skal allokeres i systemet, så gøres dette ved, at man laver en søgning i MDS'en og finder en gruppe, som opfylder de ønsker og krav opgaverne måtte have. Herefter sendes en forespørgsel til gruppens ressourcemanager med et ønske om, at opgaverne bliver udført. Det er så ressourcemanageren, der sørger for at opgaverne bliver allokeret til maskinerne i gruppen. Globus er et meget skalerbart system, men der er ikke indbygget nogle former for optimering mellem grupperne i systemet.

Condor[15] er et normalt *lokalt* ressourceallokeringssystem, med en udvidelse kaldet *Condor-G*[11]. Med denne udvidelse kan Condor bruges i Globus. Condor-G benyttes til at opbygge virtuelle grupper, hver med sit eget ressourceallokeringssystem. Maskinerne i gruppen findes ved søgninger i Globus' MDS og allokering af opgaver til maskinerne foregår ved, at der bliver installeret en *Condor deamon* på maskinen via almindelig Globus allokering. Condor-G giver mulighed for, at man kan have virtuelle organisationer distribueret over store afstande, men stadig se det som et system. Condor-G laver optimering mellem de maskiner som er del af den virtuelle gruppe. Men grupperne kan ikke skalere i størrelse til et globalt system og der er ingen optimeringer mellem flere Condor-G grupper.

Prophet[28][29] er også et *lokalt* ressourceallokeringssystem som Condor, men her har man indbygget muligheden for et samarbejde mellem flere Prophet-grupper. Dette foregår ved, at når en Prophet-gruppe har for mange opgaver, så migræres nogle af disse til andre Prophet-grupper, som ikke har problemer med overbelastning. På denne måde indføres der mulighed for at udjævne overbelastninger over et større arbejdsområde. I Prophet virker dette ved, at hver Prophet-gruppe kender et antal andre grupper, hvorimellem der kan foregå sådanne migræringer. Dette giver sub-globale optimeringer, men ikke direkte globale optimeringer.

Det er et generelt kendetegn ved ressourceallokeringssystemer som Globus og Condor, at de ikke indeholder nogen former for globale optimeringer. Gennem projektet undersøges mulighederne for at opbygge et ressourceallokeringssystem, hvor der bruges loadbalanceringer til at lave globale optimeringer. Vi tager endvidere højde for, at maskiner har forskellige ressourceudbud, og at opgaver har forskellige ressourcekrav. Principperne i loadbalanceringerne minder om de overordnede principper i PaLaBer, men som sagt tager vi højde for maskiners ressourceudbud og opgavers ressourcekrav, som det f.eks. gøres i Globus og Condor.

1.3 Problemformulering

Ressourceallokering i et globalt computernet er ikke en triviel opgave. Når man skal finde maskiner, som kan løse en opgave, bruger mange systemer i dag en *first fit* strategi, således at opgaven afleveres til den/de maskiner som er *tættest på* og er villige til at løse den. Dette er ikke nødvendigvis en optimal strategi.

I dette projekt vil vi undersøge, hvordan ressourceallokering i et globalt system kan udføres på en mere optimal måde. Vi undersøger muligheden for, at lave en global loadbalancering i et system med meget heterogene maskiner og opgaver. Vi vil beskrive simple og mere komplekse modeller og implementere flere af disse. Modellerne vil blive undersøgt ud fra nogle forsøg, som kommenteres nærmere.

1.4 Rapport struktur

Kapitel 2: Her vil vi introducere og analysere nogle af de vigtige begreber indenfor grid computing, herunder specielt ressourceallokering og global optimering. I kapitlet

introducerer vi ligeledes læseren for nogle definitioner og nogle begrænsninger, som bruges gennem resten af rapporten.

Kapitel 3: Indholdet er en teoretisk gennemgang af de ressourceallokeringsmodeller, som vi har lavet. Kapitlet indeholder både analyser og løsningsforslag til forskellige problemstillinger. Der opbygges flere modeller, hvor hver enkelt er en videreudvikling af en tidligere model. Ideen til udviklingen er, at der gradvist indføres flere og flere af løsningsforslagene til problemstillingerne.

Kapitel 4: Her beskrives, hvordan modellerne er blevet implementeret. Vi begynder med at designe og udvikle et *shared dataspace* (Linda), som bruges i alle modeller. Herefter beskriver vi hvordan modellerne er opbygget. Først med en generel implementering, som er grundlaget for alle modellerne. Herefter beskrives hvordan hver enkelt model afviger fra denne. I kapitlet beskriver vi desuden, hvordan vi har konstrueret et værktøj, der løbende bruges til analyse af kørende modeller.

Kapitel 5: Vi har lavet flere forsøg med de forskellige implementerede modeller. Dette kapitel beskriver og forklarer forsøgene nærmere. Vi viser at de forskellige principper fra teorien virker og demonstrerer også nogle af modellernes fejl.

Kapitel 6: Dette kapitel indeholder en beskrivelse af, hvordan vores modeller kan integreres og bruges i systemet *The Globus project*. Vi beskriver den generelle modulopbygning i Globus og viser hvordan vores modeller kan indsættes som nye moduler.

Kapitel 7: Rapporten afsluttes med en opsummering af vores arbejde. Endvidere følger en beskrivelse af nye områder, hvor vores arbejde med fordel kan udvides.

Kapitel 2

Ressourceallokering

I dette kapitel vil vi se nærmere på begrebet ressourceallokering. Vi vil endvidere definere nogle af de begreber vi arbejder med (afsnit 2.1), samt introducere den model, som danner grundlag for de resterende kapitler i rapporten (afsnit 2.5).

Vi definerer ressourceallokering som det at allokere en mængde opgaver til en mængde maskiner. Målet med ressourceallokeringen er at få løst opgaverne på de tilgængelige maskiner på en så *optimal* måde som muligt. Hvad der er optimalt, afhænger af problemets indgangsvinkel. Eksempelvis kunne man ønske, at alle opgaver udføres *hurtigst* muligt (minimering af den samlede udførelsestid). Optimalitetsbegrebet vil blive diskuteret i afsnit 2.2.

Der findes i dag en bred vifte af *scheduleringssystemer*, som kan allokere opgaver til maskiner. Disse systemer tager i deres allokering typisk hensyn til alle tilgængelige maskiner samt til en større mængde opgaver. I afsnit 2.3 vil vi beskrive scheduleringssystemer generelt, samt komme med eksempler på sådanne.

Når antallet af maskiner og opgaver bliver meget stort, vil det typiske scheduleringssystem ikke være tilstrækkeligt. Det vil ikke være muligt at tage alle maskiner og opgaver med i hver enkelt beslutningsproces. I stedet må der vælges nogle mere simple principper til at løse allokeringsproblemet. I litteraturen (f.eks. Prophet [28][29]) er en typisk løsning at opdele maskinerne i store grupper, således at schedulering stadig er mulig i hver enkelt gruppe, og så lave en koordinering mellem grupperne. Koordineringen består i at flytte opgaver fra grupper, som er overbelastede, til grupper med frie maskiner. Vi kalder den optimering, som foregår indenfor en mindre gruppe af maskiner for *lokal* optimering, og koordineringen mellem grupperne for *global* eller *subglobal* optimering. I afsnit 2.4 vil vi beskrive nogle af principperne ved global optimering mere udførligt.

2.1 Ressourcer, arbejdere og opgaver

I dette afsnit vil vi se på nogle af de egenskaber, der kendetegner opgaver og ressourcer. Begrebet *en arbejder* vil ligeledes blive introduceret.

Ressourcer og arbejdere

Vi definerer en ressource som værende en facilitet, som skal bruges af en opgave. En ressource kan f.eks. være CPU, RAM, disk, operativsystem eller en specifik database adgang. Til en ressource kan der være knyttet en numerisk værdi, som indikerer størrelsen på ressourcen (eksempelvis vil tallet 128 for ressourcen RAM kunne indikere, at der er 128MB RAM tilgængelig). Nogle ressourcer, som eksempelvis disk, vil naturligt kunne beskrives ved flere numeriske værdier (tilgængelig diskplads, læse/skrive hastighed osv.). Vi har valgt at se disse forskellige parametre som enkeltstående ressourcer. I eksemplet med disk ressourcen vil der både kunne være en ressource, som beskriver den tilgængelige mængde disk plads, samt en ressource der beskriver læse/skrive hastigheden til disken. Nogle ressourcer vil have et diskret domæne (eksempelvis operativsystem: Linux, SunOS, Windows osv.), mens andre ressourcer vil have et kontinuert domæne (som f.eks. mængden af fri RAM og disk).

Vi bruger begrebet *arbejder* om de entiteter, der er i besiddelse af en eller flere ressourcer. En arbejder kan eksempelvis være en almindelig pc, en supercomputer eller en klynge af maskiner, der set fra omverdenen danner en enkelt virtuel maskine.

Opgaver

En opgave er et stykke kode, som skal udføres, samt evt. noget data, der skal bruges til udførelsen. Til en opgave kan der være knyttet forskellige krav om ressourcebehov, som skal være tilstede for, at opgaven kan udføres (eksempelvis operativsystem=Linux, RAM=100MB osv.).

CPU ressourcen

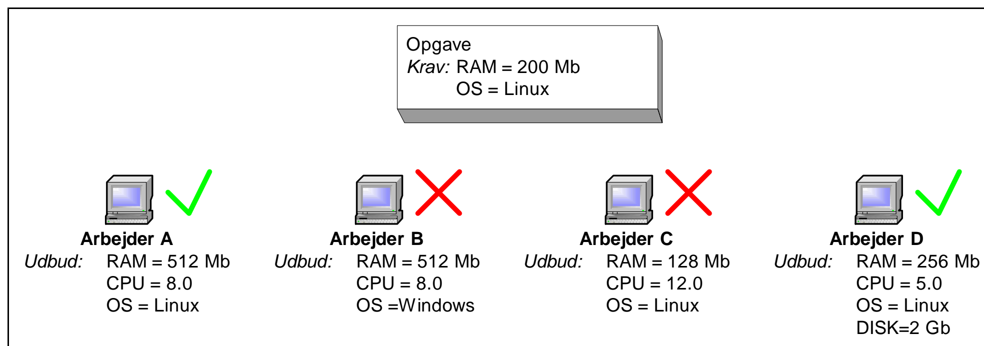
CPU ressourcen behandles i vores model lidt anderledes end de resterende ressourcer. Vi tager for givet, at alle arbejdere stiller CPU til rådighed, og at alle opgaver skal bruge noget CPU. Vi introducerer begrebet *CPUenheder*, som enhed for CPU ressourcen. En opgave antager vi skal bruge en *mængde* CPUenheder, for at blive løst, og en arbejder stiller en mængde CPUenheder til rådighed *pr. sekund*.

Match

En opgave og en arbejder siges at *matche*, hvis alle opgavens krav kan opfyldes af arbejderens ressourceudbud. Arbejderen må naturligvis gerne tilbyde flere og bedre ressourcer end dem, som opgaven skal bruge. På figur 2.1 ses et eksempel med en enkelt opgave og 4 arbejdere. Opgaven matcher arbejder A og D, men ikke B (forkert operativsystem) og C (for lidt RAM).

2.2 Optimeringsstrategi

Som nævnt er der mange forskellige måder, hvorpå man kan lave en optimal ressourceallokering. Hvilken optimeringsstrategi man vælger, afhænger bl.a. af hvor mange opgaver



Figur 2.1: Eksempel på en match mellem opgaver og arbejdere.

der skal allokeres, hvordan den indbyrdes relation er mellem opgaverne, samt hvilken grad af *ejerskab* man har over arbejderne. Nedenfor er listet nogle af de typiske optimerings strategier:

1. Uddele opgaver til ubenyttede arbejdere for at maksimere udnyttelsen af hardware.
2. Uddele opgaver så hver enkelt opgave udføres på den bedst mulige arbejder. Hvor *bedst* betyder den arbejder, som kan udføre den konkrete opgave hurtigst.
3. Uddele opgaver således, at den samlede udførelsestid for en gruppe af opgaver minimeres. Denne metode vil typisk gøre sig gældende, hvis de enkelte opgaver er del af en større applikationer.
4. Uddele opgaver således, at den totale opgavemængde udføres hurtigst muligt.
5. Uddele opgaver således, at opgaverne udføres i en prioriteret rækkefølge.
6. Uddele opgaver således, at ældste opgaver udføres først (en afledning af 5).
7. Uddele opgaver, således at *kostprisen* for den enkelte opgave (eller en opgavegruppe) minimeres. Denne metode vil typisk gøre sig gældende i systemer, hvor der er knyttet en pris for brugen af arbejdere/ressourcer.

Man vil typisk arbejde med strategier, der er en kombination af nogle af de ovenstående. Eksempelvis vil man ofte inkludere lidt af punkt 6 for at undgå, at enkelte opgaver kan blive udsultet. Flere af de ovenstående metoder vil dog kunne konflikte med hinandens mål. Det vil derfor være fornuftigt at have en indbyrdes prioritering af punkterne. Et eksempel kunne være en kombination af 1, 2, 5 og 6 som i VCE [23]. Her foreslås, at punkterne prioriteres, således at man først og fremmest prøver at udnytte alle tilgængelige ressourcer. Herefter tages højde for alder og prioritet, hvorefter man tilsidst søger at optimere efter punkt 2.

Minimering af udførelsestid

Punkt 2, 3 og 4 drejer sig alle om at minimere udførelsestiden på nogle opgaver. For at kunne lave en optimering af udførelsestiden, er det nødvendigt at kunne forudsige,

hvor lang tid det tager at udføre de enkelte opgaver. Dette afhænger selvfølgelig af hvilken arbejder, opgaven udføres på, så det er nødvendigt at kunne lave et estimat på udførelsestiden for en given opgave på en given arbejder.

Udførelsestiden for en opgave kan afhænge af mange ting, såsom CPU og IO hastigheden hos den arbejder opgavens udføres hos. For simplicitetens skyld har vi dog valgt at se bort fra alle andre bidrag end CPU. De krav en opgave har til en arbejder, har i det system vi kigger på, kun indflydelse på om opgaven kan løses hos en given arbejder eller ej, og ikke på hvor hurtigt den evt. vil kunne løses.

Hvis man ser på en enkelt opgave, så er det rimeligt simpelt at minimere den forventede udførelsestid ved simpelt at vælge den af de tilgængelige arbejdere, der forventes at kunne løse opgaven hurtigst. Ser man derimod på en større mængde af opgaver, så kompliceres optimeringen. En arbejder, der er del af den optimale løsning for én opgave, kan også være del af den optimale løsning for andre opgaver. I et sådan tilfælde vil det ikke være muligt at finde den bedste løsning, set fra hver enkelt opgaves synspunkt.

En opgave kan også være en del af en større opgavemængde. Herved er det ikke optimeringen af den enkelte opgave det handler om. I stedet handler det om at få udført den samlede opgavemængde optimalt. En opgavemængde kan være mange ting, som f.eks.:

- en mængde opgaver tilhørende én *single program multiple data* (SPMD) applikation.
- en mængde opgaver tilhørende én *paralel pipeline* (PP) applikation.
- en mængde af uafhængige opgaver.
- en kombination af ovenstående, dvs. en mængde som består af både SPMD og PP applikationer samt uafhængige opgaver.
- osv.

Har man f.eks. en SPMD applikation, som består af mange opgaver, så vil en optimering af udførelsestiden virke ved, at man minimerer tiden, indtil *alle* opgaverne i applikationen er udført (det tidspunkt hvor den sidste opgave er løst) [28, 29].

Optimeringsproblemet er sværere at definere, når man iagttager en gruppe opgaver bestående af flere applikationer og flere enkeltstående opgaver. Man kan vælge at se på hele gruppen og optimere den samlede udførelsestid. Dette er klart den mest simple løsning, men det betyder, at der ikke tages hensyn til de enkelte applikationers præferencer. Opgaver tilhørende samme applikation vil nemt kunne blive udført over et alt for langt tidsrum. Set fra de enkelte applikationer vil dette ikke være en optimal løsning.

Det er også vigtigt at undersøge, hvordan antallet af opgaver forholder sig i forhold til antallet af arbejdere. Eller set på en anden måde: Er det et allokeringsproblem, hvor man har mange opgaver, som skal udføres på få arbejdere, eller er der mange arbejdere, som skal løse få opgaver. I det første tilfælde vil en optimal løsning f.eks. kunne opnås, ved at udnytte alle arbejdere så godt som muligt. I det andet tilfælde gælder det om at finde den bedste delmængde af arbejderne til opgaverne. Vi opfatter det førstnævnte som den største udfordring i et *stort* ressourceallokeringssystem.

2.3 Schedulering

Schedulering er en *planlægning* af udførelsen af en mængde opgaver på en mængde arbejdere. Resultatet af en schedulering vil kunne være alt lige fra en detaljeret plan over, hvor og hvornår hver enkelt opgave skal udføres, til eksempelvis en inddeling af opgaverne i en prioriteret kø.

I litteraturen [3] ses det, at en schedulering i et heterogent computernet ikke direkte kan sammenlignes med *almindelig* schedulering af multiprocessorer, idet et sådan *net* er homogent og scheduleren *ejer* ressourcerne. I det heterogene computernet skal der tages højde for at forudsigelser om ting kan og vil være fejlbehæftede. Således kan mange faktorer fjernes fra optimeringsberegninger, idet disse faktorers indflydelse vil være mindre end de afvigelser, som vil komme fra fejl i forudsigelser af andre faktorer.

For at lave en schedulering kræver det information om tilstanden for de arbejdere, man har til rådighed og de opgaver, som skal scheduleres. Der skal dannes et *billede* af både arbejdere og opgaver.

Billede af opgaver: Set fra en scheduler, så kan opgaver opstå og forsvinde (en opgave kan være flyttet fra det område scheduleren på et bestemt tidspunkt fokuserer på). Herudover er opgaver en statisk entitet, indtil udførelsen af opgaven påbegyndes.

Billede af arbejdere: Arbejdere kan opstå, forsvinde (f.eks. hvis en PC slukkes eller fjernes fra datanettet), samt ændre sig (f.eks. mængden af ledig RAM og CPU).

Det billede, man skal bruge til en schedulering, er faktisk et "fremtidsbillede" – hvordan ser systemet ud det næste stykke tid, hvilke ressourcer vil der være til rådighed. Til at skabe sig et sådan billede, er man nødt til at lave nogle forudsigelser. En sådan forudsigelse kunne eksempelvis laves ved, at man tager et øjebliksbillede/snapshot af arbejdernes tilstand, og antager at dette øjebliksbillede vil være gældende et stykke tid ud i fremtiden – måske indtil næste øjebliksbillede er genereret.

Arbejderne kan skifte egenskaber hele tiden, da systemet deles med andre programmer (såsom et tekstbehandlingsprogram lokalt på en arbejder-maskine). Man kan derfor ikke regne med, at det øjebliksbillede, som bruges i scheduleringen, stadig er gældende når scheduleren er færdig. Det er ikke svært at se, at jo flere forskellige entiteter der er, som kan påvirke den enkelte arbejder, jo mere sandsynligt skifter arbejderen egenskaber mellem to øjebliksbilleder. Således vil der være større sandsynlighed for, at en opgave mod forventning ikke vil kunne udføres hos en arbejder, hvis der er flere schedulere, der uddeler opgaver til denne. Hvis man f.eks. betragter et tilfælde, hvor hver enkelt arbejder broadcaster oplysninger om sine ledige systemressourcer ud til mange mulige opgaveudbydere, så vil flere af udbyderne måske prøve at allokere arbejderen, men kun én af dem med succes.

Vi vil nu beskrive nogle af de eksisterende scheduleringssystemer som findes og bruges.

Condor [15] er et dynamisk scheduleringssystem, som søger at maksimere udnyttelsen af de arbejdere, der er til rådighed. Dette gøres bl.a. ved at flytte allerede allokerede opgaver fra maskiner med høj belastning, hen til (evt. nye) maskiner, som har mindre belastning (gøres ved hjælp af *checkpointing* og *migræring* af de enkelte opgaver).

Prophet [28][29] er en specialiseret scheduler, der udelukkende kan schedulere SPMD applikationer. Her er målet at minimere den enkelte applikations udførelsestid. Applikationerne scheduleres i den rækkefølge, de er stillet (FIFO-princip). Prophet tager, udover forventet CPU forbrug, også højde for netværkskommunikation i sin schedulering. Et af schedulerens mangler er dog, at der ikke tages højde for optimering af flere applikationer samtidigt (de placeres simpelt i en FIFO kø).

SmartNet [10] er et scheduleringssystem og en ressourcemanager. Scheduleren i SmartNet kan opsættes til at bruge forskellige optimeringsstrategier såsom maksimering af udnyttelsen af ressourcerne og minimering af den forventede gennemsnitstid for de enkelte opgaver. Det er ligeledes muligt at vælge mellem et stort antal forskellige algoritmer, som kan bruges til at opnå det ønskede optimerings mål. Scheduleren gør brug af og genererer selv avancerede forudsigelser om udførelsestider for de enkelte opgaver hos arbejderne.

AppLes [2] er en scheduler, der angriber problemet fra applikationens synspunkt. Det er en generaliseret scheduleringsagent, der kan tilpasses de enkelte applikationers behov. Til at foretage scheduleringen bruger AppLes informationer leveret af brugeren, såsom applikationstype, io-aktivitet, hukommelses forbrug, bruger præferencer, osv. samt dynamiske informationer om ressourcer, netværkstrafik og lignende (leveret af en *Network Weather Service*). Modsat eksempelvis SmartNet er AppLes ikke også en ressourcemanager, men er designet til at bruge eksisterende ressourceallokeringssystemer som eksempelvis Globus.

PaLaBer [18] er et ressourceallokeringssystem som bygger på loadbalancering. Modellen tager ikke hensyn til hvilke opgavetyper som kommer ind i systemet, men prøver i stedet at udnytte al tilgængelig hardware. Dette gøres ved at flytte rundt på opgaver fra arbejdere, som er tungt belastede, til arbejdere, der er let belastede. Herved undgår man, at der på nogle arbejdere ligger opgaver, som ikke bliver løst (eller får meget lidt CPU tid), hvis der samtidigt er frie arbejdere.

Generelt for de fleste af disse systemer er, at de er lavet til ressourceallokering i et lokalt område med et relativt lille antal arbejdere (i forhold til et globalt net).

2.4 Global optimering

I de foregående afsnit har vi beskrevet forskellige optimeringsstrategier og kommet med eksempler på eksisterende scheduleringssystemer. De fleste af disse systemer har dog den ulempe, at de kun virker med en begrænset mængde arbejdere og ikke kan skaleres til at virke i et globalt system. Problemerne opstår, fordi systemerne bruger én entitet til ressourceallokeringen. Denne entitet skal indsamle information om alle arbejdere og i dens scheduleringsfase tage højde for alle opgaver og arbejdere.

I det følgende vil vi se på forskellige muligheder for at indføre global optimering imellem nogle grupper af arbejdere, der hver især er under kontrol af en lokal ressourcemanager (eksempelvis som i Prophet).

Der findes flere måder, hvorved man kan binde forskellige grupper af arbejdere sammen til et mere globalt system. Vi vil her præsentere tre af disse metoder:

Ressource Service: Et eksempel på et system, der er designet til at kunne skalere i stor målestok, er ressourceallokeringen i Globus [6][7]. I Globus har man valgt en model, hvor der kan eksistere mange forskellige ressourceallokeringssystemer. I hvert ressourceallokeringssystem skal en ressourcemanager sørge for lokal ressourceallokering. Informationer om gruppernes tilstande, dvs. tilstanden af arbejderne i gruppen, herunder ressourceudbud, kølængder mm., gemmes i en distribueret global informationsservice kaldt MDS[5]. Når en opgave skal løses, foretages en forespørgsel i informationsservicen for at finde en eller flere grupper, som kan tilfredsstille opgavens ressourcekrav. Herefter sendes der en ny forespørgsel til en af gruppernes ressourcemanager, som så sørger for selve scheduleringen af opgaven.

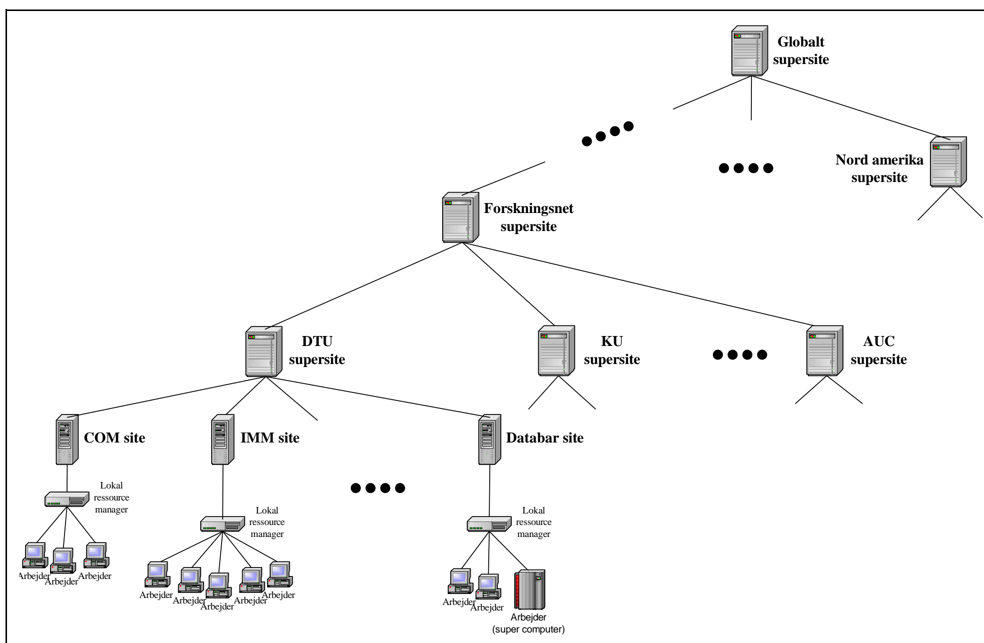
Denne måde at håndtere arbejderlokalisering på, er dog problematisk, når et system generelt er rimelig belastet. Når man laver søgninger i den distribuerede MDS, gør man det i et område af MDS'en, og ikke i hele verdens information. F.eks. kunne man have en MDS, som har information om alle arbejdere på DTU. Når man får et svar tilbage fra en søgning i MDS'en, så kan dette f.eks. fortælle, at der er fundet en arbejder, som opfylder en opgaves krav og at opgaven ser ud til at kunne påbegyndes om 25 minutter. Det betyder, at hvis opgaven sendes til denne arbejder, så vil der gå 25 minutter, før den påbegyndes. Men spørgsmålet er så, om det er det mest optimale at aflevere opgaven her? Skal man hellere udvide sin søgning til også at undersøge, om man kan finde en arbejder et andet sted? Hvis systemet her også er overbelastet, så vil der heller ikke her være nogle arbejdere frie. Problemet forstærkes, hvis der er mange opgaver, der prøver at finde arbejdere og opgaverne ikke er tilfredse med de resultater de finder i MDS'en, så bliver opgaverne nødt til hele tiden at stå og lave nye forespørgsler.

Broadcast: Modsat ressourceservicemodellen, så er opgaverne ikke autonome entiteter, som hver især søger efter arbejdere. I systemer som f.eks. Prophet[28][29], afleveres opgaver hos en lokal ressourcemanager, som opgavestilleren er tilknyttet. Ressourcemanageren prøver så at løse opgaverne lokalt indenfor dens egen arbejdergruppe, men samtidig kan den aflevere opgaver til andre ressourcemanagere, hvis nogle af disse har et overskud af computerkraft.

I Prophet foregår koordineringen mellem grupperne ved, at en ressourcemanager, som har et overskud af opgaver, laver en forespørgsel hos de andre ressourcemanagere, som den kender til, og undersøger om det er muligt, at en af dens opgaver kan løses hurtigere der.

Generelt går broadcastmodellen ud på, at der udveksles information om opgaver, typisk i form af belastningstilstande, mellem ressourcemanagere. Det kan eksempelvis gøres ved, at man broadcaster (heraf navnet til modellen) sin egen belastningstilstand til andre ressourcemanagere. Ulemperne ved denne metode er, at flere ressourcemanagere modtager den samme information og derfor vil kunne foretage beslutninger, som konflikter. Der vil ligeledes være problemer med skalerbarheden, hvis antallet af ressourcemanagere, der skal kommunikere, er stort. I [1] beskrives en metode, hvor man ved brug af multicast kan minimere dette problem.

Hierarkisk: Der opbygges et hierarki, hvor hver enkelt lokale ressourcemanager er et blad i træet. Informationer om overskydende opgaver og information om arbejdernes tilstand rapporteres opad i træet. På hvert niveau er det muligt at lave en



Figur 2.2: Eksempel på en mulig hierarkisk opbygning med DTU og dets ressourcer.

loadbalancering af undertræet, således at der flyttes opgaver fra børn, med et overskud af opgaver, til børn, som har frie arbejdere. Et eksempel på et sådan system er PaLaBer[18], der dog er lidt anderledes, da det kun har én maskine i hvert blad og ikke en ressourcemanager. En af fordelene ved en hierarkisk model er, at den meget let skalerer.

En anden fordel ved træstrukturen er, at hver enkelt knudepunkt vil kunne opføre sig som en autonom enhed. Knudepunktet vil selv kunne bestemme, hvilke informationer der gives videre op og ned i træet. Således vil en organisation som f.eks. DTU kunne gruppere sin ressourcer efter hvilken afdeling disse hører til, og herefter have et fælles knudepunkt, hvor alle afdelingerne samles. Den enkelte afdeling vil så kunne lave en politik om, at arbejdere udfører afdelingens egne opgave som første prioritet, mens overskydende arbejdere kan løse opgaver, som det fælles knudepunkt måtte have. Det fælles knudepunkt vil have til opgave at lave en loadbalancering mellem de enkelte afdelinger. Hvis organisationen som helhed har overskud af frie arbejdere, vil disse kunne stilles til rådighed til resten af verden. Et eksempel på en sådan opbygning ses på figur 2.2.

Vi har valgt at arbejde videre med den hierarkiske model, men har inden dette valg undersøgt mere flade modeller, som bruger broadcast/multicast. Valget faldt på en hierarkisk model, da vi lettest kunne forestille os, at der med denne kunne laves globale optimeringer.

Filosofien bag *global optimering* i en hierarkiske model er, at man i hvert enkelt knudepunkt i hierarkiet laver en lokal optimering (loadbalancering) af undertræet. Da dette gøres af alle knudepunkter, inklusiv roden af træet, vil man opnå en god global optimering [18].

2.5 Vores model

Som nævnt går vores model ud på, at vi vil lave en loadbalancering imellem grupper af arbejdere. I dette afsnit vil vi introducere nogle af de overordnede principper i modellen.

Modellen tager udgangspunkt i, at det globale computernet opdeles i grupper, hvor der indenfor hver gruppe bruges et ressourceallokeringssystem til håndtering af lokale ressourceallokeringer. Opdelingen i grupper minder iøvrigt meget om opbygningen i Globus. Mellem grupperne foretages en loadbalancering, hvor der flyttes opgaver fra grupper med alt for mange opgaver til grupper, som har frie arbejdere. Loadbalanceringen foregår i et hierarkisk træ, hvor hver gruppe er et blad i træet. Den globale optimeringsstrategi i modellen er, at maksimere udnyttelsen af de tilgængelige arbejdere.

Vi definerer et knudepunkts belastning, som værende *længden* af den opgavekø knudepunktet har. Dvs. hvor lang tid det tager, før den sidste opgave i knudepunktets undertræ vil blive *påbegyndt*. Loadbalanceringen i et knudepunkt foregår ved, at det prøver at udjævne dets børns opgavekølængder ved at flytte opgaver fra børn med en lang kø til børn med en kort eller ingen kø. For at holde modellen simpel, har vi indført følgende afgrænsninger:

- Alle opgaver er uafhængige.
- Udførelselstiden af en opgave hos en arbejder afhænger alene af hvor meget CPU opgaven skal bruge og arbejderens CPU udbud. Dvs. udførelsestiden kan beregnes ud fra udtrykket $\frac{O_{CPU\ enheder}}{A_{CPU\ enheder/s}}$.
- En arbejder udfører kun én opgave af gangen.

I næste kapitel vil vi foretage en detaljeret teoretisk gennemgang af vores hierarkiske loadbalanceringsmodel.

Kapitel 3

Hierarkisk loadbalancering

I dette kapitel vil vi beskrive de teorier, som ligger til grund for vores hierarkiske loadbalanceringsmodeller. Figur 3.1 illustrerer den hierarkiske topologi, vi ser på. Denne figur vil blive brugt gennem kapitlet til at beskrive teorien. Den viser desuden vores navnekonventioner for de forskellige dele i den hierarkiske struktur.

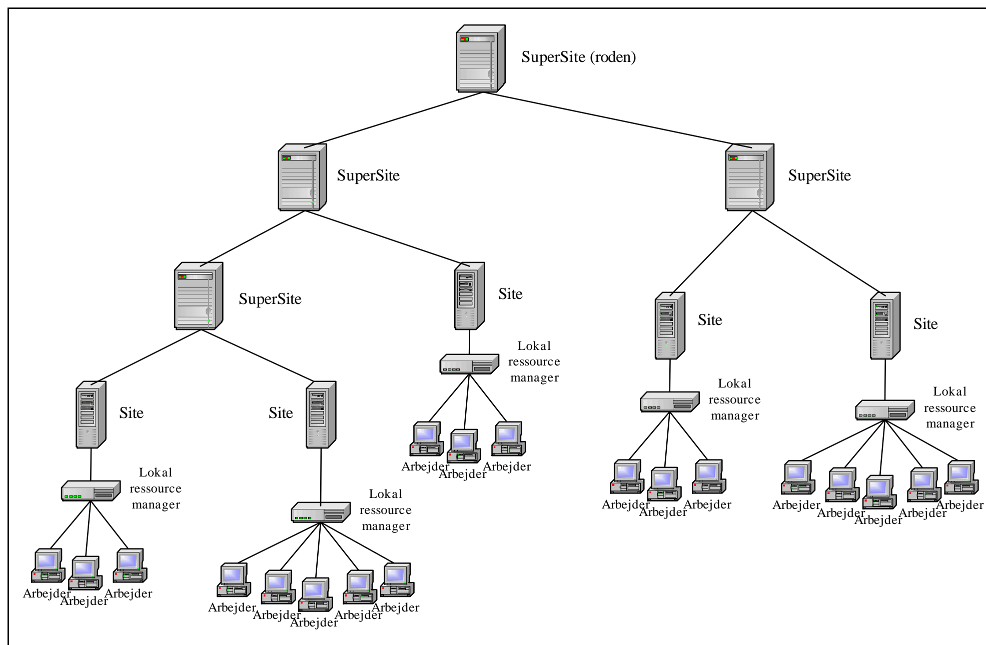
Et site indkapsler en ressourcemanager, der står for schedulering og optimering i det lokale område. Her vil man eksempelvis kunne bruge en eksisterende ressourcemanager såsom Condor [15]. Et site har til opgave, (1) at fodre sin lokale ressourcemanager med opgaver, (2) at modtage nye opgaver fra brugere af systemet, (3) at holde og styre en buffer med opgaver, som endnu ikke er scheduleret hos ressourcemanageren og (4) at informere sin forælder om sitets tilstand (incl. tilstanden af den lokale ressourcemanager). Oven på de enkelte sites opbygger vi en hierarkisk struktur af supersites, som i fællesskab skal stå for de globale optimeringer. Disse har til opgave at foretage en loadbalancering af deres børn (sites/supersites), dvs. de skal flytte opgaver fra overbelastede børn, til børn som er underbelastede. Supersites har også til opgave, at rapportere status om tilstanden for hele deres undertræ op til deres respektive forælder således, at forælderen også kan foretage en loadbalancering.

I kapitlet vil de tre modeller, vi har arbejdet med, blive præsenteret. Overordnet er målet i alle tre modeller at lave en loadbalancering mellem sitene. Design og udviklingsfasen af disse modeller har været en iterativ proces således, at model 2 er en videreudvikling af model 1, og model 3 er en videreudvikling af model 2.

Loadbalanceringen i modellerne virker ved, at opgavernes kølængder balanceres. Således vil vi flytte opgaver fra et site, hvor en opgave f.eks. først kan påbegyndes efter 20 minutter, til et site, hvor den kan startes med det samme eller indenfor meget kort tid.

Model 1 er en meget simpel model, der bl.a. har det formål, at den fungerer som introduktion til vores loadbalanceringsprincipper. Modellen arbejder udelukkende med uniforme opgaver og arbejdere.

Model 2 tager højde for forskellige størrelser af opgaver (opgaverne skal bruge forskellige mængder af CPU) samt forskellige størrelser arbejdere (arbejderne kan levere forskellige mængder CPU pr. sekund).



Figur 3.1: Hierarkisk topologi

Model 3 er den mest komplekse model, som udover størrelser på opgaver og arbejdere, også tager højde for de forskellige *ressourcekrav* opgaver har, og de forskellige *ressourceudbud* arbejdere giver. I modellen tages det ikke for givet, modsat de to andre modeller, at alle opgaver kan løses hos alle arbejdere. Der tages også højde for, at der kan være forskellige kølængder for opgaver med forskellige ressourcekrav.

Når vi i dette kapitel nævner opgaver, arbejdere og ressourcer, mener vi ikke de fysiske entiteter, men nogle logiske beskrivelser, der repræsenterer entiteterne i vores system.

3.1 Generel model

I introduktionen beskrives den logiske opbygning af systemet. Vi beskriver nu nogle af de grundlæggende funktionaliteter.

I hierarkiet laver vi loadbalanceringer mellem bladene. Det, vi ønsker at loadbalancere, er, hvornår de forskellige opgaver kan påbegyndes. Dvs. at når det skal bestemmes, om der skal flyttes en opgave fra et site til et andet, så handler det om at bestemme, hvor opgaven hurtigst vil kunne blive påbegyndt.

Herunder beskrives de funktionaliteter, som sites og supersites generelt indeholder i alle vores modeller.

Information: I hierarkiets blade (sites) indsamles og behandles information, som sendes videre til dets forælder (et supersite). Denne information indeholder oplysninger fra den lokale ressourcemanager, samt information om opgaver, der måtte ligge i sitets lokale kø.
Supersites laver en informationsindsamling og behandling som i sitet, dog foregår indsamling ved en form for summation af informationen, dets børn har sendt op.

Loadbalancering: Balanceringerne i modellen foregår ved, at supersites kan hente opgaver fra sine børn og fra sin forælder. I sites, som ikke har nogle børn, kan der kun hentes opgaver fra forælderen. Et supersite henter opgaver op fra et barn, hvis det kan se at barnet er overbelastet. Sites og supersites henter opgaver ned fra deres forælder, hvis de selv mener, at de er underbelastede. I afsnit 3.3 og i model 1 (afsnit 3.5) vil dette blive yderligere uddybet.

Nye opgaver: I vores modeller bliver nye opgaver afleveret til sites. Dette vil blive yderligere uddybet og begrundet i afsnit 3.4.

Opgaver til den lokale ressourcemanager: I alle sites er der en proces, som sørger for at fodre den lokale ressourcemanager med opgaver. Processen afleverer alle de opgaver, som den er sikker på skal løses lokalt. Dvs. de opgaver som ligger i sitets lokale kø og som kan påbegyndes indenfor *kort* tid.

I hvert knudepunkt i systemet (både sites og supersites) bruges et *delt dataområde* til at holde opgaver. Således vil alle opgaver, som hentes fra et barn eller en forælder, blive afleveret og opbevaret i dette. Når der hentes en opgave fra et andet knudepunkt, så gøres det ved at hente en opgave fra knudepunktets *delte dataområde*.

3.2 Information i hierarkiet

Et af de væsentligste punkter i den hierarkiske loadbalancering er den information, som sendes rundt i det hierarkiske træ. Vi har i vores model valgt at lade al kommunikation følge den hierarkiske struktur. Informationen genereres i træets blade (sites), hvorfra den sendes op i træet. På hvert niveau i træet, samles og behandles den modtagne information, hvorefter den sendes videre op i træet.

Den information, som et knudepunkt i træet modtager fra sine børn, skal bruges til to væsentlige ting:

1. Til at træffe sine egne loadbalanceringsvalg.

2. Til at generere overblik over egen tilstand, så denne kan rapporteres videre op i træet.

For at kunne holde opdateringsfrekvensen rimeligt nede, kræver det desuden, at man i et supersite kan bruge den modtagne information til at tage loadbalanceringsbeslutninger i et stykke tid ud i fremtiden. Ligeledes skal man også kunne registrere egne handlinger, så efterfølgende beslutninger tager højde for tidligere handlinger.

I de næste afsnit belyser vi nogle af de potentielle faldgruber ved informationsudvekslingen, som man skal være meget opmærksom på.

3.2.1 Tidsforsinkelse

Tiden er en vigtig spiller for information, der sendes rundt i systemet. Den information der genereres i bladene, er kun et øjebliksbillede af, hvordan tilstanden er. Som tiden går er det ikke sikkert at informationen stadig er rigtig. Da vi som tidligere beskrevet ønsker at holde opdateringsfrekvensen nede på et rimeligt niveau, er det vigtigt, specielt med hensyn til punkt 1 og 2 beskrevet ovenfor, at der tages højde for tiden. Hvis der f.eks. i et supersite ikke tages højde for denne, vil man let kunne komme til at træffe forkerte loadbalanceringsbeslutninger, som f.eks. ikke at registrere, at en arbejder, der blev rapporteret op som værende belastet, efter et stykke tid kan være blevet fri.

En måde hvorpå man nærmest kan eliminere problemet med tidsforsinkelse er, at tidsstemple al information. Hvergang der tages en beslutning (punkt 1) eller sendes information videre (punkt 2) er det så muligt at tage tiden med i betragtning.

En tidsstempling af informationen kan gøres på to måder: enten ved hjælp af et *globalt* ur, som bruges i hele systemet, eller ved brug af et lokalt ur i de enkelte sites og supersites. Med den første løsning vil man kunne tidsstemple informationen, når den genereres i de enkelte blade, og lade et sådan tidsstempel gælde i hele træet. Ulempen er dog, at det kræver en ikke triviel synkronisering af de forskellige entiteters ure med et globalt ur.

I den anden løsning tidsstemples informationen ikke inden den sendes fra et site, men når den modtages i et andet knudepunkt. Dette betyder at tiden, det tager at sende informationen (netværkskommunikation), vil blive *glemt*. Dette fejlbidrag vil dog være af mindre betydning i forhold til andre potentielle fejlkilder i modellen (såsom arbejdere og opgaver der ikke opfører sig som forventet).

3.2.2 Kompleksitet

Et af de væsentlige argumenter for en hierarkisk model er skalerbarhed. For informationen betyder det, at informationsmængden ikke må eksplodere i størrelse jo højre man kommer i træet. Det vil eksempelvis være uhensigtmæssigt at sende enkeltstående information om hver enkelt arbejder i systemet helt til toppen af træet.

Hvor meget information, der skal sendes rundt i systemet, er naturligvis en svær balance. Sendes der for lidt, vil modtageren af informationen ikke kunne træffe optimale beslutninger. Sendes der for meget, vil systemet næppe kunne skalere.

Vi minimere bl.a. den totale informationsmængde i hele systemet ved at holde opdaterings frekvensen nede. Dette løser dog ikke de potentielle skaleringsproblemer med, at hver enkelt information bliver for stor.

En måde at holde hvert enkelt informationsmængde nede på er at have en øvre grænse for størrelsen af informationen, der sendes mellem to knudepunkter, som ikke overskrides på noget sted i træet. Hvis man f.eks. ved at informationen altid vil være af en helt bestemt størrelse, så vil dette gælde.

3.3 Belastningstilstande

I knudepunkterne i den hierarkiske model (f.eks. som i figur 3.1), definerer vi en *belastningstilstand*, som beskriver i hvilken tilstand knudepunktet og dets undertræ er. Vores

definition af en belastningstilstand minder meget om den, der bruges i PaLaBer[18]. Et knudepunkt er i en af følgende tre belastningstilstande:

Let belastet: Et knudepunkt, som er *let* belastet betyder, at knudepunktet har et kraftoverskud (har arbejdere, som snart vil være/er frie), som det gerne vil stille til rådighed til sit forælderknudepunkt.

Normalt belastet: Knudepunktet er i en tilstand, hvor det har rigeligt med opgaver til alle sine arbejdere og derfor ikke ønsker at få mere arbejde. Samtidig har det heller ikke for mange opgaver, og ønsker derfor ikke at aflevere opgaver til sin forælder.

Tungt belastet: Knudepunktet har for mange opgaver i forhold til sine arbejdere, og vil derfor gerne aflevere nogle af disse til andre, som vil hjælpe.

En simpel loadbalancering i et knudepunkt (supersite), vil være at tage opgaver fra et *tungt* belastet barn og flytte dem til et *let* belastet barn.

Et eksempel på hvordan belastningstilstanden kan beregnes, kunne være: ($opgaver_i$ og $arbejdere_i$ er mængderne af opgaver henholdsvis arbejdere, som findes i knudepunktet i og dets undertræ. w_j er CPU forbruget opgave j skal bruge, og r_j er CPU udbudet arbejder j giver)

$$\begin{aligned}
W_i &= \sum_{j \in opgaver_i} w_j \\
R_i &= \sum_{j \in arbejdere_i} r_j \\
Belastning_i &= \frac{W_i}{R_i} \\
BelastningsStatus_i &= \begin{cases} Let & \text{if } Belastning_i < X_{min} \\ Normal & \text{if } Belastning_i \in [X_{min}, X_{max}] \\ Tung & \text{if } Belastning_i > X_{max} \end{cases}
\end{aligned}$$

Hvor X_{min} og X_{max} er tærskelværdier, hvorom der gælder $0 < X_{min} < X_{max}$.

3.4 Opgaver stilles i bunden

Når opgaverne ankommer i hierarkiet sker det i bunden (i et site). Vi har valgt at gøre dette af flere grunde:

Autonomitet: Hvis man stiller opgaverne i sit *eget* site, vil opgaver komme i det lokale sites kø og sitet kan garantere, at det vil løse egne opgaver førend det begynder at hjælpe andre sites.

Fordelt belastning: Hvis alle opgaver i systemet bliver afleveret i f.eks. rod-knudepunktet, så vil dette knudepunkt hurtigt kunne blive overbelastet og systemet ville ikke være skalerbart. Ved i stedet at aflevere opgaverne i mange forskellige sites, sikres at belastningen bliver bedre fordelt. Problemer, der kan løses lokalt, vil blive løst lokalt uden at resten af verden bliver belemret.

Afstande: Hvis opgaver stilles i et nærliggende site, så vil en loadbalancering i hierarkiet kunne sørge for, at opgaverne ikke bliver flyttet til den anden ende af verdenen for at blive løst, foruden dette er mest optimalt. På denne måde kan man sørge for, at hvis der stilles opgaver i både USA og danmark, så vil de danske opgaver blive løst i danmark, og de amerikanske i USA. Men hvis der ikke stilles opgaver i USA, så kan danmark selvfølgelig udnytte arbejdere der.

Der er selvfølgelig også nogle ulemper ved at lade opgaver ankomme i træets blade:

Fejl i information: Når der pludselig ankommer nye opgaver i et blad i træet, så vil den information, som findes hos bladets forælder, bedsteforælder osv. være fejlagtig, indtil ny information er blevet sendt op gennem træet. Dette ville ikke ske, hvis opgaverne ankom i toppen (som det eksempelvis gør i PaLaBeR[18]).

Det er værd at bemærke, at hver gang en opgave flyttes mellem eksempelvis to supersites, kræver det en form for koordinering mellem de to. Der vil naturligvis være en øvre grænse for, hvor mange opgaver et knudepunkt kan nå at behandle indenfor et fast tidsrum og hvor mange opgaver der kan flyttes mellem to knudepunkter. Ideelt set bør der derfor ikke flyttes flere opgaver end højst nødvendigt. En opgave, der på et tidspunkt ender med at blive løst i det site, hvor den er stillet, vil derfor ideelt set aldrig skulle forlade sitet.

3.5 Model 1 - Simpel loadbalancering

Den første loadbalanceringsmodel, vi vil undersøge, er en meget simpel model. Modellen virker ved, at man i hvert site prøver at opretholde en buffer med opgaver, hvis størrelse er afhængig af antallet af arbejdere i sitet.

$$w_j = \text{forventet CPU forbrug for opgave } j \tag{3.1}$$
$$r_i = \text{forventet CPU udbud pr. sek. for arbejder } i \tag{3.2}$$
$$w_j = W \qquad \forall j \in Opgaver \tag{3.3}$$
$$r_i = R \qquad \forall i \in Arbejdere \tag{3.4}$$

Desuden forudsættes det, at alle opgaver kan løses på alle arbejdere. Arbejderne har ikke en opgavekø og kan først få en ny opgave, når arbejderen er fri.

Da vi forudsætter, at alle opgaver har samme CPU forbrug (3.3) og alle arbejdere giver samme CPU udbud (3.4), laver vi en loadbalancering, hvor vi kigger på *antallet* af opgaver i forhold til *antallet* af arbejdere.

Vi definerer belastningstilstanden fra afsnit 3.3 som:

$$N_i = \text{antal opgaver i knudepunktet } i \tag{3.5}$$
$$N_{born,i} = N_i + \sum_{j:undertrae(i)} N_j \tag{3.6}$$

$$R_i = \text{antal arbejdere i knudepunktet } i \tag{3.7}$$

$$R_{born,i} = R_i + \sum_{j:undertrae(i)} R_j \tag{3.8}$$

$$Belastning_i = \frac{N_{born,i}}{R_{born,i}} \tag{3.9}$$

$$BelastTilstand_i = \begin{cases} Let & \text{if } Belastning_i < X_{min} \\ Normal & \text{if } Belastning_i \in [X_{min}, X_{max}] \\ Tung & \text{if } Belastning_i > X_{max} \end{cases} \tag{3.10}$$

Når et supersite observerer, at et af dets børn er *tungt* belastet og et andet af børnene er *let* belastet, fungerer loadbalanceringen således, at supersitet flytter en opgave fra det tungt belastede barn til det let belastede barn. Vi har valgt at opsplitte dette i to simple regler for, hvornår opgaver sendes rundt i systemet:

$$BelastTilstand = let \Rightarrow HentOpgaveFraForaelder() \tag{3.11}$$

$$BelastTilstand_i = tung \Rightarrow HentOpgaveFraBarn_i() \tag{3.12}$$

hvor i er et barn.

Ligning (3.11) gælder for både sites og supersites, og betyder at et let belastet site/supersite må prøve at hente en opgave fra sin forælder. Ligning (3.12) gælder kun for et supersite, og den indikerer, at et supersite der har et overbelastet barn, må prøve at hente en opgave hos det pågældende barn[1].

Opsplitningen i de to regler tjener bl.a. det formål, at der opnås en asynkronitet. På den måde vil en kommunikation, der ellers ville involvere tre knudepunkter, opsplittes i to kommunikationer, der hver foregår mellem to knudepunkter. Ligeledes vil et supersite, der kun har tungt belastede børn, kunne virke som en buffer for sin forælder. Uden buffere i supersites vil eksempelvis hver enkelt af de opgaveforespørgsler, roden af træet foretager til sine børn, skulle sendes hele vejen ned igennem træet til sitene.

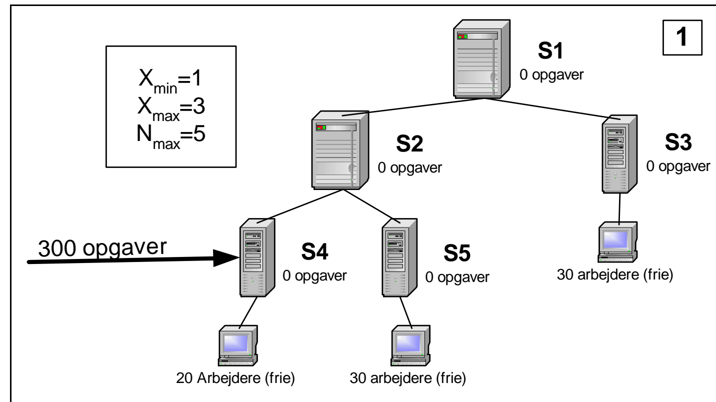
Ulempen ved opsplitning i de to regler og indførslen af buffere i supersitene er, at i et system, der totalt set er overbelastet, vil alle overskydende opgaver blive flyttet mod toppen af træet for i sidste ende at havne i roden af træet. Dette giver de samme problemer som beskrevet i afsnit 3.4. Problemet kan undgås ved at indføre et maksimalt antal opgaver N_{max}, som må befinde sig i et supersite, når der hentes opgaver. De ovenstående ligninger bliver så:

$$BelastTilstand = let \wedge N < N_{max} \Rightarrow HentOpgaveFraForaelder() \tag{3.13}$$

$$BelastTilstand_i = tung \wedge N < N_{max} \Rightarrow HentOpgaveFraBarn_i() \tag{3.14}$$

Hvis man f.eks. vælger $X_{min} = 1.0$ og $X_{max} = 3.0$, vil reglerne betyde, at de sites, hvor der ikke stilles opgave, vil prøve at holde én opgave klar til hver arbejder i sitet. I de sites hvor der stilles opgaver, vil forældersupersitet prøve at afhjælpe sitet, indtil sitet kun har 3 opgaver pr. arbejder.

[1] Det er værd at bemærke, at de viste ligninger kunne udtrykkes på andre måder. Vi har valgt at se enhver flytning af en opgave fra modtagerens synspunkt, således at det er den modtagne part, der initialisere kommunikationen ved en opgaveflytning.



Figur 3.2: Start tilstand. Alle arbejdere er frie og der er ingen opgaver i systemet. Der stilles 300 opgaver i S4.

Til beregningen af belastningstilstanden for et knudepunkt bruges i ligning (3.6) og (3.8) information om knudepunkterne i knudepunktets undertræ. I et site genereres informationen ud fra hvor mange arbejdere der er i sitet, samt hvor mange opgaver der ligger i kø. I et supersite genereres informationen ud fra en simpel summation af informationen, supersitet har modtaget fra sine børn, samt de opgaver, det måtte have liggende i sin lokale buffer. Da der hele tiden bliver allokeret opgaver til arbejderne i et site, bliver man nødt til at sende ny information med høj frekvens, således at forælderen, bedsteforælderen osv. hurtigst muligt bliver bekendt med ændringerne i antallet af opgaver.

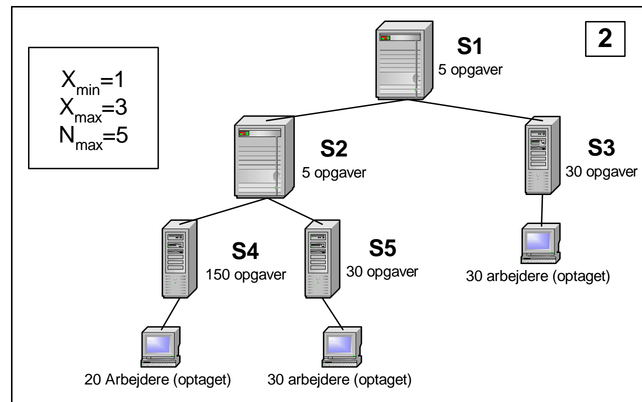
Det ses simpelt, at informationen i denne model ikke vil eksplodere, da hver enkelt information, som sendes opad i træet, er af konstant størrelse.

3.5.1 Model eksempel

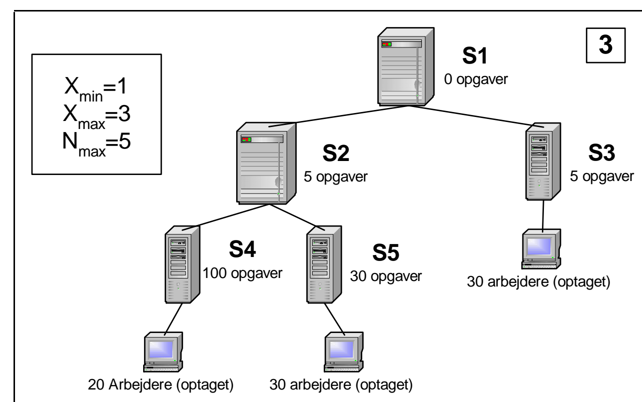
På figur 3.2, 3.3 og 3.4 vises et eksempel på, hvordan modellen fungerer. Eksemplet består af 3 sites, som tilsammen har 80 arbejdere, og 2 supersites. X_{min} og X_{max} er valgt til henholdsvis 1 og 3, mens N_{max} er sat til 5. Til at starte med er der ingen opgaver i systemet (figur 3.2).

Der stilles nu 300 opgaver i S4. På figur 3.3 ses en tilstand hvor systemet er i balance. Alle 80 arbejder har påbegyndt udførelsen af en opgave. Bufferne hos alle sites og supersites er fyldt i henhold til balanceringsreglerne. Der vil derfor ikke blive flyttet flere opgaver, før arbejderne bliver færdige med udførelsen af deres første opgave. Det bemærkes at den overskydende opgavemængde stadig er placeret i S4.

I Figur 3.4 vises en ny tilstand, som fremkommer efter at alle arbejdere har løst deres første opgave, og alle er påbegyndt en ny opgave. S3 har trukket de 5 opgaver, der var i S1, ned, men har ikke fået fyldt sin buffer op. Til trods for at S1 ikke har nogle opgaver i sin buffer, vil S1 ikke trække opgaver op fra S2. S2's tilstand er nemlig ikke længere *tungt* belastet, men derimod *normalt* belastet ($5+100+30\ opgaver/50\ arbejdere = 2.7 < X_{max}$). Resten af opgavemængden vil derfor blive klaret af S2 og dennes børn.



Figur 3.3: Tilstand 2. Systemet er i balance. Der kan ikke flyttes nogle opgaver før arbejderne igen bliver frie.



Figur 3.4: Tilstand 3. Også her er systemet i balance, til trods for at bufferne hos S1 og S3 ikke er fyldt op.

3.5.2 Vurdering af Model 1

Modellen virker ved, at der holdes et antal opgaver i kø for hver arbejder. Hvis opgaverne tager lang tid at udføre på arbejderne (f.eks. 30 minutter), betyder det, at de opgaver, der ligger i køerne, først vil kunne løses efter lang tid (når de andre opgaver er færdige). Dette er ikke prisværdigt, da der kan eksistere frie arbejdere i et andet site, som det kan betale sig at bruge hvis ventetiden er f.eks. 30 minutter.

Modellen forudsætter, at alle opgaverne skal bruge den samme mængde CPUenheder og at alle arbejdere udbyder samme mængde CPUenheder/s. Dette er dog ikke strengt nødvendigt. Modellen vil stadig opføre sig korrekt, selvom opgaverne og arbejderne er lidt forskellige.

Modellen har en væsentlig fejl, som opstår når et supersite genererer sin information. Da belastningstilstanden bestemmes ud fra en gennemsnitsbetragtning, vil det være muligt at belastningstilstanden bestemmes til normal, også selvom der er frie arbejdere et sted i undertræet. I det følgende eksempel vises et eksempel på fejlen.

Der haves to sites under et supersite. Vi sætter $X_{min} = 1.0$ og $X_{max} = 3.0$. I det ene site er der er 20 arbejdere, men ingen opgaver i kø. I det andet er der 40 arbejdere og 120 opgaver i kø (således at belastningstilstanden er normal, men næsten tung). Belastningstilstanden bestemmes nu i supersitet:

$$N_{born,i} = 0 + 120 + 0 \quad (3.15)$$
$$R_{born,i} = 40 + 20 \quad (3.16)$$
$$Belastning_i = \frac{120}{60} = 2.0 \quad (3.17)$$
$$BelastTilstand_i = Normal \quad (3.18)$$

Således er supersitets belastningstilstand *normal*, også selvom det har et barn, der er let belastet. For at løse problemet, kunne man indføre en ny information, hvor man holdt øje med hvor mange opgaver der egentligt kan flyttes. I model 2 ser vi ikke kun på rene gennemsnitsbelastninger og problemet bliver herved elemineret.

3.6 Model 2 - Opgaver og arbejdere med forskellige størrelser

I den simple model ovenfor har vi kun kigget på uniforme opgaver og arbejdere. Vi vil nu kigge på opgaver, som er af forskellige længder med hensyn til, hvor mange CPUenheder de skal bruge, samt arbejdere, der har forskelligt CPU udbud (CPUenheder pr. sekund). Igen forudsættes det at alle opgaver kan løses på alle arbejdere.

Modellen er en videreudvikling af model 1 og derfor vil den endelige model ligne model 1 meget. Forbedringerne i denne model er, at vi ikke ser på rene gennemsnitsberegninger, men istedet tager mere højde for hver enkelt arbejder og opgave. I modellen tages der også højde for at *tiden går*, således at et knudepunkt kan beregne, hvordan dets undertræ ser ud til et givent tidspunkt, også selvom informationen, det har fået fra sine børn, er

generet et stykke tid i forvejen. Således kan frekvensen, hvormed der sendes information, minimeres meget i forhold til model 1. Vi vil i afsnittet forklare disse nye principper og afslutningsvis beskrive den samlede model.

I model 1 blev site-bufferne brugt til at holde et vist antal opgaver pr. arbejder. Dette er det samme som at sige, at disse buffere blev brugt til at sørge for, at alle arbejdere kunne modtage opgaver til mindst de næste t sekunder ($t = X_{min}^{model\ 1} * t_{opgave}$) fra denne buffer. Dette resultere i at ligningerne (3.5) - (3.10) ændres til

$$N_i = \sum_{j \in Opgaver_i} w_j \tag{3.19}$$

$$N_{born,i} = N_i + \sum_{j \in undertrae(i)} N_j \tag{3.20}$$

$$R_i = \sum_{j \in Arbejdere_i} r_j \tag{3.21}$$

$$R_{born,i} = R_i + \sum_{j \in undertrae(i)} R_j \tag{3.22}$$

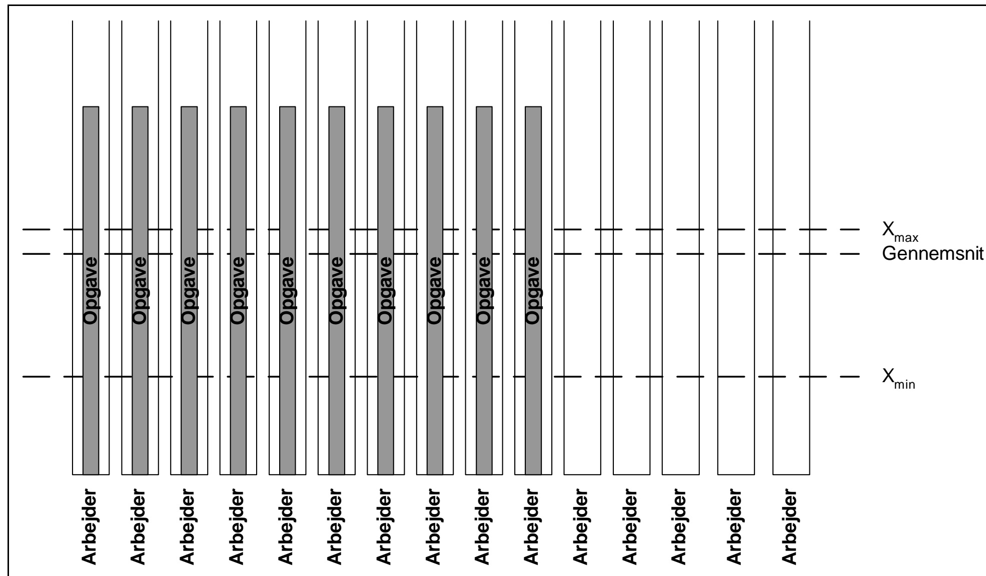
$$Belastning_i = \frac{N_{born,i}}{R_{born,i}} \tag{3.23}$$

$$BelastTilstand_i = \begin{cases} Let & \text{if } Belastning_i < X_{min} \\ Normal & \text{if } Belastning_i \in [X_{min}, X_{max}] \\ Tung & \text{if } Belastning_i > X_{max} \end{cases} \tag{3.24}$$

Hvor $Belastning_i$ således udtrykker en gennemsnitsbelastning pr. arbejder, dvs. hvor lang tid der er, til arbejderne i gennemsnit er frie.

Disse ligninger holder desværre ikke når der kommer betydelige forskelle i længden af opgaverne og/eller i arbejdernes CPU udbud, samt når opgaverne er relativt *lange* i forhold til arbejdernes CPU udbud. En eller flere store opgaver vil kunne resulterer i at belastningstilstanden bliver *tung* eller *normal*, også selvom der er helt frie arbejdere. Vi illustrerer problemstillingen med et eksempel: Der haves et site, som er i besidelse af 15 arbejdere (1 CPUenhed/s). I sitet stilles 10 opgaver, der hver skal bruge 50 CPUenheder. Vi ønsker, at der altid er mindst 10 sekunders opgaver klar til hver arbejder ($X_{min} = 10\ sek.$). For sitet vil så gælde:

$$\begin{aligned} N_{site} &= \sum_{j \in Opgaver_{site}} w_j \\ &= 10\ opgaver * 50\ CPUenheder/opgave = 500\ CPUenheder \\ N_{born,site} &= N_{site} \\ R_{site} &= \sum_{j \in Arbejdere_{site}} r_j \\ &= 15\ arbejdere * 1\ CPUenhed/s/arbejder = 15\ CPUenheder/s \\ R_{born,site} &= R_{site} \\ Belastning_{site} &= \frac{N_{born,site}}{R_{born,site}} \\ &= \frac{500\ CPUenheder}{15\ CPUenheder/sek} = 33.3\ s \end{aligned}$$



Figur 3.5: Eksemplet viser hvorledes gennemsnitsbelastningen giver anledning til en forkert beregning af belastningstilstanden.

Belastningstilstanden burde være *let*, idet der stadig er 5 frie arbejdere.
Men da $Belastning_{site} > X_{min}$ er dette ikke tilfældet med definitionen i ligning (3.24). Eksemplet er også illustreret på figur 3.5.

Hvis man ligeledes ser på arbejdere, der giver forskelligt CPU udbud, eller opgaver med stor forskel i deres længde, så vil problemer analogt til ovenstående kunne opstå.

3.6.1 Belastningstilstande og skemalægning

For at undgå ovenstående problem, bliver man nødt til at tage højde for hver enkelt opgaves *længde* og hver enkelt arbejders CPU udbud. Dette kan gøres ved at lægge et skema for, hvordan opgaverne vil blive løst på arbejderne. Herved kan man se, hvornår de enkelte opgaver vil blive påbegyndt, samt hvornår de enkelte arbejdere er frie. Informationerne, man skal bruge for at lægge et sådan skema, indeholder oplysninger om opgaverne (deres længde), samt oplysninger om arbejderne (CPU udbud, samt hvornår de forventes frie).

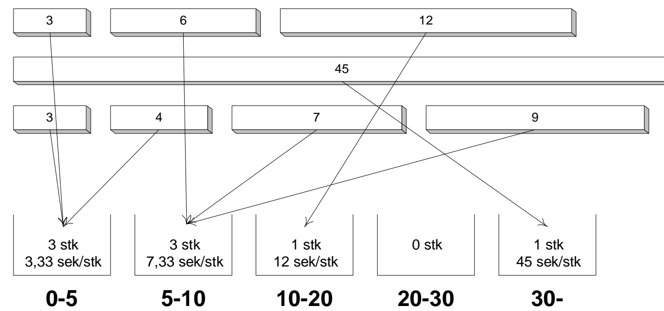
Ud fra et sådan skema kan man så redefinere belastningstilstanden fra afsnit 3.3:

$$\langle NyeOpgaver, NyeArbejdere\rangle = LaegSkema(Opgaver, Arbejdere) \tag{3.25}$$

$$BelastningsTilstand = \begin{array}{l} \text{if } SidsteStartTid(NyeArbejdere) > X_{max} \text{ then } Tung \\ \text{else if } NaesteFriTid(NyeArbejdere) < X_{min} \text{ then } let \\ \text{else } Normal \end{array} \tag{3.26}$$

Hvor *NaesteFriTid* fortæller, hvornår den tidligst frie arbejder vil være fri. *SidsteStartTid* fortæller, hvornår den sidste opgave vil blive påbegyndt.

LaegSkema-funktionens opgave er, at lægge et skema for hvordan opgaverne vil blive løst på arbejderne. Dette skema bruges vigtigst af alt til at bestemme hvilken belastningstilstand knudepunktet har og det er derfor ikke så vigtigt, at skemaet er lagt 100% optimalt, men behøver kun at være en rimelig tilnærmelse. Det er ikke muligt for et supersite at



Figur 3.6: Inddeling af opgaver i spande.

lægge sit skema så opgaverne bliver løst på præcist de arbejdere, som det egentligt står i skemaet, da det ikke er supersitet der laver den endelige allokering til arbejderen. Derfor er en tilnærmelse god nok.

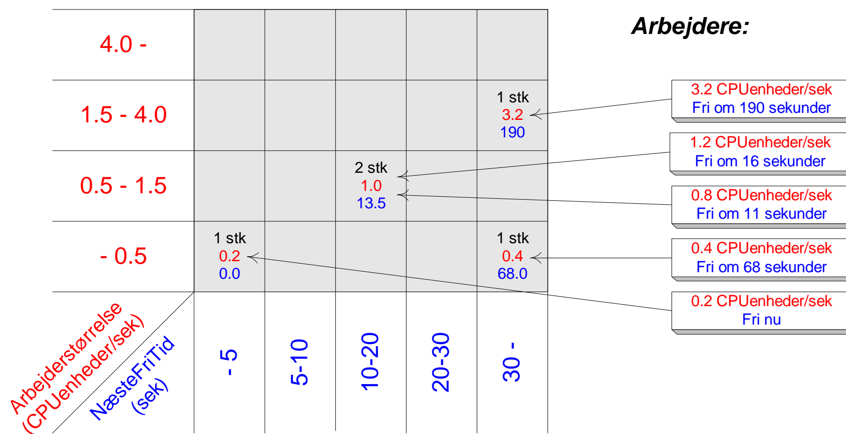
Vi har valgt den simple algoritme *Opportunistic Load Balancing (OLB)* i en lidt modificeret udgave til vores skemalægning. Algoritmen virker simpelt ved, at man tager den mindste opgave og placerer den på den arbejder vi først forventer fri. Vi har også set på andre almindelige skemalægningsalgoritmer [4], men da vores skemalægning kun skal bruges til bestemmelse af belastningstilstanden, har vi valgt den simple. Fordelen ved den simple algoritme er desuden at den er ret hurtig (af orden $\mathcal{O}(n * log(n))$ - n=antal opgaver, m=antal arbejdere, når $n \gg m$), hvor algoritmerne beskrevet i [4] typisk er af mindst $\mathcal{O}(n^2 * m)$. Desuden, som det vil blive beskrevet i næste afsnit, vil en stor del af informationerne om opgaver og arbejdere blive grupperet og simplificeret når de kommer op gennem træet, hvilket betyder at datagrundlaget til skemalægningen allerede er *ødelagt* og derfor vil en bedre algoritme ikke nødvendigvis give et bedre resultat.

3.6.2 Informationsniveau

For at kunne lægge skemaer, som beskrevet ovenfor, er det nødvendigt at have information om hver enkelt opgave og hver enkelt arbejder. Dette er ikke noget problem i et site, idet det *ejer* disse. Men i et supersite bliver nødt til at modtage information om alle opgaver og arbejdere fra dets børn, hvilket i roden af træet bliver alt for meget information i et stort system.

Det med at se på opgaver og arbejdere enkeltvis indfører vi for at undgå problemer med, at f.eks. nogle få opgaver kan få et helt site til at se tungt belastet ud, når det egentligt kunne være let belastet som beskrevet i introduktionen til afsnit 3.6. Disse problemer opstår når der er få opgaver og/eller når der er store forskelle i opgavernes længde og ligeså med store afvigelser i arbejderne. Det er således disse tilfælde, der er vigtige at få med i informationerne, som udveksles med et forælderknudepunkt. På baggrund af dette har vi valgt at minimere informationerne ved at inddele opgaverne i et fast antal opgavespande, hvor hver spand indeholder opgaver afhængig af deres længde. Opgaverne kunne f.eks. inddeles i fire spande: små, mellem, store og meget store opgaver. Et eksempel med fem spande ses i figur 3.6.

For arbejderne er det lidt mere besværligt, idet der er en ekstra dimension på informationen (CPU udbud og NaesteFriTid). Denne information kan inddeles i et 2-dimensionelt



Figur 3.7: Gruppering af arbejdere, der *minder* om hinanden.

array, hvor den ene dimension er NaesteFriTid og den anden er CPU udbud. Ligesom med opgaverne minimeres informationerne ved at sammenlægge arbejdere, som ligger i samme firkant. Et eksempel på dette ses i figur 3.7.

Med disse minimeringer af informationsmængden ses det tydeligt, at størrelsen af informationen aldrig kan eksplodere, da informationen, som sendes mellem knudepunkterne, aldrig vil kunne bliver størrere end størrelsen af de to ovenfor beskrevne strukturer, som er af konstant størrelse.

I de to figurer er inddelingsintervallerne kun eksempler. Disse intervaller skal vælges ud fra nogle estimater på mulige opgaver og arbejdere. For arbejderinformationen bør arbejderstørrelse-intervallerne vælges ud fra kendskab til de mindste og største arbejdere i systemet (eller det man kan forestille sig). Man kan f.eks. vælge, at lade de dårligste maskiner have CPU størrelsen ca. 1.0 og de bedste ca. 1000. Intervallerne bliver så:

$$[0, 1.0, ..., 1000, \infty] \tag{3.27}$$

Jo større springende er mellem de mellemliggende værdier jo mere upræcis bliver skemalægningen. Vi forstiller os at værdierne kan være eksponentiel voksende (f.eks. kunne eksponentielkoefficienten være 1.5).
For *næste fri tiden* bør intervallerne være bestemt af den maksimale X_{max} i systemet. Arbejdere, som har en fri tid der er en del større end den største X_{max}, er ikke interessante når belastningstilstanden skal bestemmes (skemalægning), da de ikke er frie det næste lange stykke tid. Således kan næste fri tid intervallerne f.eks. være: (i sekunder)

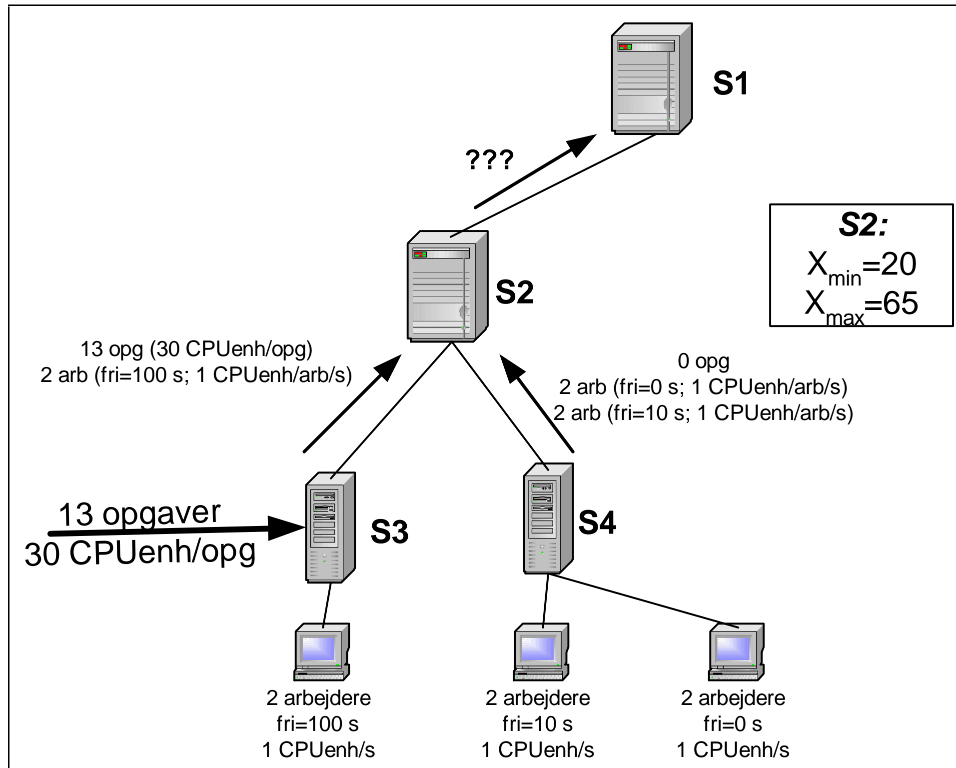
$$[0, 5.0, 10, 15, 20, 25, ..., X_{max}, X_{max} + 5, ..., X_{max} + opdateringfrekvens, \infty] \tag{3.28}$$

Det sidste interval starter ved $X_{max} + opdateringfrekvens$, da vi ønsker *rimelig* præcis information om arbejderne indtil dette tidspunkt.

For inddelingsintervallerne for opgaverne bør det sidste intervals startværdi bestemmes ud fra størrelsen af de bedste arbejdere og X_{min}. Denne værdi kan bestemmes ved:

$$O_{max} = X_{min} * CPU_{bedste\ arbejder} \tag{3.29}$$

Hvis en opgave skal bruge mere end O_{max} CPUenheder, så betyder det at lige meget hvilken arbejder denne ender på, så vil arbejderen ikke være være let belastet. Som



Figur 3.8: Systemopbygningen til eksempel på skemalægning.

for intervallerne for inddelingen af arbejderne bruges også her eksponentielt voksende værdier. Opgaveinddelingen kunne f.eks. være:

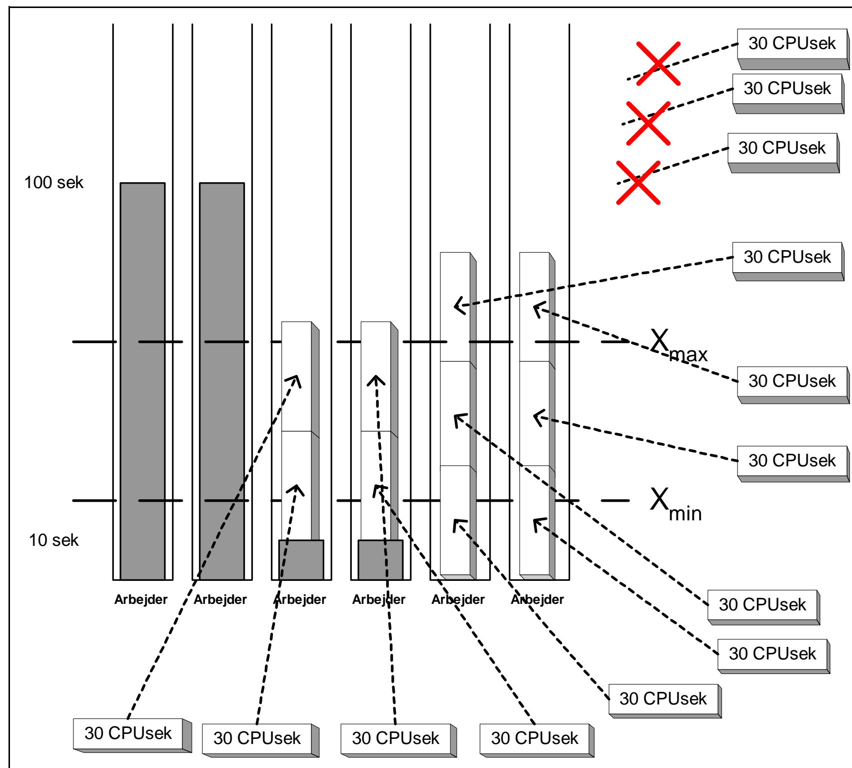
$$[0, 1.0, 1.0 * 1.5^1, 1.0 * 1.5^2, 1.0 * 1.5^3, ..., \sim O_{max}, \infty] \tag{3.30}$$

3.6.3 Eksempel på skemalægning

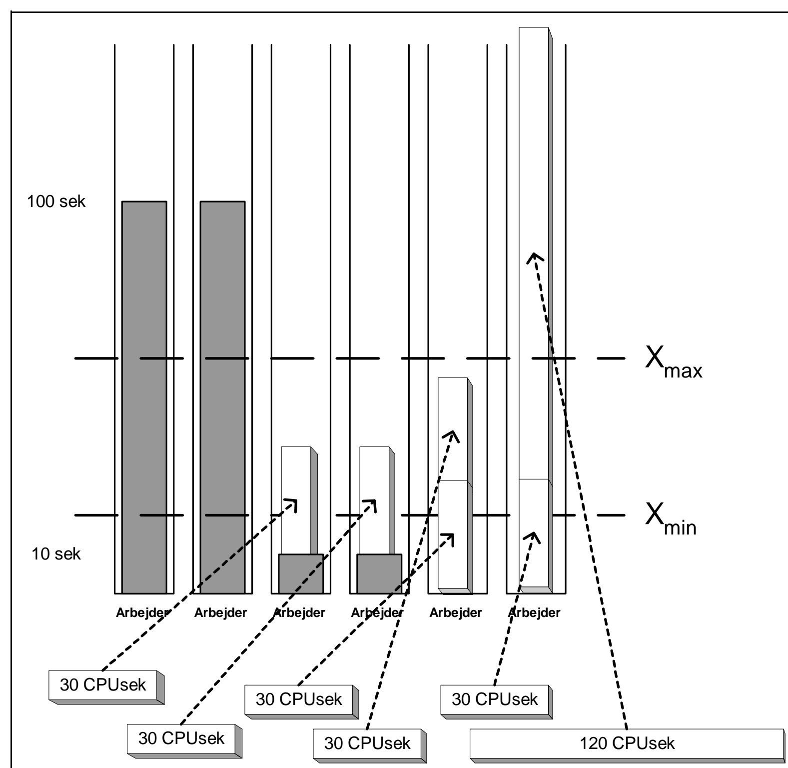
I figur 3.8 ses et eksempel med to sites og to supersites. I det følgende vil vi vise hvordan S2 ud fra information fra sine børn og skemalægning kan bestemme sin egen belastningstilstand, samt hvordan informationen, som skal sendes til S1, genereres.

I S2 haves information fra dens to børn S3 og S4, som vist på figuren. S2 kan nu lægge sit skema, hvor den placere opgaver på arbejdere, som forventes færdige indenfor $X_{max} = 65\ sekunder$. Et eksempel på dette er vist i figur 3.9. I figuren ses det at der er 3 opgaver som ikke kan placeres indenfor de 65 sekunder, og derfor bliver informationen om opgaver, som S2 sender op til sin forælder, 3 opgaver á 30 CPUenheder. Arbejderne ser nu ud som følgende: 2 stk fri om 70 sekunder, 2 stk fri om 90 sekunder og 2 stk fri om 100 sekunder. Hvis man i minimeringen af arbejdere bruger intervallerne $[..., 30, 45, 60, 95, 150, ...]$, så er informationen om arbejdere, som sendes op: 4 stk arbejdere (fri=80 s, 1 CPUenhed/s), samt 2 stk arbejdere (fri=100 s, 1 CPUenhed/s).

Hvis der istedet havde været 5 opgaver á 30 CPUenheder og 1 opgave á 120 CPUenheder, så ville skemalægningen se ud som på figur 3.10 og den rapporterede information ville så være: 0 opgaver, 2 stk arbejdere (fri=40 s, 1 CPUenheder/s), 1 stk arbejder (fri=60 s, 1 CPUenheder/s), 2 stk arbejdere (fri=100 s, 1 CPUenheder/s), samt 1 stk arbejder (fri=150 s, 1 CPUenhed/s).



Figur 3.9: Eksempel på skemalægning, hvor der bliver opgaver i overskud.



Figur 3.10: Eksempel på skemalægning, hvor der ikke er opgaver i overskud.

3.6.4 Håndtering af tiden

Da vi kun ønsker at sende information fra et knudepunkt til et forælderknudepunkt en gang imellem (eksempelvis hvert 10. sekund), vil informationen hurtigt blive forældet jo længere op i træet man kommer. Derfor ønsker vi, at man kan tager højde for den tid, der er gået, siden informationen blev genereret/var gældende.

Det er ikke så svært at tage højde for dette. Den information, som bruges i systemet, kan tidsstemples når den bliver genereret (se afsnit 3.2.1). Når informationen skal bruges, så tages der højde for den tid, som er gået.

I systemet er der information om opgaver og om arbejdere. For informationen om opgaverne er det simpelt at tage højde for tiden. Denne information indeholder oplysninger om opgaver, der ligger i kø. Da disse ikke er påbegyndt endnu, og derfor ikke ændres med tiden, førend de bliver afleveret til en arbejder, behøves der ikke gjort noget ved deres information. For arbejdere er det derimod vigtigt at tage højde for den tid, der er gået. Hvis informationen om en arbejder f.eks. er, at den er fri om 48 sekunder, så betyder det, at hvis der er gået 21 sekunder siden denne information blev generet, så er den faktisk er fri om 27 sekunder fra nu af. I resten af rapporten bruger vi funktionen *TimeChange*, som en funktion der ændrer tiden for arbejdere. For en enkelt arbejderinformation vil denne se ud som:

$$\Delta t = nu - oprettelsestid \tag{3.31}$$

$$TimeChange(arb) = arb.nextfreetime - \Delta t \tag{3.32}$$

Hvor *oprettelsestid* er tidspunktet, hvor arbejderinformationen blev genereret.

3.6.5 Modellen

Vi vil nu beskrive model 2, som tager udgangspunkt i model 1, men tager højde for problemstillingerne beskrevet ovenfor. Den væsentligste ændring fra model 1 berører generering af informationen samt behandlingen af denne. Udtrykkene for hvornår der skal hentes opgaver fra et knudepunkts forælder, samt for hvornår der må hentes opgaver fra et barn, er de samme som i model 1, dog med nye definitioner af belastningstilstanden.

Vi vil nu beskrive principperne i modellen og starter med hvordan informationen genereres. Informationen indeholder information om opgaver og arbejdere. Informationen om både opgaver og arbejdere er mængder. Hvert element i opgavemængden indeholder et antal og en størrelse (CPUenheder pr. opgave). Elementerne i informationsmængden for arbejderne indeholder et antal, en størrelse (CPUenheder/arbejder/sekund) samt forventet næste fri tid. Der er rimelig forskel i hvordan informationen genereres afhængig af om der er tale om et site eller et supersite.

Indsamling af information i et site

Informationen i et site indeholder information om opgaverne, der ligger i sitets lokale kø, samt information om arbejderne hos den lokale ressourcemanager.

$$ArbejderInfo = HentArbejderInfo() \tag{3.33}$$

$$OpgaveInfo = HentOpgaveInfo() \tag{3.34}$$

Hvor *HentArbejderInfo* henter information om arbejderne hos den lokale ressourcemanager og *HentOpgaveInfo* henter information om de opgaver, der ligger i sitets lokale kø. Informationen vil herefter se ud som:

$$\begin{aligned} \mathit{ArbejderInfo} &= \{\langle 1\ stk, 2.4\ CPUenheder/arb/s, 12\ s\rangle, \ldots\} && (3.35) \\ \mathit{OpgaveInfo} &= \{\langle 1\ stk, 261\ CPUenheder/opg\rangle, \ldots\} && (3.36) \end{aligned}$$

Indsamling af information i et supersite

I et supersite indsamles informationen ved at summere informationen, som supersitets børn har sendt til det. Da informationen fra børnene kan være genereret et stykke tid i forvejen, så tages der højde for denne tid. Den komplette information skal desuden indeholde information om de opgaver, der måtte ligge i supersitets buffer.

$$\begin{aligned} \mathit{ArbejderInfo} &= \bigcup_{i \in boern} TimeChange(ArbejderInfo_i) && (3.37) \\ \mathit{OpgaveInfo} &= \mathit{HentOpgaveInfo}() \cup \bigcup_{i \in boern} OpgaveInfo_i && (3.38) \end{aligned}$$

Hvor *TimeChange* ændrer fri-tiderne for arbejderne (beskrevet i afsnit 3.6.4) og *HentOpgaveInfo* henter information om de opgaver, der måtte ligge i supersitets lokale buffer. Informationen vil herefter kunne se ud som:

$$\begin{aligned} \mathit{ArbejderInfo} &= \{\langle 7\ stk, 2.1\ CPUenheder/arb/s, 3\ s\rangle, \ldots\} && (3.39) \\ \mathit{OpgaveInfo} &= \{\langle 18\ stk, 221\ CPUenheder/opg\rangle, \ldots\} && (3.40) \end{aligned}$$

Behandling af information

Når informationen er blevet genereret, så skal den *behandles*. Efter behandlingen skal informationen sendes til forælderknudepunktet, samt den skal gemmes lokalt, således at man det næste stykke tid kan beregne belastningstilstanden. Informationen, forælderknudepunktet skal have, skal indeholde oplysninger om hvilke opgaver der kan hentes (de opgaver der er i overskud), samt hvordan arbejderne ser ud. Det er her vigtigt at knudepunktet ikke rapporterer opgaver op, som det selv mener det vil kunne påbegynde førend X_{max}. Dette betyder også, at knudepunktet ikke må rapportere arbejdere op, der er frie før X_{max}, hvis det har opgaver, der egentlig kan løses på dem. Til dette bruges skemalægningen.

$$\begin{aligned} &\langle \mathit{NyArbejderInfo}, \mathit{NyOpgaveInfo}\rangle = \\ &\qquad \mathit{LaegSkema}(\mathit{ArbejderInfo}, \mathit{OpgaveInfo}, X_{max}) && (3.41) \end{aligned}$$

Når informationen er blevet behandlet, gemmes denne i knudepunktet til brug ved bestemmelse af knudepunktets egen belastningstilstand.

Send information til forælder

Informationen skal nu også sendes til forælderknudepunktet. Som beskrevet i afsnit 3.6.2 minimeres informationen inden den sendes til forælderen. Minimeringen gør at hvis vi f.eks. har mange opgaver som er *rimelig* ens, så kan disse beskrives med ét element i opgavemængden.

$$MinArbejderInfo = Minimize(NyArbejderInfo) \tag{3.42}$$
$$MinOpgaveInfo = Minimize(NyOpgaveInfo) \tag{3.43}$$
$$SendInfo(MinArbejderInfo, MinOpgaveInfo) \tag{3.44}$$

Modtag information

Når et supersite modtager information fra et af dets børn, tidsstemples dette og gemmes til brug ved næste informationsgenerering, samt til at kunne bestemme barnets belastningstilstand. Tidsstemplingen bruges til, at kunne tage højde for den tid der er gået, når informationen skal bruges som beskrevet i afsnit 3.6.4.

Flytning af opgaver

Der er to måder, hvorpå der flyttes opgaver. Man kan hente opgaver fra sin forælder og man kan hente opgaver fra sine børn. Modellen bruger samme opgaveflytningsprincipper som i model 1 (ligning (3.13) og (3.14)). I disse ligninger bruges belastningstilstanden og den definerer vi nu ud fra den lokale information, som er gemt i knudepunktet. I ligning (3.26) kom vi med den generelle regel for belastningstilstanden, men vil her komme med en hurtigere version. Da vi har lagt skema op til X_{max} for informationen, må det betyde, at hvis der stadig er opgaver tilbage, så er vi tungt belastet. Ligeledes hvis der ikke er nogle opgaver, så kan vi ikke være tungt belastet.

Førend belastningstilstanden kan bestemmes, skal tages højde for den tid der er gået, siden informationen blev genereret. Vi bliver derfor nødt til at ændre arbejdernes næste fri tid og lægge skema igen, for at være sikre på, at vi ikke har opgaver liggende samtidig med, at der er arbejdere, der er frie før X_{max}.

$$tmpArbejderInfo = TimeChange(NyArbejderInfo) \tag{3.45}$$
$$\langle NyOpgaveInfo, NyArbejderInfo\rangle = LaegSkema(NyOpgaveInfo, tmpArbejderInfo, X_{max}) \tag{3.46}$$

Belastningstilstanden defineres nu som:

$$BelastningsTilstand = \begin{array}{l} \text{if } antal(NyOpgaveInfo) > 0 \text{ then } Tung \\ \text{else if } NaesteFriTid(NyArbejderInfo) < X_{min} \text{ then } Let \\ \text{else } Normal \end{array} \tag{3.47}$$

Hvor *NaesteFriTid* fortæller, hvor lang tid der vil være, til der er en fri arbejder.

Registrering af flyttede opgaver

Et meget vigtigt punkt er, at knudepunkterne registrerer, når de har hentet en opgave og når der er blevet hentet en opgave fra dem. Hvis dette ikke gøres, så vil et supersite f.eks. ikke opdage, at det efter at have hentet et antal opgaver fra et barn, ikke skal hente flere (barnet er ikke længere tungt belastet). Ligeledes er det vigtigt, at informationen om et barn bliver opdateret korrekt, da denne information bruges til at generere de nye informationer, som sendes til et knudepunkts forælder.

Når man har hentet en opgave fra sin forælder, er det fordi man er let belastet og opgaven registreres ved at man lægger den på den arbejder, som har den mindste næste fri tid.

$$\begin{aligned} OpgaveCPU &= s\ CPUenheder && (3.48)\\ arb &= \mathit{fjernFoersteFrieArbejder}(\mathit{NyArbejderInfo}) && (3.49)\\ arb.nextFreeTime &= arb.nextFreeTime + \frac{OpgaveCPU}{arb.cpu} && (3.50)\\ \mathit{NyArbejderInfo} &= \mathit{NyArbejderInfo} \cup \{arb\} && (3.51) \end{aligned}$$

hvor *fjernFoersteFrieArbejder* fjerner en arbejder fra den arbejderinformation, som har den mindste næste fri tid.

Når man har hentet en opgave op fra et barn, så skal det registreres, at barnet ikke længere har denne opgave. Dette lyder simpelt. Men da informationen, som dette skal registreres i, er blevet minimeret (da barnet sendte den op), så vil der ikke nødvendigvis være elementer i opgavemængden, som ligner denne på en prik. Man skal derfor finde et opgaveelement, som minder om opgaven med hensyn til størrelsen, og i dette element fjerne én opgave. Det er vigtigt at den totale mængde af CPUenheder i hele opgavemængden er korrekt, og derfor beregnes et nyt gennemsnit for elementet.

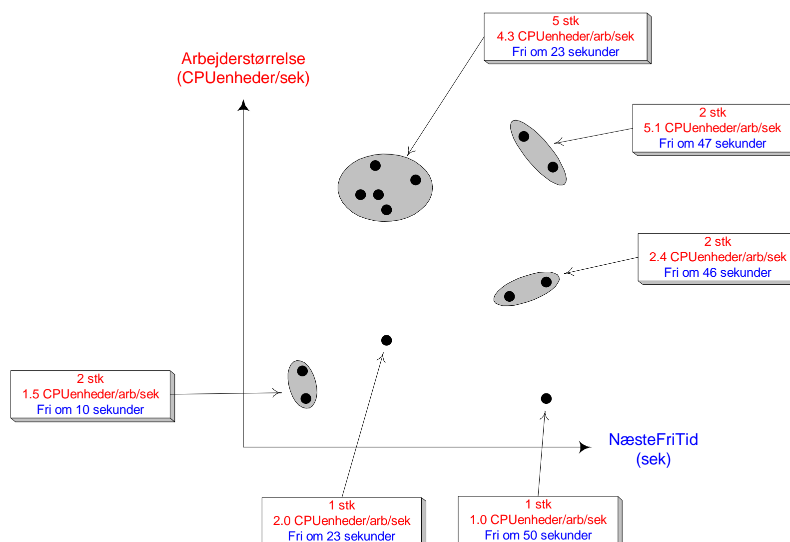
$$\begin{aligned} FjernOpgaveCPU &= s\ CPUenheder && (3.52)\\ OrigOpgaveElem &= \langle X\ stk, S\ CPUenheder\rangle && (3.53)\\ NytOpgaveElem &= \langle X-1\ stk, \frac{S*X-s}{X-1}\ CPUenheder/opg\rangle && (3.54) \end{aligned}$$

I de tilfælde hvor *OrigOpgaveElem* kun indeholder information om én opgave, vil ovenstående resulterer i en ny opgaveinformation med 0 opgaver og en resterende CPUenhedsmængde som højst sandsynligt ikke er 0 (kan både være negativ og positiv). Vi har valgt for simplicitetens skyld at *glemme* den resterende del CPU, men der burde findes en løsning, således at den resterende CPU-mængde stadig vil være registreret.

Aflevere opgaver til lokal ressourcemanager

Den sidste del i modellen er afleveringen af opgaver til den lokale ressourcemanager i et site. Opgaver afleveres til ressourcemanageren, hvis denne har en arbejder, som kan påbegynde opgaven indenfor X_{max} sekunder. Dette gøres, da vi har defineret, at et site altid vil løse opgaver selv, som kan påbegyndes indenfor X_{max} sekunder.

$$\begin{aligned} &\mathit{NaesteFriTid}(\mathit{HentArbejderInfo}()) \le X_{max} \wedge \mathit{HarOpgaver} && (3.55)\\ &\qquad \Rightarrow \mathit{AfleverOpgave} \end{aligned}$$



Figur 3.11: Eksempel på hvordan arbejdere kunne inddeles.

3.6.6 Vurdering af model 2

Hvis man sammenligner denne model med model 1, så er der nogle klare forbedringer. Med indførslen af skemalægning, hvor vi tager højde for forskellige CPU forbrug hos opgaver og forskellige udbud af regnekraft hos arbejderne, er vi kommet udenom problemerne fra model 1 med, at opgaver kan ligge længe i kø førend de bliver påbegyndt. I denne model flyttes opgaverne først ned mod en lokal ressourcemanager, når systemet mener, at der er X_{min} sekunder til, at en arbejder kan påbegynde opgaven. Således afhænger flytningen af opgaver ikke af, hvor mange opgaver det er, der ligger i køer rundt omkring (som i model 1), med derimod om, hvornår arbejdere forventes frie.

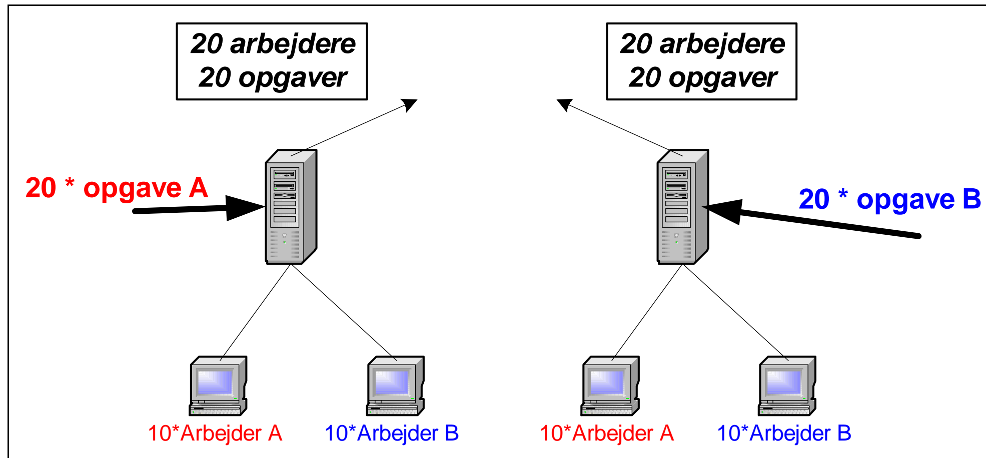
Da modellen tager højde for den tid der er gået, fra informationer er blevet indsamlet, til de bruges, kan man nedsætte opdateringsfrekvensen for informationsudvekslingen mellem knudepunkter betydeligt i forhold til model 1.

Intervallerne til minimering af opgaver og til minimering af arbejder burde undersøges nærmere, således at disse måske kan vælges mere optimalt. I stedet for at bruge faste intervaller, kunne man bruge mere flydende intervaller eller måske finde en helt tredje måde at samle opgaverne/arbejderne på. Arbejderne kunne f.eks. inddeles efter, hvor de *klumper sig sammen* som vist på figur 3.11.

3.7 Model 3 - Opgaver og arbejdere med forskellige krav/udbud

Denne model er vores endelige og en udvidelse af model 2. I modellen tages nu højde for, at ikke alle arbejdere kan løse alle opgaver, hvilket gør loadbalanceringen meget mere vanskelig. Modellen vil opføre sig som model 2, hvis alle arbejdere kan løse alle opgaver.

Der findes i dag systemer, som laver en loadbalancering, der minder om model 2 (f.eks. Palaber[18]). Problemet med systemer som disse er, at de ikke tager højde for opgavers



Figur 3.12: Eksempel på loadubalance.

krav til og arbejderes udbud af ressourcer. Dette giver anledning til, at der opstår loadubalance (engelsk: load imbalancing), selvom systemet mener, at der ikke er noget at balancere. I figur 3.12 ses et eksempel på, hvordan en sådan fejl kan opstå. Begge sites fortæller, at de har 20 arbejdere og 20 opgaver, hvilket lyder som om at der ikke er nogle problemer. Men i virkeligheden er der kun 10 arbejdere der kan løse de 20 opgaver i hvert site. Derfor burde der foregå en loadbalancering mellem de to steder, således at de hver havde 10 opgaver af hver opgavetype.

3.7.1 Krav og udbud

I modellen tages der højde for at opgaver har krav til ressourcer og at arbejdere har et ressourceudbud.

En arbejders ressourceudbud består af en mængde enkeltstående udbud, en for hver ressourcetype den har. Hvert af disse udbud består af en beskrivelse af ressourcen samt en værdi. Værdidomænet vil for nogle ressourcer være diskret, mens det for andre vil være kontinuert. I følgende tabel vises nogle eksempler på enkeltstående ressourceudbud.

Navn	Type	Enhed	Eksempler
RAM	kontinuert	MB	34, 512, 86
FriDisk	kontinuert	MB	1000, 8000, 500
CPU	kontinuert	CPUenheder	7.4, 9.1, 2.0
OS	diskret	Navn	Windows 95, Linux, SunOS
Java	diskret	Version	1.1, 1.2, 1.3, 1.4

En arbejders ressourceudbud vil være en kombination af flere enkelte ressourceudbud. Et eksempel på en arbejder kan være:

Navn	Værdi
RAM	421
FriDisk	3000
CPU	4.1
OS	Linux
Java	1.3

Opgavers ressourcekrav virker på samme måde. En opgaves ressourcekrav vil f.eks. være:

Navn	Værdi
RAM	60
Java	1.3

Navnene er unikke hele systemet, således at f.eks. RAM altid hedder RAM og ikke nogle gange *hukommelse*. Ligeledes skal enheder og typer være fastsat på forhånd for hvert navn.

Disse krav og udbud vil være opbygget som et sæt af (navn,værdi). De to ovenstående eksempler vil repræsenteres som:

$$\begin{aligned} U = udbud(arb) &= \{\langle RAM, 421\rangle, \langle FriDisk, 3000\rangle, \langle CPU, 4.1\rangle, \\ &\quad \langle OS, Linux\rangle, \langle Java, 1.3\rangle\} \qquad (3.56) \\ K = krav(opg) &= \{\langle RAM, 60\rangle, \langle Java, 1.3\rangle\} \qquad (3.57) \end{aligned}$$

Ligeledes indfører vi at $name(k)$ og $value(k)$ returnerer navnet henholdsvis værdien for et krav- eller udbudselement k.

Vi har nu indført, hvordan man beskriver opgavers ressourcekrav og arbejderes ressourceudbud. Vi vil nu definere, hvordan man undersøger, om en opgave kan udføres hos en arbejder. K er opgavens ressourcekrav og U er arbejderens ressourceudbud.

$$\begin{aligned} K &= krav(opg) = \{k_1, k_2, \ldots\} \qquad (3.58) \\ U &= udbud(arb) = \{u_1, u_2, \ldots\} \qquad (3.59) \\ Match(K, U) &\equiv \forall k \in K \bullet \exists u \in U \bullet \qquad (3.60) \\ &\quad name(k) = name(u) \wedge singlematch(k, u) \end{aligned}$$

Hvor *singlematch* afhænger af, om det er en type med kontinuert eller diskret værdidomæne:

$$\begin{aligned} singlematch_{kontinuert}(k, u) &\equiv value(k) \leq value(u) \qquad (3.61) \\ singlematch_{diskret}(k, u) &\equiv value(k) = value(u) \qquad (3.62) \end{aligned}$$

Som det ses af ligning (3.60), så accepteres det, at arbejdere har flere udbud, end en matchene opgaver har krav. Ligeledes ses det af ligning (3.62), at vi også accepterer, at der for kontinuerte værdidomæner er mere af en ressource end opgaven skal bruge (f.eks. er det acceptabelt, at en opgave, der kræver 50 MB RAM, matcher en arbejder, som udbyder 200 MB RAM).

Vi har nu defineret hvordan opgaver og arbejdere har krav henholdsvis udbud. Ligeledes har vi defineret, hvordan man undersøger, om en opgave kan løses hos en arbejder.

3.7.2 Gruppering af opgaver

For at undgå, at der er alt for mange forskellige opgavetyper i systemet, indfører vi begrebet *opgavegruppe*. En opgavegruppe er en gruppe af opgaver, hvor opgaver, der

minder om hinanden med hensyn til deres ressourcekrav, er i samme gruppe. Ved at lave en sådan gruppering, vil mange opgaver, der har en lille smule forskel i deres krav, kunne repræsenteres med ens krav.

Vi definerer nu et udtryk, som fortæller om to opgaver minder om hinanden. For opgaverne o_1 og o_2 gælder:

$$\begin{aligned} Similar(o_1, o_2) &\equiv SimilarKrav(krav(o_1), krav(o_2)) && (3.63) \\ SimilarKrav(K_1, K_2) &\equiv names(K_1) = names(K_2) \wedge && (3.64) \\ &\quad \forall n \in names(K_1) \bullet singleSimilar(K_1(n), K_2(n)) \end{aligned}$$

Hvor *singleSimilar* undersøger, om hvert enkelt ressourcekrav minder om hinanden. For krav, der har et diskret værdidomæne, siger vi, at kravene minder om hinanden, hvis og kun hvis værdierne er ens. For krav, som har et kontinuert værdidomæne, transformerer vi værdierne over i et diskret værdidomæne og siger at de minder om hinanden, hvis transformationen fører til samme værdi.

$$\begin{aligned} singleSimilar_{kontinuert}(k_1, k_2) &\equiv \lambda(value(k_1)) = \lambda(value(k_2)) && (3.65) \\ singleSimilar_{diskret}(k_1, k_2) &\equiv value(k_1) = value(k_2) && (3.66) \end{aligned}$$

En sådan transformeringsfunktion λ kan for RAM f.eks. være:

$$\lambda_{RAM}(val) \equiv \begin{cases} 32 & val \leq 32 \\ 64 & val \in]32; 64] \\ 128 & val \in]64; 128] \\ ... \end{cases} \quad (3.67)$$

Der køres med en nedre grænse, for at opgaver, som kræver meget lidt af en ressource, ender op med at minde om andre opgaver, som også kræver meget lidt af den pågældende ressource. Dette også selvom forskellen i to værdier er flere hundrede procent. Hvis man f.eks. har to opgaver, hvor den ene kræver 16KB RAM og den anden 1MB RAM, så kræver opgaverne bare *lidt* RAM.

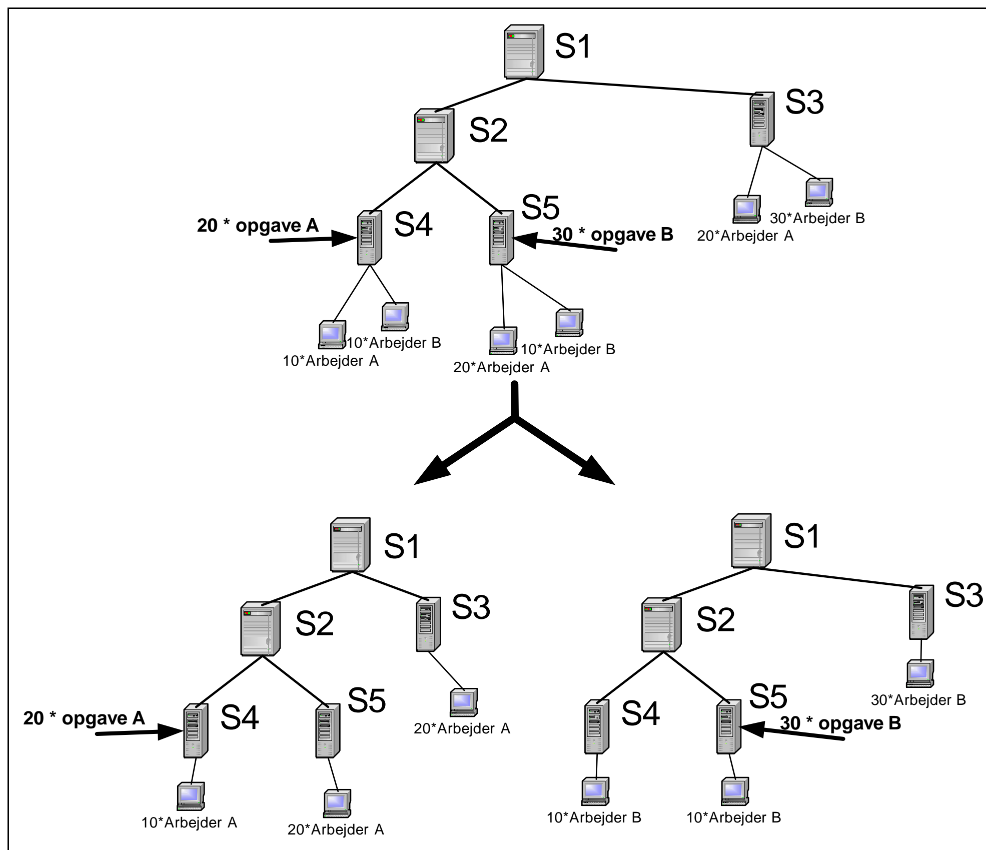
Vi definerer nu en opgavegruppe, som værende alle de opgaver, der *minder* om hinanden. Til en opgavegruppe hører også en opgaverepræsentant. Dette er en pseudo-opgave, hvis krav repræsenterer gruppens krav. Vi definerer nu de krav denne opgave skal have: (K_{opgave} er ressourcekravene for en opgave i gruppen)

$$\begin{aligned} K_{opgave} &= \{k_1, k_2, \ldots\} && (3.68) \\ K_{reprae} &= \bigcup_{k \in K_{opgave}} \text{if } type(k) = kont \text{ then } \langle name(k), \lambda(value(k)) \rangle \text{ else } k && (3.69) \end{aligned}$$

Det ses, at lige meget hvilken opgave der vælges fra gruppen, så vil det give samme resultat. Kravene er valgt således, at det altid er sikkert, at hvis en arbejder matcher opgaverepræsentanten, så matcher den også alle opgaverne i opgavegruppen.

3.7.3 Flere loadbalanceringer

Som beskrevet i indledningen til denne model, er den en udvidelse af model 2. Vi viser nu et eksempel, hvori model 2 optræder to gange (se figur 3.13). I eksemplet haves to



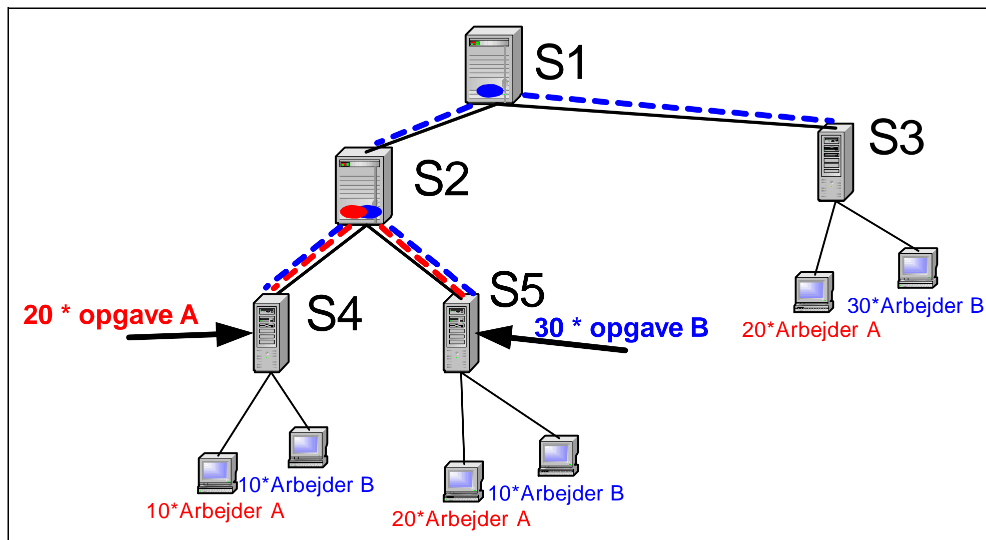
Figur 3.13: Eksempel på hvordan loadbalanceringen af to opgavegrupper kan splittes op i to separate systemer som model 2.

typer opgaver (to opgavegrupper). Den ene opgavegruppe O_a skal bruge 32 MB RAM, og den anden opgavegruppe O_b skal bruge 100 MB fri harddisk. Ligeledes haves to typer arbejdere, hvor den ene A_a tilbyder 100 MB RAM og den anden A_b tilbyder de 1 GB fri harddisk. Øverst i figuren, ses det hvordan eksemplet ser ud. Da A_a kun kan løse opgaver fra O_a og A_b kun kan løse opgaver fra O_b, kan man lave to adskilte loadbalanceringer som i model 2. Dette ses i bunden af figuren.

3.7.4 Aktive opgavegrupper

I eksemplet ovenfor kan man indføre to systemer som model 2. Herved rapporteres de to typer arbejdere og opgaver opad i systemet, som var der egentlig to helt separate systemer af model 2.

Der findes uendeligt mange forskellige kombinationer af opgavekrav og kombinationer af arbejderudbud. Derfor skulle der også bruges uendeligt mange systemer af model 2, et for hvert unikke krav/udbud. Dette er dog ikke muligt, da der skal sendes information om hver enkelt system.
For at komme uden om dette problem, indfører vi, at et knudepunkt skal fortælle sine børn, hvilke oplysninger det ønsker. Det betyder, at et knudepunkt ikke må sende oplysninger til sin forælder, førend forælderen har bedt om dem.
I eksemplet ovenfor betyder det, at knudepunkt S2 vil fortælle sine børn, at de skal sende



Figur 3.14: Figuren viser hvor opgavegrupperne i figur 3.13 er aktive.

information om arbejdere, som kan løse opgaver tilhørende opgavegrupperne O_a og O_b. Herved vil S4 og S5 kun rapportere information om arbejderne A_a og A_b, også selvom de måtte have haft nogle andre arbejdere (f.eks. A_c, hvor A_c ikke kan løse opgaverne i O_a og O_b). Ligeledes vil knudepunkt S1 kun fortælle, at den vil vide noget om, hvilke arbejdere S2 og S3 kan tilbyde, som kan løse opgaver fra opgavegruppen O_b, idet knudepunkt S2 kan klare problemet i opgavegruppen O_a selv.

Vi indfører nu begrebet en *aktiv* opgavegruppe. En opgavegruppe siges at være aktiv i et knudepunkt, hvis knudepunktet skal have information op fra sine børn om, hvilke arbejdere børnene kan tilbyde, der kan løse opgaver fra den givne opgavegruppe. I figur 3.14 ses, hvor de to opgavegrupper i eksemplet er aktive.

På figuren kan man se, at opgavegruppen O_a har sit toppunkt i S2. Toppunktet er i netop dette knudepunkt, fordi S2 kan beregne, at belastningstilstanden i opgavegruppen ikke er tung, og at den derfor ikke behøver hjælp fra sin forælder.

For at et knudepunkt kan bestemme sig for, om belastningstilstanden er tung indenfor en opgavegruppe, er det nødvendigt for det at vide hvilke arbejdere, der er i dets undertræ, som kan løse opgaver fra opgavegruppen, samt hvordan disse arbejdere ser ud. Dette gøres ved at knudepunktet aktiverer opgavegruppen i *hele* dets undertræ, hvorefter knudepunktets børn vil sende information om arbejderne. Desuden er det nødvendigt at blive ved med at modtage informationen fra børnene, indtil problemet i opgavegruppen er løst, da der f.eks. kan stilles nye opgaver, arbejdere kan forsvinde/komme, opgaverne tager kortere/længere tid end forventet osv. Derfor indfører vi, at der altid gælder:

$$G_i \supseteq G_{foraelder(i)} \tag{3.70}$$

hvor G_i er mængden med aktive opgavegrupper i knudepunkt i.

Ligning (3.70) betyder, at når en opgavegruppe er aktiv i et knudepunkt, så vil den også altid være aktiv i alle knudepunktets børn, børnebørn osv.

3.7.5 Åbning og lukning af aktive opgavegrupper

Vi har nu beskrevet, hvordan man ud fra belastningstilstandene i hvert knudepunkt, kan bestemme hvilke opgavegrupper, der bør være aktive hvor i træet. Da opgaver bliver løst, nye opgaver ankommer, arbejdere kommer og forsvinder, osv, vil belastningstilstandene hele tiden ændre sig. Hermed vil det også hele tiden ændre på hvilke opgavegrupper, der er aktive hvor i træet. Vi indfører derfor et system, som sørger for at *åbne* og *lukke* aktive opgavegrupper.

Åbning og lukning af aktive opgavegrupper foregår altid oppefra og ned. Dvs. at hvis et barn opdager, at det har et problem, kan det ikke selv gøre opgavegruppen aktiv. Det kan fortælle dets forælder om problemet. Forælderen kan herefter diktere dets børn, at opgavegruppen skal gøres aktiv. Når et barn får at vide, at det skal aktivere opgavegruppen, så gør barnet det. Barnet må dog først rapportere op, at opgavegruppen er aktiv, når opgavegruppen er aktiv hos alle dens egne børn og den har fået information fra disse. I sites, som ikke har nogle børn, kan man aktivere opgavegruppen og sende information med det samme.

Åbning af en aktiv opgavegruppe:

Når et site opdager at det er tungt belastet i en opgavegruppe og denne opgavegruppe ikke er aktiv til forælderen, så sender sitet en forespørgsel til forælderknudepunktet om at aktivere opgavegruppen. I ligningen er i en opgavegruppe.

$$BelastningsTilstand_i^{opg} = Tung \wedge i \notin AktiveOpgavegrupper^{foraelder} \Rightarrow BedForaelderAktivere(i) \tag{3.71}$$

Hvor $BelastningsTilstand_i^{opg}$ er belastningstilstanden indenfor opgavegruppen i. $AktiveOpgavegrupper^{foraelder}$ indeholder de opgavegrupper, som er aktive til forælderen.

Når forælderen (et supersite) modtager en aktiveringsforespørgsel, så beder det alle dets børn aktivere opgavegruppen. Opgavegruppen er først rigtig aktiv, når *alle* børnene har rapporteret op, at den er aktiv. Først når opgavegruppen er rigtig aktiv, kan knudepunktet bestemme sig for, om det også har brug for, at opgavegruppen er aktiv til dets forælder. Dette bestemmes af:

$$i \in AktiveOpgavegrupper \wedge i \notin AktiveOpgavegrupper^{foraelder} \wedge BelastningsTilstand_i^{opg} = Tung \Rightarrow BedForaelderAktivere(i) \tag{3.72}$$

Når et site får at vide fra dens forælder, at det skal aktivere en opgavegruppe, så kan dette ske med det samme. Et supersite kan først rapportere, at opgavegruppen er aktiv, når alle dens børn har sagt, at opgavegruppen er aktiv.

Lukning af en aktiv opgavegruppe:

Når en opgavegruppe skal inaktiveres, så foregår det altid fra toppunktet af det aktive træ. En opgavegruppe inaktiveres, når supersitet kan se, at der ikke længere er nogle af dets børn, der er tungt belastede i opgavegruppen. Supersitet fortæller alle sine børn, at

opgavegruppen ikke længere er aktiv hos deres forælder. Herved bliver alle børnene nu selv toppunkter for opgavegruppen. De kan så hver især bestemme sig for, om de også ønsker at lukke videre ned gennem hierarkiet.

For et supersite gælder for en opgavegruppe i:

$$\begin{aligned} & i \in \mathit{AktiveOpgavegrupper} \wedge i \notin \mathit{AktiveOpgavegrupper}^{foraelder} \wedge \\ & \forall b \in \mathit{boern} \bullet \mathit{BelastTilstand}_{b,i} \neq \mathit{Tung} \Rightarrow \mathit{SendInaktiverGruppe}(i) \end{aligned} \tag{3.73}$$

Hvor $\mathit{BelastTilstand}_{b,i}$ fortæller belastningstilstanden i opgavegruppen i hos barnet b. $\mathit{SendInaktiverGruppe}(i)$ sender en *inaktiver opgavegruppe* i-besked til alle dets børn.

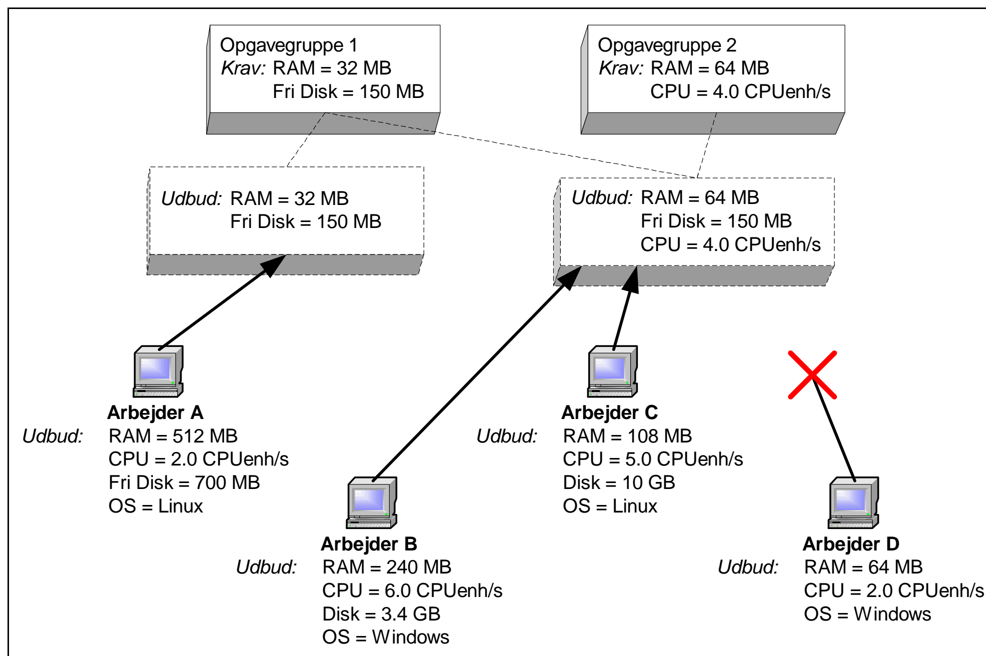
I et site er der ikke egentlige aktive opgavegrupper, da det ikke har nogle børn. Derfor kan et site glemme alt om opgavegruppen, når den modtager en inaktiver besked.

3.7.6 Arbejdere kan løse flere forskellige opgaver

Indtil nu har vi set på eksempler, hvor hver arbejder kun kunne løse en specifik opgavegruppe. Men man bliver nødt til at tage højde for, at en arbejder kan løse opgaver fra flere forskellige opgavegrupper. Dette kan f.eks. være en arbejder der udbyder 256 MB RAM. Denne arbejder kan løse opgaver fra både en opgavegruppe, der kræver 64 MB RAM, en som kræver 128 MB RAM og mange flere. Ligeledes kan en arbejder, som både udbyder RAM og disk, løse opgaver, som kun kræver RAM, og opgaver, som kun kræver disk, samt kombinationen.
Når et knudepunkt skal sende information om disse arbejdere op til dets forælder, kan det ikke vide, om arbejderen skal bruges til at løse opgaver for den ene opgavegruppe eller den anden. Det eneste det ved er, at dets forælder har fortalt, at det skal rapportere oplysninger op om arbejdere, der kan løse opgaver fra de aktive opgavegrupper. Derfor bliver knudepunktet nødt til at få rapporteret arbejderen op således, at forælderen selv kan bestemme, til hvilken opgavegruppe arbejderen skal bruges.
Da man ved, at forælderen *kun* ønsker oplysninger om de aktive opgavegrupper, kan man lave en *indordning* af arbejderne til disse opgavegrupper. Med indordning menes, at når en arbejder rapporteres op, så ændres dens ressourceudbud således, at ressourceudbudet netop udspænder det udbud, som skal til for at opfylde de matchene opgavegruppers krav. Fordelen ved at lade arbejderne *indordne* sig til opgavegrupperne er, at arbejdere, som er forskellige, kan beskrives som ens, hvis de kan løse opgaver fra nøjagtig de samme opgavegrupper. Dette gør at antallet af forskellige arbejdere, som skal rapporteres op, bliver minimeret og at de som skal rapporteres op kan beskrives i grupper, istedet for enkeltstående arbejdere med forskellige ressourceudbud. Indordningen af arbejdere til aktive opgavegrupper kan beskrives som følgende:

$$\begin{aligned} G_{match} &= \{ g \in \mathit{AktiveOpgavegrupper}^{foraelder} \mid \\ &\quad\; match(udbud(arb), krav(g)) \} && (3.74) \\ K_{match} &= \bigcup_{g \in G_{match}} krav(g) && (3.75) \\ NytUdbud &= span(K_{match}) && (3.76) \end{aligned}$$

Hvor $span(\{k_0, k_1, ..., k_n\})$ er en funktion, som laver det mindste udbud, som kan opfylde kravene $k_0, k_1, ..., k_n$. Skulle G_{match} være tom, så betyder det, at arbejderen ikke kan løse nogle af opgavegrupperne. Arbejderen kan derfor fjernes helt fra informationen.



Figur 3.15: Indordning af arbejdere til aktive opgavegrupper.

I figur 3.15 ses nogle eksempler på, hvordan en sådan indordning virker.

Når alle arbejderne har fået lavet deres nye ressourceudbud, vil de optræde i grupper. Herved kan informationen sendes op til forælderen, som en mere simpel information end hvis de skulle rapporteres op enkeltvis. Informationen som sendes op om *hver* arbejdergruppe er den samme, som blev sendt op for arbejderne i model 2.

3.7.7 Belastningstilstande og skemalægning

Ligesom i de andre modeller bruger vi begrebet belastningstilstand til at bestemme, hvornår der skal flyttes opgaver. I denne model definerer vi en belastningstilstand indenfor hver enkelte aktive opgavegruppe, således at man f.eks. kan sige, at vi er tungt belastet i opgavegruppen 32 MB RAM. Ligeledes definerer vi også en belastningstilstand indenfor hver aktive arbejdergruppe, således at man f.eks. kan sige, at arbejdergruppen 64 MB RAM er let belastet.

Definitionen af belastningstilstandene er en smule anderledes end i de andre modeller. Belastningstilstandene bruges i to tilfælde: Når opgaver skal hentes fra et barn, og når opgaver skal hentes fra en forælder.

Når et supersite henter opgaver fra et barn, så er det opgaver tilhørende en bestemt opgavegruppe, der skal hentes. Til at bestemme om en opgave skal hentes, bruges belastningstilstanden indenfor opgavegruppen hos barnet. Hvis denne er tung, må der hentes opgaver og ellers må der ikke. Vi nøjes derfor med at definere om en opgavegruppe er tungt belastet eller ej.

Når et knudepunkt skal hente opgaver fra dets forælder, så gøres dette ud fra en arbejdergruppe. Opgaver må hentes så længe arbejdergruppen er let belastet. Vi definerer derfor belastningstilstanden indenfor en arbejdergruppe som være let eller ej.

Ligesom i model 2 bestemmes belastningstilstandende ved, at der lægges skema. Skemalægningen i denne model minder en del om den i model 2. Den store forskel er, at der nu skal tages højde for, at ikke alle opgaver kan løses på alle arbejdere.

Man kan nu efter skemalægning bestemme belastningstilstandene for arbejder- og opgavegrupper.

For den opgavegruppen o i et knudepunkt bestemmes belastningstilstanden ved:

$$BelastningsTilstand_o^{opg} = \text{if } SidsteStartTid(Arbejdere, o) > X_{max} \text{ then } Tung \text{ else } ikketung \tag{3.77}$$

Hvor $SidsteStartTid(Arbejdere, o)$ fortæller det sidste tidspunkt, hvor en opgave tilhørende opgavegruppen o bliver påbegyndt på en arbejder i $Arbejdere$.

For den aktive arbejdergruppe a i et knudepunkt bestemmes belastningstilstanden ved:

$$BelastningsTilstand_a^{arb} = \text{if } NaesteFriTid(Arbejdere, a) < X_{min} \text{ then } Let \text{ else } ikkelet \tag{3.78}$$

Hvor $NaesteFriTid(Arbejdere, a)$ fortæller, hvornår den tidligst frie arbejder tilhørende arbejdergruppen a vil være fri.

Vi har valgt at bruge næsten den samme skemalægningsalgoritme som i model 2. Den er dog blevet ændret således, at der nu tages højde for, at en opgave ikke kan løses på alle arbejdere. Vi ved, at denne algoritme ikke er optimal. Det er fordi vores algoritme simpelt tager de mindste opgaver og lægger disse på de tidligst frie arbejdere, som kan løse opgaverne. Herved kan man komme til at lægge opgaver, som f.eks. kun kræver 16 MB RAM, på en arbejder, som udbyder 1 GB RAM og 2 T disk. Dette betyder at andre opgaver som f.eks. kræver meget RAM og disk ikke vil kunne lægges ind i skemaet på en arbejder, der er fri før X_{max}. Dette problem vil vi belyse mere i afsnit 3.8.

3.7.8 Informationsniveau

De to andre modeller vi har vist, havde begge den fordel, at størrelsen på informationen, som sendes opad i træet, er konstant/har en maksimal størrelse. I denne model er der ikke en øvre grænse på størrelsen af informationen, da denne afhænger af antallet af opgave- og arbejdergrupper, der skal rapporteres op. Antallet af arbejdergrupper, som kan fortælles opad i træet, kan eksplodere, da antallet af mulige arbejdergrupper teoretisk er:

$$2^{antal\ aktive\ opgavegrupper} \tag{3.79}$$

I virkeligheden er der dog ikke nær så mange. Hvis der f.eks. er en gruppe opgaver, som kræver OS=Linux og en anden som kræver OS=Windows, så kan der ikke eksistere en arbejdergruppe, som kan dække begge disse (det understøtter denne model i hvert fald ikke). Men stadig vil antallet af mulige arbejdergrupper være stærkt voksende med antallet af aktive opgavegrupper. Vi mener dog at dette kan undgås, ved at indføre nogle regler for, hvordan antallet af arbejdergrupper kan minimeres:

1. Hvis en arbejdergruppe bidrager med mindre end $x\%$ til en opgavegruppe, så kan man fjerne opgavegruppen fra indordningen (ligning (3.74)-(3.76)), så arbejdergruppen ikke kan løse denne opgavegruppe. Således vil arbejdergruppen evt. kunne lægges sammen med en anden gruppe.

2. I supersites vil man, når et barn anmoder om at få aktiveret en opgavegrupper, kunne bestemme, om man behøver aktivere opgavegruppen hos *alle* supersitets børn og dermed *hele* undertræet. Ud fra historik og nuværende information i supersitet vil man i mange tilfælde kunne bestemme, om det problem barnet har, måske kan klares ved kun at åbne ned til nogle få andre børn.

3. Det vil også være muligt, at man kan afskære visse dele af træet, hvis man kan bestemme, at denne del af træet ikke vil kunne afhjælpe et problem (eller kun i meget ringe grad).

4. Indførelse af dynamiske størrelser af X_{min} og X_{max}, således at nogle opgavegrupper ikke åbnes førend der virkelig er store problemer.

5. Lave en grænse for hvor mange opgavegrupper og arbejdergrupper, der må være aktive i et knudepunkt samtidig. Herved kan man lukke de aktive opgavegrupper, som har de mindste problemer.

3.7.9 Modellen

Vi vil nu lave en samlet beskrivelse af model 3. I modellen bruges de samme generelle principper som for model 2. Men med indførslen af krav og udbud, ændrer vi på, hvordan et knudepunkt henter opgaver hos dets forælder og dets børn. Informationen adskiller sig også ved, at den nu har en information svarende til den for opgaver og arbejdere i model 2 for *hver* unikke opgavekrav henholdsvis arbejderudbud.

$$\begin{aligned}
OpgaveInfo &= \{O_1, O_2, O_3, \ldots\} && (3.80)\\
O_i &= \{\langle RAM, 35\rangle, \langle OS, Linux\rangle\} \mapsto && (3.81)\\
&\qquad \{\langle X\ \ stk, S\ \ CPUenheder/opg\rangle, \ldots\} \\
ArbejderInfo &= \{A_1, A_2, A_3, \ldots\} && (3.82)\\
A_i &= \{\langle RAM, 132\rangle, \langle OS, Linux\rangle, \langle Disk, 1800\rangle\} \mapsto && (3.83)\\
&\qquad \{\langle Y\ \ stk, Z\ \ CPUenheder/arb/s, F\ \ s\rangle, \ldots\}
\end{aligned}$$

Informationen der sendes opad i træet, minimeres ved, at der kun sendes information om *aktive* opgavegrupper. Ligeledes inddeles opgaver i opgavegrupper (afsnit 3.7.2) og arbejdere indordnes efter de aktive opgavegrupper (afsnit 3.7.6).

I knudepunkterne i hierarkiet opretholdes mængden $AktiveOpgavegrupper^{foraelder}$. Denne mængde indeholder de opgavegrupper, som et knudepunkt skal sende information om til dets forælder.

Indsamling af information i et site

Indsamlingen af information i denne model minder meget om den i model 2. Forskellen er, at vi nu grupperer opgaver og arbejdere ud fra deres krav.

$$ArbejderInfo = HentArbejderInfo() \tag{3.84}$$
$$OpgaveInfo = HentOpgaveInfo() \tag{3.85}$$

Hvor *HentArbejderInfo* henter information om arbejderne hos den lokale ressourcemanager og *HentOpgaveInfo* henter information om de opgaver, der ligger i sitets lokale kø. Informationen vil herefter f.eks. se ud som:

$$ArbejderInfo = \begin{cases} \{\langle RAM, 121\rangle, \langle OS, Linux\rangle\} \mapsto \\ \quad \{\langle 1\ stk, 1.4\ CPUenheder/arb/s, 12\ s\rangle\} \\ \{\langle RAM, 126\rangle, \langle OS, Linux\rangle\} \mapsto \\ \quad \{\langle 1\ stk, 1.7\ CPUenheder/arb/s, 32\ s\rangle\} \\ \{\langle RAM, 381\rangle, \langle OS, Windows\rangle, \langle Disk, 1000\}\rangle \mapsto \\ \quad \{\langle 1\ stk, 2.1\ CPUenheder/arb/s, 124\ s\rangle, \ldots\} \\ \ldots \end{cases} \tag{3.86}$$

$$OpgaveInfo = \begin{cases} \{\langle RAM, 12\rangle, \langle OS, Windows\rangle\} \mapsto \\ \quad \{\langle 1\ stk, 83\ CPUenheder/opg\rangle\} \\ \{\langle RAM, 150\rangle, \langle DISK, 850\rangle, \langle OS, Linux\rangle\} \mapsto \\ \quad \{\langle 1\ stk, 104\ CPUenheder/opg\rangle, \ldots\} \\ \ldots \end{cases} \tag{3.87}$$

Indsamling af information i et supersite

I et supersite minder indsamlingen også meget om den i model 2, men igen hvor der indføres en gruppe for hvert krav-/udbudsmateriale.

$$ArbejderInfo = \bigcup_{i \in boern} TimeChange(ArbejderInfo_i) \tag{3.88}$$
$$OpgaveInfo = HentOpgaveInfo() \cup \bigcup_{i \in boern} OpgaveInfo_i \tag{3.89}$$

Informationen vil herefter f.eks. se ud som:

$$ArbejderInfo = \begin{cases} \{\langle RAM, 128\rangle, \langle OS, Windows\rangle\} \mapsto \\ \quad \{\langle 13\ stk, 0.87\ CPUenheder/arb/s, 9.4\ s\rangle, \\ \quad\ \langle 41\ stk, 1.93\ CPUenheder/arb/s, 0\ s\rangle\} \\ \{\langle RAM, 64\rangle, \langle Disk, 1000\rangle, \langle OS, Linux\rangle\} \mapsto \\ \quad \{\langle 9\ stk, 2.19\ CPUenheder/arb/s, 4\ s\rangle, \ldots\} \\ \ldots \end{cases} \tag{3.90}$$

$$OpgaveInfo = \begin{cases} \{\langle RAM, 16\rangle, \langle OS, Windows\rangle\} \mapsto \\ \quad \{\langle 71\ stk, 82.3\ CPUenheder/opg\rangle, \\ \quad\ \langle 39\ stk, 74.1\ CPUenheder/opg\rangle\} \\ \{\langle RAM, 19\rangle, \langle Java, 1.3\rangle\} \mapsto \\ \quad \{\langle 1\ stk, 82.3\ CPUenheder/opg\rangle\} \\ \{\langle RAM, 256\rangle, \langle DISK, 1000\rangle, \langle OS, Linux\rangle\} \mapsto \\ \quad \{\langle 41\ stk, 108\ CPUenheder/opg\rangle, \ldots\} \\ \ldots \end{cases} \tag{3.91}$$

Behandling af information

Behandlingen af information er nu en del anderledes end den i model 2. Vi starter med at lægge skema op til X_{max} således, at man i den nye opgave- og arbejderinformation kan se hvilke opgaver, der er i overskud, og hvilke arbejdere, der er let belastede.

$$\langle \mathit{NyArbejderInfo}, \mathit{NyOpgaveInfo} \rangle = \mathit{LaegSkema}(\mathit{ArbejderInfo}, \mathit{OpgaveInfo}, X_{max}) \tag{3.92}$$

Når skemaet er lagt, så minimerer vi informationen ved at gruppere opgaverne i opgavegrupper, som beskrevet i afsnit 3.7.2. I sites vil dette betyde, at antallet af forskellig opgavekrav-elementer i opgaveinformationen bliver reduceret væsentligt, ved at opgaver, der *minder* om hinanden, samles. I supersites udføres grupperingen kun for, at føre information om de opgaver, som lå i supersitets buffer, ind i grupperne. Informationen, som kom fra børnene, er nemlig allerede grupperet rigtigt.

$$\mathit{NyOpgaveInfo} = \mathit{Group}(\mathit{NyOpgaveInfo}) \tag{3.93}$$

Aktivering af opgavegruppe

Hvert knudepunkt har en liste med de opgavegrupper, som er aktive til forælderknudepunktet. Efter skemalægningen og grupperingen af opgaver kan knudepunktet nu undersøge, om det har nogle opgavegrupper, hvor den er tungt belastet, og hvor gruppen ikke er aktiv op til forælderen. Eksisterer der nogle sådanne, så skal knudepunktet nu sende en besked til forælderen om at få disse aktiveret, som beskrevet i afsnit 3.7.5.

Send information til forælder

Informationen skal nu sendes til forælderknudepunktet. Som i model 2 minimeres arbejder- og opgaveinformationen, men inden dette gøres, så indordnes arbejdergrupperne efter de aktive opgavegrupper (de opgavegrupper som forælderknudepunktet ønsker information om) som beskrevet i afsnit 3.7.6.

Først fjernes informationen om de opgavegrupper, som ikke er aktive til forælderen, da man kun må informere forælderen om dets aktive opgavegrupper:

$$\mathit{OpgInfo} = \bigcup_{g \in \mathit{AktiveOpgavegrupper}^{foraelder}} \mathit{NyOpgaveInfo}(g) \tag{3.94}$$

Herefter indordnes alle arbejdergrupperne til de aktive opgavegrupper:

$$\mathit{ArbInfo} = \bigcup_{a \in NyArbejderInfo} \mathit{indordning}(a, \mathit{OpgInfo}) \tag{3.95}$$

Hvor $\mathit{indordning}(a, \mathit{OpgInfo})$ betyder, at arbejdergruppen a *indordner* sig efter opgavegrupperne i *OpgInfo*, som det er defineret i ligning (3.74) til (3.76) i afsnit 3.7.6.

Nu er arbejderne indordnet efter de aktive opgavegrupper. Denne information gemmes i knudepunktet, til brug ved bestemmelse af belastningstilstande, når der skal udveksles opgaver med forælderknudepunktet.

Herefter er informationen næsten klar til at blive sendt til forælderen. Men lige inden dette gøres, så minimeres hver enkelt opgavegruppe og hver enkelt arbejdergruppe på samme måde som i model 2.

$$MinOpgaveInfo = \bigcup_{g \in OpgInfo} \langle krav(g) \mapsto Minimize(info(g)) \rangle \quad (3.96)$$

$$MinArbejderInfo = \bigcup_{g \in ArbInfo} \langle udbud(g) \mapsto Minimize(info(g)) \rangle \quad (3.97)$$

$$SendInfo(MinArbejderInfo, MinOpgaveInfo) \quad (3.98)$$

Informationen modtages hos forælderknudepunktet på samme måde som i model 2.

Flytning af opgaver

Vi vil nu beskrive hvordan opgaverne flyttes rundt i denne model. Dette afviger en del fra model 2.

I model 2 hentes opgaver fra et barn, hvis det er tungt belastet, og opgaver fra forælderen, hvis knudepunktet og dets undertræ generelt er let belastet. Dette udvides i denne model, således at dette gøres indenfor hver enkelt aktive opgavegruppe og arbejdergruppe. Herved ændres reglerne for, hvornår der skal hentes opgaver.

Når der skal hentes opgaver fra forælderen, så gøres dette ud fra de let belastede arbejdergrupper i *ArbInfo* (som er de aktive arbejdergrupper). Reglen bliver så for en arbejdergruppe:

$$BelastningsTilstand_a^{arb} = let \Rightarrow HentOpgaveFraForaelder(a) \quad (3.99)$$

hvor a er arbejdergruppen, og $HentOpgaveFraForaelder(a)$ betyder, at der må hentes en opgave, som den indordnede arbejdergruppe a kan løse. Belastningstilstanden for en arbejdergrupper bestemmes, som i afsnit 3.7.7, ved:

$$BelastningsTilstand_a^{arb} = \text{if } NaesteFriTid(ArbInfo_a) < X_{min} \text{ then } Let \text{ else } ikkelet \quad (3.100)$$

Når opgaver skal hentes hos et barn, så er reglen ligesom i model 2, dog gælder reglen i denne model for hver enkelt opgavegruppe, som er aktiv i knudepunktet. I et supersite gælder så for den aktive opgavegruppen o og barnet i (hvor i er barn til supersitet):

$$BelastningsTilstand_{i,o} = tung \wedge N_o < N_{max} \Rightarrow HentOpgaveFraBarn_{i,o}() \quad (3.101)$$

hvor $BelastningsTilstand_{i,o}$ er belastningstilstanden hos barnet i i opgavegruppen o, som kendes ud fra den sidste information man har modtaget fra barnet.
Ligeledes betyder $HentOpgaveFraBarn_{i,o}()$, at der skal hentes en opgave tilhørende opgavegruppen o fra barnet i.

Registrering af flyttede opgaver

Ligesom i model 2 er dette punkt ekstremt vigtigt. Når et supersite henter en opgave fra et barn i en bestemt opgavegruppe, så kan dette registreres ved, at der ændres i en opgaveinformation tilhørende opgavegruppen på samme måde som i model 2 (ligning 3.52 til 3.54).

Men når vi kommer til hvordan man skal registrere, at en opgave er blevet hentet fra et knudepunkts forælder, så opstår der problemer. Opgaven hentes ud fra en arbejdergruppe (ligning (3.99)), men da vi ikke ved, om opgaven faktisk vil blive løst på en arbejder, der er indordnet til denne gruppe, kan man ikke simpelt lave en registrering i denne gruppe (som det gøres i model 2 i ligningerne (3.48)-(3.51)). Det store problem er, at vi ikke har kontrol over de lokale ressourcemanagere og derfor ikke oppe i træet kan bestemme hvilken af de mulige arbejdere/arbejdergrupper, der i den sidste ende skal have opgaven. I afsnit 3.8.1 vil vi uddybe denne problemstilling og komme med et løsningsforslag, når man kan bestemme hos hvilken arbejder, en opgave skal udføres.

Vi har derfor valgt, at vi i denne model følger nogenlunde samme princip som i model 2. Vi finder den arbejderinformation, der kan løse opgaven og har den mindste næste fri tid. Registreringen foregår så ved: (*opg* er opgaven, der skal registreres)

$$arb = fjernFoersteFrieArbejder(ArbInfo, krav(opg)) \quad (3.102)$$

$$arb.nextFreeTime = arb.nextFreeTime + \frac{CPUforbug(opg)}{CPUudbud(arb)} \quad (3.103)$$

$$ArbInfo = ArbInfo \cup \{arb\} \quad (3.104)$$

hvor *fjernFoersteFrieArbejder* fjerner en arbejder fra den arbejderinformation, som har den mindste næste fri tid, samt tilhører en arbejdergruppe, der opfylder opgavens ressourcekrav.

Lukning af aktive opgavegrupper

Et supersite holder hele tiden øje med, om der er nogle af dets aktive opgavegrupper, der er blevet overflødige. Reglerne for hvornår supersitet må lukke en aktiv opgavegruppe er givet ved ligning (3.74) i afsnit 3.7.5.

Aflevere opgaver til lokal ressourcemanager

Ligesom i model 2 er der også i denne model en process, som afleverer opgaver til den lokale ressourcemanager. Principperne er de samme, men hvor der her tages højde for opgavernes ressourcekrav og arbejdernes ressourceudbud.

3.7.10 Vurdering af model 3

Inddelingen af opgaver i grupper giver fordelen af, at store mængder af opgaver kan beskrives rimeligt simpelt. Og med indførelsen af begrebet aktive opgavegrupper, vil opgavemængder kun komme til at påvirke systemet i et område nær mængden. Med indførslen af arbejdergrupper og deres indordning til aktive opgavegrupper får vi også

her fordelen af, at mange forskellige arbejdere kan beskrives som ens. Ligeså betyder indordningen også, at det kun er arbejdere, der er brug for længere oppe i hierarkiet, der vil blive rapporteret op. Fordelen med indordning til de aktive opgavegrupper er, at vi ikke mister nogen som helst information om arbejderne, men derimod kun fjerner de informationer, som man ved ikke skal bruges.

I modelafsnittet er der nævnt et væsentligt problem ved modellen. Dette gør, at vi mister lidt af autonomiteten i undertræerne i hierarkiet. Det betyder også, at der vil være opgaver, der flyttes unødigt rundt i træet, hvilket ikke er hensigtsmæssigt som beskrevet i afsnit 3.4. Dette problem vil vi undersøge og komme med en løsning til i afsnit 3.8.1.

Den måde hvorpå vi inddeler opgaver i grupper, burde undersøges nærmere. For ressourcetyper, hvor kravet er en værdi i et kontinuert interval (f.eks. RAM, fri disk osv.), bør man undersøge hver enkelt og bestemme hvordan inddelingen skal være.

I model 1 og 2 bruges der en maksimal buffer i supersites, således at der ikke kan hentes for mange opgaver op fra dets børn og ned fra dets forælder. I model 3 bruger vi også denne bufferregel, når der hentes opgaver op fra et barn (ligning (3.101)), men ikke når vi henter opgaver ned fra en forælder (ligning (3.99)). Dette giver anledning til, at der kan komme til at ligge mange opgaver i bufferen i et supersite. Dog skal det stadig bemærkes, at der kun hentes opgaver ned, hvis supersitet mener, at der er en arbejder, der har næste fri tid $< X_{min}$. I de andre modeller vil denne regel betyder, at der kun hentes få opgaver ned, hvis der er sket en større ændring i tilstanden et sted i undertræet, således at nogle arbejdere mod forventning alligevel ikke bliver frie.

Vi er blevet nødt til at udelade bufferreglen, når der skal hentes en opgave ned fra forælderen, på grund af den måde modellen er designet. Når opgaver hentes ned, så gøres dette ved, at et knudepunkt beder om en opgave ud fra en arbejdergruppes ressourceudbud. Herved er der nogle opgavegrupper, som ikke kan udelukkes, da f.eks. et ressourceudbud på 64 MB RAM inkluderer både 64 og 32 MB RAM.

For at indføre bufferreglen igen, så burde man kunne fortælle hvilke opgavegrupper, man *ikke* ønsker at modtage opgaver fra. Dette kan gøres, da man præcist ved hvilke opgavegrupper, der er aktive hos forælderen.

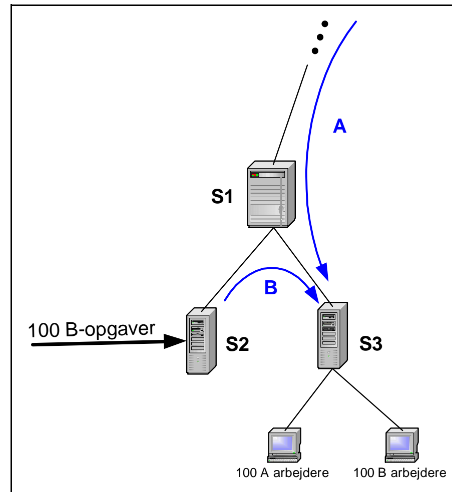
Vi mener at denne rimelig simple ændring vil kunne løse problemet.

3.8 Udvidet model 3

Vi vil i dette afsnit beskrive nogle mulige ændringer til model 3. Vi undersøger og kommer med et løsningsforslag til problemet beskrevet i modelafsnittet i model 3.
Når der i model 3 hentes opgaver fra et knudepunkt, så afleveres simpelt den tidligst ankomne opgave, som kan løses af arbejdergruppen. Vi undersøger muligheden for en bedre fordeling af opgaver ved at bruge en *smartere* skemalægningsalgoritme og herefter fordele opgaver efter skemaet.

3.8.1 Kontrolleret flytning af opgaver

Som beskrevet i modelafsnittet til model 3 har modellen et problem. I bund og grund opstår problemet, fordi vi ikke kan bestemme hvilken type arbejder, en opgave skal udføres



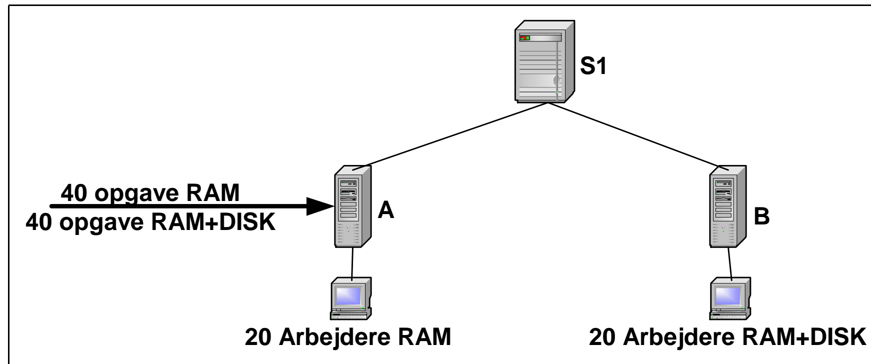
Figur 3.16: Figuren illustrerer en mulighed for, at ikke alle B-opgaver med sikkerhed vil påbegyndt i S3 også selvom dette burde ske.

hos, hos en lokal ressourcemanager. Vi vil nu give et simpelt eksempel på problemet. Eksemplet er illustreret i figur 3.16.

Et supersite har to aktive opgavegrupper O_a og O_b. Opgavegruppen O_a er også aktiv til supersitets forælder. O_b's krav overlapper O_a's krav således, at arbejdere, som kan løse opgaver fra opgavegruppen O_b, også kan løse opgaver fra gruppen O_a. I supersitets undertræ er der et site som har 100 opgaver tilhørende O_b. Supersitet kan udfra sin skemalægning se, at det kan løse problemet indenfor O_b selv ved at flytte opgaverne fra S2 til S3. Ligeledes kan det se, at det har frie arbejdere, der kan løse opgaver tilhørende O_a. Mens S1 er igang med loadbalanceringen indenfor O_b, prøver det også at hente opgaver til arbejdergruppen A_a fra dets forælder. S3 vil derfor modtage opgaver af både typen O_a og O_b. Når S3 afleverer dem til den lokale ressourcemanager, så vil denne schedulere opgaver tilhørende O_a på arbejdere af både typen A_a og A_b. Den lokale ressourcemanager ved nemlig ikke hvilke opgaver, der vil komme i fremtiden, og kan derfor ikke lægge opgaverne på en måde, som vil være optimal. Herved vil der pludselig være arbejdere, som S1 havde forestillet sig skulle løse opgaverne i O_b, som løser opgaver fra O_a. Dermed er S1 pludselig tungt belastet i O_b, da der nu ikke er arbejdere nok til at løse opgaverne. Dette betyder, at S1 faktisk har mistet lidt af sin autonomitet og at principperne om, at et knudepunkt først klarer lokale problemer i undertræet, inden det afhjælper resten af træet, ikke overholdes.

For at løse dette problem, bliver man nødt til at kunne fortælle en lokal ressourcemanager på hvilken arbejder/arbejdertype, man mener, at en opgave bør udføres. Hvis man har denne mulighed, så vil man kunne *styre* opgaverne ned gennem træet mod de arbejdere, som man mener bør løse dem.

Når et knudepunkt henter opgaver ned, så er det fordi, det er let belastet i en arbejdergruppe. Dette betyder, at man ikke selv ønsker at bruge arbejderne og derfor vil stille disse til rådighed for forælderen. Derfor bør man sørge for at følge forælderens anvisning således, at den opgave man får ned til en arbejdergruppe, også bliver udført på en arbejder, der er indordnet til denne. Dette bør ligeledes gøres, da forælderen så vil kunne bestemme præcis hvordan den vil udnytte den overskydende arbejdskraft.



Figur 3.17: Figuren illustrerer en situation hvor balanceringen muligvis ikke foregår så optimalt som muligt.

Men nok allermest vigtigt er det, at en opgave, der er blevet hentet ned til en let belastet arbejdergruppe, ikke ender med at blive udført hos en arbejder fra en arbejdergruppe, som egentlig *ikke* er let belastet. Hvis dette sker, så betyder det, at opgaven måske kan gøre, at nogle af de lokale opgavegrupper bliver mere belastede end de var før (som vist i eksemplet).

En løsning på problemet kan være, at man markerer opgaver, når de bliver hentet ned fra en forælder. Markeringen skal indikere, hvilken arbejdergruppe opgaven er blevet hentet ned med. Når en opgave er blevet hentet ned til et supersite, så skal supersitet sørge for, at opgaven kun må hentes ned til et barn af en af barnets arbejdergrupper, som er blevet indordnet til den markerede arbejdergruppe. I et site skal man sørge for at fortælle den lokale ressourcemanager, at opgaven skal udføres hos en af de arbejdere/arbejdertyper, som er blevet indordnet til den markerede arbejdergruppe.

Dette er dog ikke nok til at løse problemet. Det er ud fra skemalægningen vi bestemmer om en arbejdergruppe er let belastet. Hvis skemalægningen står med en opgave, som kan udføres af to forskellige arbejdere fra hver deres arbejdergrupper og disse begge er let belastet, så vælger den den ene og bestemmer herefter den anden som let belastet. Nu vil knudepunktet bestemme sig for at der må hentes en opgave ned til den let belastede arbejder. Hvis opgaven fra skemalægningen pludselig bliver allokeret til den anden arbejder, så har vi hentet en forkert opgave ned. Det er derfor nødvendigt at ændre modellen, således at opgaver afleveres afhængig af hvordan skemaet er lagt.

3.8.2 Forbedret fordeling af opgaver

Når der i model 3 bliver hentet en opgave i et knudepunkt, afleveres simpelt den tidligst ankomne opgave, som *matcher* arbejdergruppens ressourceudbud (*FIFO* princippet). Dette er ikke optimalt.

Vi vil vise betydningen af dette ved et simpelt eksempel. I figur 3.17 ses et simpelt system med to sites forbundet via et supersite. Supersitet S1 vil hente opgaver fra begge opgavegrupper op. Arbejdergruppen RAM+DISK i site B vil bede S1 om opgaver indtil alle arbejderne har fået en opgave. S1 vil aflevere opgaver af både typen RAM og typen RAM+DISK. Når arbejderne er færdige med deres første opgave, så vil der ikke være opgaver nok til alle RAM arbejderne, men mere end en opgave i kø hos RAM+DISK

arbejderne. S1 burde fra starten kunne beregne sig frem til, at det var smartest, at den kun afleverede RAM+DISK opgaver til B og lade A få alle RAM opgaverne.

Det er ikke lige sådan at indføre en sådan optimering i model 3. For at dette kan lade sig gøre, skal der først og fremmest bruges en anden skemalægningsalgoritme. Dernæst kræver det også at ændringsforslagene fra afsnit 3.8.1 om at opgaver afleveres i henhold til skemalægningen, bliver indført.

Kapitel 4

Implementering

I dette kapitel vil vi beskrive, hvordan vi har implementeret prototyper af de tre modeller beskrevet i kapitel 3.

Til implementeringen af modellerne, har vi valgt at bruge et *delt dataområde (shared dataspace)* efter principperne i *Linda* [24, 22]. Vi bruger det delte dataområde, som koordination- og kommunikationsmedie, og hvert site og supersite er i vores implementering bygget op om et delt dataområde. Vi designer og udvikler et *Linda*-system i afsnit 4.2.

Vi starter kapitlet med en kort gennemgang af, hvilke teknologier vi har brugt til implementeringerne (afsnit 4.1).

Efter den overordnede beskrivelse af vores Linda-system forklarer vi i afsnit 4.3, hvordan opgaver og arbejdere repræsenteres i Linda-systemet. I afsnit 4.4 forklarer vi, hvordan opgaver og information kommunikeres mellem knudepunkter i det hierarkiske system.

I afsnit 4.5 vil vi komme med en kort beskrivelse af, hvordan vi i vores forsøg simulerer lokale ressourcemanagere og arbejdere, samt beskrive hvordan vi indsamler karakteristiske systeminformationer og hvordan disse præsenteres.

Vi afslutter kapitlet med en gennemgang af, hvordan vi har implementeret vores modeller (afsnit 4.6). Vi gør dette ved at forklare den generelle opbygning, som er brugt i alle modellerne. Herefter beskriver vi kort selve implementeringen af de tre modeller.

4.1 Teknologi

I dette afsnit vil vi beskrive de teknologier, som er anvendt ved implementeringen.

Vi har valgt ikke at lave en større analyse af, hvilket programmeringssprog, der ville være det bedste at implementere vores modeller i. Vi ønsker at kunne lave nogle meget store forsøg med mulighed for mindst 100 knudepunkter og 5000-10000 simulerede arbejdere. På DTU har vi adgang til nogle store SUN computere[1] og vi har fra starten ønsket, at vores programmer skal kunne køre på disse. Vi har derfor valgt Java, der med sin

[1] DTU's G-bar, som består af 2 stk. Sun Fire 6800 hver med 24 UltraSparc III 750 MHz CPU og 24 GB RAM.

platformsuafhængighed giver os mulighed for at afvikle programmerne på de store DTU-computere, samtidigt med at vi har kunne udvikle og afprøve programmerne på vores egne PC'ere.

Al netværkskommunikation er implementeret ved hjælp af Java RMI [27]. *Ikke* på grund af RMI's performance egenskaber og stabilitet, men fordi det gjorde implementeringen mere simpel.

Vi har i implementeringen desuden brugt JavaCSP [14], en CSP [25] implementering i Java.

Til præsentation af indsamlet data (se afsnit 4.5.2) har vi brugt komponenter fra JFreeChart [21].

4.2 Linda

Vi vil i dette afsnit designe og udvikle et delt dataområde (*shared dataspace*) til brug i vores modeller. Vores delte dataområde bygger på principperne i *Linda*[24, 22].

Et *delt dataområde* er et sted, hvor processer kan gemme og hente information. En af de store fordele ved et delt dataområde er, at når to processer skal kommunikere, så kan det gøres asynkront. Dette er muligt, da en proces kan aflevere en information i området, uden at en anden proces er klar til at modtage denne.

I Linda kalder man det delte dataområde for et *tupelspace*. I dette kan processer aflevere information i form af *tupler*. Andre processor kan herefter hente informationen i tuplerne ved hjælp af *antitupler*.

En tupel er en ordnet liste af elementer, hvor hvert element består af en type og en værdi. En antitupel er bygget op på samme måde som en tupel.

I et Linda tupelspace er der nogle få operationer, som bruges til at aflevere og hente data. Disse er:

out(*tupel*) Placerer tuplen i tupelspacet.

in(*antitupel*) Fjerner og returnerer én tupel fra tupelspacet, som *matcher* antituplen. Hvis der ikke eksisterer en sådan, så ventes der på, at der ankommer en.

rd(*antitupel*) Samme som `in`-operationen, men denne fjerner ikke tuplen fra tupelspacet. Den returnerer kun en kopi.

eval(*tupel*) En variant af `out`, hvor et eller flere af tupelelementerne ikke er værdier, men istedet er funktioner. Hver af disse funktioner udføres af tupelspacet og deres resultat lagres i tupelelementet som en værdi istedet for funktionen. Når alle funktionerne er blevet udført, bliver tuplen placeret i tupelspacet som ved en normal out-operation.

En tupel og en antitupel siges at *matche*, når følgende betingelser er opfyldt:

1. Der er samme antal elementer i tupel og antitupel.

2. Typerne på de enkelte elementer i tupel og antitupel er de samme.

3. Hvert element i tuplen har samme værdi, som det tilsvarende element antituplen, eller en af værdierne er et såkaldt wildcard felt. Wildcards matcher en hvilken som helst værdi af en given type.

Følgende er et eksempel på brugen af et Linda-tupelspace.

```
out("MyID", 5, 9)
out(32, "Peter", 29)
in("OtherID",?i)
in("MyID",?i, 9)
```

De første to operationer afleverer hver en tupel til tupelspacet. Den tredje operation returnerer ikke, da der ikke er nogen tupel, der matcher denne. Den fjerde operation vil matche tuplen, som blev indsat i første operation. Tuplen fjernes fra tupelspacet og operationen returnerer. Wildcard-variablen i vil herefter indeholde værdien 5.

4.2.1 Linda varianter

Der findes mange forskellige varianter af Linda systemet. Systemerne adskiller sig både ved udbud af nye services og ved konkret implementering, men de har alle den overordnede Linda-idé tilfælles. Af afvigelser fra den oprindelige Lindamodel kan nævnes:

- Størstedelen af alle implementeringerne supporterer ikke `eval` funktionen (eksempelvis TuCSoN [16] og JavaSpaces [26]).

- Tilføjelse af en ny funktion `readAll(antitupel)`, som returnerer alle tupler, der matcher antituplen. I den oprindelige model, er det ikke muligt at læse alle tupler, som matcher til en given antitupel, præcis en gang. Funktionen løser dette problem(MARS [12]).

- Objektorienterede lindaimplementeringer bruger objekter i stedet for tupler og antitupler. Nedarvning tillades således, at en tupel gerne må være en underklasse af antituplen ved en match (JavaSpaces [26]).

- Mulighed for timeout ved `in` og `rd` operationer, så der maksimalt ventes et bestemt stykke tid på en matchene tupel (bl.a. JavaSpaces [26]).

- Mulighed for events og reaktioner (MARS [12]).

- mange flere.

Se [22] for en mere uddybende gennemgang af nogle af de forskellige systemer.

4.2.2 Ressourcer, opgaver og Linda

Man kan rimelig let lave et simpelt ressourceallokeringssystem i et Linda-system. Opgaver kan placeres i et tupelspace som opgavetupler og arbejdere kan hente opgaverne med opgaveantitupler. Ligeledes kan de udførte opgaver aflevere deres resultat i tupelspacet, som så kan afhentes af opgavestilleren.

```
Opgavestiller:   out("opgave", opgaveID, opgaven)
                 in("opgavesvar", opgaveID, ?svar)

Arbejder:        in("opgave", ?id, ?opg)
                 svar = ...opg...
                 out("opgavesvar", id, svar)
```

En sådan simpel løsning har væsentlige mangler i forhold til den problemstilling vi fokuserer på. Opgavetuplen og arbejderantituplen indeholder ikke nogen form for information om mulige krav og udbud af ressourcer. En løsning kunne være at tupler og antitupler udvides til at indeholde informationer om ønsker/udbud af ressourcer. Eksempelvis vil tuplen

```
("task", "CPU", "RAM", opgave)
```

give en match med antituplen

```
("task", "CPU", "RAM", ?opgave)
```

men ikke med antituplen

```
("task", "CPU", ?opgave)
```

hvilket sikrer, at en arbejder, der kun kan tilbyde CPU, ikke får tildelt opgaver, som også kræver RAM.

I den originale Linda version er tupler ordnede, dvs. tuplen

```
("RAM", "CPU")
```

er forskellig fra tuplen

```
("CPU", "RAM")
```

Der skal desuden være de samme antal elementer i en tupel og en antitupel for, at der kan opstå en match, hvilket betyder, at et en maskine, som stiller CPU og RAM til rådighed, ikke vil kunne matches med en opgave, der kun skal bruge CPU. Da vi som beskrevet i afsnit 3.7.1 har brug for, at en opgave kan matches med en arbejder, som udbyder flere ressourcer end opgaven kræver, fjerner vi disse krav. Når ordningen af tupler og antitupler fjernes, så bliver man nødt til på en anden måde at kunne identificere de enkelte felter.

Dette gør vi ved at indføre, at et felt har et *navn*, som er unikt i tuplen/antituplen. Herved beskrives en tupel/antitupel nu som en mængde af felter, hvor hvert felt har et navn, en type og en værdi. En tupel kan f.eks. se ud som:

$$Tupel = \{\langle RAM, num(32)\rangle, \langle CPU, num(5.2)\rangle\} \tag{4.1}$$

Og en antitupel:

$$Antitupel = \{\langle RAM, num(32)\rangle, \langle CPU, num(?i)\rangle\} \tag{4.2}$$

Som beskrevet i afsnit 3.7.1 vil der til beskrivelsen af et ressourcekrav/ressourceudbud typisk høre en numerisk værdi, som fortæller noget om mængden af den ønskede/tilgængelige ressource (f.eks. RAM). I forhold til standard Linda (og de fleste kendte Linda-lignende implementeringer) giver dette dog et problem, da man her kun har mulighed for at lave en eksakt match eller en wildcard match på de enkelte felter i tuplen. I et sådan system er det f.eks. ikke muligt for en opgave at udtrykke:

`jeg ønsker` ***minimum*** `32 MB RAM`

men derimod kun

`jeg ønsker` ***præcis*** `32 MB RAM`

eller

`jeg ønsker` *RAM*

For at løse dette problem, har vi valgt at indføre datatypen *numerisk interval*. Med indførslen af denne og en ændring i matchreglerne vil det være muligt at matche en tupel og en antitupel, som repræsenterer henholdsvis en opgave og en arbejder, hvor opgaven siger, at den skal bruge 32 MB RAM og arbejderen vil løse opgaver, som maksimalt skal bruge 100 MB RAM.

$$Tupel_{opgave} = \{\langle RAM, num(32)\rangle, \langle Opgave, opg(...)\rangle\} \tag{4.3}$$

$$Antitupel_{arbejder} = \{\langle RAM, numInterval(0, 100)\rangle, \langle Opgave, opg(?i)\rangle\} \tag{4.4}$$

Med den nye type, ændrer vi på den grundlæggende regel om, at det kun er to elementer af samme type, som kan matche. Vi indfører nu, at der er en match mellem en numerisk værdi x og et numerisk interval I, hvis den numeriske værdi x ligger i intervallet I ($x \in [I_{min}, I_{max}]$).

4.2.3 Krav til tupelspace

Problemstillingerne beskrevet i sidste afsnit har medført, at vi har valgt at implementere vores egen version af Linda, da ingen anden implementering gav os de muligheder, vi har brug for. Til vores implementering har vi stillet følgende krav:

1. Ordningen af en tupel og en antitupel skal ikke have betydning for, om der kan opstå en match.
2. Antitupler må ved en match gerne indeholde flere elementer end der er i tuplerne, men skal naturligvis som minimum opfylde kravet i tuplen.
3. For ethvert tupelelement i antituplen skal det være muligt at specificere, om elementet *skal* have et matchene element i tuplen eller om det er et match-frit tupelelement (til brug i 2).
4. Indførelse af datatypen: numerisk interval.
5. Match-reglerne mellem tupler og antitupler udvides fra standard Linda (eksakt- og wildcardmatch) til også at tillade en match mellem en numerisk værdi og et numerisk interval.
6. De almindelige Linda operationer skal understøttes, dog ikke `eval`.
7. Det skal være muligt at angive et timeout for operationerne `in` og `rd`, således at en antitupels levetid kan begrænses.

I realiteten er der ikke nogen forskel på en tupel og en antitupel. I implementeringen har vi derfor valgt at lade tupel og antitupel være ens.

Punkt 2 og 3 har vi løst ved, at lade en tupel/antitupel bestå af nogle *påkrævede* tupelelementer og nogle *udbuds* tupelelementer. De påkrævede tupelelementer er dem, som *skal* opfyldes af modparten ved en match. Udbuds elementer stilles blot til rådighed for den tupel/antitupel der matches med.

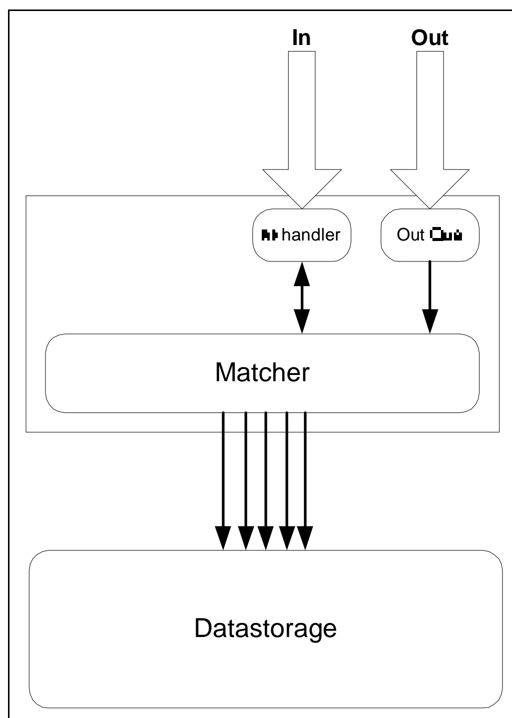
4.2.4 Linda implementering

På figur 4.1 ses den logiske opbygning af vores Linda implementering. Systemet er opdelt i to hovedkomponenter, en *matcher* og et *datastorage*. Matcheren har til opgave at modtage tupler og antitupler via *out*-, *rd*- og *in*-operationer. Det er også matcheren, der står for at låse *in*- og *rd*-operationer indtil der er fundet en match, eller indtil der forekommer et timeout for den pågældende antitupel.

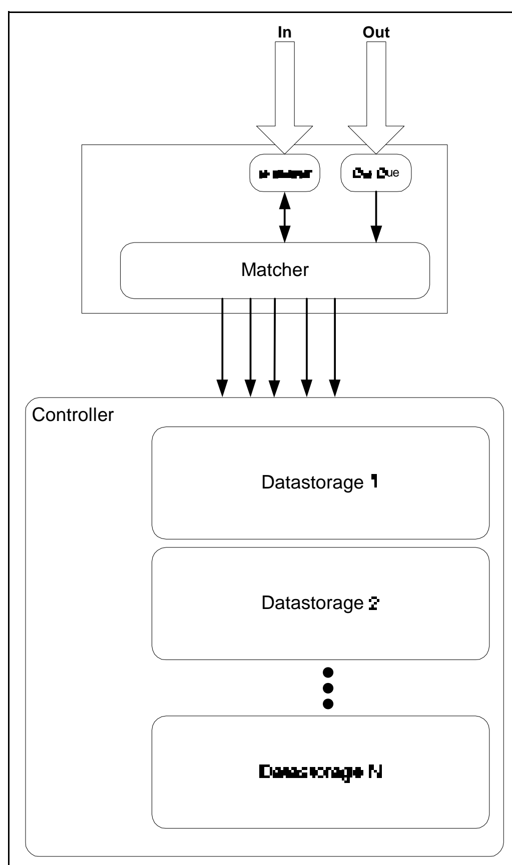
Datastorage har til opgave at opbevare alle de tupler og antitupler, der er i systemet. Matcheren vil ved *in*-, *rd*- og *out*-operationer forespørge datastoraget, om der er en tupel/antitupel, der matcher en given tupel/antitupel. Det er således datastoraget, der bestemmer *hvilken* tupel/antitupel der matches, mens det er matcheren der bestemmer, hvornår muligheden for en match skal undersøges.

Grunden til denne opdeling er, at vi på den måde simpelt kan udskifte det enkelte datastorage og bruge forskellige versioner, som har forskellige politikker for hvilke tupler/antitupler der matches, når der er flere valgmuligheder.

I vores modelimplementeringer vil vi typisk arbejde med mere end et datastorage samtidigt. Hvert datastorage styrer en bestemt type af tupler/antitupler, mens der udenom disse er en *controller*, der styrer hvilket datastorage, der på et givet tidspunkt skal kommunikeres med. Dette er illustreret på figur 4.2. Et simpelt eksempel med to datastorages



Figur 4.1: Figuren viser opbygningen af vores linda implementering.



Figur 4.2: Figuren viser opbygningen af vores linda implementering, når der bruges flere samtidige datastorages.

kan være, at det ene bruges til at håndtere alle arbejderantitupler og opgavetupler, mens det andet datastorage tager sig af alle andre tupler og antitupler. Det bemærkes, at den på figuren viste *controller* supporterer det samme interface mod matcheren, som et enkelt datastorage har. Så på den måde er controlleren i sig selv også et datastorage.

Vi har valgt at understøtte seks forskellige datatyper i vores implementering. De tre første er helt almindelige typer: *Tekst*, *numerisk* og *boolsk*. Ligeledes er der vores specielle intervaltype: *numerisk-interval*. Da tupelspacet er blevet implementeret i Java, har vi ligeledes valgt at understøtte typen *objekt*, som kan holde et Java-objekt. Den sidste type er lidt speciel og den hedder *agent*. Denne type bruges til at holde information om en opgave.
Typerne hedder i vores implementering: RealVal (numerisk), RealRange (numerisk-interval), StringVal (tekst), ObjectVal (objekt), BoolVal (boolsk) og AgentVal (agent). Vi har ligeledes indført navne for wildcard-værdierne. Disse er: RealWild, StringWild, ObjectWild, BoolWild og AgentWild.

I bilag A ses en komplet formalisering af vores Linda version i *RAISE Specifikation Language*[13]. I specifikationen kan reglerne for hvornår en tupel og en antitupel matcher ses. Med disse regler, er det muligt at udtrykke *krav og udbud*, som beskrevet i afsnit 3.7.1

4.3 Opgaver og arbejdere

I modelimplementeringerne har vi valgt at repræsentere opgaver med tupler og arbejdere med antitupler.

En opgavetupel er en tupel, som har et påkrævet tupelelement med navnet *AGENT* af typen *AgentVal*. Da elementet er påkrævet, betyder det, at det kun kan matches med antitupler, der også har et *AGENT* tupelelement. AgentVal kan indeholde et Java objekt, som bruges ved initialisering af den opgave, som skal udføres. Oplysningerne i objektet bruges ikke i loadbalanceringssystemet, men informationerne gives videre til den arbejder som løser opgaven. Objektet vil eksempelvis kunne indeholde informationer om, hvor selve opgaven og dens data skal hentes fra samt andre parametre for opgaven. En opgavetupel har også et udbudstupelelement med navnet *CPUTIME*, som angiver det estimerede antal CPUenheder, opgaven skal bruge for at blive løst. Sidst men ikke mindst kan en opgave have nogle påkrævede tupelelementer, som beskriver opgavens ressourcekrav.

En opgave, der eksempelvis forventer at skulle bruge 2000 CPUenheder og skal bruge en arbejder, der kører Linux og mindst har 64 MB RAM (ressourcekrav), vil blive repræsenteret ved følgende tupel:

```
ReqElems[
  <AGENT,AgentVal(JobInfo)>,
  <RAM,RealVal(64)>,
  <OS,StringVal('Linux')>
],
OptElems[
  <CPUTIME,RealVal(2000)>
]
```

En arbejderantitupel er en antitupel, som har det påkrævede tupelelement *AGENT*. Men modsat opgavetuplen så er denne af typen *AgentWild*. Dette betyder, at en arbejderantitupel kun vil kunne matche med en tupel, som indeholder en opgave. Arbejderens ressourceudbud repræsenteres ved udbudstupelelementer i antituplen. En arbejder der eksempelvis kan levere 128 MB RAM må matches med alle opgaver, som enten ikke kræver RAM eller som kræver mellem 0 og 128 MB RAM. Et sådan udbud vil derfor blive repræsenteret ved et tupelelement af typen *RealRange* med værdierne 0 og 128.

Et eksempel på en arbejderantitupel, som vil kunne matches med den ovenfor viste opgavetupel er:

```
ReqElems[
  <AGENT,AgentWild()>
],
OptElems[
  <RAM,RealRange(0, 128)>,
  <DISK,RealRange(0, 2000)>,
  <OS,StringVal('Linux')>
  <CPU,RealRange(0, 5.3)>
]
```

4.4 Kommunikation

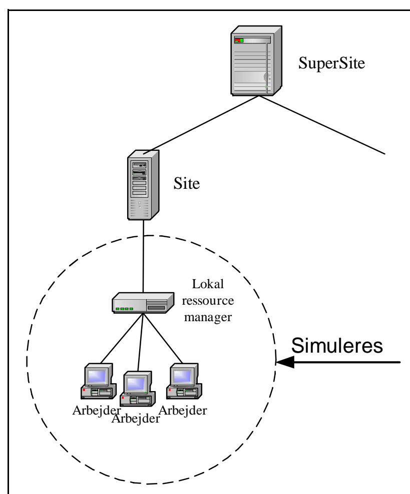
Som nævnt i indledningen af dette kapitel, så har hvert site og supersite sit eget tupelspace. Opgavetupler flyttes rundt i systemet mellem sites/supersites, ved at modtageren foretager en *in*-operation i det tupelspace opgaven skal hentes fra. Når eksempelvis et site ønsker en opgave til en af sine arbejdere, foretager det en *in*-operation hos sin forælder. *In*-operationen indeholder en antitupel, som angiver de ressourcer sitets arbejder stiller til rådighed. På den måde sikres (via Linda matchreglerne), at opgaven som hentes, vil kunne løses af arbejderen.

Almindelig information (såsom belastningstilstandsinformation) sendes ved at lave en *out*-operation med en informationstupel[2] i modtagerens tupelspace. Modtageren af informationen vil typisk skulle modtage information fra flere afsendere (et supersite modtager information fra alle sine børn). I de implementeringer vi beskriver senere, vil der typisk være en proces i hvert supersite, som har til opgave at samle informationen fra børnene. Dette gøres ved at lave en lokal *in*-operation i supersitets eget tupelspace. Herved kan der ventes på information fra alle børnene samtidigt.

4.5 Testmiljø

For at kunne afprøve de modeller vi har udviklet, er vi blevet nødt til at simulere dele af systemet. Da vi ikke har adgang til flere tusinde computere, har vi valgt at simulere de

[2]En informationstupel er en helt almindelig tupel, som har et *INFO* tupelelement af typen *ObjektVal*. ObjektVal kan på samme måde som AgentVal indeholde et Java objekt. I en informationsstupel, indeholder Java objektet selve information.



Figur 4.3: Viser hvilke dele af systemet som simuleres.

enkelte arbejdere og udførslen af opgaver hos arbejderne. Rent teknisk har vi gjort det, at vi har implementeret en *lokal ressourcemanager*, som udover at kunne schedulere opgaver til arbejdere, også simulere udførslen af opgaverne. På 4.3 er vist de dele af systemet som bliver simuleret.

I vores lokale ressourcemanager kan man registrere et antal arbejdere med forskellige ressource udbud. Der kan både fjernes og adderes nye arbejdere med tiden. Et site kan aflevere opgaver til ressourcemanageren, som så schedulerer dem til arbejderne. Ligeledes kan man spørge ressourcemanageren om dens tilstand (information om arbejderne). Vi har pt. implementeret to forskellige schedulerings/allokerings rutiner i den lokale manager. Den ene (*FirstStartTime*) allokerer opgaverne i den rækkefølge, den modtager dem, hos den arbejder der forventes først at kunne påbegynde opgaven. Den anden (*FirstEndTime*) allokerer opgaverne i samme rækkefølge, men hos den arbejder der forventes at kunne løse opgaven hurtigst.

Den lokale ressourcemanager holder en opgavekø for hver arbejder i sitet. Når den allokerer opgaver, placeres disse bagerst i en arbejders opgavekø.

Vores lokale ressourcemanager kan desuden sættes op til at lave afvigelser på opgavernes udførelsestid, således at den faktiske udførelsestid for opgaverne afviger fra den estimerede.

4.5.1 Simulator

Alle vores forsøg er udført på DTU's G-bar (erlang og bohr serverne). For ikke at skulle starte en java proces for hvert enkelt site, supersite og lokale ressourcemanager, har vi implementeret en *TotalSimulator*. TotalSimulatoren er et javaprogram, der kan starte og køre flere sites, supersites og lokale ressourcemanagere samtidigt. På den måde kan vi have nogle hundrede af disse kørende i 10-20 javaprocesser. To sites/supersites, som er direkte forbundet og dermed skal kommunikere med hinanden, er i vores forsøg placeret i hver deres TotalSimulator på hver sin fysiske server. På den måde har vi stadig elementet netværkskommunikation med i forsøgene.

Vi kommunikerer med TotalSimulatorene via et dedikeret tupelspace (CommandTS). TotalSimulatoren henter kommandoer ved hjælp af *in*-operationer i CommandTS. Vi (brugeren) placerer kommandoer i CommandTS i form af tupler (hertil bruges et lille hjælpe program, CommandScripter, som parser en tekstbaseret kommando, og omformer den til *out*-operation i CommandTS). En kommando vil eksempelvis kunne fortælle en TotalSimulator, at den skal starte et site, *s7*, hvis forælder skal være *s3*, eller at den lokal ressourcemanager i *s7* skal have tilført 30 arbejdere af type 5[3].

4.5.2 Dataindsamling og -behandling

Et vigtigt element i forsøgene er at få samlet data ind, så resultatet af forsøgene kan analyseres. I vores implementeringer er der et stort antal parametre, som kan være interessante at overvåge, såsom belastningstilstande i de forskellige sites/supersites eller det antal opgaver et site har hentet ned fra sin forælder.

Alle entiteter i vores system, der skal overvåges, implementerer et *remote logningsinterface*. Via interfacet er det muligt at hente alle de parametre, som skal overvåges.

Et site/supersite har en debug proces, som kan sættes til at hente alle logningsparametre med et fast tidsinterval. Processen gemmer alle de hentede parametre sammen med samplingstidspunktet.

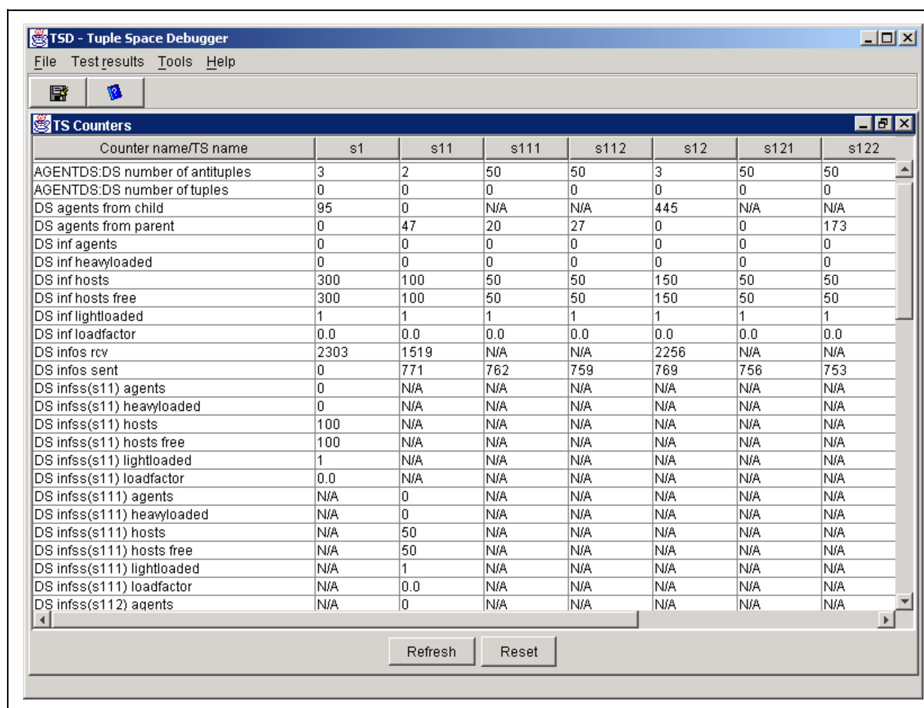
Til at præsentere de indsamlede data, har vi implementeret et lille grafisk program (TSD). Programmet kan bl.a. vise en tabel over alle parametre, og deres værdier i de forskellige sites/supersites (figur 4.4). Tabellen kan sættes til automatisk at opdatere sig selv hvert sekund. Denne funktionalitet kan bruges til at overvåge simuleringen, mens den er igang, men kun meget grov information kan (med det trænede øje) udledes af tabellen.

For mere forfinede og detaljerede oplysninger, kan der via TSD startes en remote logning, dvs. den før omtalte debug proces i alle sites og supersites sættes til at logge alle data med en højere samplingsfrekvens (f.eks. 100 ms.). Når et forsøg er afsluttet, kan logningen stoppes igen, og alle data fra de remote debug processer i sites og supersites kan loades fra TSD. TSD indeholder et værktøj, der ud fra de indsamlede data kan generere nogle grafer (figur 4.5). Værktøjet kan vise et vindue med op til tre forskellige grafer, og i hver graf kan der være flere forskellige kurver. Til hver graf, vælges hvilke parametre der skal vises og fra hvilke sites/supersites, de skal vises. X-aksen på graferne er altid tiden, mens y-akserne varierer afhængigt af hvilke parametre, der vises. Det er ligeledes muligt at *zoome* ind, og kun se graferne for et mindre tidsrum. På figur 4.6 ses et eksempel på en graf.

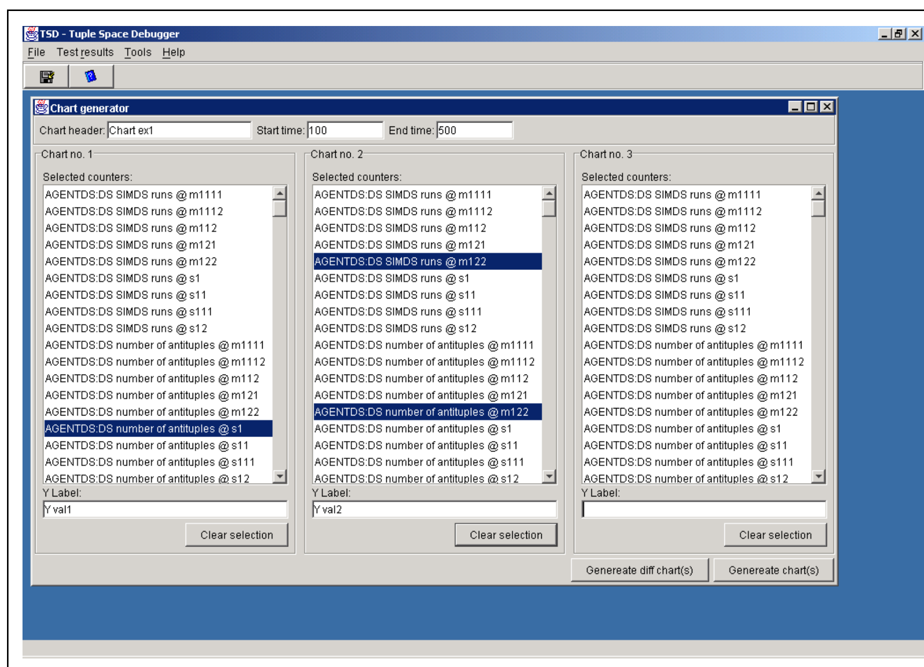
4.6 Implementering af modeller

I dette afsnit vil vi komme med en kort beskrivelse af vores modelimplementeringer. Vi starter med et afsnit, som beskriver den generelle opbygning af modellerne. Herefter kommer et lille afsnit om hver enkelt model, hvor vi fremhæver nogle vigtige ting om den aktuelle model. En udførlig gennemgang af de enkelte modeller findes i bilag B.

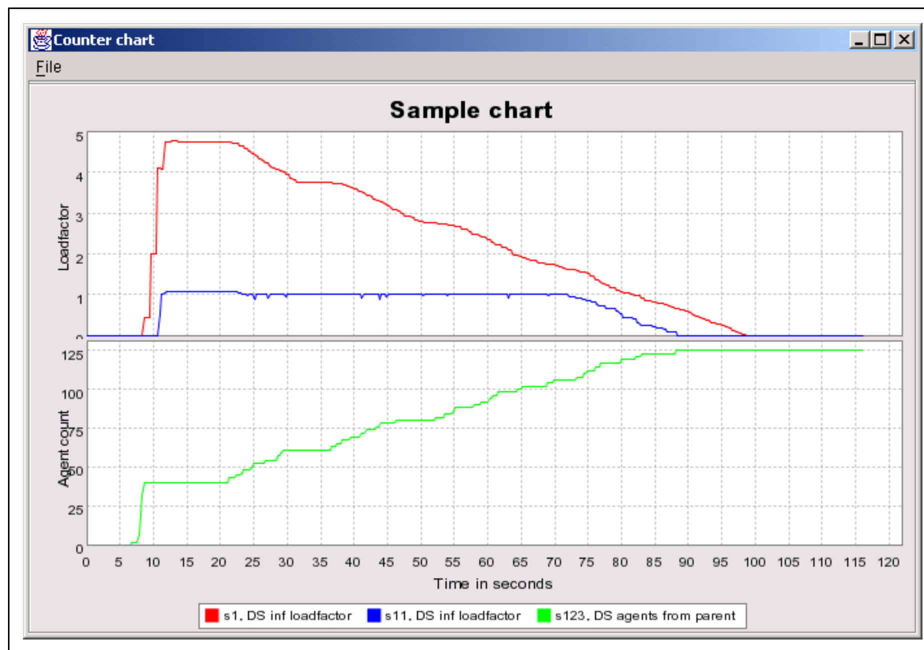
[3]For bedre at kunne følge med i hvad der sker i systemet, bruger vi i vores forsøg arbejdere, hvor ressource udbudet er kendt på forhånd. Vi har et fast antal forskellige typer af arbejdere, som kan startes i de enkelte sites. Dette beskrives mere udførligt i kapitel 5.



Figur 4.4: TDS med tabel over de parametre som kan overvåges ved en simulering.



Figur 4.5: TDS med det værktøj der bruges til at generere grafer. Alle parametre som kan plottes, findes i listerne som *parameter navn@site/supersite navn*.



Figur 4.6: Eksempel på en graf. Til den pågældende simulering er der samplet med et interval på 250 ms. På den øverste graf er plottet loadfaktorene for to forskellige sites. Den nederste viser antallet af opgaver et tredje site har modtaget fra sin forælder. Graferne er plottet ved hjælp af komponenter fra JFreeChart [21].

4.6.1 Generel model

I dette afsnit vil vi vise grundstenene i implementeringen af vores modeller. Modellernes generelle opbygning er meget lig hinanden og afviger mest af alt ved måden, hvorpå information genereres og behandles.

I modellerne bruges et Linda tupelspace i hvert knudepunkt (sites og supersites).

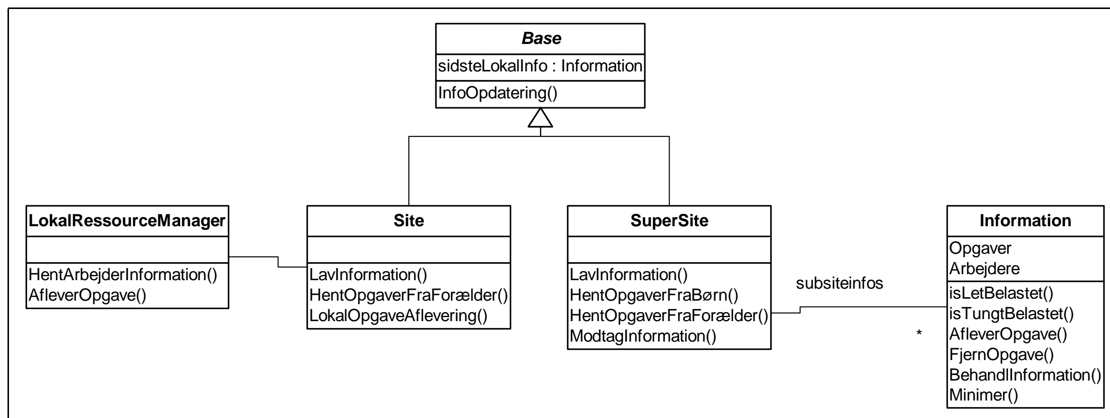
Modellerne opdeles i to dele: en for supersites og en for sites. Som det ses i teorien er der dog mange ting, som er fælles for både sites og supersites og disse samler vi i en del kaldet *Base*. Ud fra teorien kan man se, hvilke dele sites og supersites bør bestå af, samt hvilke af disse dele, der er fælles.

Det, som adskiller vores tre modeller mest, er informationen om arbejdere og opgaver. Vi definerer i dette generelle afsnit kun nogle egenskaber for denne information og uddyber disse egenskaber i hver enkelt model.

Vi vil herunder komme med beskrivelser af de funktionaliteter, som er fælles for modellerne, og forklare hvorledes de virker. På figur 4.7 vises et simplificeret klassediagram for, hvordan modellerne generelt ser ud.

Modtage information fra børn

I et supersite er der en proces, som modtager information fra supersitets børn. Børnene afleverer informationen ved at lægge dem i supersitets tupelspace (*out*-operation). Supersitet henter så informationen ved en simpel *in*-operation. Når informationen er blevet hentet, så gemmes denne (overskriver eventuelt en gammel information fra barnet).



Figur 4.7: Simplificeret klassediagram for den generelle modelimplementering.

```
SuperSite::ModtagInformation() =
  while true
    AntiTupel infoantitupel = .......
    Tupel infotupel = myTS.in(infoantitupel);
    senderid = getsenderid(infotupel)
    subsiteinfos(senderid) = getinfo(infotupel)
```

Indsamling af information

I alle knudepunkter skal der indsamles information, som skal bruges både til selv at tage beslutninger, samt til at informere forælderknudepunktet. Der er rimelig forskel på, hvordan informationen indsamles i et site og et supersite. Informationen i et site består af information fra den lokale ressourcemanager, samt information om opgaver i sitets lokale kø. I et supersite genereres informationen ud fra information, som det har modtaget fra sine børn, samt eventuelle opgaver i dets lokale buffer (datastorage).

Indsamling af information i et supersite:
Et supersite skal desuden tage højde for, at tiden går (undtagen model 1). Når det skal lægge informationen sammen fra sine børn, er de forskellig informationer genereret på forskellige tidspunkter. Dette tages der højde for ved at ændre arbejdernes næste fri tider, således at de refererer til samme tidspunkt (tidspunktet *nu*).

```
SuperSite::LavInformation() =
  info.Opgaver = LokalOpgaveKø.hentOpgaveInfo()
  for each subinfo in subsiteinfos
    Opgaver = Opgaver + subinfo.Opgaver
    tmpArbejdere = timeChange(subinfo.Arbejdere)
    info.Arbejdere = info.Arbejdere + tmpArbejdere
  return info
```

Indsamling af information i et site:

```
Site::LavInformation() =
  info.Opgaver = LokalOpgaveKø.hentOpgaveInfo()
  info.Arbejdere = LokalRessourceManager.HentArbejderInformation()
  return info
```

Opdatering af information

I alle knudepunkterne på nær toppunktet, er der en proces, som sørger for at indsamle information, behandle informationen, gemme den lokalt og sende den til forælderknudepunktet. Den information, der gemmes lokalt, bruges til at bestemme belastningstilstanden for knudepunktet, når der er brug for det.

```
Base::InfoOpdatering() =
  while true
    info = LavInformation()
    info.behandlinfo()
    sidsteLokalInfo = info
    info.minimer()
    Tupel infotupel = ....(info)
    parentTS.out(infotupel)

    sleep OpdateringsPause
```

Hente opgaver fra forælderknudepunkt:

I alle knudepunkter, både sites og supersites, skal der være en proces, som henter opgaver fra forælderknudepunktet, når knudepunktet mener, at det er underbelastet (gælder dog ikke topknudepunktet, da det ingen forælder har). Processen er lidt forskellig for sites og supersites, hvorfor vi har valgt at beskrive dem hver for sig. Opgaverne hentes fra forælderen enkeltvis ved at lave en *in*-operation i forælderens tupelspace.

Supersite: I supersitene i modellerne er der indført en maksimal bufferstørrelse således, at der ikke kan hentes opgaver, hvis der allerede er et vist antal opgaver i knudepunktets lokale kø. Når en opgave er blevet hentet, så afleveres den til det lokale tupelspace (*out*-operation) og knudepunktets information opdateres ved at tilføje information om opgaven til den sidste generede information. Processen bliver således:

```
SuperSite::HentOpgaverFraForælder() =
  while true
    if LokalOpgaveKø.antalOpgaver<MAXBUFFERSIZE
         & sidsteLokalInfo.isLetBelastet() then
      AntiTupel arbejderantitupel = .......
      Tupel opgavetupel = parentTS.in(arbejderantitupel)
      if opgavetupel!=null then
        Opgave opg=opgave(opgavetupel)
        sidsteLokalInfo.AfleverOpgave(opg)
        myTS.out(opgavetupel);
```

Site: I et site er processen lidt anderledes. Der bruges ikke en maksimal bufferstørrelse, da vi ikke henter opgaver ned for at lægge dem i vores lokale buffer. Når en opgave er blevet hentet, så afleveres den direkte til den lokale ressourcemanager og knudepunktets information opdateres ligesom i et supersite. Processen bliver således:

```
Site::HentOpgaverFraForælder() =
  while true
    if sidsteLokalInfo.isLetBelastet() then
      AntiTupel arbejderantitupel = .......
      Tupel opgavetupel = parentTS.in(arbejderantitupel)
      if opgavetupel!=null then
        Opgave opg=opgave(opgavetupel)
        sidsteLokalInfo.AfleverOpgave(opg)
        lokalressourcemanager.AfleverOpgave(opg)
```

Hente opgaver fra børn:

I supersites skal der være en proces, der sørger for at afhjælpe supersitets børn, når disse er tungt belastede. Når der er fundet et barn, der er tungt belastet, så hentes der en opgave fra barnet. Ligesom når der hentes opgaver fra et forælderknudepunkt, så bruger vi igen reglen om et maksimalt antal opgaver i supersitets lokale kø. Opgaverne hentes enkeltvis ved at lave en *in*-operation i barnets tupelspace. Når en opgave er blevet hentet, så registreres det i informationen om barnet.

```
Supersite::HentOpgaverFraBørn() =
  while true
    if AntalLokaleOpgaver<MAXBUFFERSIZE then
      barnid=FindTungtBelastetBarn()
      if barnid!=null then
        TupelSpace barnTS=tupelspace(barnid)
        AntiTupel arbejderantitupel = .......
        Tupel opgavetupel = barnTS.in(arbejderantitupel)
        if opgavetupel!=null then
          myTS.out(opgavetupel)
          Opgave opg=opgave(opgavetupel)
          subsiteinfos(barnid).FjernOpgave(opg)
```

Hvor *FindTungtBelastetBarn* undersøger om der i *subsiteinfos* (informationerne modtaget fra børnene) er et barn, som er tungt belastet. I implementeringerne har vi sørget for, at funktionen er *fair* således, at hvis der er flere børn, der er tungt belastede, så returnerer funktionen skiftevis disse.

Aflever opgaver til lokal ressourcemanager:

I sites er der en proces, som afleverer opgaver til sitets lokale ressourcemanager[4]. Processen holder øje med, hvornår der er arbejdere hos ressourcemanageren, som har mindre end X_{max} sekunder til de er frie. Når dette sker, så undersøger processen, om der er en opgave i den lokale kø, som kan afleveres til arbejdere og gør dette, hvis en sådan eksisterer.

[4]Model 1 bruger dog ikke en lokal ressourcemanager og arbejderne bliver derfor simuleret på anden vis (vil blive beskrevet i modellen).

```
Site::LokalOpgaveAflevering() =
  while true
    Arbejdere = LokalRessourceManager.HentArbejderInformation()
    Arbejdere = subset(Arbejdere, Xmax)
    for each arb in Arbejdere
      AntiTupel arbejderantitupel = arb.ArbejderAntitupel
      Tupel opgavetupel = myTS.in(arbejderantitupel)
      if opgavetupel!=null then
        Opgave opg=opgave(opgavetupel)
        LokalRessourceManager.afleverOpgave(opg)
```

4.6.2 Model 1 - simpel load balancering

Implementeringen af model 1 er rimelig simpel. Kildekoden ligger i java-pakken *framework.tuplespaces.ts3*. I appendix B.1 har vi lavet en detaljeret gennemgang af, hvordan modellen er implementeret.

Da modellen ikke tager højde for, at tiden går, så er det meget vigtigt, at der sendes information med en høj frekvens. Vi har derfor valgt at denne sendes én gang i sekundet.

Vi har lavet en lille ændring i forhold til teorien. I teorien trækker vi opgaver ned således, at der altid ligger et vist antal opgaver i kø for hver arbejder. Man bliver nødt til at tage højde for, at nogle opgaver bliver trukket ned for at blive påbegyndt med det samme og således *ikke* kommer til at ligge i kø. Vi har derfor indført, at informationen også skal indeholde antallet af frie arbejdere.

$$Information: \begin{pmatrix} Antal\ opgaver\ i\ kø \\ Antal\ arbejdere \\ Antal\ frie\ arbejdere \end{pmatrix} \tag{4.5}$$

Når der hentes en opgave ned fra et supersite, så undersøges det, om opgaven vil blive lagt i en kø, eller om den kan påbegyndes med det samme. Herved defineres AfleverOpgave-funktionen som:

```
Information::AfleverOpgave(opgave) =
  if AntalFrieArbejdere > 0 then
    AntalFrieArbejdere = AntalFrieArbejdere - 1
  else
    AntalOpgaver = AntalOpgaver + 1
```

4.6.3 Model 2 - udvidet load balancering

Denne model tager højde for størrelsen af CPU udbuddet fra arbejdere og CPU forbruget for opgaver. Hermed bliver strukturen af informationen en del mere kompliceret end i model 1. Ligeså er behandlingen af informationen også mere kompliceret.

Da model 2 tager højde for, at tiden går, så har vi valgt, at der kun genereres og sendes information hvert 5. sekund.

I Bilag B.2 har vi lavet en beskrivelse af implementeringen, hvor vi viser alle definitionerne på funktionerne fra den generelle model. Implementeringen ligger i java-pakken *framework.tuplespaces.ts4*.

4.6.4 Model 3 - kompleks load balancering

Dette er den mest komplekse af vores modeller, hvorfor det også har været den mest komplicerede at implementere.

I hvert knudepunkt er der, for hver aktive opgavegruppe og arbejdergruppe en proces, hvis opgave er, at hente opgaver fra forælderen henholdsvis børnene. Hver gang der genereres ny information i knudepunktet, så undersøges det, om der skal oprettes nye processer og om der er nogle, der skal nedlægges. Dette gøres også, når knudepunktet modtager information fra dets forælder om, at det skal åbne eller lukke en aktiv opgavegruppe.

I et supersite's tupelspace inddeler vi opgaverne i opgavegrupper (hvor opgaver *minder* om hinanden) og for hvert af disse, har vi et dedikeret datastorage. Dette datastorage bruges blandt andet til at holde øje med, hvor mange opgaver tilhørende opgavegruppen, der i øjeblikket ligger i buffer i supersitet.

Implementeringen af modellen ligger i java-pakken *framework.tuplespaces.ts7*. I bilag B.3 viser vi definitionerne på funktionerne fra den generelle model.

4.7 Afrunding

I dette kapitel har vi beskrevet, hvordan vi har implementeret vores modeller. Kildekoden og kommentarer i form af *JavaDoc* findes på den vedlagte CD-rom (se bilag D for en komplet beskrivelse af indholdet på CD-rommen).

Under implementeringen stødte vi ind i nogle uforudsete fejl i både RMI og i den version af JavaCSP, som vi havde til rådighed. Den nuværende version af RMI (J2SE 1.4) er ikke helt stabil. Når VM'en belastes med mange samtidige RMI kald, smides til tider en *IllegalStateException* fra en timer klasse, som bruges i RMI. Fejlen vil i nogle tilfælde få vores program til at gå ned. Fejlen skulle blive rettet i næste frigivelse.

Den implementering af JavaCSP, som vi havde til rådighed, indeholdt også et par fejl, heriblandt et deadlock problem. Vi havde heldigvis adgang til kildekoden, og fik efter nogle dages intensivt arbejde løst problemerne. Den modificerede kildekode af JavaCSP findes også på CD-rommen.

Kapitel 5

Resultater

For at afprøve vores modeller, foretager vi en række forsøg, der har til formål at vise, at modellerne opfører sig som forventet i henhold til teorien fra kapitel 3. For hvert forsøg vi foretager, vil vi beskrive formålet med forsøget, fremgangsmåden, samt kommentere resultatet i forhold til det forventede resultat. I hvert forsøg genererer vi en række grafer, som kommenteres i forbindelse med resultatet af de enkelte forsøg. Disse grafer vil sammen med forsøgsopstillingerne kunne findes i bilag C.

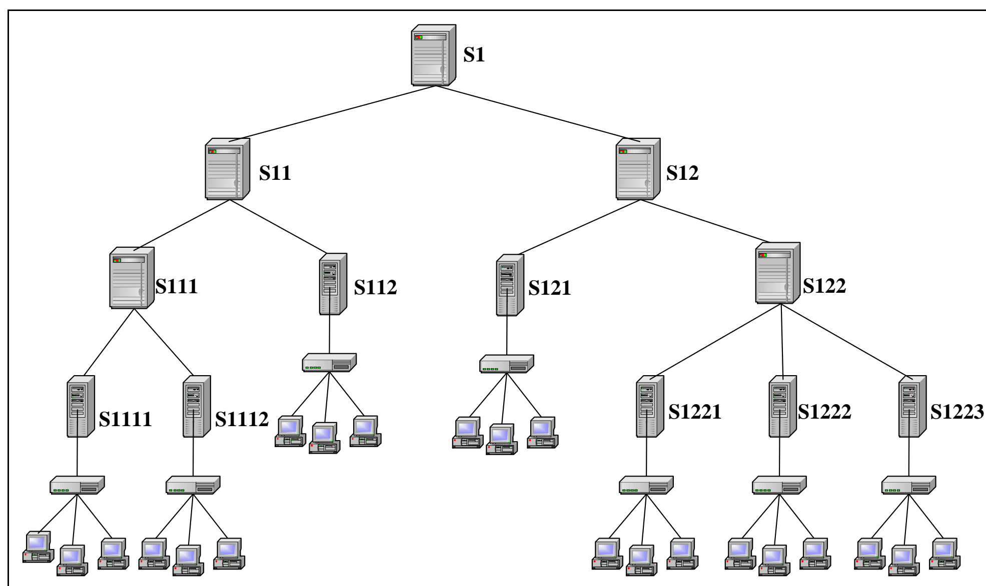
I afsnit 5.1 gennemgår vi de generelle fremgangsmåder ved forsøgene. I afsnit 5.2, 5.3 og 5.4 vil forsøgene vedrørende de tre modeller blive beskrevet. Til slut vil vi i afsnit 5.5 opsummere og kommentere resultaterne. Her vil vi desuden diskutere, hvilken betydning det vil have, når vi bevæger os fra vores testmiljø til et virkeligt miljø.

5.1 Fremgangsmåde

I hvert forsøg opbygger vi en hierarkisk struktur af sites og supersites. Vi starter et kendt antal arbejdere hos hver lokale ressourcemanager, som alle er frie, når forsøgene starter.

Der stilles opgaver i et eller flere sites. I nogle forsøg stilles alle opgaver fra starten af forsøget, mens der i andre forsøg også stilles opgaver senere i forløbet. Opgaver stilles ved at lave *out*-operationer med opgavetupler i et site. For hver opgave der stilles, foretages der én *out*-operation.

De opgaver og arbejdere der bruges i forsøgene for model 3, har forskellige ressourcekrav og -udbud. For bedre at kunne overskue hvad der foregår i forsøgene, bruger vi opgaver og arbejdere med kendte krav/udbud. Arbejdere og opgaver udvælges ud fra nogle faste typer. I tabel C.1 og C.2 ses ressourcekrav og -udbud for de opgave- og arbejdertyper vi bruger. For hver type opgave viser tabellen, udover ressourcekrav, også de opgavegrupper, som de enkelte opgavetyper tilhører. Hver opgavegruppe har to id'er: et logisk id fra 1 og fremefter, og et id der genereres automatisk ud fra opgavegruppens ressourcekrav. Desværre vil der i de grafer, som vi genererer i forsøgene, både blive brugt det logiske id og det autogenererede id. Dette skyldes, at vi i vores implementering ikke har været stringente med hvilket af de to id'er, der skulle bruges. Tabel C.1 vil derfor også kunne bruges til at se sammenhæng mellem de to id'er for opgavegrupperne.



Figur 5.1: Figuren viser den hierarkiske opbygning, som bliver brugt ved flere af forsøgene

Parameter beskrivelser

I forsøgene indsamler vi oplysninger om forskellige parametre som beskrevet i afsnit 4.5.2. I det følgende beskriver vi betydningen af de forskellige parametre:

Antal frie arbejdere: Antallet af frie arbejdere, som findes i et knudepunkt og alle dets børn. Bruges kun i model 1.

Belastningsfaktor: Belastningsfaktoren i et knudepunkt, dvs. antallet af endnu ikke schedulerede opgaver i knudepunktet og dets børn, divideret med det totale antal arbejdere i undertræet. Bruges kun i model 1.

Belastningstilstand: Angiver belastningstilstanden for et knudepunkt. I model 1 og model 2 er der kun én belastningstilstand i hvert knudepunkt. I model 3 er der en belastningstilstand for hver af opgavegrupperne i de enkelte knudepunkter. Belastningstilstanden kan antage værdierne 1 for let, 2 for normal og 3 for tung belastning.

Opgaver fra forælder: Antallet af opgaver et site/supersite modtager fra sin forælder. I model 3 haves både det totale antal, samt antallet for de forskellige opgavegrupper.

Opgaver til forælder: Antallet af opgaver et site/supersite afleverer til sin forælder. I model 3 haves både det totale antal, samt antallet for de forskellige opgavegrupper.

Næste Fri Tid (NFT): Bruges i model 2 og i model 3. For en arbejder, er NFT den tid, der er scheduleret opgaver hos arbejderen, dvs. den tid der er til at arbejderen kan påbegynde en ny opgave, som endnu ikke er scheduleret. NFT for et site, er den mindste NFT for alle arbejdere i sitet, når de lokale opgaver er skemalagt. I et supersite er NFT den mindste NFT for alle arbejdere i supersitets undertræ. Igen lægges først et skema for, hvordan balanceringen af børnenes opgaver samt de lokale opgaver skal foregå.

I model 2 er der i hvert site/supersite kun én NFT, mens der i model 3 vil være en NFT for hver aktiv opgavegruppe. I model 3 bruger vi desuden den generelle NFT for den lokale ressourcemanager i et site. Den generelle NFT er NFT'en på tværs af opgavegrupper, dvs. den arbejder hos den lokale ressourcemanager, som totalt set har den mindste NFT.

Total opgavemængde: Den totale opgavemængde (i CPUenheder) der findes i et site eller supersite og dets undertræ, inklusiv den opgavemængde som er scheduleret hos arbejderne.

Tilstand for en opgavegruppe: Bruges kun i model 3. Denne tilstand er *ikke* det samme som belastningstilstanden. Tilstanden for en opgavegruppe angiver hvilken logisk tilstand et site/supersite er i med hensyn til en given opgavegruppe. Tilstanden kan antage følgende værdier:

1 - lukket: Opgavegruppen er ikke aktiv i det pågældende site/supersite.

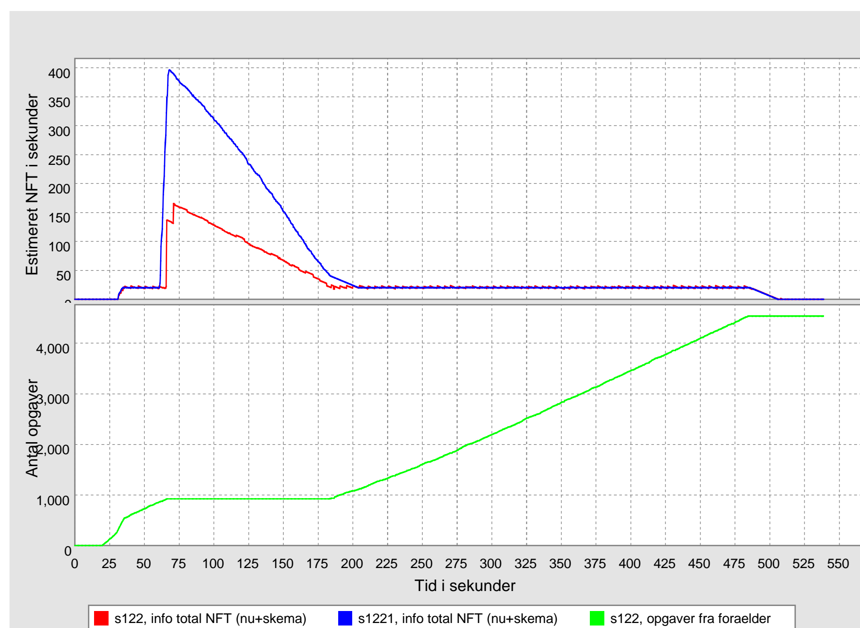
2 - åbner børn: Opgavegruppen er ved at blive aktiveret i et supersites undertræ.

3 - børn åbne: Opgavegruppen er aktiv i sitet/supersitet, og hos alle børn, men den er ikke aktiv hos dets forælder.

4 - lukker børn: Der er startet en inaktivering af opgavegruppen hos supersitet og dets børn.

5 - børn og forælder åbne: Opgavegruppen er aktiv hos alle børn og hos forælderen.

6 - åbnes hos forælder: Opgavegruppen er aktiv hos alle knudepunktets børn, og der er sendt en forespørgsel til forælderen om at få aktiveret opgavegruppen.



Figur 5.2: Figuren er taget fra forsøg 2-3 og er identisk med figur C.16. Øverst viser figuren NFT for et site og for et supersite fra forsøget. Nederst ses antallet af opgaver, som supersitet henter fra sin forælder.

I figur 5.2 ses et eksempel på en figur genereret under forsøgene. Figuren er taget fra forsøg 2-3 (se nedenfor), og viser NFT for et site (*S1221*) og for dennes forælder (*S122*), samt antallet af opgaver som *S122* henter fra sin forælder. Denne graf vil sammen med de resterende grafer der genereres under forsøgene kunne findes i bilag C.

5.2 Model 1

Forsøg 1-1

Mål: (1) vise de simple loadbalanceringsprincipper (2) vise at belastningstilstandene opfører sig som forventet, herunder at et site der ikke selv har opgaver opretholder sin buffer på X_{min}, og at et site hvor der stilles opgaver ikke afhjælpes, når bufferen i sitet når ned under X_{max}.

Fremgangsmåde: I dette forsøg bruger vi en opbygning som i figur 5.1, dvs. med 7 sites som er forbundet via 5 supersites. I hvert site er tilknyttet 100 arbejdere. Der stilles 4000 opgaver i site *S1111*. Model 1 tager som nævnt for givet, at alle arbejdere er ens, og at alle opgaver er ens. De opgaver vi stiller i denne simulering tager ca. 30 sekunder at udføre hos en arbejder[1].

X_{min} er i forsøget sat til 1 opgave, X_{max} til 3 opgaver og N_{max} til 10 opgaver.

Resultat: Det vigtigste mål med dette forsøg, var at vise, at den simple loadbalancering virker, dvs. at der flyttes opgaver fra de tungt belastede sites/supersites til de let belastede sites. I figur C.2 ses, at alle arbejdere i systemet er i gang med at løse opgaver i al den tid, hvor venstre del af træet er tungt belastet. Der foregår altså en flytning af opgaver fra venstre side af træet til arbejderne i højre side af træet.

På den nederste del af figur C.3 ses desuden, at i de sites, hvor der ikke stilles opgaver, forsøges der opretholdt en belastningsfaktor på $X_{min} = 1.0$. I figuren ses en jævn belastning for *S1112*, mens der er nogle store udsving i belastningen for *S1223*. Dette skyldes, at *S1112* er placeret direkte som et barn til et supersite, der selv er forfader til det site, hvor opgaverne stilles. *S1223* har derimod to supersites imellem sig selv og en forfader til opgave sitet (*S1111*). *S1112*'s forælder vil trække opgaver op til sin egen buffer så længe *S1111* er tungt belastet, der vil derfor være en stor chance for, at der er opgaver hos *S111*, når *S1112* skal bruge en opgave. Da vi i model 1 ikke tager højde for tiden, vil supersitet i højre side af træet (*S12*), ikke trække opgaver ned fra *S1* før end *S12* selv er let belastet. Når *S1223* bliver let belastet, skal der derfor hentes opgaver fra toppen af træet. Da alle opgaver i forsøget er næsten lige lange, vil alle arbejdere blive frie på omtrent den samme tid, hvilket forstærker udsvingene. Det ses endvidere at udsvingene for *S1223*'s NFT har en tendens til at blive af mindre størrelse, som tiden går, mens den tid udsvingene varer forøges en smule. Dette skyldes, at der trods alt er en lille afvigelse i opgavelængden, og arbejderne vil som tiden går, blive færdige på mere og mere tilfældige tidspunkter.

[1]Der er en lille afvigelse (på max. 5%) fra de 30 sekunder i udførelsen af hver enkelt opgave. Denne afvigelse er lagt ind, for at bryde sekvenser hvor alle arbejdere i systemet periodisk præcis hvert 30. sekund skal have allokeret en opgave.

Et site, hvor der stilles opgaver (*S1111*) afhjælpes kun i den periode hvor sitet er tungt belastet. Dette ses i figur C.4, hvor der øverst haves belastningstilstanden for *S1111* og nederst antallet af opgaver som *S111* henter fra *S1111*. Det ses at *S111* stopper med at hente opgaver fra *S1111* når *S1111* bliver normalt belastet.

Forsøg 1-2

Mål: (1) at afprøve balanceringen når der stilles opgaver i flere sites samtidigt, (2) vise at et site, der selv har opgaver, ikke trækker opgaver ned, før dets egne opgaver er løst, (3) at et supersite med belastede børn ikke afhjælper andre før supersitet som helhed er let belastet.

Fremgangsmåde: Vi bruger igen opbygningen som i figur 5.1, og igen er der 100 arbejdere i hvert site. Der stilles nu 5000 opgaver i *S1111* og 500 opgaver i *S1112*. Der ventes i ca. 20 sekunder, hvorefter der stilles 2000 opgaver i *S1221*.

Med de stillede opgaver, bør der i *S1* foregå en balancering mellem *S11* og *S12* i starten af forsøget. Når opgaverne stilles i *S1221*, som gør hele *S12*'s træ overbelastet, bør balanceringen stoppe, i en periode indtil *S12* igen bliver let belastet.

X_{min} er igen sat til 1 opgave, X_{max} til 3 opgaver og N_{max} til 10 opgaver.

Resultat: Forsøget viser, at loadbalanceringen også virker, når der stilles opgaver forskellige steder i træet. Forsøget illustrerer desuden, at et site ikke henter opgaver fra dets forælder, så længe sitet selv er normalt eller tungt belastet (figur C.8).

I figur C.6 og C.7 ses, at der på et tidspunkt i forsøget var frie arbejdere i *S121* (ca. 25 arbejdere), samtidigt med at hele venstre del af træet var tungt belastet. Dette skyldes problemerne med brug af gennemsnits betragtninger (beskrevet i afsnit 3.5.2), som det også illustreres på de to figurer.

Den midterste del og den nederste del af figur C.7 viser desuden, at et supersite, der p.g.a. et af sine børn er tungt belastet, ikke henter opgaver ned fra sin forældre. I stedet henter supersitet opgaver fra sit tungt belastede barn (og fordeler dem til de let belastede børn).

5.3 Model 2

Forsøg 2-1

Mål: (1) vise at model 2 opfører sig som model 1, når alle opgaver er ens og alle arbejdere er ens.

Fremgangsmåde: Opbygningen er som i figur 5.1 med 100 arbejdere i alle sites. Alle arbejdere er ens (de kan levere 1000 CPUenheder/s). Der stilles 4000 ens opgaver i *S1111* (30000 CPUenheder, hvilket betyder at en udførsel af en opgave vil tage 30 sekunder.).

X_{min} er sat til 20 sekunder, X_{max} til 40 sekunder og N_{max} til 10 opgaver.

Resultat: Forsøget viser, at modellen opfører som model 1, når alle opgaver er ens og alle arbejdere er ens. Figur C.10 viser at alle sites i højre side af træet har en NFT, som overholder $X_{min} = 20s$ i den periode, hvor venstre side af træet er tungt belastet. Det ses desuden at der ikke er de samme nedadgående udsving i NFT'en for de enkelte sites som for belastningsfaktoren i model 1[2]. Dette skyldes, at model 2 tager højde for at tiden går, og på den måde kan et supersite trække opgaver ned fra sin forælder, før end dets egne børn bliver let belastede.

I figur C.11 ses NFT for venstre side af træet (*S11*), højre side af træet (*S12*) samt for hele træet (*S1*). Det ses at venstre side er meget belastet, mens højre side af træet, aldrig har en NFT der kommer over 50 sekunder (X_{min} plus længden af en opgave). Det ses også at NFT for *S1* er en *udglatning* af de to børns NFT.

NFT for et supersite fortæller som sagt, hvor lang tid der er til at en ny opgave vil kunne blive påbegyndt i sitets undertræ. For et supersite, som ikke påvirkes udefra (dvs. som ikke har en forælder, der trækker opgaver op fra supersitet, og har børn/børnebørn, hvor der ikke stilles nye opgaver), betyder det, at hældningen på NFT grafen bør være -1. I figur C.11 ses til tiden t=20 sekunder, at *S1* mener der er lige knap 150 sekunder til en opgave kan påbegyndes i hele systemet. Nulpunktet nås lige før t=170 sekunder, altså en hældning på ca. -1. Det ses også at *S1*'s forudsigelser, passer godt med det tidspunkt, hvor der rent faktisk er frie arbejdere i højre side af træet.

Forsøg 2-2

Mål: (1) vise at systemet kan klare en kombination af opgaver med stor forskel i deres størrelse, således at problemstillingen illustreret i figur 3.5 ikke opstår.

Fremgangsmåde: Der haves to sites A og B, som er forbundet via et enkelt supersite. I hvert site er der 100 arbejdere som kan levere 1000 CPUenheder pr. sekund. Der stilles 50 store opgaver (600000 CPUenheder) i A. På den måde vil halvdelen af arbejderne i A være optaget i ca. 10 minutter, mens den anden halvdel er frie. Med $X_{min} = 20s$, $X_{max} = 40s$ og $N_{max} = 10$, ses det også, at rene gennemsnitsbetragtninger ville gøre at A nu er tungt belastet. Efter 60 sekunder stilles der 1000 opgaver á 30000 CPUenheder i B.

Resultat: Forsøget viste, at der til trods for de 50 meget lange opgaver i A, foregik en balancering mellem A og B. I figur C.13 ses at A's NFT kun er nul, så længe der ikke er opgaver i B. Så snart der stilles opgaver i B, starter balanceringen, og A's NFT holdes over $X_{min} = 20s$. Figuren viser også, at til trods for, at den totale opgave CPU-mængde (det totale antal opgave CPUenheder) er størst i A, så foregår balanceringen fra B til A og ikke omvendt.

[2]De opadgående udsving i NFT fremkommer hver gang der allokeres en opgave hos en arbejder. Da alle opgaver er lige lange, vil alle arbejdere i et site skulle have allokeret opgaver samtidigt. Dvs. til et tidspunkt hvor der allokeres opgaver til arbejderne i et site, vil alle arbejderne have en NFT på ca. $X_{min} = 20s$. Da de alle får allokeret en opgave på 30 sekunder, vil alle arbejdernes NFT efter allokeringen være ca. 50 sekunder. Det bemærkes igen, at NFT for et site, er den mindste NFT for alle arbejderne i sitet.

Forsøg 2-3

Mål: (1) vise at loadbalanceringen virker med en bred vifte af forskellige arbejdere og opgaver, samt opgaver der stilles forskellige steder i træet.

Fremgangsmåde: Opbygningen er som i figur 5.1. I *S1111*, *S1112*, *S1221*, *S1222* og *S1223* er der 100 arbejdere, med et CPU udbud mellem 1000 og 5000 CPUenheder pr. sekund (for hver arbejdere beregnes en tilfældig værdi mellem 1000 og 5000 CPUenheder, som er arbejderens CPU udbud). I *S112* er der 10 arbejdere med et CPU udbud mellem 10000 og 15000 CPUenheder og i *S121* 250 arbejdere med et udbud mellem 500 og 2000 CPUenheder.

Der stilles 10000 opgaver i *S1111* fordelt tilfældigt mellem 40000 og 100000 CPUenheder. Efter ca 10 sekunder stilles der 500 opgaver fordelt tilfældigt mellem 200000 og 500000 CPUenheder i *S1112*. Efter yderligere 20 sekunder stilles der 2000 opgaver tilfældigt fordelt mellem 40000 og 60000 CPUenheder i *S1221*.

Den mindste opgave i systemet er altså på 40000 CPUenheder. Den vil på den langsomste arbejder (500 CPUenheder pr. sekund) have en estimeret udførelsestid på $40000/500 = 80$ sekunder, og på den hurtigste arbejder $40000/15000 = 2,6$ sekunder. Den største opgave i systemet (500000 CPUenheder) vil tilsvarende have estimeret udførelsestider på 1000 og 33,3 sekunder på den mindste henholdsvis største arbejder.

I forsøget er X_{min} sat til 20 sekunder, X_{max} til 40 sekunder og N_{max} til 10 opgaver.

Resultat: Forsøget viser, at der også med forskellige størrelser af opgaver og arbejdere foregår en loadbalancering, som involverer alle sites i systemet. I figur C.15 ses belastningstilstanden for venstre side af træet, dvs. den side af træet hvor de fleste opgaver stilles, samt NFT for de sites i højre side af træet, hvor der ikke stilles opgaver. Som det fremgår af figuren, så er alle arbejdere i sitene i gang med at udføre opgaver, så længe venstre side af træet er tungt belastet.

Forsøget viser desuden, at et site, hvor der pludselig stilles opgaver (*S1221*), stopper med at trække opgaver ned fra sin forælder, indtil sitet igen bliver let belastet (ses i figur C.16). På samme måde viser figur C.17 at et supersite (*S12*), der på et tidspunkt får et tungt belastet barn, som også gør supersitet tungt belastet, også stopper med at trække opgaver ned fra sin forælder. I stedet startes der en lokal balancering af supersitets børn, dvs. der flyttes opgaver fra det tungt belastede barn (i dette tilfælde S122) over til de/det let belastede børn/barn (her *S121*). Figur C.18 illustrere desuden, at NFT'en for de tungt belastede sites/supersites i venstre side af træet falder langsommere i den periode, hvor højre side af træet er tungt belastet. Dette skyldes naturligvis, at *S1* i den pågældende periode ikke foretager en loadbalancering mellem *S11* og *S12* da de begge er tungt belastede.

Figur C.19 illustrere betydningen af store opgaver for et site med et relativt lille antal arbejdere (*S112*). Øverst i figuren ses det totale antal opgaver CPUenheder der findes i sitet og i midten sitets NFT. Den nederste del af figuren viser det antal opgaver der hentes fra *S1112* (det site hvor de store opgaver stilles), dvs. antallet af *store* opgaver, som kommer ud i systemet. Det ses at den totale opgave CPU mængde er betydeligt større i den periode, hvor der leveres store opgaver til systemet. Det bemærkes endvidere, at der ikke er de samme udsving i NFT for

S112. Dette skyldes, at sitet både modtager store opgaver, der stammer fra *S1112*, men også de mindre opgaver som stammer fra *S1111*.

Forsøget viste desuden, at et supersite (*S111*) med to tungt belastede børn, afhjælper begge børn. Vi har valgt den implementeringsmæssige strategi, at der i et sådan tilfælde forsøges hentet lige mange opgaver op fra hvert af de tungt belastede børn (hvilket også fremgår af figur C.21). Det betyder i dette tilfælde, hvor der er stor forskel i *hvor* tungt belastede de to børn er, at der total set bliver flyttet flere opgaver, end hvis man f.eks. havde valgt kun at hente opgaver fra det mest belastede barn. Med andre ord flyttes der i forsøget opgaver fra *S1112*, mens det er tungt belastet, for senere, når sitet bliver let belastet, at flytte (andre) opgaver til *S1112*.

5.4 Model 3

Forsøg 3-1

Mål: (1) vise at model 3 overholder model 2, (2) vise at modellen kan håndtere to balanceringer, som ikke interfererer.

Fremgangsmåde: Der bruges samme fremgangsmåde som i forsøg 2-3, med følgende ændringer: For hver arbejder i forsøg 2-3 haves nu 2 arbejdere, en af type 1 (kan levere 100 MB RAM) og en af type 7 (kan levere 2 GB DISK). Arbejdernes CPU udbud fordeler sig på samme måde som i forsøg 2-3. Eksempelvis er der i *S112* 10 arbejdere af type 1, som kan levere mellem 10000 og 15000 CPUenheder og 10 arbejdere af type 7 som også kan levere mellem 10000 og 15000 CPU.enheder. For hver opgave det stilles i forsøg 2-3 stilles nu 2 opgaver, en af type 1 (skal bruge 32 MB RAM) og en af type 7 (skal bruge 200 MB Disk).

Det ses at opgavetype 1 kun kan løses af arbejdertype 1 og opgavetype 7 kun af arbejdertype 7. Det forventede resultat af forsøget er derfor to samtidige balanceringer der foregår på samme måde som balanceringen i forsøg 2-3.

Resultat: Forsøgt er som sagt meget analogt til forsøg 2-3, i dette tilfælde med to ikke interfererende opgavegrupper, der hver for sig bør fungere som forsøg 2-3. Figur C.23, C.24, C.25 og C.26 viser de samme informationer som figur C.15, C.16, C.17 og C.18 henholdsvis, men for hver af de to opgavegrupper (100990=1 og 2420125=7).

Det ses at der er overordentligt god overensstemmelse (med undtagelse af nogle få afvigelser, som beskrives nedenfor) mellem resultaterne fra forsøg 2-3 og resultaterne for hver af grupperne i dette forsøg. Balanceringen i model 3 fungere på samme måde som i model 2, når alle opgaver og alle arbejdere tilhører samme typer. Forsøget viser desuden at model 3 håndtere tiden, samt størrelser på arbejdere og opgaver på samme måde som model 2.

Der er enkelte afvigelser fra resultatet af dette forsøg og resultatet af forsøg 2-3. I figur C.26 ses, at belastningstilstanden for *S12* svinger mellem let og normal belastning i de perioder, hvor der ikke er overskydende opgaver i *S12*'s undertræ. Dette gør sig ikke gældende for den tilsvarende figur for forsøg 2-3 (figur C.18). Grunden til denne forskel skyldes at implementeringen af model 3 (som beskrevet i afsnit 3.7.10) ikke arbejder med en øver grænse på bufferstørrelsen i et supersite, når

der trækkes opgaver ned fra forælderen. Den eneste begrænsning er, at supersitet skal være let belastet, når det henter en opgaver fra sin forælder. En enkelt opgave vil kunne få et supersite til at skifte fra let til normal belastning, hvorfor disse tilstandsskift fremkommer. I model 2 haves en øvre grænse for bufferstørrelsen på N_{max}, der vil derfor ikke blive hentet opgaver ned indtil supersitet er normalbelastet, men kun indtil den lokale buffer i supersitet er fyldt op.

Forsøg 3-2

Mål: (1) vise at problemerne beskrevet i afsnit 3.7 med belastningsubalance ikke forekommer.

Fremgangsmåde: Der haves to sites A og B, som er forbundet via et enkelt supersite. I både A og B er der 50 arbejdere af type 1 og 50 arbejdere af type 7. Alle arbejdere kan levere 1000 CPUenheder pr. sekund. I A stilles der 100 opgaver af type 1 som skal bruge 120000 CPUenheder, og i B stilles der 100 opgaver af type 7 som også skal bruge 120000 CPUenheder.

X_{min} er sat til 20 sekunder, X_{max} til 40 sekunder og N_{max} til 10.

Resultat: Forsøget viser, at der foregår en balancering mellem de to sites, til trods for at antallet af opgaver i begge sites ikke overstiger antallet af frie arbejdere i sitene. Dette fremgår af figur C.28[3], som viser NFT for både A og B, samt antallet af opgaver der udveksles. Forsøget viser at model 3 håndtere den beskrevne belastnings ubalance fint.

Forsøget illustrere desuden at en aktiv opgavegruppe lukker igen så snart alle opgaver er balanceret. Dette fremgår af den nederste del af figur C.28, som viser en af de to opgavegruppers tilstand. (Denne tilstand er ikke det samme som belastningstilstanden for opgavegruppen. Se afsnit 5.1 for en nærmere beskrivelse). Opgavegruppen er kun aktiv i et ganske kort tidsrum, indtil alle opgaver, der kan flyttes, er blevet flyttet.

Forsøg 3-3

Mål: (1) vise at balanceringen fungere med to opgavetyper og to arbejdertyper, (2) vise at site autonomiteten fungerer, (3) illustrere problemet med at den arbejdertype, som kan løse flere typer opgaver, ikke kun løser den *rigtige* opgavetype.

Fremgangsmåde: Der haves to sites A og B. I A er der 100 arbejdere som kan levere 100 MB RAM (type 1) og i B er der 100 arbejdere som kan levere 100 MB RAM og 2 GB Disk (type 9). I A stilles der 500 opgaver der skal bruge 32 MB RAM (type 1) og 300 opgaver der skal bruge 32 MB RAM og 200 MB Disk (type 6). I B stilles der 200 opgaver der skal bruge 32 MB RAM (type 1). Alle arbejdere i systemet kan levere 1000 CPUenheder pr. sekund, og alle opgaver skal bruge 30000 CPUenheder.

X_{min} er sat til 20 sekunder, X_{max} til 40 sekunder og N_{max} til 10.

[3] Som det fremgår af figuren, så var logningen slået til i ca. 50 sekunder før opgaverne blev stillet. Selve forsøget starter derfor først kort før t=50s.

Resultat: Forsøget viser at der foregår en balancering af opgaver mellem A og B. Til trods for, at der bliver stillet opgaver i et site, som ikke kan løses i sitet, så bliver alle opgaver i systemet løst.

Forsøget illustrere desuden site autonomiteten, i og med at B først løser sine egne opgaver, inden sitet begynder at afhjælpe A (se figur C.30).

Med de stillede opgaver og de angivne værdier for X_{min} og X_{max}, illustrere forsøget desuden at B ikke afhjælper A på den mest optimale måde. Dette indses på følgende måde (se figur C.30): De 200 opgaver der stilles i B, vil blive scheduleret med det samme (2 på hver arbejder, som så alle har NFT=60s). På tilsvarende måde vil der blive scheduleret 200 opgaver af type 1 i A. Efter ca. 20 sekunder er NFT hos alle arbejdere i begge sites ca. 40 sekunder. Det betyder at der i A, hvor der stadig findes ikke schedulerede opgaver, vil blive scheduleret yderligere 100 opgaver, og arbejdernes NFT i A nu er 70 sekunder. Efter yderligere 20 sekunder, er NFT for arbejderne i B 20 sekunder, og der begynder en balancering af opgaver fra A til B. På det tidspunkt findes der i A 200 opgaver af type 1 og 300 opgaver af type 2. Arbejderne i A har alle en NFT på ca. 50 sekunder. Det vil derfor på dette tidspunkt være mest optimalt hvis B kun løser opgaver af type 6. På den måde vil alle opgaver af type 1 være færdige efter 50+30+30=110 sekunder, og alle opgaver af type 6 efter 20+30+30+30=110 sekunder. Men som det fremgår af figur C.30, så modtager B lige mange opgaver af type 1 og af type 6. Resultatet bliver heraf at B kommer til at arbejde i en længere periode end A, og at den sidste opgave først er løst i B efter ca. 140 sekunder.

Forsøg 3-4

Mål: (1) vise at balanceringen virker med flere forskellige arbejdertyper og flere forskellige opgavetyper, (2) illustrere åbning og lukning af de aktive opgavegrupper

Fremgangsmåde: Der bruges en opbygning som i figur 5.1. *S1111*, *S1112*, *S1221*, *S1222* og *S1223* er ens site med hver 100 arbejdere fordelt på følgende måde:

Arbejdertype	CPU udbud	Antal
1	1000-5000	10
7	1000-5000	10
4	1000-5000	15
5	1000-5000	15
5	500-1000	10
5	7000-8000	10
10	1000-5000	5
11	1000-5000	5
13	1000-5000	10
14	1000-5000	10

I *S112* er der 10 arbejdere af type 16, og 5 arbejdere af type 18, som alle har et CPU udbud mellem 10000 og 15000 CPUenheder/s. I *S121* er der 100 arbejdere af type 1 med et CPU udbud mellem 500 og 2000 CPUenheder/s, samt 1 arbejder af type 20 med et CPU udbud på 5000 CPUenheder/s.

I *S1111* stilles der 2500 opgaver af type 1 og 2500 opgaver af type 3. I *S1112* stilles der 1000 opgaver af type 5, 1000 opgaver af type 8, og 500 opgaver af type 6. Alle disse opgaver skal bruge mellem 40000 og 100000 CPUenheder. Efter 30 sekunder stilles der 15 lange opgaver (300000 CPUenheder) i *S1221* af type 9, samt 1 opgave af type 10 i *S112* på 600000 CPUenheder.

Resultat: Figur C.32, C.33 og C.34, viser belastningstilstanden i venstre side af træet (*S11*) for de fem opgavetyper der bliver stillet i *S1111* og *S1112*. Figurerne viser desuden NFT for samme opgavegrupper hos et udsnit af de sites der findes i højre side af træet. Figurerne illustrerer, at i den tid, hvor venstre side af træet er overbelastet, holdes NFT for de samme opgavegrupper omkring X_{min} hos sitene i højre side af træet. Dvs. der foregår en balancering af opgaver fra venstre side af træet til højre side af træet.

Efter et stykke tid, hvor der kun har været stillet opgaver i venstre side af træet, stilles der nogle lange opgaver af type 9 (RAM=32, Disk=2000, OS=Linux) hos *S1221*. Der stilles 15 opgaver, hvilket er tilpas mange til at *S1221* bliver tungt belastet, men ikke nok til at *S122* bliver mere end normalt belastet[4]. Der startes en lokal balancering af type 9 opgaverne, som har *S122* som toppunkt. Balanceringen er rimeligt hurtigt overstået, og den aktive opgavegruppe lukkes igen. Denne lokale balancering er illustreret i figur C.35. Det er hver at bemærke, at der på intet tidspunkt under balanceringen af type 9 opgaverne, er involveret andre dele af systemet. End ikke *S122*'s forælder kender til opgavernes eksistens, men kan kun konstatere, at nogle af de arbejder, der hidtil har været brugt i andre balanceringer, i en kortere periode er belastet.

Figur C.36 viser den generelle NFT hos de lokale ressourcemanagere i alle sites i systemet. Den generelle NFT hos de lokale ressourcemanagere fortæller hvornår ressourcemanageren tidligst vil kunne påbegynde en endnu ikke scheduleret opgave. NFT'en fortæller derimod ikke hvilken type opgave der kan påbegyndes, dvs. der kan godt være en større NFT for nogle typer opgaver hos ressourcemanageren. *S1111* er det eneste site i dette forsøg, som selv har opgaver til alle de arbejdertyper, der findes i sitet. *S1111* holder derfor en generel NFT omkring X_{max}. De resterende sites er alle afhængige af opgaver fra andre sites for at udnytte deres arbejdere fuldt ud. De holder derfor en generel NFT omkring X_{min}.

Figur C.37 viser NFT i *S121* for den ene specielle opgave (type 10), der stilles i *S112*. Opgaven af type 10 kan kun løses af én arbejder i hele systemet. Det er arbejderen af type 20, som findes i *S121*. Figur C.37 illustrerer, at den ene arbejder er indordnet efter de aktive opgavegrupper i den periode, hvor opgaven af type 10 ikke findes i systemet. Både før og efter at arbejderen bruges til at løse den store opgave af type 10, er der ingen andre end *S121*, der kender til den specielle arbejders komplette ressource udbud.

Forsøg 3-5

Mål: (1) vise at balanceringen fungere i et større system med en bred vifte af forskellige arbejdere/opgaver.

[4] *S1221* har selv 10 arbejdere, der kan løse opgaver af type 9. Det samme gør sig gældende for *S1222* og *S1223*, så ialt har *S122* 30 arbejdere under sig, som kan løse opgaver af type 9

Fremgangsmåde: Til dette forsøg bruges en træstruktur, som har 4 niveauer. I hvert knudepunkt forgrener træet sig i 4 undertræer. Dvs. i nederste niveau haves ialt 64 sites. Alle sites på nær to er standard sites, med samme fordeling af arbejdere som i forsøg 3-4:

Arbejdertype	CPU udbud	Antal
1	1000-5000	10
7	1000-5000	10
4	1000-5000	15
5	1000-5000	15
5	500-1000	10
5	7000-8000	10
10	1000-5000	5
11	1000-5000	5
13	1000-5000	10
14	1000-5000	10

I *S1222* er der 500 arbejdere af type 8, som hver kan levere mellem 500 og 1000 CPUenheder. I *S1333* er der 50 arbejdere af type 14, som hver kan levere mellem 10000 og 20000 CPUenheder. Alt i alt er der i forsøget derfor 6750 arbejdere.

Der stilles i alt 68000 opgaver, som alle har et CPU behov mellem 50000 og 200000 CPUenheder. Opgaverne stilles i følgende sites:

Opgavetype	Antal	Site
1	10000	*S1111*
1	10000	*S1144*
3	10000	*S1211*
3	10000	*S1244*
5	4000	*S1311*
8	4000	*S1411*
6	10000	*S1444*
6	10000	*S1344*

Med denne fremgangsmåde er der ingen sites i systemet, som har opgaver til alle sine egne arbejdere. Alle sites skal derfor modtage opgaver, som er stillet i andre sites, for at udnytte alle deres arbejdere.

Resultat: Forsøget viser, at balanceringen også virker, når antallet opgaver og arbejdere i systemet forøges markant.

I figur C.39 vises NFT for de lokale ressourcemanagere i nogle udvalgte sites. Figuren viser bl.a. at det tager noget tid (ca. 50 sekunder fra de første opgaver er i systemet) indtil alle arbejdere er belastet. Dette skyldes to ting: Dels tager det omtrent den tilsvarende tid at stille alle opgaverne, og dels så tager det tid at flytte opgaver rundt i systemet. Her skal det også bemærkes, at de sites, hvor der eksempelvis stilles 10000 opgaver, i starten er meget belastede med hensyn til deres eget CPU forbrug. Dette skyldes, at samtidigt med at de bliver bombarderet med nye opgaver, så begynder deres forælder hurtigt på at trække opgaver op fra dem. Efter denne

indledende fase er der dog, som det fremgår af figuren, ingen problemer med at få distribueret opgaver nok rundt i systemet.

Figur C.39 illustrer desuden at et site (*S1333*), som sidder tæt på en opgavekilde, der kan levere opgaver til alle sitets arbejdere, meget hurtigt bliver belastet.

5.5 Vurdering af resultater

Vi har i dette kapitel beskrevet en række forsøg, som vi har foretaget med de tre modeller. Overordnet viser forsøgene god overensstemmelse med den teori, som er beskrevet i kapitel 3. Forsøgene viser, at de løsninger vi foreslår i kapitel 3 virker efter hensigten. Samtidigt illustrerer forsøgene de fejl og mangler, der er i de tre modeller og som også beskrives i kapitel 3.

Resultaterne skal naturligvis ses i lyset af, at visse dele af systemet er simuleret, og at systemet derfor ikke nødvendigvis opfører sig præcis som det ville gøre i et virkeligt system. Vi vil i det efterfølgende se på betydningen af disse simuleringer.

I vores forsøg eksisterer der ingen fysiske opgaver og ingen fysiske arbejdere. Opgaver og arbejdere er hver især repræsenteret ved en beskrivelse, og når opgaverne *udføres* hos en arbejder simuleres dette som beskrevet i afsnit 4.5. I vores ideelle testmiljø, starter opgaverne med det samme i det øjeblik, de bliver allokeret til en fri arbejder. I den virkelige verden, vil det typisk kræve en form for koordinering mellem opgave/opgavestiller og arbejderen, før selve udførelsen kan påbegyndes. Eksempelvis skal den fysiske opgave (program samt data) flyttes hen til den arbejder, som skal udfører opgaven. Denne forsinkelse i opgaveudførelsen vil have den generelle indvirkning på vores system, at de enkelte knudepunkter i træet, vil tro at de er mindre belastede end de i virkeligheden er. Det betyder, at der for tidligt (i forhold til X_{min}) vil blive trukket opgaver ned mod let belaste sites. Man bør derfor i de estimater, der genereres for opgaveudførelserne, tage højde for det bidrag der kommer, når opgaverne flyttes og startes hos den enkelte arbejder. Dette er ikke nogen triviel opgave, da f.eks. netværksbåndbredden mellem opgavens fysiske placering og den arbejder, der skal løse opgaven, ikke kan antages for kendt. Det er dog vigtigt at pointere, at det kun er ved allokering af opgaver på frie (og næsten frie) arbejdere, at ovenstående problem indtræffer. Hvis der f.eks. på en arbejder er en opgave igang, som man forventer tager 60 sekunder, så vil man kunne påbegynde initialiseringen af den nye opgave inden den gamle opgave afsluttes, og problemet vil ikke indtræffe[5].

I forsøgene har vi for overskuelighedens skyld valgt at bruge statiske arbejdere i de enkelte sites. I den virkelige verden, vil arbejderne ikke være statiske. Som tiden går, vil der både opstå og forsvinde arbejdere. Arbejdere vil også ændre ressourceudbud (eksempelvis vil deres CPU udbud kunne både stige og falde). Et site vil hurtigt opdage ændringer i den lokale ressourcemanagers udbud af arbejdere, da synkroniseringen mellem et site og den lokale ressourcemanager foregår med høj frekvens. Et site vil derfor med det samme ændre sine handlinger, så de passer med den ændrede tilstand hos den lokale ressourcemanager.

[5]Det kræver naturligvis at den nye opgave kan initialiseres på under 60 sekunder uden at det går ud over udførelsen af den igangværende opgave. Hvis den igangværende opgave har behov for meget netværksbåndbredde, er det ikke sikkert at man kan flytte en ny opgave til arbejderen, uden at påvirke den igangværende.

Sitets forælder og bedsteforældre er derimod afhængige af information fra sitet om den ændrede tilstand. Indtil den opdaterede information når op gennem træet, vil der hos bedsteforældrene kunne blive taget forkerte beslutninger.
Overordnet set er der to mulige fejltilstande: (1) Et supersite tror fejlagtigt, at sitet er normalt/tungt belastet og vil derfor muligvis ikke trække opgaver ned fra sin forælder. (2) Et supersite tror fejlagtigt, at sitet er let/normalt belastet og afhjælper derfor ikke det pågældende site, men trækker muligvis opgaver ned fra sin forælder, som er tiltænkt sitet. (1) betyder, at sitets arbejdere muligvis ikke udnyttes optimalt for en periode. (2) betyder, at opgaver, som muligvis vil kunne løses andre steder i træet, ligger i kø i for lang tid. Vi ser ikke (1) som et stort problem, mens (2) er ret uhensigtmæssigt. Med indførelse af en maksimal bufferstørrelse når der trækkes opgaver ned (som bekrevet i afsnit 3.7.10), vil vi dog kunne begrænse det antal opgaver, som trækkes ned i tilfælde (2).

De data som er indsamlet under forsøgene, og som er brugt til at generere de viste grafer, kan findes på den vedlagte CD-ROM. Graferne kan genereres ud fra disse datafiler ved hjælp af vores grafiske værktøj (TSD).

Kapitel 6

Globus integration

I dette kapitel vil vi analysere muligheden for, at integrere vores loadbalanceringsmodel i et eksisterende ressourceallokeringssystem som Globus. Globus er et meget modulært system, hvor snitfladerne mellem de enkelte moduler er pænt defineret. Ideen bag modulopbygningen i Globus er, at de enkelte moduler kan udskiftes efter behov, uden at resten af systemet skal ændres.

Vi har tidligere kort beskrevet grundprincipperne i Globus, men vil her komme med en mere udførlig forklaring af det overordnede design. På figur 6.1 ses opbygningen af Globus' ressourceallokeringssystem.

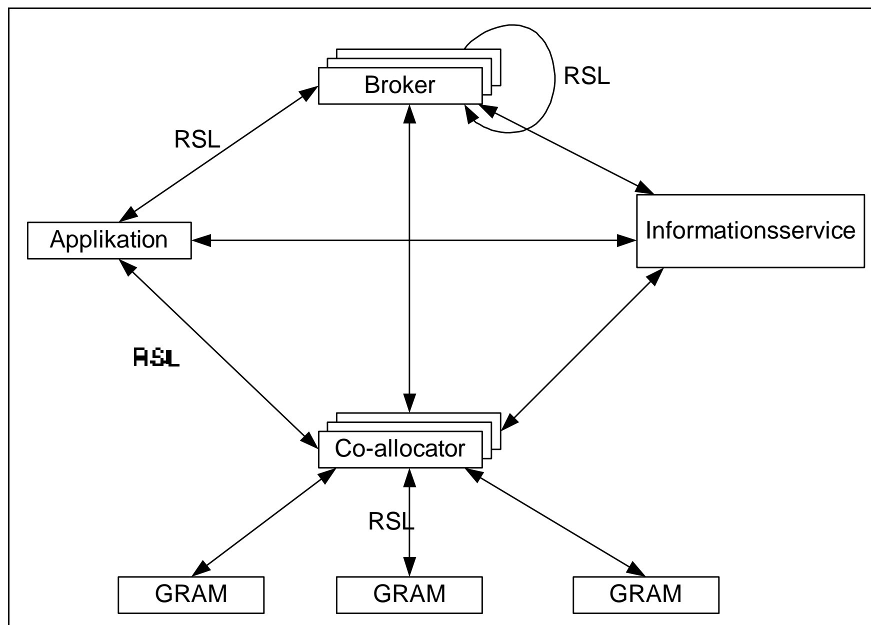
Applikation: Applikationen er den entitet, som ønsker at få løst nogle opgaver. Til dette skal bruges nogle arbejdere.

Broker: En *broker* er en specialiseret agent, der kan hjælpe applikationer med at få konkretiseret deres ressourcebehov til arbejdere. En brokers typiske opgave er, at omforme et (abstrakt) ressourcebehov fra en applikation til et mere konkret ressourcebehov.

Informationsservice: *Informationsservicen* bruges af applikationer og brokere til at lokalisere arbejdere ved hjælp af søgninger. Informationsservicen opdateres hele tiden med informationer om alle arbejdere i systemet.

GRAM: En *GRAM* (Globus Resource Allocation Manager) er en lokal Globus ressourcemanager. Denne har til opgave, at modtage opgaver fra applikationer og allokere disse hos lokale arbejdere. Den skal ligeledes overvåge og kontrollere udførelserne af opgaver, og eventuelt rapportere status og eventuelle fejlmeddelelser til opgavernes applikationer. En GRAM skal desuden sørge for, at holde informationsservicen opdateret med hensyn til de arbejdere GRAM'en administrerer (dette gøres af en *GRAM Reporter*).

Co-allocator: En *Co-allocator* bruges typisk, når en applikation har mange delopgaver. En co-allocators opgave er, at administrere udførelsen af opgaverne, når disse er allokeret hos flere forskellige GRAM'er. En co-allocator vil f.eks. kunne have til opgave at sikre, at alle delopgaverne startes *samtidigt* hos forskellige GRAM'er.



Figur 6.1: Figuren viser opbygningen af Globus' ressourceallokeringsframework. Figuren er lånt fra [6].

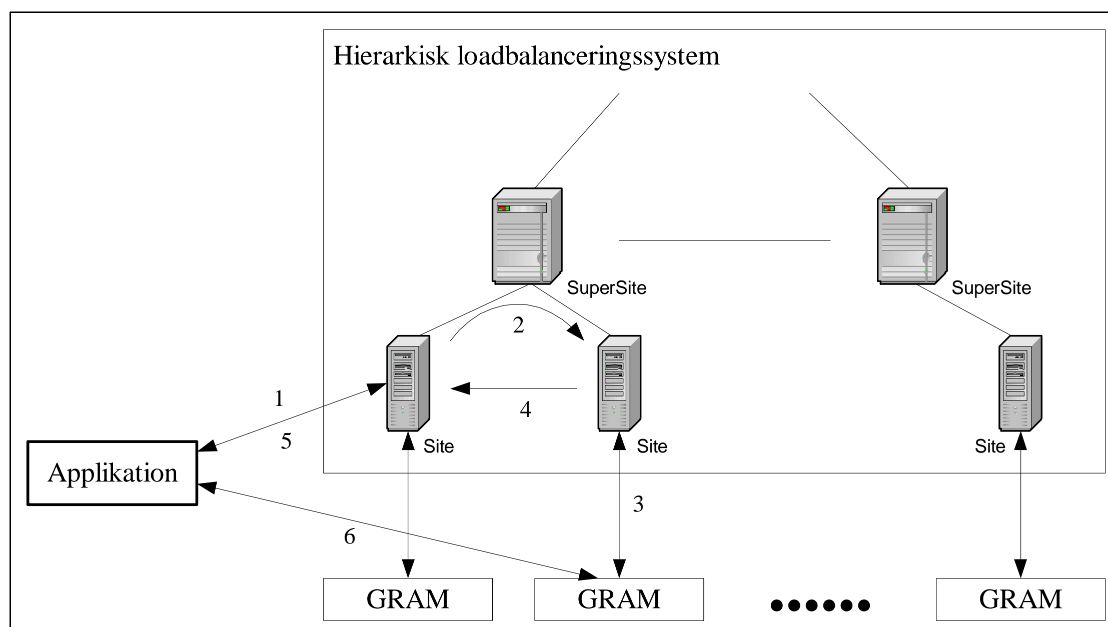
Når en applikation/broker laver en søgning i informationsservicen og når de prøver at lave en allokering af en opgave hos en GRAM, så bruger man et specielt designet sprog til at beskrive de ressourcekrav man har. Dette sprog hedder RSL - *Resource Specification Language* [19, 20].

Der findes på nuværende tidspunkt kun få brokere og co-allocatorer. De, som findes, er meget simple. Derfor indbygger applikationer typisk selv deres egen broker og co-allocator, som er designet specielt til applikationens formål.

En ressourceallokering i Globus foregår typisk på den måde, at en applikation foretager en søgning efter arbejdere i informationsservicen. Søgningen foretages ud fra et RSL udtryk. Søgningen returnerer oplysninger om en eller flere GRAM'er, der kan opfylde de ønskede ressourcekrav fra RSL-udtrykket. Herefter sendes der en RSL-forespørgsel til en udvalgt GRAM med et ønske om en udførelse af en opgave. Hvis GRAM'en accepterer, så allokeres opgaven til den ønskede arbejder. Under udførelsen af opgaven, kan applikationen kontakte GRAM'en og få informationer om status for udførelsen.

Vi ser vores hierarkiske loadbalanceringsmodel, som en avanceret informationsservice. En informationsservice hvor svaret fra en *søgning*, er en *reservation* og ikke blot ren information. En forespørgsel i Globus' informationsservice ændre ikke noget ved tilstanden af informationsservicen. Dvs. at en forespørgsel ikke har indflydelse på resultatet af en senere forespørgsel. I vores hierarkiske loadbalanceringsmodel, vil de enkelte forespørgsler blive registreret, og på den måde have indflydelse på hinanden.

På figur 6.2 ses vores forslag til en overordnet opbygning af Globus, hvor vores hierarkiske loadbalanceringssystem fungerer som informationsservice. På figuren er brokeren og co-allocatoren udeladt, da applikationen selv agerer sin egen broker og co-allocator.



Figur 6.2: Figuren viser den overordnede opbygning af Globus, når vores hierarkiske loadbalanceringssystem bruges som informationsservice. På figuren er udeladt både *broker* og *co-allocator*, da disse roller også kan varetages af applikationen selv.

Hver GRAM er forbundet til sit eget site i den hierarkiske struktur. Informationer om tilstanden af de arbejdere, som den enkelte GRAM administrerer, rapporteres af en GRAM Reporter til det tilknyttede site. Dette foregår på samme måde, som når informationen opdateres i Globus' oprindelige informationsservice.

Figur 6.2 illustrerer desuden hvordan en ressourceallokering foregår. Når en applikation foretager en søgning i det hierarkiske loadbalanceringssystem, sendes en RSL forespørgsel til et site (1). I et site konverteres RSL udtrykket til en opgavetupel, og der findes en arbejder til opgaven efter principperne beskrevet i kapitel 3 (2). På figur 6.2 allokeres der en arbejder til opgaven i et andet site, end det hvor opgaven blev stillet. Allokeringen i et site består i, at foretage en reservation hos sitets tilknyttede GRAM (3). Information om reservationen, en *reservationsbillet*, sendes herefter tilbage til det site, hvor opgaven blev stillet (4). Herfra returneres reservationsbilletten til applikationen (5). Resten af forløbet fungere på samme måde som i den oprindelige Gobus model, med den ene udvidelse, at reservationsbilletten sendes med i forespørgslen fra applikationen til GRAM'en (6).

Reservation

Integreringen af vores hierarkiske loadbalanceringsmodel i Globus giver anledning til én vigtig ændring i Globus. Vores model er afhængig af muligheden for at kunne foretage reservationer hos en GRAM. En løsning på dette reservationsproblem er allerede foreslået og implementeret i GARA (Globus Architecture for Reservation and Allocation) [9]. I GARA specificeres bl.a. de ændringer, der skal til for, at en GRAM kan supportere reservationer. En sådan modificering af GRAM'en vil kunne bruges ved integreringen af vores hierarkiske loadbalanceringsmodel i Globus. Modificeringen af GRAM'erne i GARA indeholder tilmed mere avancerede former for reservationer end dem, der er behov for i

vores model.

Med indførelsen af reservationer, så er der også behov for en enkelt yderligere udvidelse. Informationsopdateringerne fra en GRAM til et site skal synkroniseres med udstedelsen af reservationer. Alternativt vil et site ikke kende den præcise tilstand af GRAM'en.

Udvidelser til det hierarkiske loadbalanceringssystem

For at vores model kan integreres med Globus, skal der laves nogle få udvidelser af vores system:

RSL konverteringsmodul: Der skal udvikles et modul, der kan konvertere RSL forespørgsler fra applikationer til vores models interne repræsentation (opgavetupler). Det vil her ikke være muligt at understøtte den komplette RSL grammatik. Eksempelvis understøtter vores model ikke sammensatte ressourceforespørgsler, som f.eks. indeholder en *eller*-operator.

Informationsmodul: Der skal udvikles et modul, som kan modtage og konvertere de informationer om arbejdernes tilstand, som GRAM'en sender til de enkelte sites.

Koordineringsmodul: Modulet skal sørge for at modtage RSL-forespørgsler fra applikationer. Herefter konverteres RSL-forespørgslen til en tupel via RSL konverteringsmodulet. Tuplen afleveres til det lokale site, som sender den ud i systemet. Herefter venter modulet på en reservationsbillet, som reservationsmodulet afleverer. Når reservationsbilletten er modtaget, så afleveres den tilbage til applikationen.

Reservationsmodul: Dette modul virker i alle sites og skal sørge for, at lave reservationen hos den lokale GRAM, når sitet har en opgave der skal allokeres. Når reservationen er gennemført, så skal reservationsmodulet sende reservationsbilletten tilbage til det tilhørende koordineringsmodul.

Vi mener at vores hierarkiske loadbalanceringsmodel, som beskrevet ovenfor, rimeligt simpelt kan integreres i Globus. Naturligvis er vores model begrænset, i og med at vi kun supporterer uafhængige opgaver. Man vil derfor med integreringen ikke kunne opnå den fulde funktionalitet af Globus. Til gengæld mener vi, at vores model med indførelsen af global optimering vil kunne bidrage med et yderst interessant aspekt til Globus.

Kapitel 7

Konklusion

Med denne rapport har vi præsenteret en hierarkisk loadbalanceringsmodel, som introducerer global optimering i et (stort) ressourceallokeringssystem. Modellens grundidé er at udnytte eksisterende ressourcemanagere, som håndterer lokale optimeringer indenfor et site. Ovenpå disse bygges et hierarki af supersites, som hver især foretager en loadbalancering af deres undertræ.

Den endelige model (model 3) er fremkommet gennem en iterativ proces. I modellen begrænser vi os til at se på uafhængige opgaver. Vi har desuden valgt den afgrænsning, at en opgaves udførelsestid på en arbejder kun er afhængig af, hvor stort et CPU behov (CPU-enheder) opgaven skal bruge og hvor stort et CPU udbud (CPUenheder/s) arbejderen stiller til rådighed.

Modellen tager højde for, at opgaver har forskellige *ressourcekrav* og at arbejdere har forskellige *ressourceudbud*. Og endvidere at opgaver har forskelligt CPU behov og arbejdere har forskelligt CPU udbud.

I modellen introducerer vi begrebet en *opgavegruppe* om en gruppe af opgaver, som har næsten ens ressourcekrav. Modellen virker ved, at der laves loadbalanceringer indenfor de enkelte opgavegrupper. For ikke at skulle tage højde for alle opgavegrupper i alle hierarkiets knudepunkter, har vi indført begrebet en *aktiv* opgavegruppe. Hvert knudepunkt har et antal aktive opgavegrupper, som indikerer, at knudepunktet er med i en loadbalancering indenfor hver af disse opgavegrupper. En opgavegruppe er kun aktiv i den del af hierarkiet, hvor det er nødvendigt, at der laves en loadbalancering for at afhjælpe et sites overbelastning i opgavegruppen. Vi har ligeledes introduceret begrebet en *arbejdergruppe* om en gruppe af arbejdere, der kan løse opgaver fra de samme aktive opgavegrupper. Vi viser hvordan en *indordning* af arbejdergrupperne kan sikre, at der kun sendes relevant information om arbejdernes ressourcer rundt i systemet. For at bestemme belastningstilstanden indenfor de forskellige grupper har vi brugt skemalægningsprincipper, som tager højde for, at arbejdergrupper kan løse opgaver fra flere aktive opgavegrupper.

I projektforløbet har vi implementeret tre prototyper af de modeller vi har lavet. Gennem nogle forskellige forsøg med prototyperne, har vi vist, at modellerne opfører sig som forventet.

I kapitel 6 har vi vist, hvordan modellen med relativt få ændringer kan integreres med det eksisterende ressourceallokeringssystem Globus.

7.1 Vurdering af model

De forsøg vi har beskrevet i kapitel 5 illustrerer, at de overordnede loadbalanceringsprincipper fungerer. Forsøgene viser, at så længe hele det hierarkiske loadbalanceringssystem er overbelastet, så findes der ingen frie arbejder i systemet, som vil kunne løse opgaver fra de tungt belastede opgavegrupper. Forsøgene viser desuden, at modellen løser problemer med loadubalance og problemer med store forskelle i opgave- og arbejderstørrelser tilfredsstillende som beskrevet i teorien.

I afsnit 3.8 beskrev vi nogle mulige forbedringer af modellen. I afsnit 3.8.1 foreslog vi, at en bedre styring af de opgaver, der sendes ned gennem træet ved en balancering, vil kunne optimere modellen yderligere. Det illustreredes desuden, at denne mangel i den nuværende model kan medføre, at et knudepunkt kan miste dele af sin autonomitet. Vi mener helt klart, at dette problem bør løses i modellen og et løsningsforslag blev givet i afsnittet. Desværre blev vi for sent opmærksomme på problemstillingen, og har derfor grundet tidsmangel ikke implementeret denne løsning.

I afsnit 3.8.2 forklarede vi hvordan en bedre skemalægningsalgoritme med stor sandsynlighed vil kunne optimere loadbalanceringen yderligere. I afsnittet illustreredes ved et simpelt eksempel, hvordan den nuværende algoritme kan resultere i en løsning, der ikke er helt optimal. For at kunne bruge en *smartere* skemalægningsalgoritme, så skal den ovenfor beskrevne styring af opgaverne ned gennem træet implementeres. Vi mener dog, at det kan være yderst interessant at afprøve modellen med andre og mere intelligente skemalægningsalgoritmer.

7.2 Fremtidigt arbejde

Udover de ovenfor beskrevne udvidelser til modellen, så vil vi her kommenterer nogle muligheder for fremtidigt arbejde.

Afhængige opgaver: Vores model arbejder kun med uafhængige opgaver. Det vil helt givet være interessant at undersøge mulighederne for, hvordan der i modellen kan tages højde for forskellige typer af afhængige opgaver.

Optimering af opgaveflytninger: I den nuværende implementering af modellen, flyttes opgavebeskrivelserne enkeltvist rundt i systemet. Der vil være en øvre grænse for, hvor mange opgavebeskrivelser et knudepunkt kan modtage og behandle pr. sekund. I vores prototyper ligger grænsen på ca. 200 opgavebeskrivelser pr. sekund. I et globalt system kan det blive en flaskehals, da man snildt kan forestille sig en generel belastningsskævhed, som kræver flytning af store mængder opgaver.
En mulig løsning på problemet er at kunne samle opgavebeskrivelser i *opgaveklumper*. Herved vil knudepunkterne kunne flytte og behandle en klump med f.eks. 1000 opgavebeskrivelser på én gang. Størrelserne af opgaveklumperne kunne afhænge af, hvor grov belastningsskævheden er, samt hvor højt i hierarkiet man befinder sig.

Estimater af udførelsestid: Modellen afhænger af rimelige præcise estimater på udførelsestiderne for opgaverne. Hvis udførelsestiderne afviger mærkbart fra estimaterne, kan det give problemer for modellen. Opgaver vil dels kunne ligge for lang tid i kø i

de enkelte knudepunkter, og dels kunne blive blive ført *for sent* ned til frie arbejdere. Der bør laves en analyse af (1) den faktiske betydningen af afvigelser i de forventede udførelsestider for opgaverne og (2) hvordan modellen kan tage højde for, at der er afvigelser i estimaterne.

Bilag A

Formalisering af tupelspace i RAISE

A.1 Base Linda

```
scheme BaseLinda =
  class
    type
      TupleElemID = Text,
      Agent,
      Obj,
      TupleElemReal ==
        val(Real) | range(Real × Real) | wild,
      TupleElemString == val(Text) | wild,
      TupleElemBool == val(Bool) | wild,
      TupleElemAgent == val(Agent) | wild,
      TupleElemOther == val(Obj) | wild,
      TupleElem ==
        RealTE(TupleElemReal) |
        TextTE(TupleElemString) |
        BoolTE(TupleElemBool) |
        AgentTE(TupleElemAgent) |
        OtherTE(TupleElemOther),
      TupleElemMap = TupleElemID →m TupleElem,
      BaseTuple ::
        ReqElems : TupleElemMap   OptElems : TupleElemMap,
      Tuple = BaseTuple,
      AntiTuple = BaseTuple

    value
      Match : Tuple × AntiTuple → Bool,
      ElemMatch : TupleElem × TupleElem → Bool

    axiom
      [ Element_match ]
        ∀
          te1, te2 : TupleElem, tr1, tr2, tr3 : Real,
```

```
        t1, t2 : Text, a1, a2 : Agent, o1, o2 : Obj,
        b1, b2 : Bool
    •
        ElemMatch(te1, te2) ≡
          case (te1, te2) of
            (RealTE(val(tr1)), RealTE(val(tr2))) →
              tr1 = tr2,
            (RealTE(val(tr1)), RealTE(range(tr2, tr3))) →
              tr1 ≥ tr2 ∧ tr1 ≤ tr3,
            (RealTE(val(tr1)), RealTE(wild)) → true,
            (RealTE(_), RealTE(val(_))) →
              ElemMatch(te2, te1),
            (TextTE(val(t1)), TextTE(val(t2))) → t1 = t2,
            (TextTE(val(_)), TextTE(wild)) → true,
            (TextTE(_), TextTE(val(_))) →
              ElemMatch(te2, te1),
            (BoolTE(val(b1)), BoolTE(val(b2))) → b1 = b2,
            (BoolTE(val(_)), b_wild) → true,
            (BoolTE(_), BoolTE(val(_))) →
              ElemMatch(te2, te1),
            (AgentTE(val(a1)), AgentTE(val(a2))) →
              a1 = a2,
            (AgentTE(val(_)), AgentTE(wild)) → true,
            (AgentTE(_), AgentTE(val(_))) →
              ElemMatch(te2, te1),
            (OtherTE(val(o1)), OtherTE(val(o2))) →
              o1 = o2,
            (OtherTE(val(_)), OtherTE(wild)) → true,
            (OtherTE(_), OtherTE(val(_))) →
              ElemMatch(te2, te1),
            (_, _) → false
          end,
[ Match ]
    ∀
        t : Tuple, at : AntiTuple, tID1 : TupleElemID,
        tID2 : TupleElemID
    •
        Match(t, at) ≡
          let
            atelems = ReqElems(at) ∪ OptElems(at),
            telems = ReqElems(t) ∪ OptElems(t)
          in
            dom ReqElems(t) ⊆ dom atelems ∧
            dom ReqElems(at) ⊆ dom telems ∧
            (tID1 ∈ dom ReqElems(t) ⇒
               ElemMatch(ReqElems(t)(tID1), atelems(tID1))) ∧
            (tID2 ∈ dom ReqElems(at) ⇒
               ElemMatch(telems(tID2), ReqElems(at)(tID2)))
```

```
                  end
end
```

A.2 Datastorage

```
context: BaseLinda

scheme DATASTORAGE(BL : BaseLinda) =
  class
    type
      Timeout = Nat, /* Timeout in milliseconds */
      UniqueID,
      InternalAntiTuple = Timeout × BL.AntiTuple × Bool × UniqueID,
      InternalTuple = BL.Tuple × UniqueID,
      AntiTupleMatch == Nil | AT(InternalAntiTuple),
      TupleMatch == Nil | T(InternalTuple),
      InternalStorage

    channel GetUniqueID : UniqueID

    variable IS : InternalStorage

    value
      AddTuple : InternalTuple ~→ Unit,
      AddAntiTuple : InternalAntiTuple ~→ Unit,
      RemoveTuple : UniqueID ~→ Unit,
      RemoveAntiTuple : UniqueID ~→ Unit,
      GetAntiTupleMatch : BL.Tuple → AntiTupleMatch,
      GetTupleMatch : BL.AntiTuple → TupleMatch,
      System : Unit → Unit,
      InternalSystem : Unit → Unit,
      UniqueIDGenerator : Unit → Unit,
      GetAllTuples :
        InternalStorage → UniqueID →m InternalTuple,
      GetAllAntiTuples :
        InternalStorage → UniqueID →m InternalAntiTuple

    axiom
      [ System ]
        System() ≡ UniqueIDGenerator() ∥ InternalSystem(),
      [ UniqueIdGenerator ]
        UniqueIDGenerator() ≡
          local
            type UniqueDB

            value
              NoID : UniqueDB,
              GetNextID : UniqueDB → UniqueID × UniqueDB,
              is_ok : UniqueID × UniqueDB → Bool,
              insert : UniqueID × UniqueDB → UniqueDB,
```

```
                  sys : UniqueDB → out GetUniqueID Unit

               axiom
                  ∀ d : UniqueID • is_ok(d, NoID) ≡ true,
                  ∀ d, d2 : UniqueID, db : UniqueDB •
                     is_ok(d, insert(d2, db)) ≡
                        d ≠ d2 ∧ is_ok(d, db),
                  ∀ d : UniqueID, db : UniqueDB •
                     GetNextID(db) ≡
                        let d : UniqueID • is_ok(d, db) in
                           (d, insert(d, db))
                        end,
                  ∀ d : UniqueID, db : UniqueDB •
                     sys(db) ≡
                        let (d, newdb) = GetNextID(db) in
                           GetUniqueID!d ; sys(newdb)
                        end
            in sys(NoID) end,
      [ AddTuple ]
         ∀ id : UniqueID, t : BL.Tuple •
            AddTuple((t, id)) post
               id ∈ dom GetAllTuples(IS) ∧
               GetAllTuples(IS)(id) = (t, id)
            pre id ∉ dom GetAllTuples(IS),
      [ RemoveTuple ]
         ∀ id : UniqueID •
            RemoveTuple(id) post id ∉ dom GetAllTuples(IS)
            pre id ∈ dom GetAllTuples(IS),
      [ GetTupleMatch ]
         ∀ at : BL.AntiTuple •
            GetTupleMatch(at) ≡
               if
                  (∃ (t, id) : InternalTuple •
                      (t, id) ∈ rng GetAllTuples(IS) ∧
                      BL.Match(t, at))
               then
                  let
                     (t, id) : InternalTuple •
                        (t, id) ∈ rng GetAllTuples(IS) ∧
                        BL.Match(t, at)
                  in
                     T((t, id))
                  end
               else Nil
               end
end
```

A.3 Linda

```
context: DATASTORAGE, BaseLinda

scheme Linda2(DS : DATASTORAGE(BL), BL : BaseLinda) =
  class
    type RetTuple == Tup(BL.Tuple) | Nil

    object
      WAIT_OBJECTS[ i : DS.UniqueID ] :
        class channel c : RetTuple end

    channel
      OutQue : DS.InternalTuple,
      SystemInQue : DS.InternalAntiTuple,
      TimeOutChan : DS.UniqueID

    value
      Out : BL.Tuple → out OutQue in any Unit,
      In :
        BL.AntiTuple × DS.Timeout ~→
          out SystemInQue, TimeOutChan
          in {WAIT_OBJECTS[ i ].c | i : DS.UniqueID} RetTuple,
      Rd :
        BL.AntiTuple × DS.Timeout ~→
          out SystemInQue, TimeOutChan
          in {WAIT_OBJECTS[ i ].c | i : DS.UniqueID} RetTuple,
      System : Unit → in any out any write any Unit,
      InRdIntern :
        BL.AntiTuple × DS.Timeout × Bool ~→
          out SystemInQue, TimeOutChan
          in {WAIT_OBJECTS[ i ].c | i : DS.UniqueID} RetTuple

    axiom
      [ Out ]
        ∀ t : BL.Tuple •
          Out(t) ≡
            let myuniqueid = DS.GetUniqueID? in
              OutQue!(t, myuniqueid)
            end,
      [ InRdIntern ]
        ∀
          at : BL.AntiTuple, ti : DS.Timeout, slet : Bool
        •
          InRdIntern(at, ti, slet) ≡
            local
              variable returnVal : RetTuple
```

```
            value
              Is_timedout : DS.Timeout → Bool
          in
            let myuniqueid = DS.GetUniqueID? in
              SystemInQue!(ti, at, slet, myuniqueid) ;
              (let t = WAIT_OBJECTS[ myuniqueid ].c? in
                 returnVal := t
               end
               ⌷
               (if ti > 0
                 then
                   (if Is_timedout(ti)
                    then
                      ((let
                          t = WAIT_OBJECTS[ myuniqueid ].c?
                        in
                          returnVal := t
                        end
                        ⌷
                        TimeOutChan!(myuniqueid) ;
                        returnVal := Nil))
                    else stop
                    end)
                 else stop
                 end))
            end ;
            returnVal
          end,
    [ In ]
      ∀ at : BL.AntiTuple, ti : DS.Timeout •
        In(at, ti) ≡ InRdIntern(at, ti, true),
    [ Rd ]
      ∀ at : BL.AntiTuple, ti : DS.Timeout •
        Rd(at, ti) ≡ InRdIntern(at, ti, false),
    [ System ]
      System() ≡
        local
          channel SystemOutQue : DS.InternalTuple

          value
            OutQueManager : DS.InternalTuple* → Unit,
            Matcher :
              Unit → out any in any write any Unit

          axiom
            [ OutQueManager ]
              ∀ curbuf : DS.InternalTuple* •
```

```
        OutQueManager(curbuf) ≡
          let t = OutQue? in
            OutQueManager(curbuf ⌢ ⟨t⟩)
          end
          ⌷
          if curbuf ≠ ⟨⟩
          then
            let
              (t, tmptail) =
                (hd (curbuf), tl (curbuf))
            in
              SystemOutQue!t ;
              OutQueManager(tmptail)
            end
          else stop
          end,
    [ Matcher ]
      Matcher() ≡
        (let (t, id) = SystemOutQue? in
          local
            variable letsstop : Bool := false
          in
            do
              let
                retat = DS.GetAntiTupleMatch(t)
              in
                case retat of
                  DS.Nil →
                    letsstop := true ;
                    DS.AddTuple((t, id)),
                  DS.AT((ti, at, slet, uniqueid)) →
                    DS.RemoveAntiTuple(uniqueid) ;
                    WAIT_OBJECTS[ uniqueid ].c!
                    Tup(t) ;
                    if slet then letsstop := true
                    end
                end
              end
            until letsstop end
          end
        end
        ⌷
        let
          (ti, at, slet, uniqueid) = SystemInQue?
        in
          let rett = DS.GetTupleMatch(at) in
            case rett of
              DS.Nil →
```

```
                                        if
                                          ti = 0 /* Timeout=0 */
                                        then WAIT_OBJECTS[ uniqueid ].c!Nil
                                        else
                                          DS.AddAntiTuple(
                                             (ti, at, slet, uniqueid))
                                        end,
                                       DS.T(t, id) →
                                        WAIT_OBJECTS[ uniqueid ].c!Tup(t) ;
                                        if slet then DS.RemoveTuple(id)
                                        end
                                    end
                                 end
                              end
                              []
                              let uniqueid = TimeOutChan? in
                                 DS.RemoveAntiTuple(uniqueid)
                              end) ;
                             Matcher()
                in
                   OutQueManager(⟨⟩) ∥ Matcher() ∥ DS.System()
                end
end
```

Bilag B

Implementering

B.1 Model 1 - simpel load balancering

Model 1 er meget simpel. Implementeringen følger den teoretiske beskrivelse i afsnit 3.5. Da model 1 ikke tager højde for, at tiden går, så er det meget vigtigt, at der sendes information med en høj frekvens. Vi har derfor valgt at sende information én gang i sekundet.

Vi vil nedenfor vise hvordan arbejderne er blevet simuleret i denne model, samt hvilken type information der bruges i modellen, samt hvordan denne behandles. De resterende dele af systemet følger beskrivelserne i 4.6.1.

Arbejdere

I denne model henter arbejderne opgaver fra tupelspacet (ved hjælp af en antitupel) når de er frie. En arbejder ser således ud:

```
Arbejder() =
  while true
    AntiTupel arbejderantitupel = .......
    Tupel opgavetupel = myTS.in(arbejderantitupel)
    Opgave opg=opgave(opgavetupel)
    opgavelængde=getCPUTime(opg)
    sleep opgavelængde
```

Funktionen $getCPUTime(t)$ returnerer, udfra opgavetuplen, hvor lang tid arbejderen skal simulerer, at opgaven tager. $myTS$ er det tupelspace arbejderen er tilmeldt.

Information

I denne model er informationen, som sendes opad i træet, meget simpel. Informationen skal indeholde information om, hvor mange opgaver der ligger i kø i undertræet, samt hvor mange arbejdere der er. Når man sender en opgave nedad i træet, er det derfor vigtigt at vide, om denne opgave vil blive gemt i en kø til en arbejder, eller der er en

arbejder helt fri og derfor kan begynde at arbejde på opgaven med det samme. Dette har stor betydning, da beregningerne til loadbalanceringen bruger antallet af opgaver i kø og ikke antallet af opgaver hos arbejdere. Man burde også have information med om hvor mange opgaver der kan flyttes (da vi ikke flytter opgaver fra et site som ikke er tungt belastet), men dette gøres ikke (se fejlbeskrivelsen i afsnit 3.5.2).

Vi har derfor valgt, at information skal indeholde: Antal opgaver, antal arbejdere og antal frie arbejdere.

I et site genereres denne information simpelt:

```
Site::LavInformation() =
  info.AntalOpgaver = LokalOpgaveKø.antalOpgaver
  info.AntalArbejdere = AntalArbejdere
  info.AntalFrieArbejdere = TupelSpace.antalArbejderAntiTupler()
  return info
```

I et supersite skal informationen være en opsummering af børnenes information, samt indeholde information om opgaver, som ligger i den lokale kø (som beskrevet i ligning (3.5 - 3.8).

```
Supersite::LavInformation() =
  info.AntalOpgaver = LokalOpgaveKø.antalOpgaver
  info.AntalArbejdere = 0
  info.AntalFrieArbejdere = 0

  for each i in childs
    info.AntalArbejdere += child(i).AntalArbejdere
    info.AntalFrieArbejdere += child(i).AntalFrieArbejdere
    info.AntalOpgaver += child(i).AntalOpgaver

  return info
```

Minimer-funktionen gør i denne model intet. BehandlInformation-funktionen laver en lille udglatning, således at vi ikke fortæller frie arbejdere op, hvis vi har opgaver.

```
Information::BehandlInformation() =
  FlytOpgaver = AntalOpgaver - AntalFrieArbejdere
  if FlytOpgaver<0 then
    AntalFrieArbejdere = AntalFrieArbejdere - FlytOpgaver
    AntalOpgaver = 0
  else
    AntalOpgaver = AntalOpgaver - AntalFrieArbejdere
    AntalFrieArbejdere = 0
```

Vi indfører nu for informationen en hjælpefunktion og definerer herefter belastningstilstandefunktionerne som:

```
Information::getLoadFaktor() =
  if AntalFrieArbejdere>0 return 0.0
  if AntalArbejdere=0 return uendelig;
  return AntalOpgaver/AntalArbejdere;

Information::isLetBelastet() =
  return getLoadFaktor()<Xmin;

Information::isTungtBelastet() =
  return getLoadFaktor()>Xmax;
```

For informationen mangler vi nu kun at definere de to funktioner: AfleverOpgave og FjernOpgave. Disse er ekstremt simple, da informationen indeholder antallet af opgaver.

```
Information::AfleverOpgave(opgave) =
  if AntalFrieArbejdere > 0 then
    AntalFrieArbejdere = AntalFrieArbejdere - 1
  else
    AntalOpgaver = AntalOpgaver + 1

Information::FjernOpgave(opgave) =
  AntalOpgaver = AntalOpgaver - 1
```

B.2 Model 2 - udvidet load balancering

Information

Informationen i denne model er mere kompleks end i model 1. For en arbejder har vi information om dens $NextFreeTime$ og størrelsen af dens CPU (CPUenheder pr. sekund). For en opgave har vi *længden* af opgaven i CPUenheder.

I informationen er *Opgaver* og $Arbejdere$ mængder og hvert element i disse sæt, består udover informationen om opgaven eller arbejderen også et antal, således at man kan sige at der f.eks. er 40 opgaver á 53 CPUenheder. Informationen ser ud som:

$$\begin{aligned}
\mathit{Information} &= \langle \mathit{Arbejdere}, \mathit{Opgaver} \rangle \\
Arbejdere &= \{\langle X_1\ \mathit{stk}, C_1\ \mathit{CPUenheder/arb/s}\rangle, \\
&\qquad \langle X_2\ \mathit{stk}, C_2\ \mathit{CPUenheder/arb/s}\rangle, \ldots\} \\
\mathit{Opgaver} &= \{\langle Y_1\ \mathit{stk}, D_1\ \mathit{CPUenheder/opg}\rangle, \\
&\qquad \langle Y_2\ \mathit{stk}, D_2\ \mathit{CPUenheder/opg}\rangle, \ldots\}
\end{aligned}$$

Hvor X_n er antallet af arbejdere, og C_n betegner hvor god regnekraft arbejderne har hver især. Y_n er antallet af opgaver, og D_n beskriver hvor meget regnekraft hver af opgaverne skal bruge.

Vi bruger i det følgende ˆ-operatoren som foreningsmængdeoperator.

I et site genereres informationen ved:

```
Site::LavInformation() =
  Opgaver = < >
  for each opg in LokalOpgaveKø.opgaver
    Opgaver = Opgaver ^ <1,opg.cpu>

  Arbejdere = < >
  for each arb in LokalRessourceManager.HentArbejderInformation()
    Arbejdere = Arbejdere ^ <(1,arb.nextFreeTime, arb.cpu)>

  info.Arbejdere = Arbejdere
  info.Opgaver = Opgaver
  return info
```

Og i et supersite genereres informationen ved:

```
Supersite::LavInformation() =
  Opgaver = < >
  Arbejdere = < >

  for each opg in LokalOpgaveKø.opgaver
    Opgaver = Opgaver ^ <1,opg.cpu>

  for each subsiteinfo in subsiteinfos
    Opgaver = Opgaver ^ subsiteinfo.Opgaver
    tmpArbejdere = timeChange(subsiteinfo.Arbejdere)
    Arbejdere = Arbejdere ^ tmpArbejdere

  info.Arbejdere = Arbejdere
  info.Opgaver = Opgaver
  return info
```

BehandlInformation-funktionen kalder skemalægningsproceduren *LaegSkema* med parameteren X_{max}. *LaegSkema*-funktionen lægger skemaet med algoritmen OLB og lægger kun opgaver på arbejdere, som har en $NextFreeTime < X_{max}$. Algoritmen er implementeret efter følgende principper:

```
information::LaegSkema(threshold) =
  ' Opdel arbejdere i to sæt. Et sæt med dem som ikke er fri før
  ' threshold og et med dem som er.
  FærdigeArbejdere = SubsetOver(Arbejdere,threshold)
  RestArbejdere = SubsetUnder(Arbejdere,threshold)

  ' Sorterer resterende arbejdere stigende efter NextFreeTime
  RestArbejdere = Sorter(RestArbejdere)

  ' Sorterer opgaverne stigende efter CPU forbrug
  RestOpgaver=Sorter(Opgaver)
```

```
while antal(RestOpgaver)>0 and antal(RestArbejdere)>0
  arb = fjernførste(RestArbejdere)
  opg = fjernførste(RestOpgaver)

  ' Tag højde for at arbejderne er helt fri. Så kan de først starte
  ' opgaver nu og ikke i fortiden.
  if arb.nextfreetime < nu then
    arb.nextfreetime = nu

  nyarbejderNFT=arb.nextfreetime+opg.cpu/arb.cpu

  ' hvor mange arbejdere og opgaver er der.
  if arb.antal=opg.antal then
    ' Lige mange arbejdere og opgaver, flyt alle
    ' arbejder til ny nextfreetime
    arb.nextfreetime = nyarbejderNFT
    if nyarbejderNFT < threshold then
      RestArbejdere.addsorteret(arb)
    else
      FærdigeArbejdere.add(arb)
  else if arb.antal>opg.antal then
    ' Flere arbejdere end opgaver. Opsplit arbejderne i to dele
    nyarb.antal = opg.antal
    nyarb.nextfreetime = nyarbejderNFT
    nyarb.cpu = arb.cpu
    arb.antal = arb.antal - nyarb.antal

    ' Adder de arbejdere der ikke bruges
    ' (de er stadig under threshold)
    RestArbejdere.addsorteret(arb)

    ' Adder de nye arbejdere
    if nyarbejderNFT < threshold then
      RestArbejdere.addsorteret(nyarb)
    else
      FærdigeArbejdere.add(nyarb)
  else
    ' Flere opgaver end arbejdere.
    ' Læg en opgave til hver arbejder og
    ' læg resten af opgaverne tilbage.
    opg.antal = opg.antal - arb.antal
    RestOpgaver.addsorteret(opg)

    arb.nextfreetime = nyarbejderNFT
    if nyarbejderNFT < threshold then
      RestArbejdere.addsorteret(arb)
    else
```

```
      FærdigeArbejdere.add(arb)

  Opgaver = RestOpgaver
  Arbejdere = FærdigeArbejdere ^ RestArbejdere
```

Da informationen i supersites er minimeret, vil der være mange opgaver og arbejdere som ser præcist ens ud (de er blevet minimeret og samlet) og vi har derfor indført en optimering af skemalægningsrutinen, således at vi schedulerer flere opgaver til flere arbejdere ad gangen.

Minimer-funktionen virker i denne model som beskrevet i 3.6.2 og vist på figurene 3.6 og 3.7.

```
information::Minimer() =
  MinimerArbejdere()
  MinimerOpgaver()

information::MinimerArbejdere() =
  NyeArbejdere = < >

  ' Minimer først NextFreeTime
  for each intervalNFT in intervallerNextFreeTime
    ' Tag de arbejdere som har NextFreeTime
    ' mellem de to intervalendepunkter
    tmpArbejdere = Arbejdere.subsetNextFreeTime(intervalNFT.min,
    intervalNFT.max)
    if tmpArbejdere.antal>0 then
      gennemsnitNextFreeTime = GennemsnitNextFreeTime(tmpArbejdere)
      ' Indenfor dette subset, inddel arbejderne
      ' i CPU størrelse intervallerne
      for each intervalCPU in intervallerCPUarbejdere
        tmp2Arbejdere = tmpArbejdere.subsetCPU(intervalCPU.min,
                                                    intervalCPU.max)
        if tmp2Arbejdere.antal>0
          antal = tmp2Arbejdere.antal
          cpusize = GennemsnitCPU(tmp2Arbejdere)
          NyeArbejdere = NyeArbejdere ^
                              <antal,gennemsnitNextFreeTime,cpusize>
  Arbejdere=NyeArbejdere

information::MinimerOpgaver() =
  NyeOpgaver = < >

  ' Minimer opgavernes efter CPU størrelse
  for each intervalCPU in intervallerCPUopgaver
    tmpOpgaver = Opgaver.subsetCPU(intervalCPU.min,intervalCPU.max)
    if tmpOpgaver.antal>0
      gennemsnitCPU = GennemsnitCPU(tmpOpgaver)
      NyeOpgaver = NyeOpgaver ^ <tmpOpgaver.antal,gennemsnitCPU>
```

```
Opgaver = NyeOpgaver
```

Belastningsfunktionerne defineres nu ved:

```
information::isLetBelastet() =
  arb = findFørsteFrieArbejder(Arbejdere)
  return arb.NextFreeTime < Xmin

information::isTungtBelastet() =
  LaegSkema(Xmax)
  return antal(Opgaver) > 0
```

Funktionen *findFørsteFrieArbejder* returnerer tidspunktet for, hvornår den første arbejder forventes fri. Funktionen tager højde for den tid der er gået siden informationen blev genereret. I *isTungtBelastet* starter vi med at lægge skema. Dette gøres for at tage højde for at der er gået noget tid, således at der måske er nogle opgaver, som nu ikke mere kan flyttes.

For informationen mangler vi nu kun at definere de to funktioner: AfleverOpgave og FjernOpgave. Disse er mere komplicerede end i model 1.

Aflever opgave lægger opgaven ned på den arbejder som har den mindste *NextFreeTime*.

```
Information::AfleverOpgave(opgave) =
  arb = fjernMindsteArbejderNFT(Arbejdere)

  ’ Tag højde for at arbejderen er helt fri. Så kan den først starte
  ’ opgaven nu og ikke i fortiden
  if arb.nextfreetime < nu then
    arb.nextfreetime = nu

  arb.nextfreetime = arb.nextfreetime + opgave.cpu/arb.cpu

  Arbejdere = Arbejdere ^ <arb>
```

FjernOpgave funktionen er straks meget mere kompliceret. For det første er opgaveinformationen minimeret, således at der i informationen ikke findes en opgave der ligner eksakt den opgave vi ønsker at fjerne. Vi har valgt en rimelig simpel løsning, som i enkelte tilfælde ødelægger informationen en smule. Da informationen er minimeret, så er der typisk rigtig mange opgaver samlet i hvert enkelt opgaveinformationselement. I vores løsning finder vi den opgaveklump som minder mest om opgaven, og så fjerner vi én opgave

```
Information::FjernOpgave(opgave) =
  opginf = FjernTættesteOpgaver(Opgaver)
  if opginf.antal=1 then
    ’ Gør intet
  else
    ’ Beregn nyt CPU gennemsnit, når vi har fjernet opgaven
    nytotalcpu = opginf.antal * opginf.cpu - opgave.cpulength
```

```
    opginf.antal = opginf.antal - 1
    opginf.cpu = nytotalcpu/opginf.antal
    ' Læg opgaveklumpen tilbage
    Opgaver = Opgaver ^ <opginf>
```

B.3 Model 3 - kompleks loadbalancering

Denne model afviger på visse punkter en del fra den generelle model beskrevet i afsnit 4.6.1. I hvert knudepunkt er der, for hver aktive opgavegruppe og arbejdergruppe en proces, som sørger for at hente opgaver fra forælderen henholdsvis børnene. Hver gang der genereres ny information i knudepunktet, så undersøges det, om der skal oprettes nye processer, og om der er nogle der skal nedlægges. Dette gøres også når knudepunktet modtager information fra dets forælder om at det skal åbne eller lukke en aktiv opgavegrupper.

Informationen i denne model er meget mere kompleks end i model 2. For hvert unikke opgavekrav og hvert unikke arbejderudbud haves en information svarende til den for opgaverne henholdsvis arbejderne i model 2.

$$
\begin{aligned}
OpgaveInfo &= \{O_1, O_2, O_3, \ldots\} \\
O_i &= \{\langle Ressourcekrav_{1,1}, KravVaerdi_{1,1}\rangle, \langle Ressourcekrav_{1,2}, KravVaerdi_{1,2}\rangle\} \mapsto \\
&\qquad \{\langle X\ stk, S\ CPUenheder/opg\rangle, \ldots\} \\
ArbejderInfo &= \{A_1, A_2, A_3, \ldots\} \\
A_i &= \{\langle Ressourceudbud_{1,1}, UdbudVaerdi_{1,1}\rangle, \ldots\} \mapsto \\
&\qquad \{\langle Y\ stk, Z\ CPUenheder/arb/s, F\ s\rangle, \ldots\}
\end{aligned}
$$

Hvor X er antallet af opgaver og S er det forventede CPU forbrug hver af opgaverne skal bruge. Y er antallet af arbejdere informationen dækker over. Z er CPU udbud for denne information og F er den forventede næste fri tid.

Vi bruger igen ˆ-operatoren som foreningsmængdeoperator.

I et site genereres informationen ved:

```
Site::LavInformation() =
  Opgaver = < >
  for each opg in LokalOpgaveKø.opgaver
    Opgaver.add(krav(opg), 1, opg.cpulength)

  Arbejdere = < >
  for each arb in LokalRessourceManager.HentArbejderInformation()
    Arbejdere.add(udbud(arb),1,arb.nextFreeTime, arb.cpusize)

  info.Arbejdere = Arbejdere
  info.Opgaver = Opgaver
  return info
```

Hvor *add* for både opgavemængden og arbejdermængden undersøger om der eksisterer et element med med det aktuelle ressourcekrav henholdsvis ressourceudbud. Hvis der ikke gør, så oprettes et nyt. Herefter adderes den nye information som et nyt element i dette element.

Og i et supersite genereres informationen ved:

```
Supersite::LavInformation() =
  Opgaver = < >
  Arbejdere = < >

  for each opg in LokalOpgaveKø.opgaver
    Opgaver.add(krav(opg), 1, opg.cpulength)

  for each subsiteinfo in subsiteinfos
    Opgaver.addAll(subsiteinfo.Opgaver)
    tmpArbejdere = timeChange(subsiteinfo.Arbejdere)
    Arbejdere.addAll(tmpArbejdere)

  info.Arbejdere = Arbejdere
  info.Opgaver = Opgaver
  return info
```

Hvor *addAll* virker ved, at *add* kaldes for hvert element i informationen.

$BehandlInfo$-funktionen i denne model er mere kompleks end i de andre modeller.

```
information::behandlinfo() =
  LaegSkema(Xmax)

  ' Grupper opgaver i grupper hvor opgaver minder om hinanden
  GrupperOpgaver()

  ' Find eventuelle opgavegrupper som ikke er aktive til forælderen
  OverbelastedeOpgaveGrupper=findIkketommeOpgaveGrupper()
  ikkeaktivegrupper=subset(OverbelastedeOpgaveGrupper,
                                aktiveopgavegrupperforaelder)

  ' Hvis der var nogle sådanne, så send en
  ' aktiveringsforespørgsel til forælderen
  SendAktiverOpgaveGrupper(ikkeaktivegrupper)

  ' Informationen må kun indeholde oplysninger om de aktive
  ' opgavegrupper
  filtrerOpgaveGrupper(aktiveopgavegrupperforaelder)

  ' Indordn arbejderinformationen efter opgavegrupperne.
  IndordnArbejdere()
```

```
  ' Minimer evt antallet af arbejdergrupper, hvis der er for mange
  MinizeArbejdere()
```

LaegSkema-funktionen lægger skemaet med nogenlunde samme algoritme som i model 2. Men med indførelsen af opgavegrupper og arbejdergrupper bliver skemalægning mere besværlig. Der tages højde for at opgaverne i opgavegrupperne kun kan løses af nogle arbejdergrupper. Algoritmen er implementeret efter følgende principper:

```
information::LaegSkema(threshold) =

  ' Sorteret liste med opgavegrupper, hvor elementerne er sorteret
  ' stigende efter opgavegruppernes mindste opgave. Listen
  ' indeholder de opgavegrupper som skal scheduleres
  RestOpgaveGrupper = lavSorteretOpgavegruppeListe(OpgaveInfo)

  ' Denne liste indeholder opgavegrupper, som der ikke kan
  ' scheduleres flere opgaver fra.
  FærdigeOpgaveGrupper = < >

  while antal(RestOpgaveGrupper)>0
    ' Tag første element ud.
    opggrp = fjernførste(RestOpgaveGrupper)
    if antal(opggrp)=0 then
      FærdigeOpgaveGrupper.add(opggrp)
    else
      ' Find den tidligst frie arbejdergruppe som kan løse opgaven
      ' og er fri før threshold
      valgtearbgrp=null
      valgteminnft=threshold
      for each arbgrp in ArbejderInfo
        if bedstenextfreetime(arbgrp)<valgteminnft and
                match(opggrp,arbgrp) then
          valgtearbgrp=arbgrp
          valgteminnft=bedstenextfreetime(arbgrp)

      if valgtearbgrp=null then
        ' Der var ingen der kunne løse opgaven
        FærdigeOpgaveGrupper.add(opggrp)
      else

        arb = fjernTidligstFrie(valgtearbgrp)
        opg = fjernmindste(opggrp)

        ' Tag højde for at arbejderne er helt fri. Så kan de
        ' først starte opgaver nu og ikke i fortiden.
        if arb.nextfreetime < nu then
          arb.nextfreetime = nu
```

```
        nyarbejderNFT=arb.nextfreetime+arb.cpu*opg.cpu

        ' hvor mange arbejdere og opgaver er der.
        if arb.antal=opg.antal then
          ' Lige mange arbejdere og opgaver, flyt alle
          ' arbejder til ny nextfreetime
          arb.nextfreetime = nyarbejderNFT
          valgtearbgrp.add(arb)
        else if arb.antal>opg.antal then
          ' Flere arbejdere end opgaver. Opsplit arbejderne i to dele
          nyarb.antal = opg.antal
          nyarb.nextfreetime = nyarbejderNFT
          nyarb.cpu = arb.cpu
          arb.antal = arb.antal - nyarb.antal

          ' Adder de arbejdere der ikke bruges
          ' (de er stadig under threshold)
          valgtearbgrp.add(arb)

          ' Adder de nye arbejdere
          valgtearbgrp.add(nyarb)
        else
          ' Flere opgaver end arbejdere.
          ' Læg en opgave til hver arbejder og
          ' læg resten af opgaverne tilbage.
          opg.antal = opg.antal - arb.antal
          opggrp.add(opg)

          arb.nextfreetime = nyarbejderNFT
          valgtearbgrp.add(arb)

        ' Hvis der stadig er opgaver tilbage i opgavegruppen
        ' så skal disse stadig scheduleres.
        if antal(opggrp)>0 then
          RestOpgaveGrupper.addsorteret(opggrp)
        else
          FærdigeOpgaveGrupper.add(opggrp)


  ' Vi er nu færdige
  OpgaveInfo = FærdigeOpgaveGrupper
```

Den ovenfor viste pseudokode er i implementeringen optimeret en hel del. Ligesom i model 2, så vil skemalægningen være ret hurtig, da informationen i supersites er minimeret fra børnene.

Minimer-funktionen virker i denne model næsten som i model 2. Dog minimeres der indenfor hver enkelt arbejder- og opgavegruppe.

```
information::Minimer() =
  for each opggrp in OpgaveInfo
    opggrp.MinimerOpgaver()
  for each arbgrp in ArbejderInfo
    arbgrp.MinimerArbejdere()
```

Hvor *MinimerOpgaver*() og *MinimerArbejdere*() er som i model 2.

I denne model definerer vi belastningstilstande indenfor opgavegrupper og arbejdergrupper. Vi definerer dog kun om en opgavegruppe er let belastet eller ej, og kun om en arbejdergruppe er tungt belastet eller ej.

```
information::isLetBelastet(ArbGruppe) =
  arb = findFørsteFrieArbejder(Arbejdere,OpgGruppe)
  return arb.NextFreeTime < Xmin

information::isTungtBelastet(OpgGruppe) =
  return antal(OpgaveGrupper(OpgGruppe)) > 0
```

Funktionen *findFørsteFrieArbejder* returnerer tidspunktet for, hvornår den første arbejder forventes fri, som kan løse opgaver tilhørende *OpgGrupper*.

Vi redefinerer nu funktionerne som henter opgaver fra børn og fra forælderen. For hver aktive opgavegruppe i et supersite haves en proces som henter opgaver fra børnene. Én sådan proces er givet ved:

```
Supersite::HentOpgaverFraBørn(OpgGrp) =
  while true
    if AntalLokaleOpgaver(OpgGrp)<MAXBUFFERSIZE then
      barnid=FindTungtBelastetBarn(OpgGrp)
      if barnid then
        TupelSpace barnTS=tupelspace(barnid)
        AntiTupel arbejderantitupel = ....krav(OpgGrp)....
        Tupel opgavetupel = barnTS.in(arbejderantitupel)
        myTS.out(opgavetupel)
        Opgave opg=opgave(opgavetupel)
        subsiteinfos(barnid).fjernOpgave(opg)
```

Ligeledes skal der være en proces for hver arbejdergruppe som henter opgaver fra forælderknudepunktet. For et supersite er én sådan process givet ved:

```
SuperSite::HentOpgaverFraForælder(ArbGrp) =
  while true
    if sidsteLokalInfo.isLetBelastet(ArbGrp) then
      AntiTupel arbejderantitupel = ....udbud(ArbGrp)....
      Tuple opgavetupel = parentTS.in(arbejderantitupel)
      if opgavetupel!=null then
        Opgave opg=opgave(opgavetupel)
        sidsteLokalInfo.AfleverOpgave(opg)
        myTS.out(opgavetupel);
```

Og for et site ved:

```
Site::HentOpgaverFraForælder(ArbGrp) =
  while true
    if sidsteLokalInfo.isLetBelastet(ArbGrp) then
      AntiTupel arbejderantitupel = ....udbud(ArbGrp)....
      Tupel opgavetupel = parentTS.in(arbejderantitupel)
      if opgavetupel!=null then
        Opgave opg=opgave(opgavetuple)
        sidsteLokalInfo.laegOpgavePåArbejder(opg)
        lokalressourcemanager.afleveropgave(opg)
```

For informationen mangler vi nu kun at definere de to funktioner: AfleverOpgave og FjernOpgave. Disse er meget mere komplicerede end i model 2.

Aflever opgave lægger opgaven ned på den arbejder som har den mindste $NextFreeTime$ og som kan løse opgaven.

```
Information::AfleverOpgave(opgave) =
  arbgrp = findMindsteArbejdergrupperNFT(ArbejderInfo,opgave)

  arb = arbgrp.fjernMindsteArbejderNFT()

  ' Tag højde for at arbejderen er helt fri. Så kan den først starte
  ' opgaven nu og ikke i fortiden
  if arb.nextfreetime < nu then
    arb.nextfreetime = nu

  arb.nextfreetime = arb.nextfreetime + arb.cpu * opgave.cpu

  arbgrp.add(arb)
```

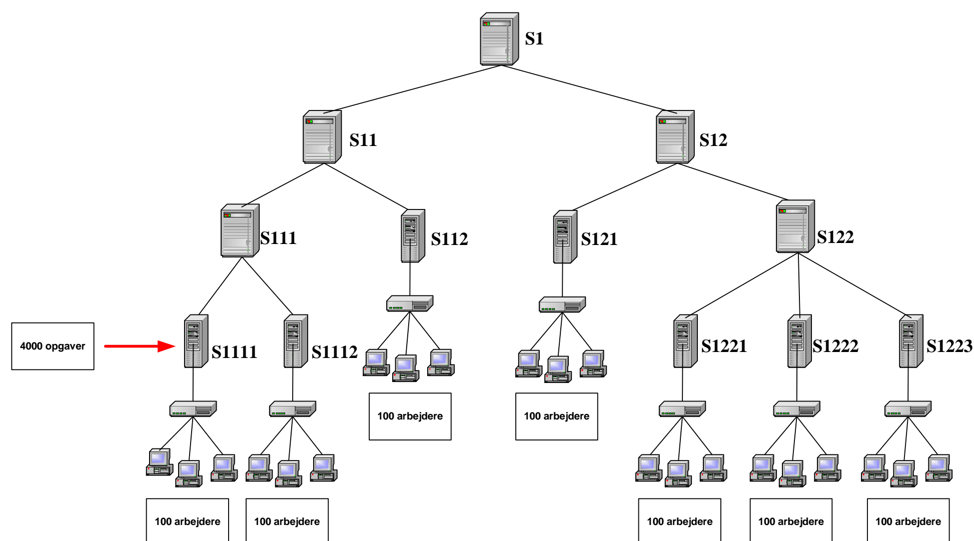
FjernOpgave funktionen minder meget om den i model 2, men hvor vi her først finder den gruppe opgaven tilhører.

```
Information::FjernOpgave(opgave) =
  opggrp = findOpgaveGruppe(opgave)
  opginf = opggrp.FjernTættesteOpgaver(Opgave)
  if opginf.antal=1 then
    ' Gør intet
  else
    ' Beregn nyt CPU gennemsnit, når vi har fjernet opgaven
    nytotalcpu = opginf.antal * opginf.cpu - opgave.cpulength
    opginf.antal = opginf.antal - 1
    opginf.cpu = nytotalcpu/opginf.antal
    ' Læg opgaveklumpen tilbage
    opggrp.add(opginf)
```

Bilag C

Forsøgsresultater

C.1 Forsøg 1-1



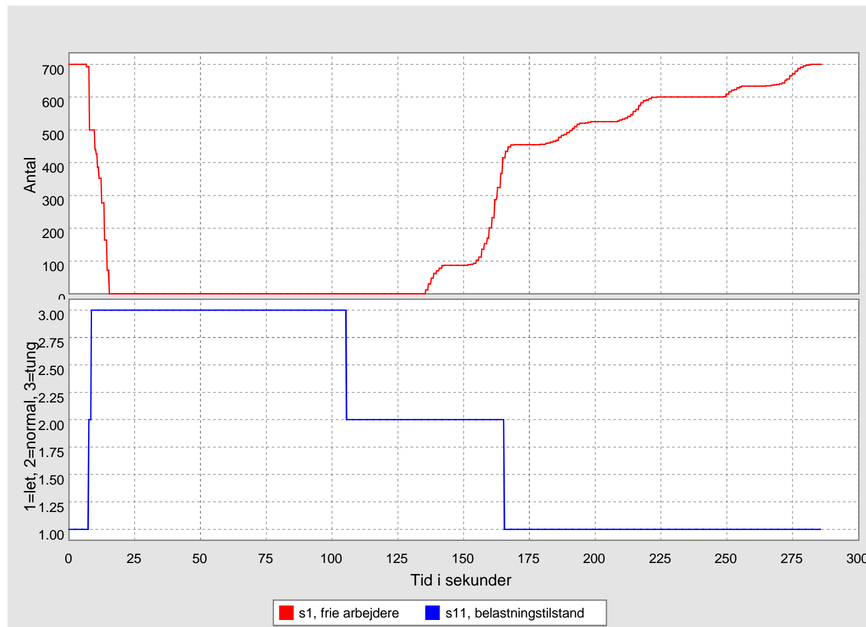
Figur C.1: Figuren viser det system som er brugt ved forsøg 1-1. Det ses bl.a. hvor mange arbejdere der er i de enkelte sites, og hvor der stilles opgaver.

Opgavetype	Ressourcekrav	Ressourcekrav for opgavegruppe	Auto id
1	RAM=32	RAM=32	100990
2	RAM=90	RAM=128	161278
3	DISK=200	RAM=256	2259357
4	RAM=150, DISK=500	RAM=256, DISK=512	2661787
5	RAM=250	RAM=256	241662
6	RAM=32, DISK=200	RAM=32, DISK=256	2360347
7	DISK=500	DISK=512	2420125
8	DISK=3000	DISK=4096	4670877
9	RAM=32, DISK=2000, OS=Linux	RAM=32, DISK=2048, OS="Linux"	76913363
10	RAM=32, DATABASE="climateDB"	RAM=32, DATABASE="climateDB"	-1686954401

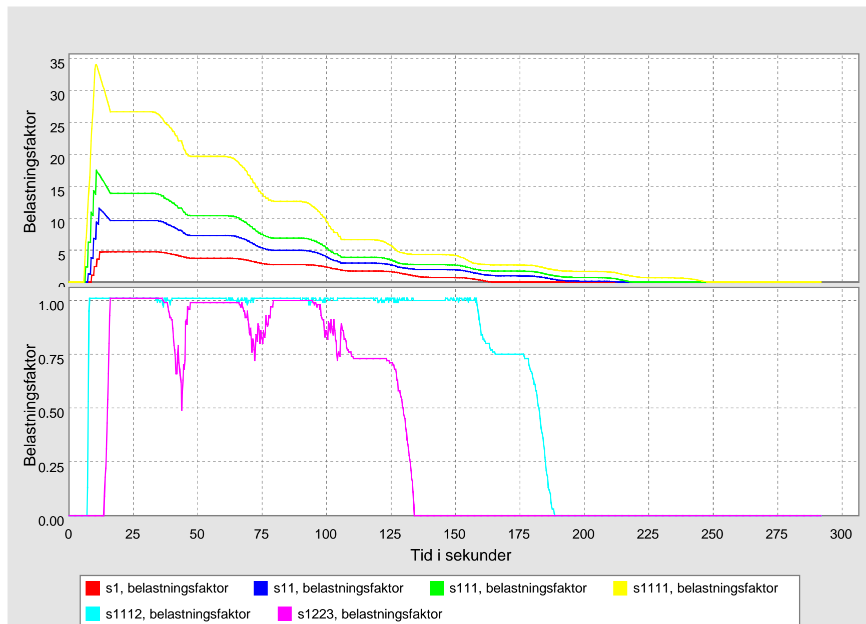
Tabel C.1: Tabellen viser de opgavetyper vi bruger under forsøgene. I tabellen ses både ressourcekravene for opgaverne, samt ressourcekravne for de tilhørende opgavegrupper.

Arbejdertype	Ressourceudbud
1	RAM=100
2	RAM=300
3	RAM=500
4	RAM=200, DISK=5000
5	RAM=250, DISK=1000
6	RAM=800, DISK=300
7	DISK=2000
8	DISK=8000
9	RAM=100, DISK=2000
10	RAM=200, DISK=1000, OS="Linux"
11	RAM=200, DISK=1000, OS="Windows"
12	RAM=500, DISK=1000, OS="Linux"
13	RAM=100, DISK=5000, OS="Linux"
14	RAM=800, DISK=1000, OS="Windows"
15	RAM=100, OS="Windows"
16	RAM=100, OS="Linux"
17	RAM=500, OS="Linux"
18	DISK=1000, OS="Linux"
19	DISK=1000, OS="Windows"
20	RAM=200, DISK=2000, OS="Linux", DATABASE="climateDB"

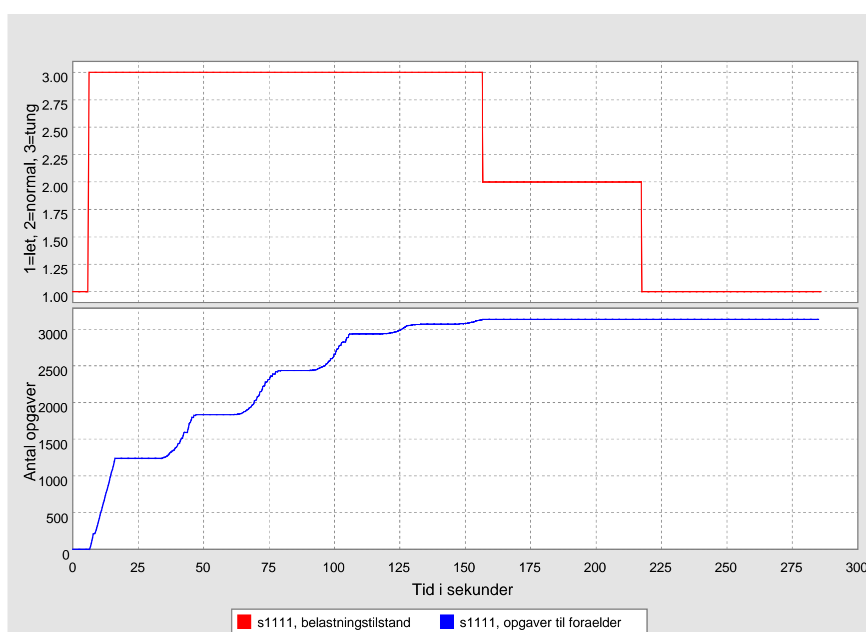
Tabel C.2: Tabellen viser de arbejdertyper vi bruger under forsøgene. I tabellen ses ressource udbudet for de enkelte arbejdertyper.



Figur C.2: Øverst viser figuren det totale antal frie arbejdere set fra *S1* (dvs. i hele systemet), som funktion af tiden. Nederst vises belastningstilstanden for venstre side af træet (*S11*), dvs. belastningstilstanden for den side af træet hvor der stilles opgaver. Det ses at i den tid hvor *S11* er tungt belastet, er der ingen frie arbejdere i hele systemet. (Med undtagelse af starten af forsøget, da det tager lidt tid at få flyttet opgaver til alle frie arbejdere.)

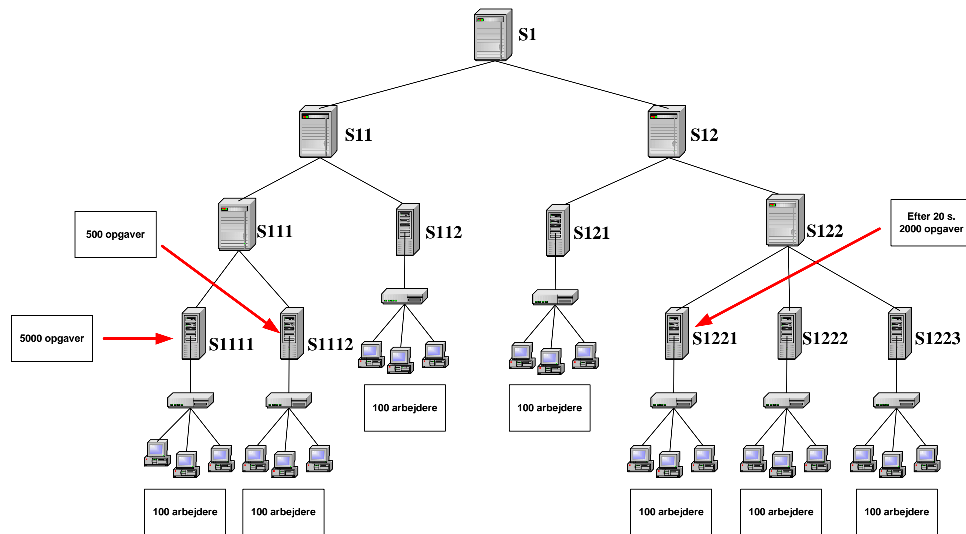


Figur C.3: Øverst viser figuren belastningsfaktorerne for det site hvor der stilles opgaver (*S1111*) og dennes forfædre (*S111*, *S11* og *S1*). Det ses, som forventet, at belastningsfaktoren falder jo tættere man kommer på toppen af træet. Nederst ses belastningsfaktorerne for to sites, hvor der ikke stilles opgaver (*S1112*, og *S1223*). Det ses at de to sites prøver at opretholde en belasntning på $X_{min} = 1.0$.

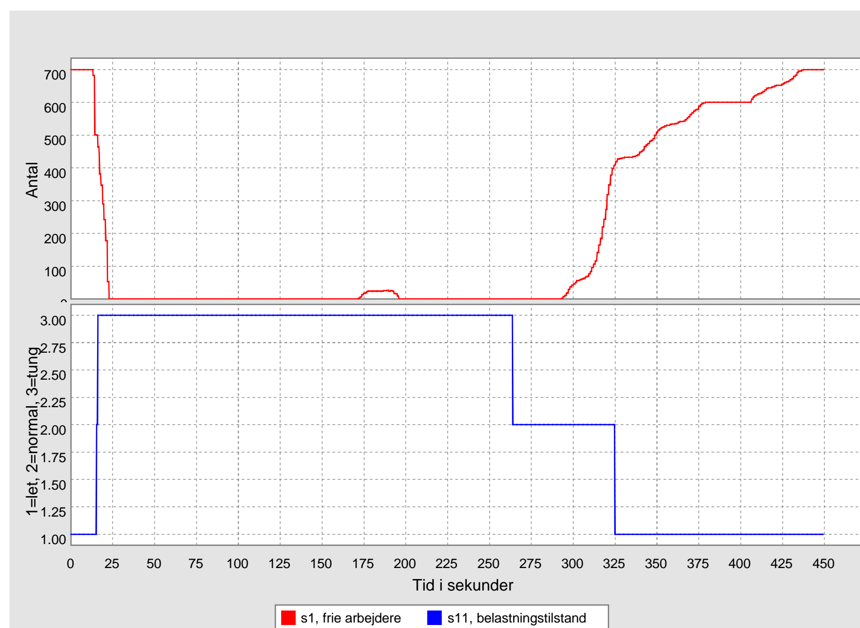


Figur C.4: Øverst viser figuren belastningstilstanden for det site hvor alle opgaverne stilles (*S1111*) og nederst vises antallet af opgaver, som *S1111*'s forælder (*S111*) trækker op fra *S1111*. Det ses at *S111* kun trækker opgaver op fra *S1111*, så længe dette site er tungt belastet.

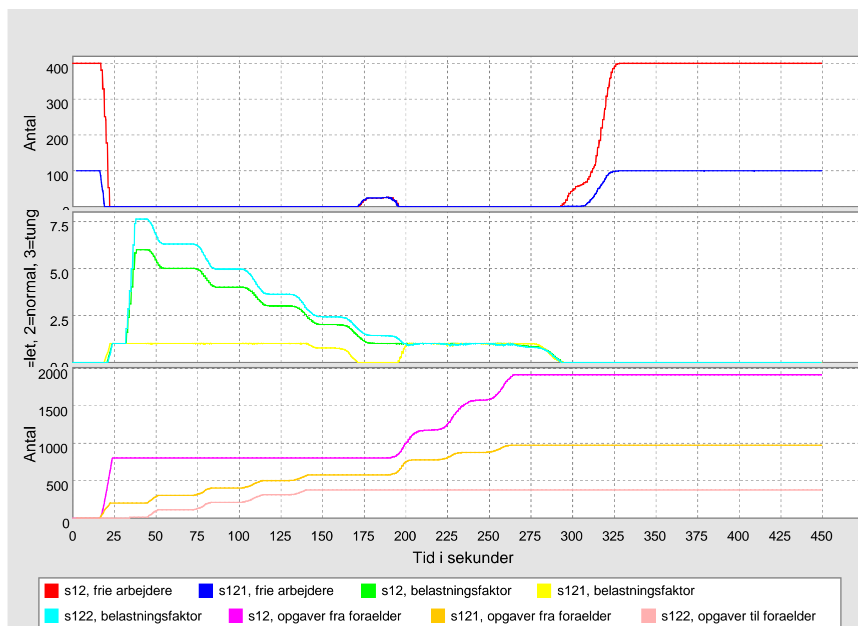
C.2 Forsøg 1-2



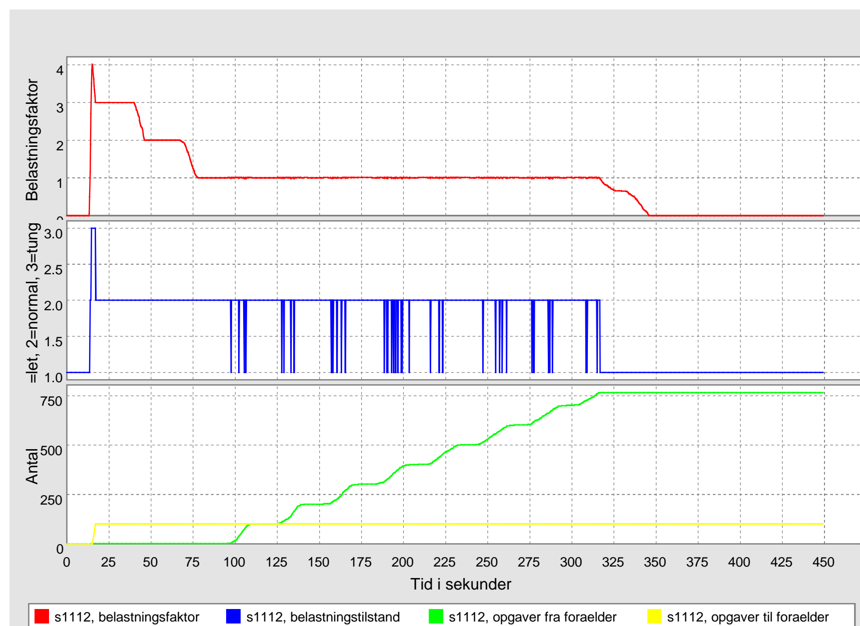
Figur C.5: Figuren viser det system som er brugt ved forsøg 1-2.



Figur C.6: Figuren viser, som på figur C.2, øverst antallet af frie arbejdere i hele systemet, og nederst belastningstilstanden for *S11*. Igen ses, at alle arbejdere stort set er i brug i hele det tidsforløb, hvor *S11* er tungt belastet. Det bemærkes, at der her er et lille tidsrum, hvor der er frie arbejdere i systemet, samtidigt med at *S11* er tungt belastet.

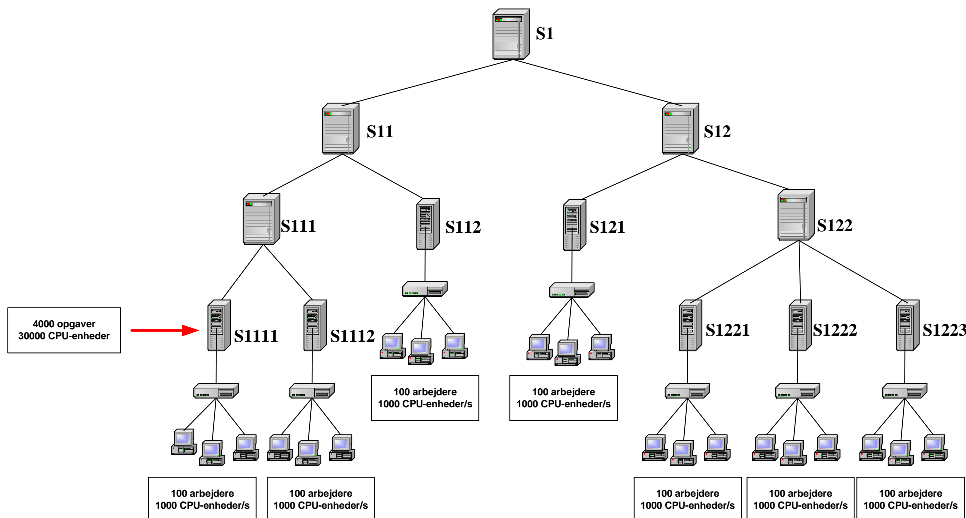


Figur C.7: Øverst ses antallet af frie arbejdere i *S12* og *S121*. Det ses at de arbejdere som var frie mens *S11* var tungt belastet på figur C.6, er frie arbejdere fra *S121*. I midten ses belastningsfaktorerne for *S12*, *S121* og *S122*. Det ses, at i den periode, hvor der er frie arbejdere i *S121*, beregner *S12* fejlagtigt sin egen belastningfaktor til være lige over 1, dvs. den ser sig selv som normalt belastet. Nederst ses det antal opgaver, som *S12* henter fra sin forælder, det antal opgaver *S121* henter fra *S12*, samt det antal opgaver *S12* henter fra *S122*. Det ses at i den periode, hvor der er frie arbejdere i *S121*, henter *S12* ikke opgaver ned fra *S1* og ikke opgaver op fra *S122*.

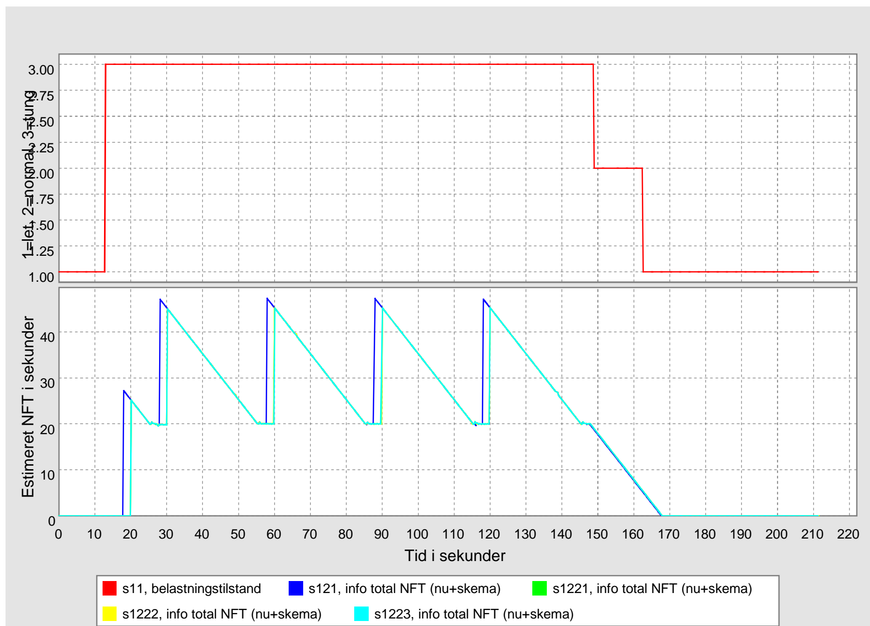


Figur C.8: Øverst viser figuren belasningsfaktoren for *S1112* og i midten belastningstilstanden for samme site. Nederst ses antallet af opgaver *S1112* henter fra sin forælder, samt antallet af opgaver der trækkes op fra *S1112* af dets forælder. Det ses at *S1112* afhjælpes sit problem i starten (indtil sitet er normalt belastet). Herefter trækkes der ikke opgaver op fra *S1112*. Ligeledes begynder *S1112* først at hente opgaver fra sin forælder når belastningsfaktoren kommer ned på $X_{min} = 1.0$.

C.3 Forsøg 2-1



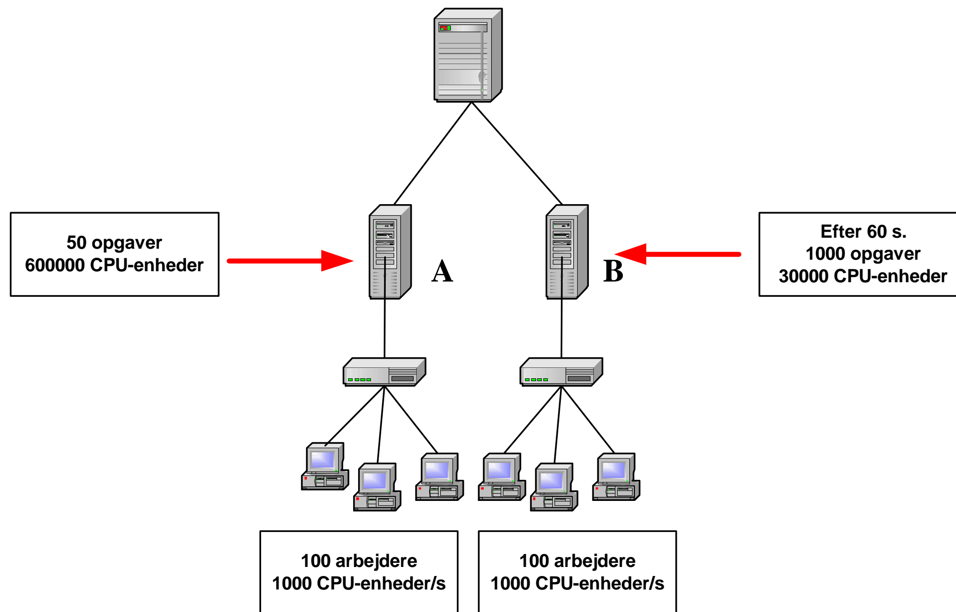
Figur C.9: Figuren viser det system som er brugt ved forsøg 2-1.



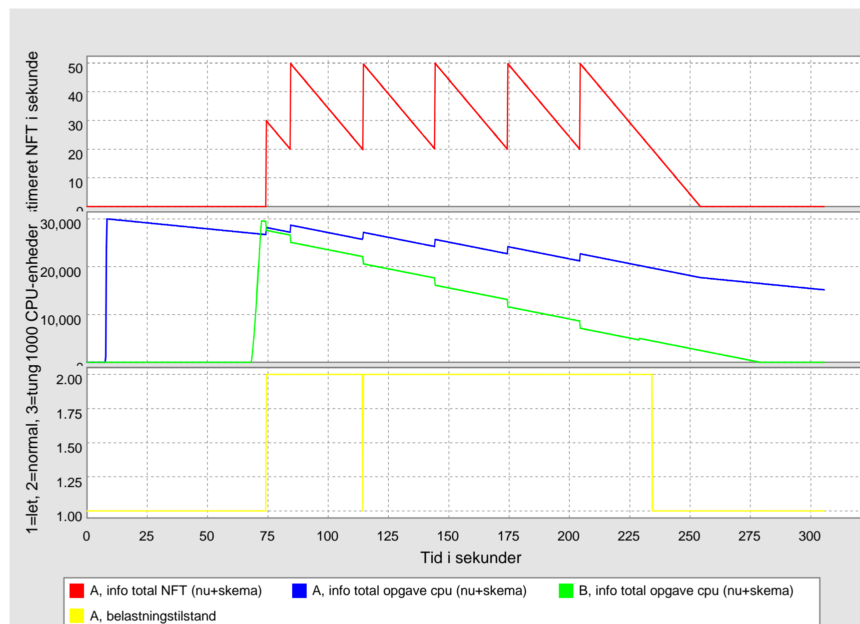
Figur C.10: Øverst viser figuren belastningstilstanden for venstre del af træet (*S11*). Nederst vises den estimerede NFT i sekunder for alle sites i højre del af træet. Det ses at ingen af sitene i højre del af træet har en NFT mindre end $X_{min} = 20s$, i den periode hvor *S11* er tungt belastet.

Figur C.11: Figuren viser de estimerede NFT'er i sekunder for *S1*, *S11* og *S12*. Det ses, som forventet, at NFT for *S1* er faldende med konstant hastighed (hældning på -1).

C.4 Forsøg 2-2

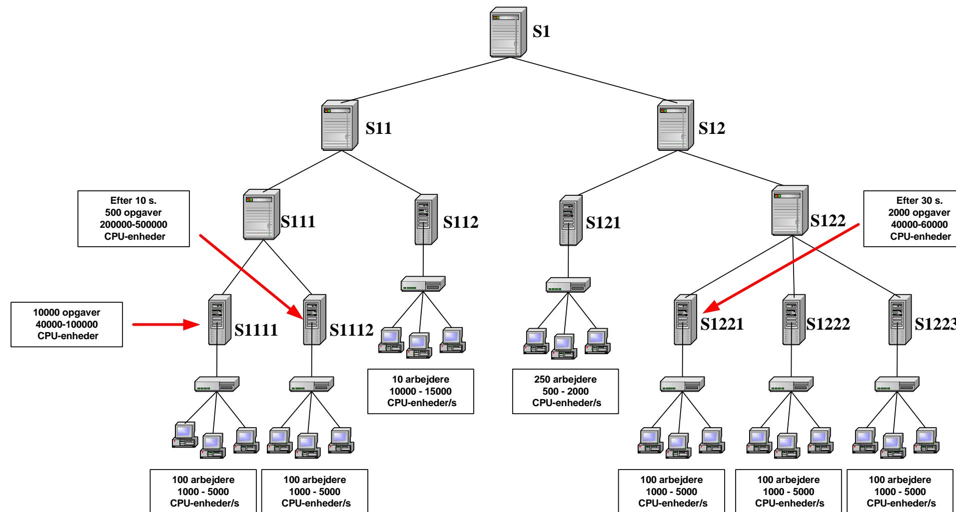


Figur C.12: Figuren viser det system som er brugt ved forsøg 2-2.

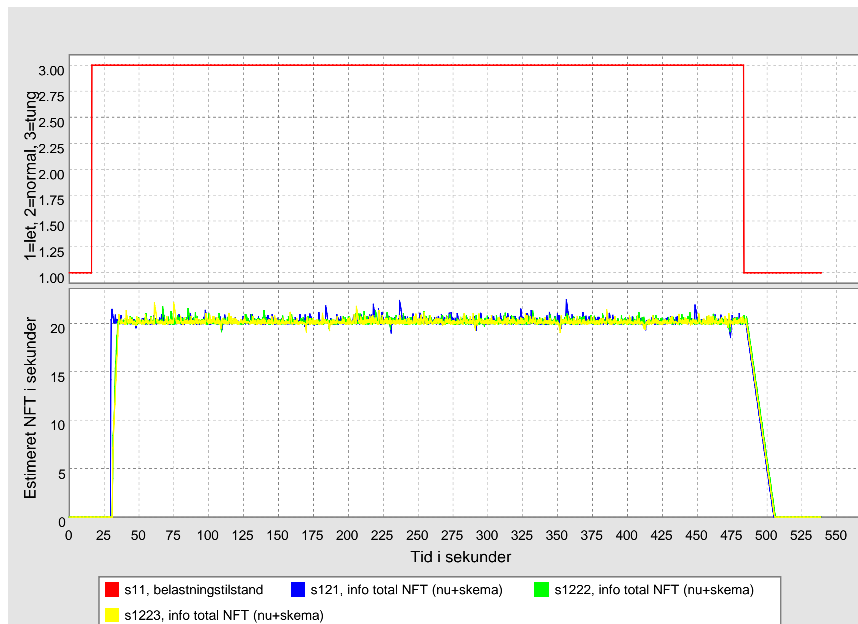


Figur C.13: Øverst viser figuren A's NFT i sekunder. I midten ses det totale antal CPUenheder i A og i B. Nederst ses A's belastningstilstand. Det ses at A er let belastet indtil der stilles opgaver i B. Det ses endvidere at A's NFT er større end X_{min} i den periode hvor B har overskydende opgaver.

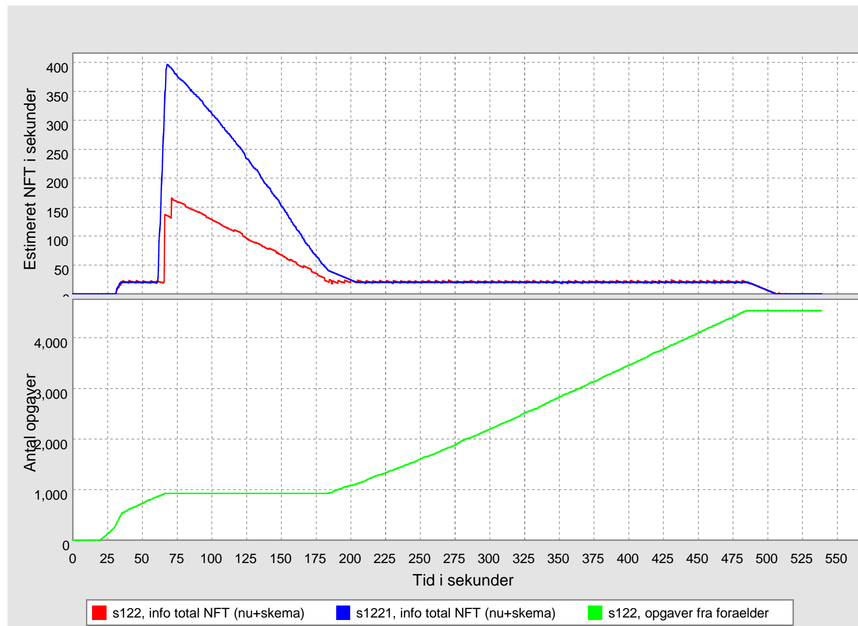
C.5 Forsøg 2-3



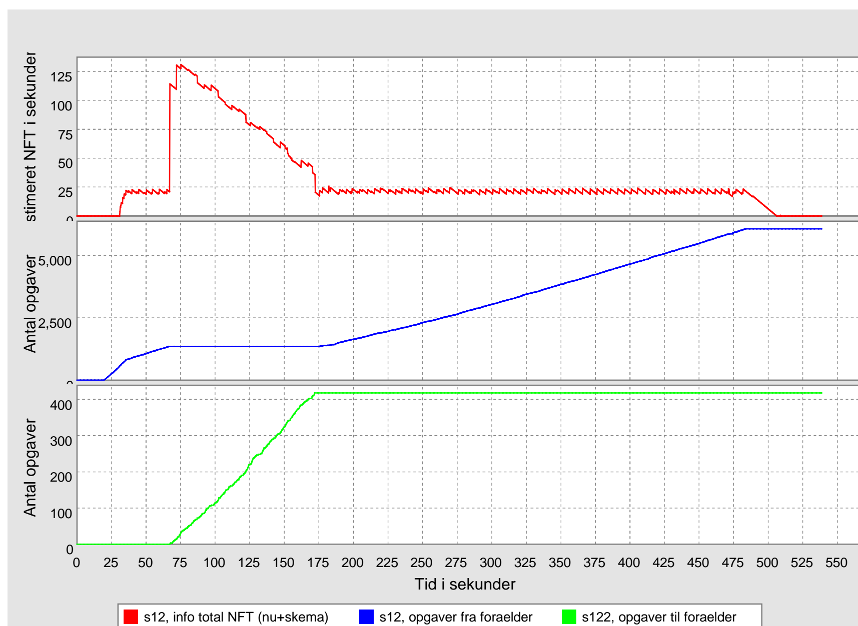
Figur C.14: Figuren viser det system som er brugt ved forsøg 2-3.



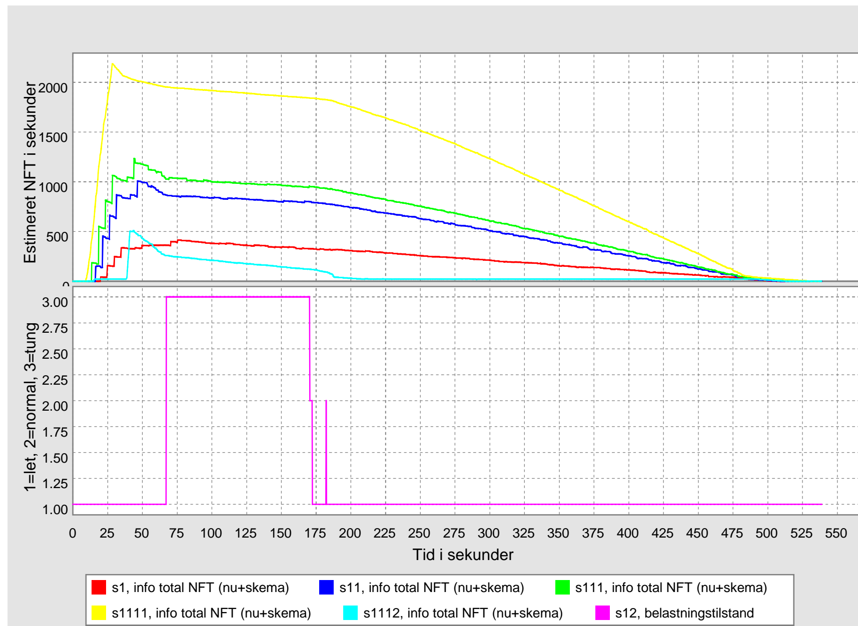
Figur C.15: Øverst viser figuren belastningstilstanden for den havdel af træet hvor de fleste opgaver stilles (*S11*). Nederst ses den estimerede NFT for de tre sites i højre del af træet, hvor der ikke stilles opgaver. Det ses at den estimerede NFT i de tre sites holder sig meget pænt omkring $X_{min} = 20s$ i den periode, hvor *S11* er tungt belastet



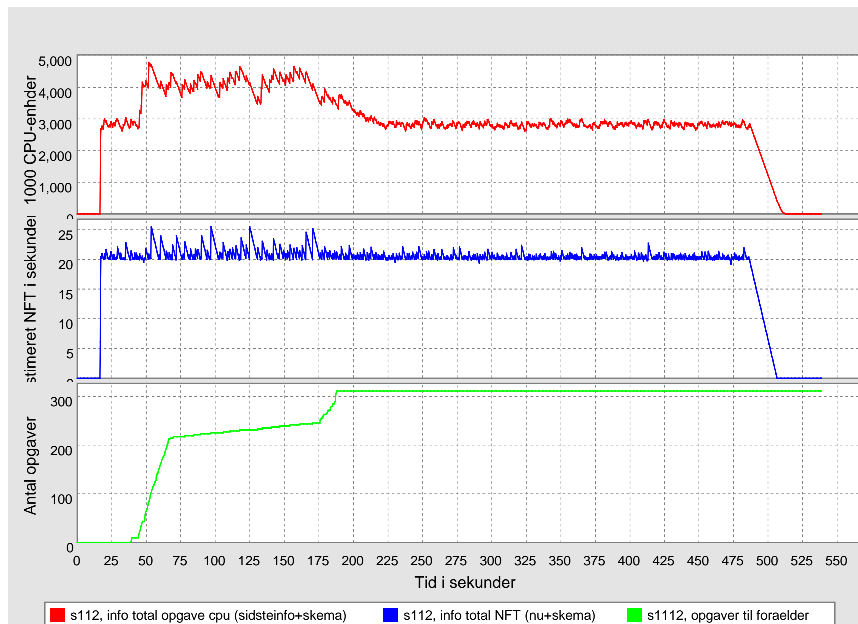
Figur C.16: Øverst viser figuren den estimerede NFT for det site i højre del af træet, hvor der stiles opgaver (*S1221*), og for dennes forælder (*S122*). Nederst ses antallet af opgaver *S122* henter fra sin forælder. Det ses, at i den periode, hvor der er for mange opgaver hos *S1221*, henter *S122* ikke opgaver fra sin foræder.



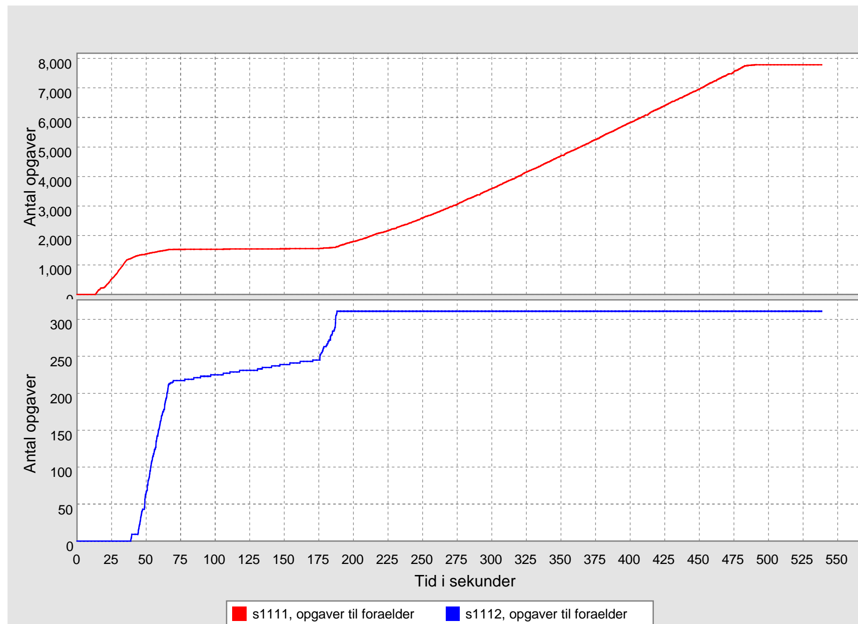
Figur C.17: Øverst viser figuren den estimerede NFT for *S12* i sekunder. I midten ses antallet af opgaver *S12* henter ned fra sin forælder, og nederst ses antallet af opgaver *S12* henter sit tungt belastede barn (*S122*). Det ses at i den periode, hvor *S12* har en NFT større end ca. 20, hentes der ikke opgaver ned fra dennes forælder, i stedet for hentes der opgaver fra det tungt belastede barn.



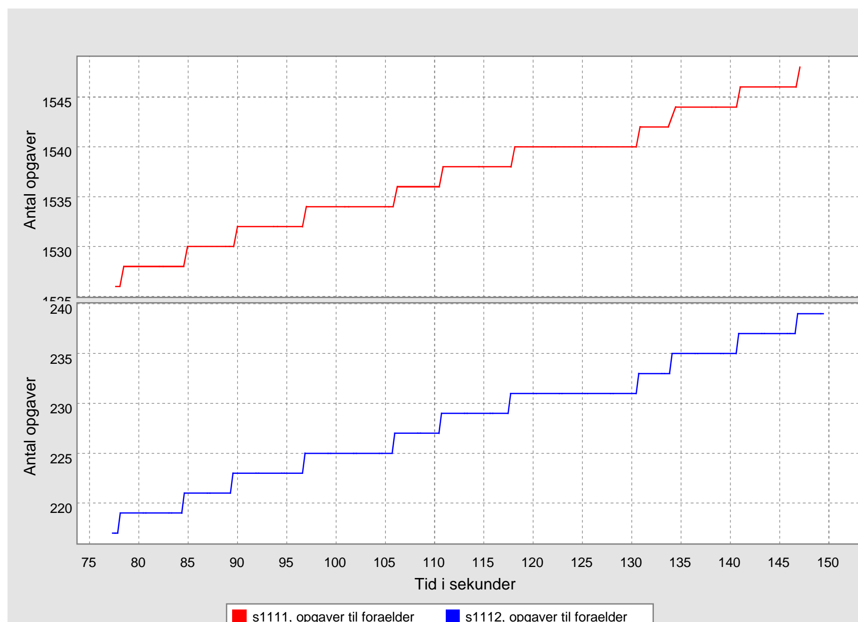
Figur C.18: Øverst vises den estimerede NFT for de sites i venstre side af træet, hvor der stilles opgaver, samt for deres forfædre. Nederst ses belastningstilstanden for højre side af træet (*S12*). Det ses at at NFT'en falder med en mindre hastighed i venstre del af træet, så længe højre side af træet selv er belastet.



Figur C.19: Øverst viser figuren det totale antal CPUenheder i *S112*. I midten ses den estimerede NFT for samme site. Nederst ses antallet af opgaver som *S1112* får trukket op fra sin forælder. Det ses at der er større udsving i den totale opgave mængde for *S112* i den periode hvor *S1112* bidrager til systemet med de meget store opgaver. Det ses også at NFT for *S112* ikke har nær de samme udsving.

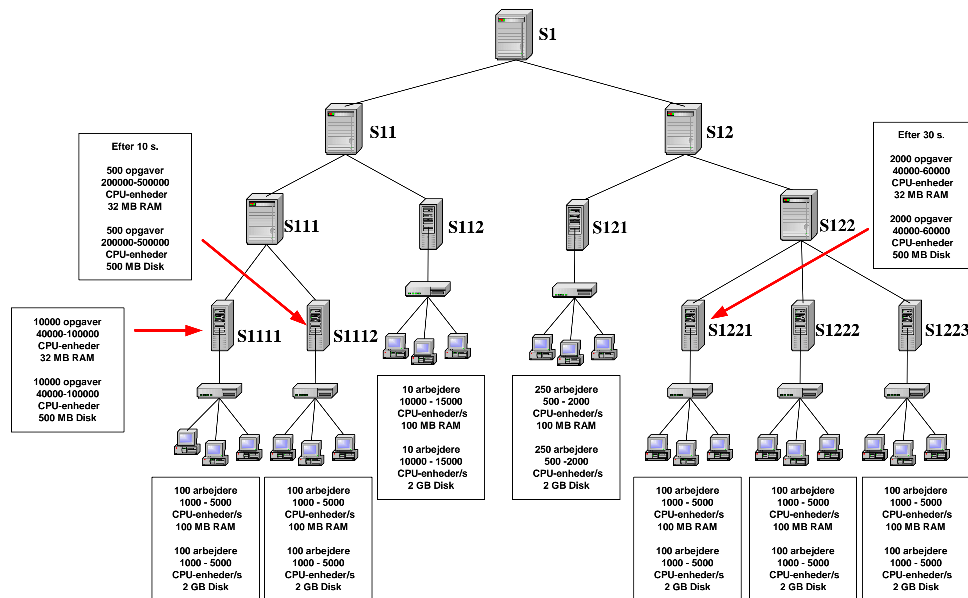


Figur C.20: Øverst viser figuren antallet af opgaver *S111* trækker op fra *S1111*, og nederst antallet af opgaver som *S111* trækker op fra sit andet barn (*S1112*).

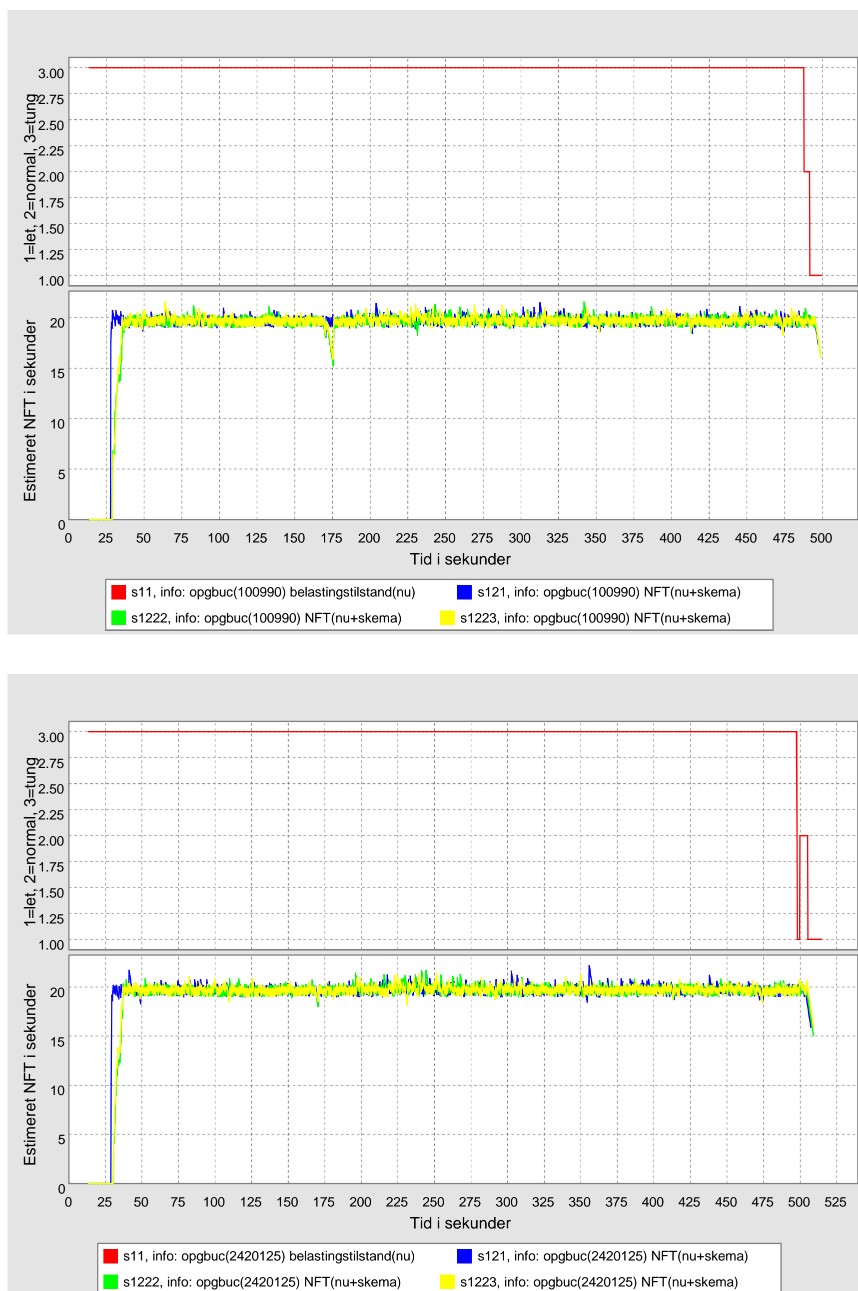


Figur C.21: Figuren viser det samme som figur C.20, dog kun i tidsrummet 75-150 sekunder (det flade stykke på figur C.20). Sammen med figur C.20 viser figuren at der trækkes opgaver op fra begge de to overbelastede sites.

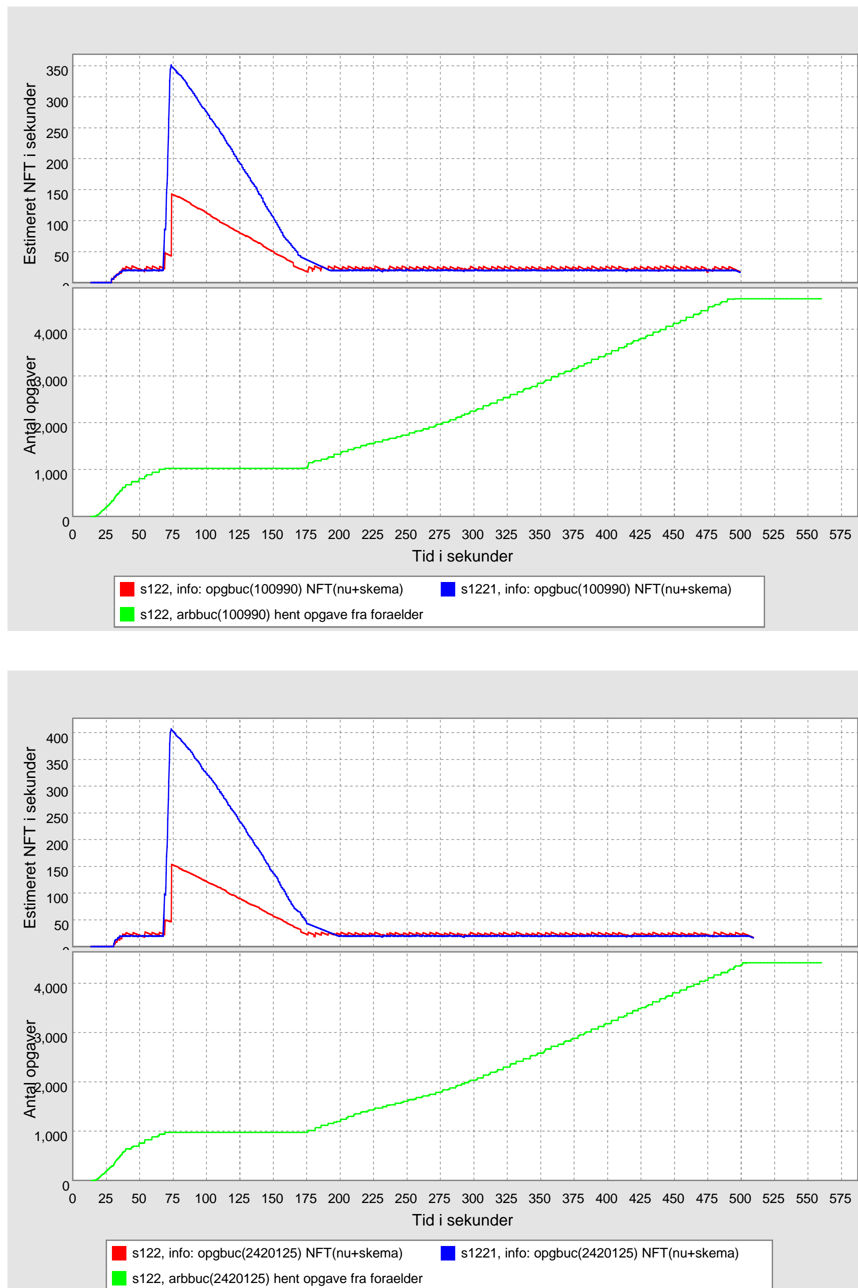
C.6 Forsøg 3-1



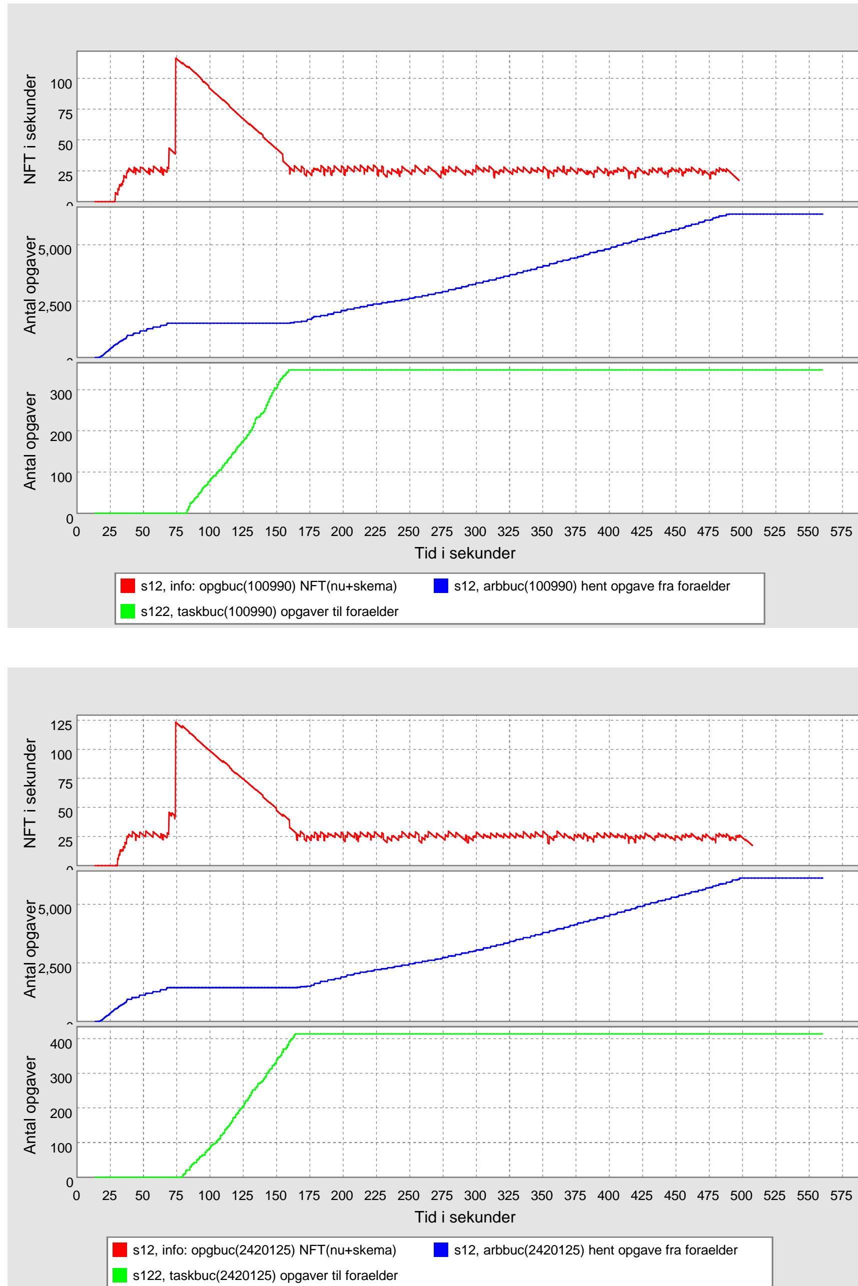
Figur C.22: Figuren viser det system som er brugt ved forsøg 3-1.



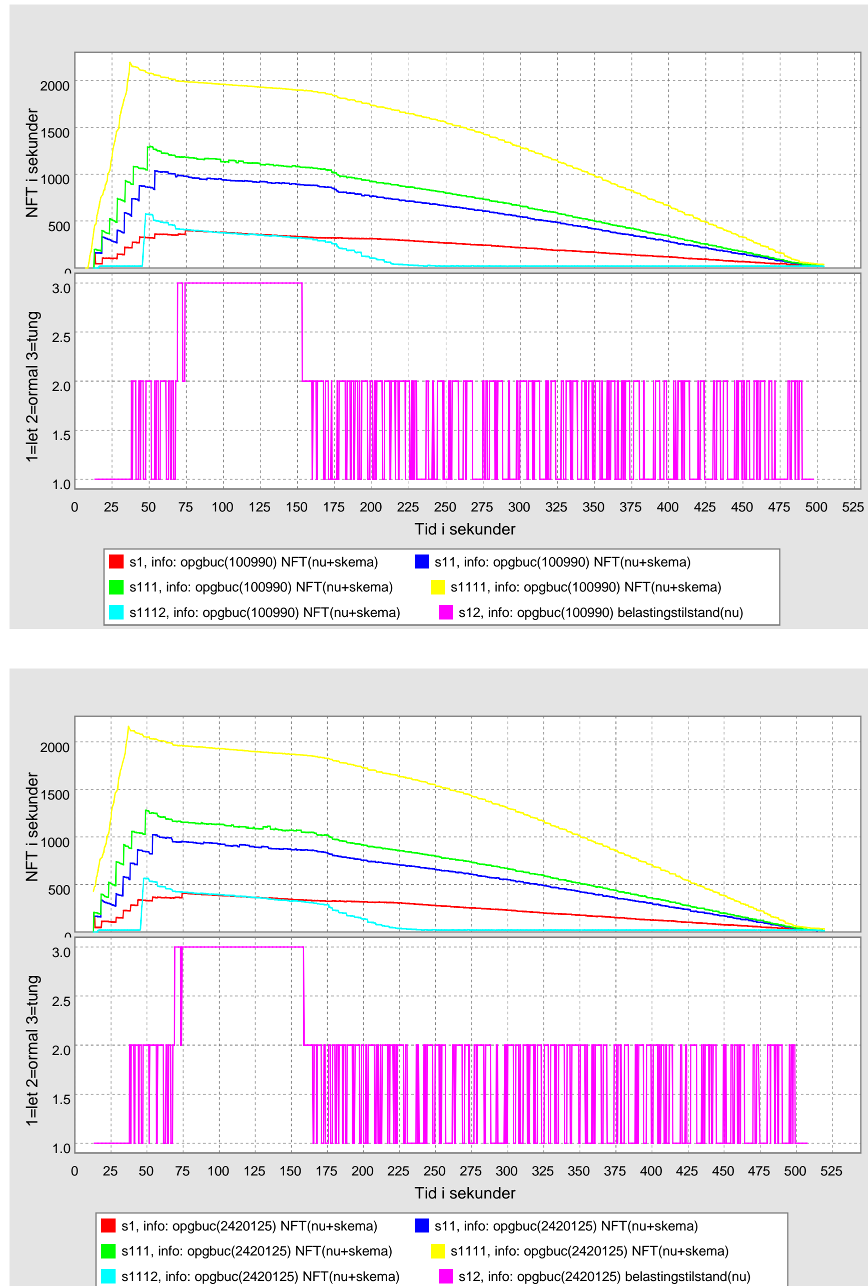
Figur C.23: Figuren viser for hver af de to opgavegrupper, de samme oplysninger som figur C.15, dvs. belastningstilstanden for *S11* samt NFT for *S121*, *S1222*og *S1223*.



Figur C.24: Figuren viser for hver af de to opgavegrupper, de samme oplysninger som figur C.16, dvs. NFT for *S122* og *S1221*, samt det antal opgaver som *S122* henter fra sin forælder.

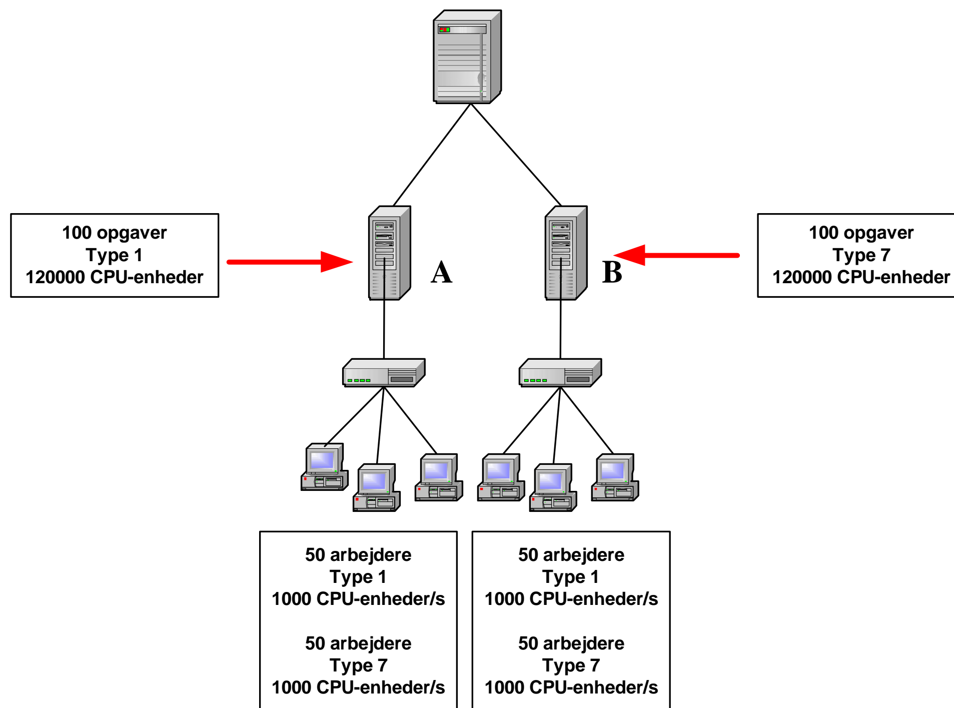


Figur C.25: Figuren viser for hver af de to opgavegrupper, de samme oplysninger som figur C.17, dvs. NFT for *S12*, det antal opgaver som *S12* henter fra sin forælder, samt det antal opgaver *S12* henter fra *S122*.

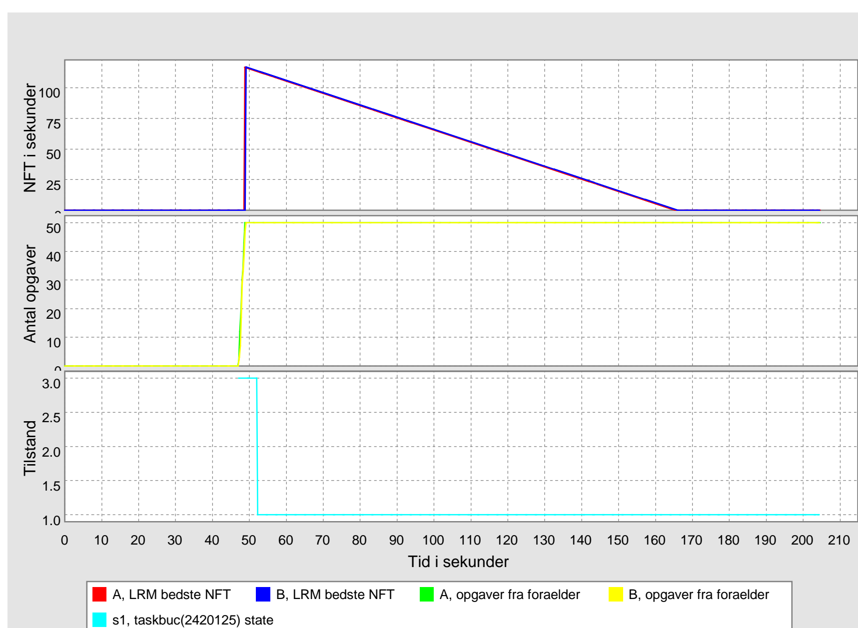


Figur C.26: Figuren viser for hver af de to opgavegrupper, de samme oplysninger som figur C.18, dvs. NFT for *S1*, *S11*, *S111*, *S1111* og *S1112*, samt belatningstilstanden for *S12*.

C.7 Forsøg 3-2

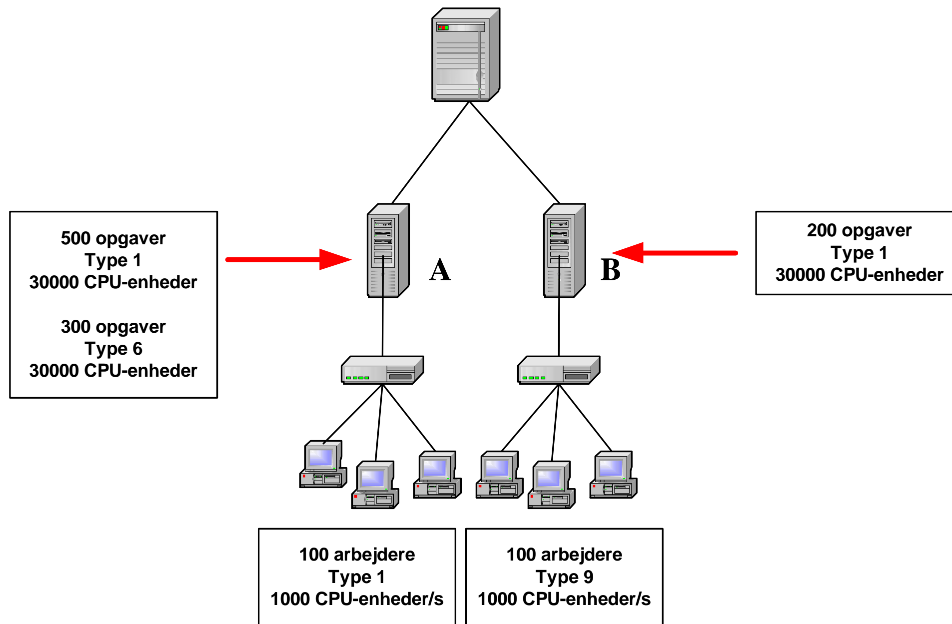


Figur C.27: Figuren viser det system som er brugt ved forsøg 3-2.

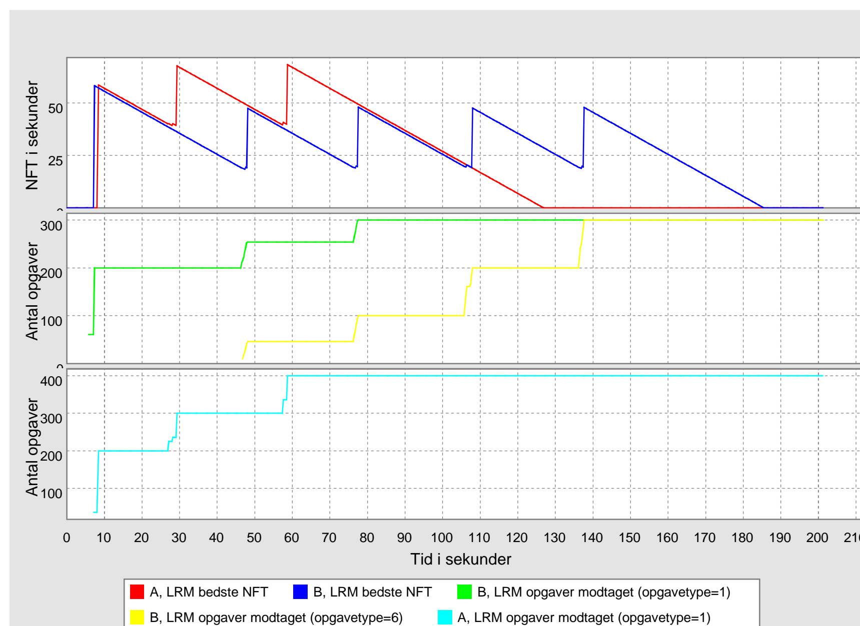


Figur C.28: Figuren viser NFT hos den lokale ressourcemanager i henholdsvis A og B (øverst). I midten viser figuren antallet af opgaver, som A og B modtager fra deres fælles forælder. Nederst viser figuren tilstande i supersitet for den ene af de to opgavegrupper (vi gør igen opmærksom på, at denne tilstand *ikke* er det samme som belastningstilstanden). Begge sites modtager 50 opgaver, og begge site udnytter derfor alle deres ressourcer. Selve balanceringen foregår indenfor et kort tidsrum (den tid hvor *S1*'s tilstand er lig 3).

C.8 Forsøg 3-3

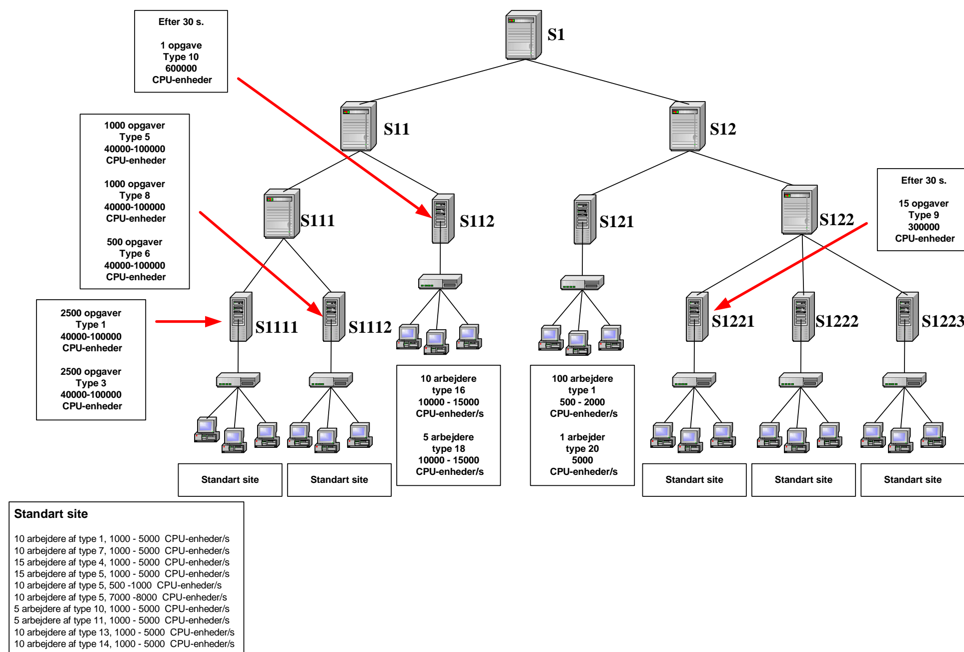


Figur C.29: Figuren viser det system som er brugt ved forsøg 3-3.

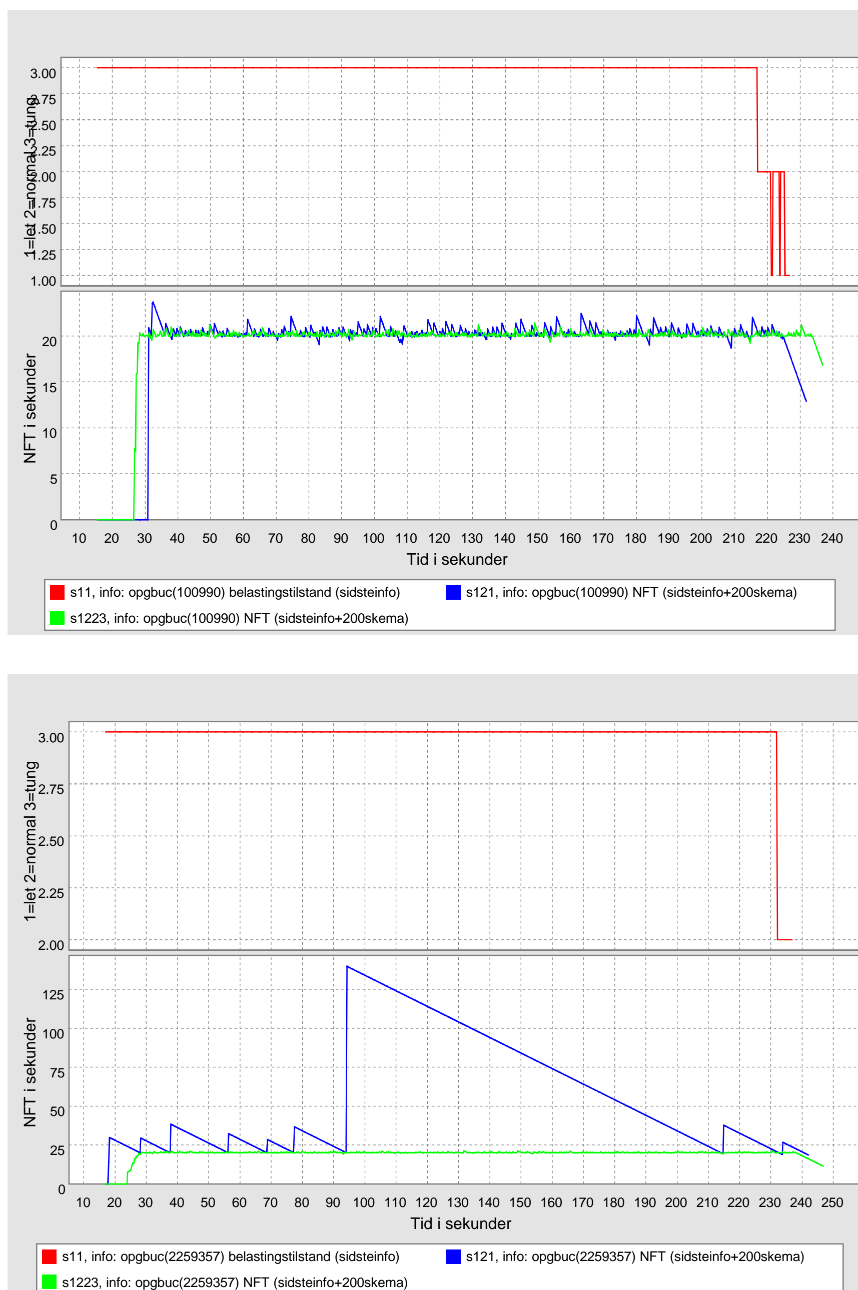


Figur C.30: Øverst viser figuren den generelle NFT for de lokale ressource managere i A og B. I midten ses antal opgaver af de to typer, som er scheduleret i B. Nederst ses antal opgaver af type 1, som er scheduleret i A (A har ikke scheduleret nogen opgaver af type 6, da denne type ikke kan løses i A). B schedulere først sine egne 200 opgaver af type 1, og først når NFT i B når under X_{min}, scheduleres der nye opgaver i B. B modtager både opgaver af type 1 og type 6 fra A, til trods får at det mest optimale ville være at B kun løser A's type 6 opgaver, og overlader type 1 opgaverne til A selv.

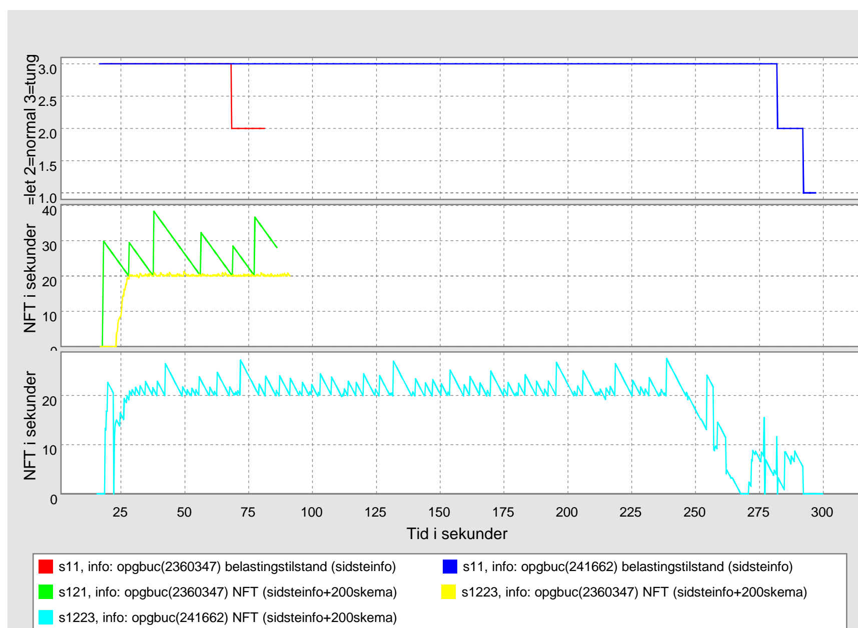
C.9 Forsøg 3-4



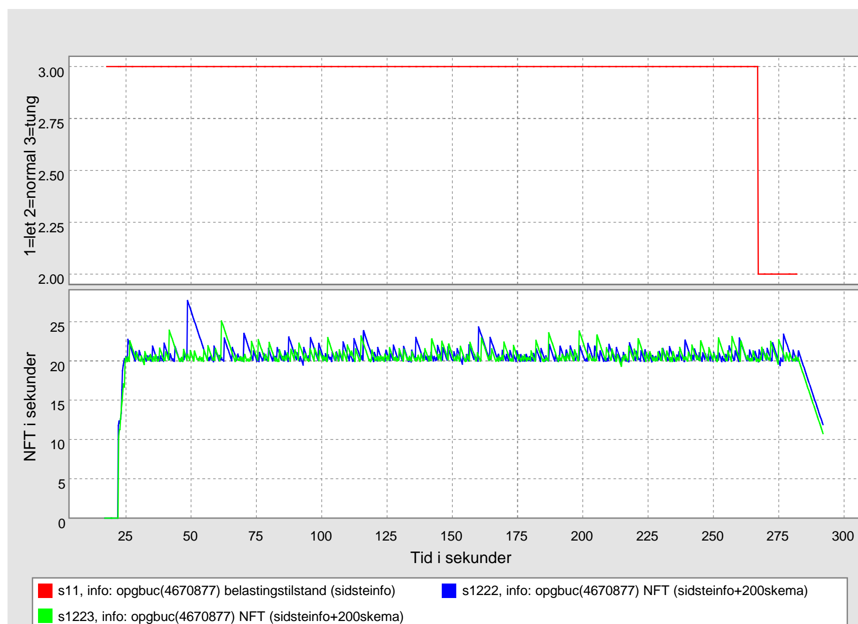
Figur C.31: Figuren viser det system som er brugt ved forsøg 3-4.



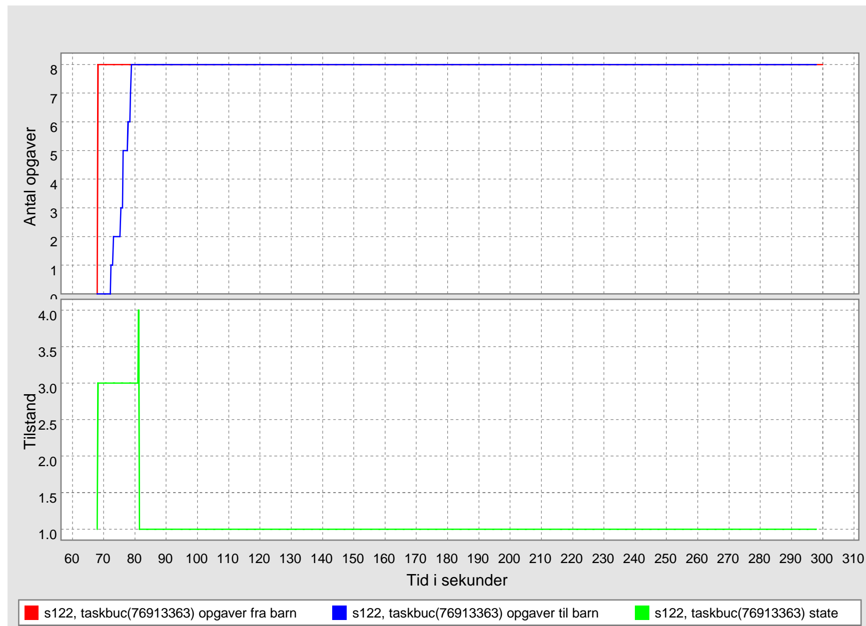
Figur C.32: De øverste dele af figurerne viser belastningstilstanden i venstre side af træet (*S11*) for to forskellige opgavegrupper (1=100990, 3=2259357). De nederste dele af figurerne viser NFT for de samme opgavegrupper i to sites (*S121* og *S1223*) placeret i højre side af træet.



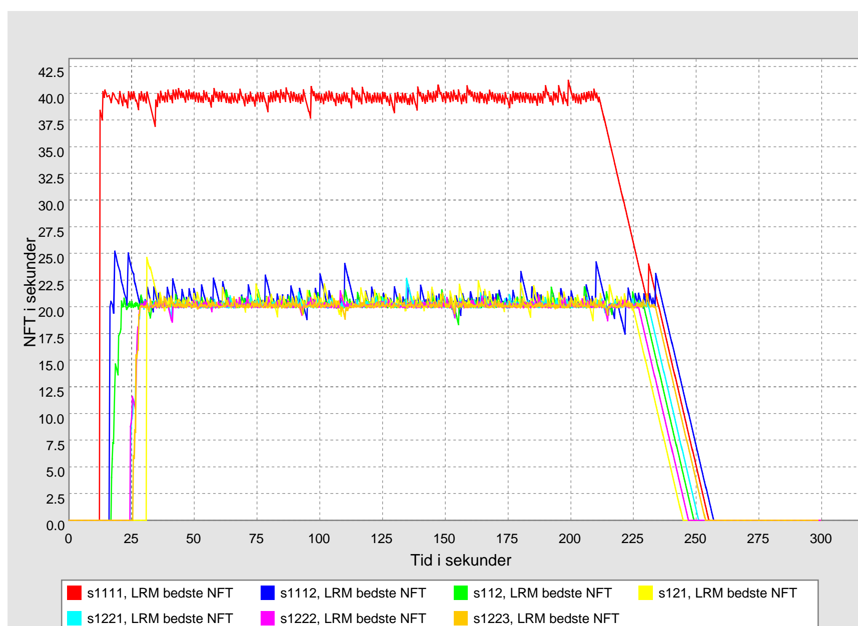
Figur C.33: Øverst viser figuren belastningstilstanden for yderligere to opgavegrupper (6=2360347, 5=241662). I midten ses igen NFT i *S121* og *S1223* for opgavegruppen med type 6. Nederst ses NFT i *S1223* for den opgavegruppen med type 5 (*S121* vises ikke her, da *S121* ikke kan løse opgaver med type 5). Der er store udsving i NFT i *S121* for opgaver af type 6. Dette skyldes at *S121* kun har én arbejder, der kan løse opgaver af type 6. NFT i sitet er derfor, for opgaver af type 6, identisk med NFT for denne ene arbejder.



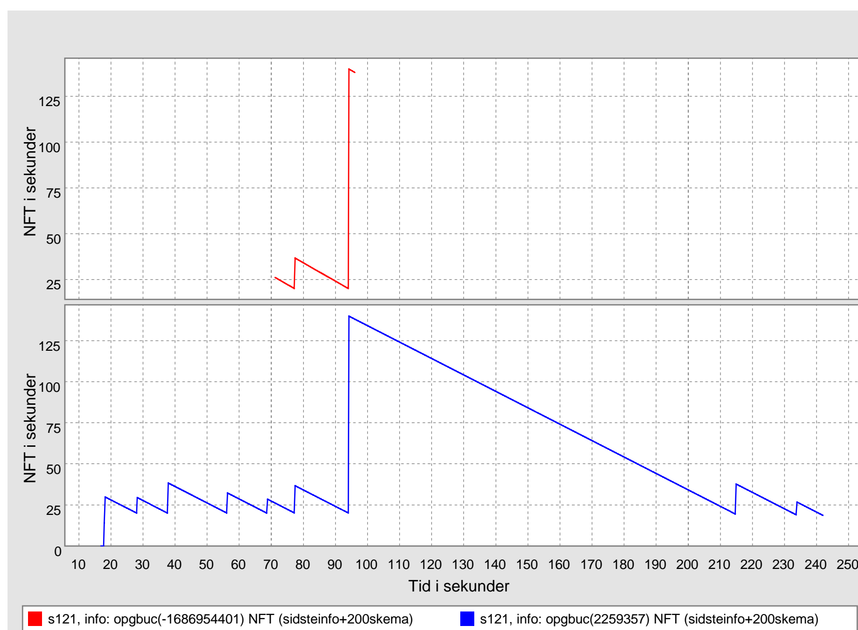
Figur C.34: Øverst viser figuren belastningstilstanden i venstre del af træet for opgavegruppe 8 (4670877). Nederst viser figuren NFT i *S1222* og *S1223* for samme opgavegruppe.



Figur C.35: Figuren illustrerer balanceringen af opgaver af type 9 (76913363). Øverst viser figuren det antal opgaver, som *S122* henter fra sine børn (i dette tilfælde *S1221*), samt det antal opgaver som der hentes fra *S122* af børnene. I midten viser figuren opgavegruppens tilstand i *S122*, og nederst belastningstilstanden ligeledes for *S122*. Denne balancering involvere aldrig *S122*'s forælder, hvilket ses ved, at alle opgaver, som *S122* modtager fra *S1221*, bliver afleveret til *S122*'s andre børn. Figuren viser desuden, at opgavegruppen kun er aktiv i et kortere tidspunkt (illustreret på figuren ved tilstanden i *S122*).

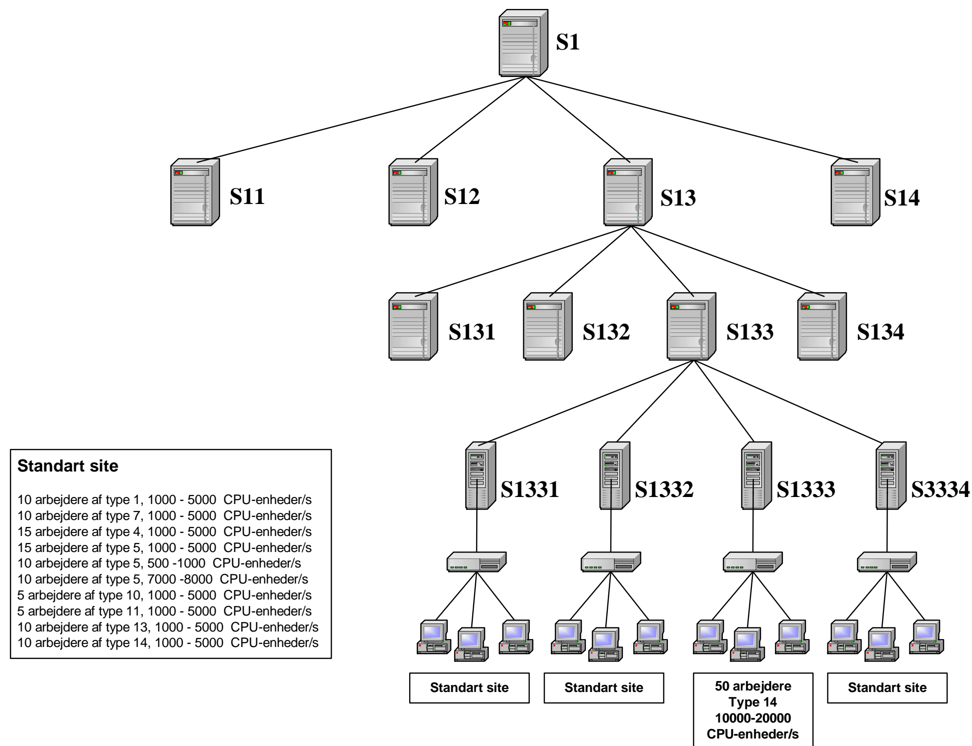


Figur C.36: Figuren viser den generelle NFT for de lokale ressourcemanagere i alle sites i systemet. *S1111* er det eneste site, der selv har opgaver, til alle sine egne arbejdere. De resterende sites skal modtage opgaver fra andre, for at udnytte alle deres arbejdere. Derfor har *S1111* en NFT omkring X_{max}, mens de resterende sites har en NFT omkring X_{min}.

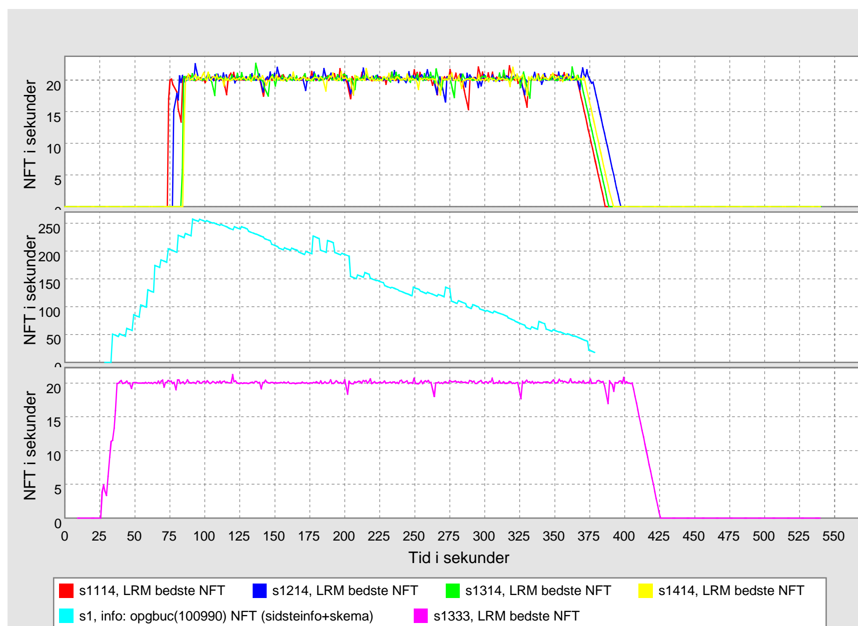


Figur C.37: Figuren viser øverst NFT for opgavetype 10 (den ene opgave, som stilles i *S112*). Nederst viser figuren NFT for opgave type 3 i *S121*, som er det site, hvor opgaven af type 10 bliver løst. I *S121* er der kun én arbejder, som kan løse opgaver af type 3. Denne ene arbejder er også den eneste arbejder i hele systemet, som kan løse opgaven af type 10.

C.10 Forsøg 3-5



Figur C.38: Figuren viser et udsnit af det system som er brugt ved forsøg 3-5.



Figur C.39: Øverst viser figuren den generelle NFT hos fire forskellige lokale ressourcemanagere (*S1114*, *S1214*, *S1314* og *S1414*). I midten ses NFT for opgaver af type 1, som det ser ud fra roden af træet. Nederst ses den generelle NFT hos den lokale ressourcemanager i *S1333*. Ud fra NFT i *S1*, ses at det tager ca. 50 sekunder før alle opgaver af type 1 er stillet (den tid hvor *S1*'s NFT vokser). Figuren viser desuden, at det tager en tilsvarende tid før der er distribueret opgaver nok rundt i systemet, så alle arbejdere er belastet. Den nederste del af figuren illustrerer, at et site, der er placeret tæt på den kilde hvorfra sitet skal modtage sine opgaver, hurtigt får opgaver nok til alle sine arbejdere.

Bilag D

Indhold af medfølgende CD-rom

Til rapporten hører en CD-rom. Denne indeholder:

Kildekode	Java-kildekoden til vores modeller, samt dokumentation af kildekoden genereret med Java's *javadoc*.
JavaCSP	Kildekoden til vores modificerede version af JavaCSP, samt javadoc dokumentation.
Jar-filer	Indeholder vores kompilerede kode, samt *JavaCSP*- og *JFreeChart*-komponenterne pakket i jar-filer.
Forsoegsdata	Datafiler fra forsøgene beskrevet i kapitel 5. Disse kan vises med vores grafiske værktøj (TSD).
Rapport	Denne rapport i PDF og PS format.

I roden af CD-rommen findes desuden en Windows bat-fil (TSD.bat), som starter det grafiske værktøj - TSD. Dette værktøj kan bruges til at læse forsøgsdataene i *Forsoegsdata* kataloget.

Alle programmerne på CD-rommen er udviklet og testet under Java 1.4.

Litteratur

[1] Collin Allison, Paul Harrington, Feng Huang og Mike Livesey. Scalable services for resource management in distributed and networked environments. *Proceedings of Third International Workshop on Services in Distributed and Networked Environments*, side 98–105, 1996.

[2] Fran Berman, Rich Wolski, Silvia Figueira, Jennifer Schopf og Gary Shao. Application-level scheduling on distributed heterogenious networks. *Proceedings of Supercomputing '96*.

[3] Francine Berman. High-performance schedulers. I Foster og Kesselman [8], side 279–310.

[4] Tracy D. Braun, Howard Jay Siegel, Noah Beck, Ladislau L. Bölöni, Muthucumaru Maheswaran, Albert I. Reuter, James P. Robertson, Mitchell D. Theys og Bin Yao. A comparison of eleven static heuristics for mapping a class of independent tasks onto heterogeneous distributed computing systems. *Journal of Parallel and Distributed Computing*, 61:810–837, 2001.

[5] Karl Czajkowski, Steven Fitzgerald, Ian Foster og Carl Kesselman. Grid information service for distributed resource sharing. I *Proceedings of the Tenth IEEE International Symposium on High-Performance Distributed Computing (HPDC-10)*. IEEE Press, august 2001.

[6] Karl Czajkowski, Ian Foster, Nick Karonis, Carl Kesselman, Stuart Martin, Warren Smith og Steven Tuecke. A resource management architecture for metacomputing systems. I *IPPS/SPDP '98 Workshop on Job Scheduling Strategies for Parallel Processing*, side 62–82, 1998.

[7] Ian Foster og Carl Kesselman. The gobus project: A status report. I *Heterogeneous Computing Workshop, 1998. (HCW 98)*, bind 7, side 4–18, 1998.

[8] Ian Foster og Carl Kesselman, redaktører. *The GRID - Blueprint for a New Computing Infrastructure*. Morgan Kaufmann Publishers, Inc., 1999.

[9] Ian Foster, Carl Kesselman, Craig Lee, Bob Lindell, Klara Nahrsted og Alain Roy. A distributed resource management architecture that supports advanced reservations and co-allocation. *Seventh International Workshop on Quality of Service*, side 27 – 36, 1999.

[10] Richard Freund, Taylor Kidd, Debbie Hensgen, Lantz Moore, Mike Gherrity, Mike Halderman og Mark Campbell. SmartNet: A scheduling framework for heterogeneous

computing. *Second International Symposium on Parallel Architectures, Algorithms, and Networks*, side 514 – 521, 1996.

[11] James Frey, Todd Tannenbaum, Miron Livny, Ian Foster og Steven Tuecke. Condor-g: A computation management agent for multi-institutional grids. *10th IEEE International Symposium on High Performance Distributed Computing*, side 55 – 63, 2001.

[12] Giacomo Gabri, Letizia Leonardu og Franco Zambonelli. MARS: a programmable coordination architecture for mobile agents. *IEEE Internet Computing*, 2000.

[13] The RAISE Language Group. *The RAISE Specification Language.* Printice Hall International, 1992.

[14] Jakob Schou Jensen. Implementering af generaliseret alternativ kommando i Java. Rapport, Institut for Informationsteknologi, Danmarks Tekniske Universitet, august 1999.

[15] Michael J. Litzkow, Miron Livny og Matt W. Mutka. Condor - a hunter of idle workstations. I *8th International Conference on Distributed Computing Systems*, side 104–111, 1988.

[16] Andrea Omicini og Franco Zambonelli. Coordination for internet application development. *Journal of Autonomous Agents and Multi-Agnet Systems*, 2(3), sep 1999.

[17] Andrea Omicini, Franco Zambonelli, Matthias Klush og Robert Tolksdof, redaktører. *Coordination of Internet Agents - Models, Technologies, and Applications.* Springer-Verlag, 2001.

[18] Rainer Pollak. A hierarchical load balancing environment for parallel and distributed supercomputer. *International Symposium on Parallel and Distributed Supercomputing*, 1995.

[19] The Globus Project. *The Globus Resource Specification Language RSL v1.0.* http://www-fp.globus.org/gram/rsl_spec1.html.

[20] The Globus Project. *GRAM RSL Parameters.* http://www-fp.globus.org/gram-/gram_rsl_parameters.html.

[21] The Object Refinery. *JFreeChart.* http://www.object-refinery.com/jfreechart-/index.html.

[22] Davide Rossi, Giacomo Gabr og Enrico Denti. Tuple-based technologies for coordination. I Omicini et al. [17], side 83–109.

[23] Philip Rousselle, Paul Tymann, Salim Hariri og Geoffrey Fox. The virtual computing environment. I *Proceedings of the Third IEEE International Symposium on High Performance Distributed Computing*, side 7–14. Dept. of Electr. & Comput. Eng., Syracuse Univ., NY, USA, 1994.

[24] Antony Rowstron. Run-time systems for coordination. I Omicini et al. [17], side 61–82.

[25] Robin Sharp. *Principles of Protocol Design*. Prentice Hall International, 1994.

[26] Sun. Javaspaces service specification. Rapport, Sun microsystems, oktober 2000.

[27] Sun. Java remote method invocation specification. Rapport, Sun microsystems, http://java.sun.com/j2se/1.4/docs/guide/rmi/index.html, 2002.

[28] Jon B. Weissman og Zhao Xin. Run-time support for scheduling parallel applications in heterogeneos nows. *http://ringer.cs.utsa.edu/faculty/jon/*, Aug 1997.

[29] Jon B. Weissman og Xin Zhao. Scheduling parallel applications in distributed networks. *http://ringer.cs.utsa.edu/faculty/jon/*, 1998.