Universitat Autònoma de Barcelona

Escola d'Enginyeria

Memòria del Projecte Fi de Carrera

2832-1 Extensió de COMP Superscalar

Memòria del Projecte Fi de Carrera
Enginyeria en Informàtica
realitzat per
Roger Rafanell Mas
i dirigit per
Porfidio Hernández Budé
Bellaterra, ... de ..... de 2010

Universitat Autònoma de Barcelona

escola d'enginyeria

El sotasignat, **Porfidio Hernández Budé**

Professor/a de l'Escola d'Enginyeria,

CERTIFICA:

Que el treball a què correspon aquesta memòria ha estat realitzat sota la seva direcció per en **Roger Rafanell Mas.**

I per tal que consti firma la present.

Signat: .....
Bellaterra, ...de.....de 2010

Agraïments

Aquest projecte significa estrictament el final d'una etapa d'aproximadament 5 anys; bé, de fet, significa molt més que tot això.

Aquest projecte significa la culminació de tots els esforços fets al llarg d'una de les etapes més interessants de la meva vida, significa també el final de la lluita per aconseguir un dels meus somnis, que ara em trobo acariciant amb les puntes dels dits... ser Enginyer Informàtic!

I doncs, ara més que mai, no puc sinó llençar un bon grapat d'agraïments per a cadascuna de les persones que han aportat el seu gra de sorra perquè jo avui em trobi escrivint aquestes línies.

En primer lloc voldria agrair sincerament a Porfidio Hernández Budé, el meu director de projecte, el seu suport, accessibilitat, celeritat per atendre'm i la seva absoluta meticulositat a l'hora d'aconsellar-me. *Gracias por tus sabios consejos Porfi!*

En segon lloc, m'agradaria mencionar especialment l'ampli suport i dedicació cedits per la Dra. Rosa Maria Badia, ja que sense els seus consells, seguiment i guiatge, aquest viatge hagués estat completament impossible. Gràcies de tot cor Rosa!

Vull agrair també l'ajuda de tots els meus companys del grup de *Grid Computing and Clusters* del BSC-CNS, sense els quals desenvolupar aquest projecte hagués estat, de ben segur, una tasca molt menys divertida.

Gràcies als meus pares, pel seu suport constant i incondicional en tots els sentits, sense el qual, de ben segur, jo no hauria arribat fins aquí. Hi ha un llarg camí des que li llegeixes contes abans d'anar a dormir al teu fill fins que se't llicencia.

A tu tiet, el primer enginyer informàtic de la família, per confiar en mi des del primer dia i permetre'm tenir-te com a referència.

Finalment gràcies a tu Henar, pel teu amor, paciència i dedicació durant aquests anys d'universitat. Ella ha sabut aguantar-me, calmar-me i donar-me esperança en els moments on el món trontollava al meu voltant. Us ben asseguro que qui cregui que és fàcil ser la parella d'un informàtic pot parlar amb ella, perquè en realitat aquesta carrera és també una mica seva.

De tot cor, moltes gràcies a tothom.

Roger Rafanell i Mas

Índex

I Introducció **1**

1 Introducció **3**
1.1 Motivació . . . 3
1.2 Objectius . . . 5
1.3 Metodologia . . . 6
1.4 Planificació inicial . . . 7
1.5 Anàlisi econòmica . . . 9
Recursos humans . . . 9
Hardware utilitzat . . . 9
Software utilitzat per desenvolupar el projecte . . . 10
Software utilitzat per a la redacció la memòria . . . 10
Material fungible . . . 11
Anàlisi del cost total . . . 11
1.6 Organització de la memòria . . . 12

II Anàlisi del projecte i estat de l'art **13**

2 Grid Computing **15**
2.1 Què és Grid Computing . . . 15
2.2 Avantatges i inconvenients del Grid . . . 16
2.3 Aplicacions del Grid al món real . . . 17
2.4 Models de programació distribuïts . . . 19
2.4.1 Model d'objectes distribuïts . . . 19
Grid Component Model (GCM) . . . 19
ProActive . . . 20
2.5 COMP Superscalar . . . 22
2.5.1 Model de programació de COMPSs . . . 22
2.5.2 L'API de COMPSs . . . 23
2.5.3 *Runtime* componentitzat . . . 24
Components . . . 24
JavaGAT . . . 25
2.5.4 Interacció entre components . . . 26
2.5.5 Definició de la interfície i selecció de tasques . . . 28

III Desenvolupament del projecte **29**

3 Disseny i implementació **31**

3.1 Punt de partida i extensions proposades 31
3.2 Gestió de rèpliques 32
3.2.1 Disseny del sistema de gestió de rèpliques 32
Anàlisi del model 32
Proposta de disseny - Requeriments funcionals 33
3.2.2 Implementació del sistema de gestió de rèpliques 35
3.3 Planificador 40
3.3.1 Anàlisi del planificador inicial 40
3.3.2 Disseny i implementació del planificador 41
Temps de transferència 41
Temps d'espera en cua 42
Temps d'execució 42
Fiabilitat dels recursos 43
Propostes de planificació 43
Implementació del planificador 45
3.4 Extracció de dades necessàries per al planificador 46
3.4.1 Predicció del temps de transferència 46
3.4.2 Mesura del temps mig d'execució 48
3.4.3 Mesura del temps d'espera en cua 49
3.5 Mesura i actualització de la velocitat de xarxa 50
3.5.1 Actualització dinàmica de la velocitat de xarxa 51
Càlcul de la velocitat de xarxa 52
Actualització de la matriu de velocitat de xarxa 54
Avantatges i inconvenients d'utilitzar aquest sistema 54
3.6 Millora de la tolerància a fallades 55
3.6.1 Disseny de mecanismes de tolerància a fallades 55
3.6.2 Recuperació de fiabilitat 57
3.6.3 Limitació en el nombre de reintents 57
3.7 Gestió de recursos 57
3.7.1 Modificacions al ProjectManager 58
3.7.2 Modificacions al ResourceManager 59
3.7.3 Implementació de l'HistoricalManager 59

4 Tests, Experiments i Resultats **61**
4.1 Entorn de proves 61
4.2 Aplicacions 62
4.2.1 HMMER 62
4.2.2 JRA4 64
4.2.3 SparseLU 65
4.3 Anàlisi del rendiment del planificador 66
4.3.1 Anàlisi del temps d'execució 66
4.3.2 Anàlisi del balanceig de tasques 68
4.4 Anàlisi del sistema de gestió de rèpliques 70
4.5 Anàlisi de la tolerància a fallades 71
4.5.1 Anàlisi sense limit de reintents 71
4.5.2 Anàlisi amb límit de reintents 72
4.6 Anàlisi del consum de recursos 73

IV Conclusions i treball futur **75**

5 Conclusions del projecte **77**
5.1 Conclusions 77
5.2 Treball futur 78
5.3 Anàlisi final de la planificació 79

V Apèndix **81**

Bibliografia **87**

Índex de figures

1.1 Planificació inicial del projecte. 8

2.1 Components que conformen un sistema de Grid Computing. 17
2.2 Interacció de components a Fractal. 20
2.3 Exemple de funcionament seqüencial, multifil i distribuït de ProActive. . . . 20
2.4 Exemple d'objecte Futur. 21
2.5 Representació del patró Master-Worker 22
2.6 Anàlisi automàtica de dependències. 24
2.7 Interacció de components del *runtime*. 25
2.8 Estructura de JavaGAT distribuïda en capes. 26
2.9 Flux d'execució d'una tasca a través del *runtime*. 27

3.1 Classes dels objectes *FileInfo*, *Version* i *Location*. 34
3.2 Classe de l'objecte *FileInstanceId*. 34
3.3 Classe simplificada del component Task Scheduler. 45
3.4 Dinàmica de la mesura del temps mig d'execució. 48
3.5 Dinàmica de la mesura del temps d'espera en cua. 50
3.6 Diagrama amb solapament de transferències. 53

4.1 Topologia del Grid de proves. 62
4.2 Graf de l'aplicació HMMER. 63
4.3 Graf de l'aplicació JRA4. 65
4.4 Graf de l'aplicació SparseLU. 66
4.5 Gràfiques de temps d'execució de l'aplicació HMMER. 68
4.6 Llegenda dels gràfics de balanceig de tasques. 68
4.7 Gràfics de balanceig de tasques en un HMMER de 8192 seqüències amb **6, 10, 14 i 22** Workers. 68
4.8 Comparativa de temps d'execució entre versions de COMPSs. 70
4.9 Exemple amb actualització dels models entre execucions. 71
4.10 Comparació del consum de CPU entre versions de COMPSs. 74
4.11 Comparació del consum de memòria entre versions de COMPSs. 74

Índex de taules

1.1 Quadre resum de costos en recursos humans. 9
1.2 Quadre resum dels costos en recursos *hardware*. 10
1.3 Quadre resum dels costos en *software* per al desenvolupament. 10
1.4 Quadre resum de costos del programari utilitzat en la redacció de la memòria. 11
1.5 Quadre de costos totals del projecte agrupats per concepte. 11

3.1 Exemple de planificació sense històric. 44
3.2 Exemple de planificació amb històric previ. 45
3.3 Exemple de matriu inicial de velocitat de xarxa (Mbps). 51

4.1 Velocitats de connexió entre els nodes del Grid. 62
4.2 Comparativa de temps d'execució de l'aplicació HMMER. 67
4.3 Temps d'execució de cada tipus de tasca en un HMMER amb 14 Workers. . 69
4.4 Temps mig d'execució inicial per un sol model. 71
4.5 Assignació de tasques segons el % de fiabilitat de la màquina bscgrid06. . . 72
4.6 Eliminació de bscgrid07 en excedir el nombre màxim d'errors permesos. . . 73

5.1 Hores finals dedicades a cada una de les etapes del projecte. 80

Part I

Introducció

Capítol 1

Introducció

Aquest projecte busca estendre l'entorn de programació paral·lela COMP Superscalar (COMPSs) dissenyat a BSC-CNS (Barcelona Supercomputing Center - Centro Nacional de Supercomputación, http://www.bsc.es) per tal de dotar-lo de funcionalitats actualment no suportades.

Aquesta extensió radica principalment en la implementació de mecanismes que permetin incrementar la flexibilitat, robustesa i polivalència del sistema. Al llarg d'aquest primer capítol es mostraran en detall els objectius que ha calgut assolir per tal de completar el projecte i en conseqüència quins han estat els problemes als quals ens hem hagut d'afrontar.

1.1 Motivació

Aquesta última dècada ha estat un període clau pel desenvolupament de models de computació paral·lela i distribuïda.

Aquests han anat aflorant a gran celeritat, intervenint actualment i gairebé de forma invisible en la nostra vida quotidiana.

La constant necessitat de capacitat de càlcul i algunes de les noves branques d'investigació, com són: l'anàlisi de models climàtics, la interacció entre proteïnes, simulacions de fluids, disseny aeroespacial o el desenvolupament de nous fàrmacs, són clars exemples de la necessitat de recrear en computadors el comportament del món real. Aquestes simulacions requereixen gran capacitat de càlcul i a la vegada una gran capacitat d'emmagatzematge que fan que sigui necessari l'ús de grans computadors o supercomputadors.

Un supercomputador és, doncs, una eina molt potent al servei de la recerca i el desenvolupament, però també és molt costosa de mantenir, tant en termes de manteniment com d'eficiència energètica.

Malauradament, doncs, l'accés a supercomputadors és moltes vegades limitat. I és per aquest fet que només una petita part d'investigadors poden accedir a aquests tipus de màquines.

Per sort, el desenvolupament de nous models de negoci basats en l'oferta de serveis de computació, coneguts també com a serveis de computació en el núvol (Cloud/Grid Computing), permeten als investigadors accedir a recursos de càlcul sense haver de tenir

en compte cap dels costos associats als supercomputadors. Només paguen pel temps que els utilitzen *("Pay-on-demand")*.

Aquests nous serveis han obligat a desenvolupar solucions que permetin la viabilitat d'aquests nous models, oferint a la vegada una bona qualitat.

Aquesta meta ha obligat a desenvolupar: nous sistemes de fitxers distribuïts, sistemes robustos i tolerants a fallades, mecanismes eficients de planificació i gestió de recursos que han permès mantenir un concepte de localitat deslocalitzada (deslocalització transparent a l'usuari).

No obstant, malgrat que es tracta d'un model sòlid i amb projecció, a dia d'avui encara hi ha elements que no estan del tot perfilats. Un d'aquests són els models de programació orientats a aquest tipus d'infraestructures.

COMPSs és, doncs, un entorn desenvolupat per BSC-CNS que proveeix d'un model de programació orientat a Grid i que permet executar aplicacions seqüencials de forma paral·lela sobre una infraestructura Grid, de forma que l'execució sigui totalment transparent a l'usuari. Això permet pensar en paral·lel, però programar en seqüencial.

Actualment COMPSs es troba en la seva primera versió i com qualsevol software que es troba en les seves primeres etapes de desenvolupament, disposa d'un nombre reduït de funcionalitats.

Disposa d'un planificador funcional, però a la vegada simple i bàsic, que no implementa cap funcionalitat que li permeti treballar amb rèpliques del fitxers inicials; és a dir, cada vegada que s'executa una aplicació, els fitxers d'entrada han de ser transferits de nou al Grid i no hi ha cap mètode que permeti indicar a COMPSs que alguns dels fitxers requerits per l'aplicació ja es troben als nodes, ja sigui perquè han estat transferits prèviament de forma deliberada o bé perquè una execució anterior ja li ha deixat còpies.

Així doncs, haver de transferir els fitxer al Grid cada vegada resulta un problema, ja que depenent de la mida del fitxer i de la velocitat de la xarxa, aquest procés pot arribar a resultar realment lent.

La motivació principal d'aquest projecte serà doncs estendre i millorar aquestes funcionalitats implementant un nou planificador basat en l'anàlisi a temps real de diferents paràmetres mesurats i extrets del Grid com: la velocitat de la xarxa entre nodes, velocitat de processament dels nodes, percentatge de disponibilitat dels nodes i el temps d'espera en cua per tal de poder executar en els nodes, que permetran a COMPSs escollir en cada moment el millor node del Grid per enviar a executar una tasca determinada.

A més a més, a causa de la carència anteriorment esmentada, desenvoluparem també un sistema per tal de permetre a COMPSs tenir *consciència* de la localització dels fitxers que distribueix al Grid; d'aquesta manera es minimitzarà el nombre de transferències en cas que els fitxers que es necessiten per executar una aplicació es trobin ja en algun node del Grid.

A la següent secció, explicarem i desglossarem amb més detall cada un dels objectius anteriorment esmentats.

1.2 Objectius

Vistes, doncs, les necessitats de càlcul en certs entorns, explicats en l'apartat anterior, l'objectiu del projecte serà aconseguir un entorn de treball capaç d'executar aplicacions en Grid de forma més eficient que en la versió actual.

La meta del projecte serà desenvolupar una nova versió del *runtime* de COMPSs, ampliant-lo en funcionalitats i millorant-lo de forma que sigui menys sensible a fallades, estenent el seu sistema de tolerància a fallades de tal manera que es permeti al planificador tenir en compte quins són els recursos del Grid que són mes fiables, i tenir-los també en compte a l'hora d'assignar-hi tasques.

Es desenvoluparà la funcionalitat que permetrà al *runtime* mantenir una consciència de la localització dels fitxers; d'aquesta manera aconseguirem minimitzar el volum de transferències al llarg de diferents llançaments de l'aplicació, evitant així que a cada execució d'una aplicació es transfereixin els fitxers necessaris.

Això s'aconseguirà mantenint una llista de rèpliques vàlides de cadascun dels fitxers, que també servirà al planificador perquè seleccioni com a preferent un recurs que té en el seu disc el màxim nombre de fitxers necessaris per poder-hi executar una tasca determinada.

S'implementaran també diferents sistemes de recollida d'informació, com per exemple la mesura dels temps mig que cal esperar per accedir al processador d'un recurs determinat, el càlcul del temps mig d'execució d'una tasca determinada a un recurs determinat, o bé els mecanismes necessaris per tal de fer la predicció del temps de transferència dels fitxers necessaris per executar una tasca.

D'aquesta manera, les dades anteriors serviran posteriorment perquè el planificador que desenvoluparem sigui capaç d'avaluar quin és el millor dels recursos d'entre tots els que disposa, entenent el millor com el que minimitza el temps d'espera per poder accedir a un recurs, el temps d'execució de la tasca i el que en maximitza la seva fiabilitat.

Amb això, la meta final del projecte és construir una nova versió de COMPSs que gestioni els recursos d'una forma més eficient i intel·ligent, obtenint a la vegada un bon compromís entre el consum de recusos i l'encert en la presa de decisions.

1.3 Metodologia

La metodologia seguida per desenvolupar aquest projecte ha estat la següent:

1. Analitzar en detall el funcionament de l'entorn COMPSs i comprendre el model de components en que està basat.

2. Definir un conjunt de funcionalitats no existents en l'entorn actual, assegurant que puguin ser implementables en el termini de duració del projecte.

3. Realitzar una anàlisi dels requisits particulars que s'hauran de complir per tal que la implementació de les extensions pugui ser acoblada correctament i de forma senzilla. Es farà també el disseny de l'arquitectura de cada extensió proposada per tal de satisfer aquests requisits.

4. L'estructuració de l'extensió a realitzar quedarà dividida en parts ben diferenciades, que seran tractades com a paquets de treball independents:

   (a) Desenvolupament del sistema de gestió de rèpliques de fitxers.

   (b) Anàlisi de diverses alternatives de planificació.

   (c) Desenvolupament del sistema d'emmagatzematge de l'històric.

   (d) Desenvolupament del sistema de planificació, desglossat en:

      i. Anàlisi dels possibles paràmetres del sistema a mesurar.

      ii. Anàlisi dels paràmetres que caldrà registrar a l'històric.

      iii. Desenvolupament del mètode de predicció del temps de trasferència de fitxers.

      iv. Desenvolupament del mètode per mesurar el temps d'espera en cua.

      v. Desenvolupament del mètode per mesurar el temps mig d'execució de cada tasca a cada node.

      vi. Desenvolupament del sistema de tolerància a fallades (càlcul de % de fiabilitat dels nodes).

      vii. Desenvolupament del planificador final.

   (e) Desenvolupament del sistema de càlcul de la velocitat de xarxa.

   (f) Fase d'avaluació del rendiment.

   (g) Fase de correccions.

   (h) Fase d'optimització del nou codi.

   (i) Redacció de la documentació.

5. Per cada paquet de treball es faran proves de validació individuals per tal d'assegurar-ne el bon funcionament.

6. Establir un entorn de proves per validar el comportament global del sistema.

7. Realitzar el conjunt de proves en un entorn real on es realitzaran assajos d'escalabilitat, rendiment, consum de memòria, etc...

1.4 Planificació inicial

Un cop introduïts els objectius i la metodologia del projecte, en aquesta secció realitzarem l'assignació temporal inicial de cada una de les tasques definides partint sempre del temps total disponible per a la realització del projecte i amb una implicació aproximada de 4 hores diàries.

Les primeres 100 hores de treball anirien dedicades a preparar tant la definició com els objectius del projecte, realitzar l'anàlisi de la documentació de COMPSs i familiaritzar-se amb l'entorn d'aquest.

En el procés de desenvolupament del prototip calculem invertir-hi aproximadament unes 616 hores, que quedarien desglossades de la següent manera:

1. Sistema de gestió de rèpliques (92 hores) desglossat en:
   (a) Desenvolupament localització de fitxers (56 hores)
   (b) Test del sistema (36 hores)
2. Desenvolupament del planificador (340 hores) desglossat en:
   (a) Anàlisi i disseny del planificador (56 hores)
   (b) Gestió d'històric (28 hores)
   (c) Implementació de millores en la tolerància a fallades (52 hores)
   (d) Càlcul dinàmic de la velocitat de xarxa (60 hores)
   (e) Mesura del temps d'espera en cua (48 hores)
   (f) Mesura del temps d'execució (28 hores)
   (g) Test del planificador (68 hores)
3. Optimitzacions realitzades al nou *runtime* (40 hores)
4. Avaluació de rendiment global (120 hores)
5. Fase de correccions finals (24 hores)

Cal tenir present a l'hora de planificar que part del projecte inclou també la redacció de la memòria final, que haurà de prendre al voltant de 244 hores.

Per tant, en total, el projecte comportarà **100** hores de familiarització i anàlisi inicial, **616** hores de desenvolupament del prototip, més les **244** hores de redacció de la memòria, que resulten finalment una previsió de **960** hores de dedicació, que es repartiran en sessions de mitja jornada de duració (4 hores) entre el mes de Febrer de 2010 i Gener de 2011 tal, i com mostra el diagrama de Gantt de la figura 1.1.

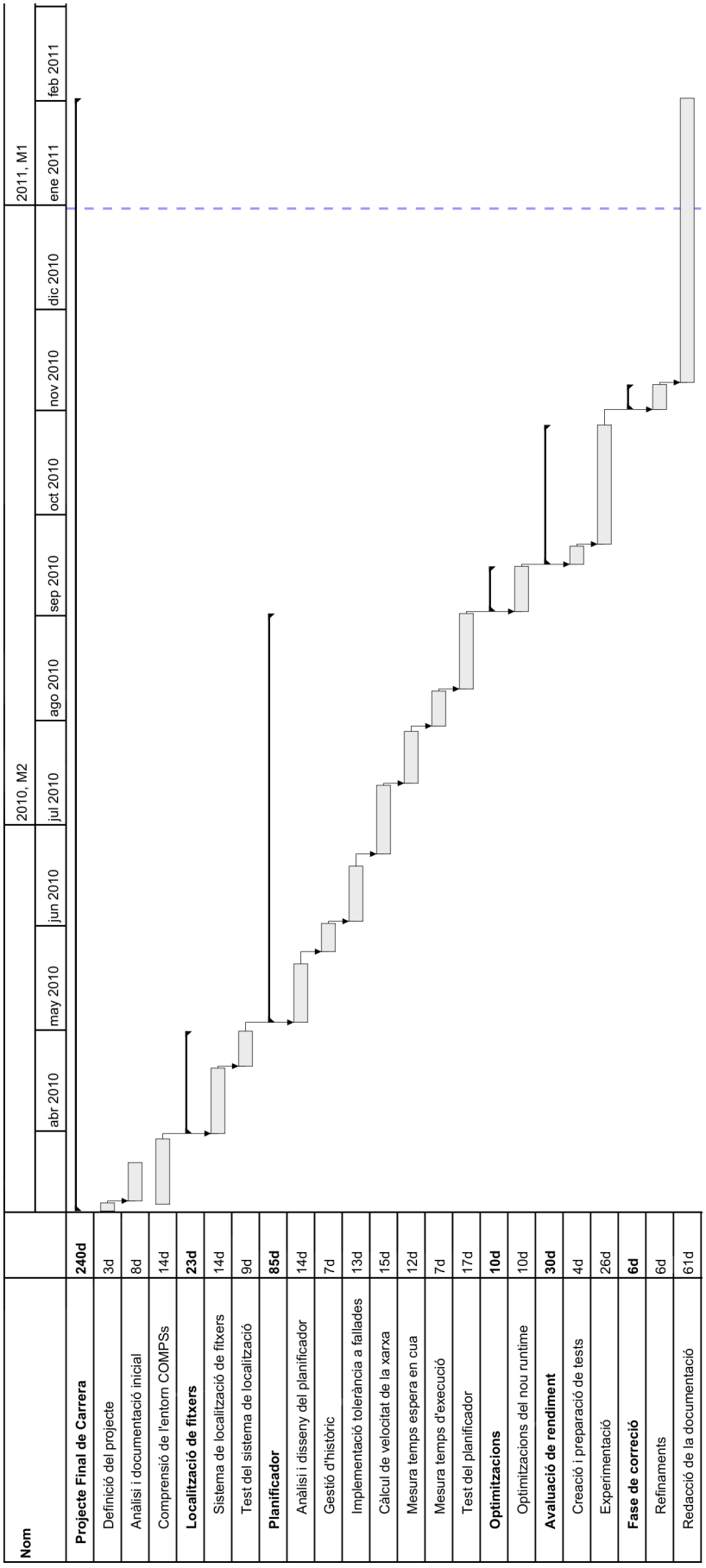


Figura 1.1: Planificació inicial del projecte.

1.5 Anàlisi econòmica

Un dels aspectes més importants a l'hora de desenvolupar un projecte és el cost econòmic que aquest pot comportar. Al llarg d'aquesta secció detallarem els costos econòmics que hagués suposat desenvolupar-lo en un entorn empresarial. Tot i la dificultat de quantificar els costos d'ús de les màquines i serveis del BSC-CNS, es revisaran el seguit de costos desglossats en: recursos humans, *hardware* i *software* utilitzats al llarg del procés de desenvolupament.

Recursos humans

Per calcular els costos de contractació de personal, s'ha classificat en tres possibles perfils les persones necessàries per desenvolupar el projecte:

- **Cap de Projecte:** pren les decisions executives a través de les diverses opcions plantejades per l'analista.
- **Analista:** pren decisions sobre la tecnologia i l'arquitectura del disseny. Proporcionar al programador la informació necessària perquè pugui desenvolupar l'aplicació.
- **Programador:** implementa el disseny de l'analista en un llenguatge de programació determinat.
- **Tècnic en sistemes:** s'encarrega de realitzar la instal·lació, configuració i manteniment dels entorns necessaris per a la resta d'actors del projecte.

En la taula 1.1 es mostra el repartiment d'hores dedicades. En la redacció de la memòria, gran part de la feina recau en l'analista, que és qui argumenta el disseny elaborat i coneix les tecnologies utilitzades.

Perfil	Cost (€/hora)	Hores dedicades	Total
Cap de projecte	75	25	1.875
Analista	60	527	31.620
Programador	42	376	15.792
Tècnic en sistemes	50	32	1.600
Subtotal			**50.887 €**

Taula 1.1: Quadre resum de costos en recursos humans.

Hardware utilitzat

Per al desenvolupament i realització tant del prototip com de les proves posteriors, s'han fet servir les màquines de l'entorn descrit en el capítol 4. Tal i com mostra la taula 1.2, aquests són els seus costos.

Perfil	Cost/Unitat (€)	Quantitat	Amortització	Total
Màquines Dual Core (BSC-CNS)	1.534	3	0.3	1.381
Màquines Quad Core (BSC-CNS)	1.897	2	0.3	1.138
Màquines 24 Core (BSC-CNS)	36.755	1	0.027	993
Computador Personal	600	1	0.33	198
Subtotal				**3.710 €**

Taula 1.2: Quadre resum dels costos en recursos *hardware*.

La infaestructura Grid utilitzada consta de 3 tipus de màquines: 3 de Dual Core, 2 de Quad Core i una màquina de 24 Cores dels quals s'han fet servir un màxim de 12. L'amortització de les màquines ha estat comptada a 3 anys donant un cost total en *hardware* de 3.710 €.

Software **utilitzat per desenvolupar el projecte**

Les llicències de *software* necessàries per desenvolupar un projecte són un altre aspecte a tenir en compte a l'hora de valorar-ne els costos. En alguns casos utilitzar *software* específic pot arribar a resultar una part important del cost total del projecte; en aquest cas no ha estat així, però cal tenir-ho en compte. A la taula 1.3 podem veure el cost de les diferents llicències de *software* necessàries per desenvolupar aquest projecte.

Software	Cost (€)
OpenSuSE 11.2 (Sistema Operatiu)	gratuït
Java J2SE Software Development Kit	gratuït
NetBeans IDE 6.8	gratuït
NetBeans UML Plug-in	gratuït
Apache-ant	gratuït
NetPerf	gratuït
Visual VM	gratuït
Subtotal	**0 €**

Taula 1.3: Quadre resum dels costos en *software* per al desenvolupament.

Software **utilitzat per a la redacció la memòria**

Realitzar el prototip no és l'únic que requereix l'ús de programari, la redacció de la memòria és també una part important del projecte i l'ús d'alguns dels programes utilitzats poden requerir també l'ús de llicències de pagament. La taula 1.4 descriu els diferents programes utilitzats a l'hora de realitzar la memòria i el seu cost.

Software	Cost (€)
Gedit	gratuït
Vi	gratuït
Latex	gratuït
Paquet Texlive (generació de pdf per latex)	gratuït
OpenOffice (Writter, Calc i Draw)	gratuït
GNUPlot (gràfics)	gratuït
MathPlotLib (gràfics)	gratuït
GIMP	gratuït
Planner (gestió de projectes)	gratuït
Subtotal	**0 €**

Taula 1.4: Quadre resum de costos del programari utilitzat en la redacció de la memòria.

Material fungible

El tòner, paper d'impressió per a la memòria i l'enquadernació d'aquesta tenen un cost total aproximat de 200 €.

Anàlisi del cost total

Concepte	Cost (€)
Recursos humans	50.887
Hardware	3.710
Software utilitzat per al desenvolupament	0
Software per redactar la memòria	0
Materials Fungibles	200
Total	**54.797 €**

Taula 1.5: Quadre de costos totals del projecte agrupats per concepte.

Tal com es mostra a la taula 1.5, el cost total del projecte puja a 54.797 €. Com podem veure, quasi el 93% del cost ve donat per Recursos humans. En concret, l'analista s'enduu gairebé el 61% de la inversió. De les 527 hores de feina que se li han assignat, 244 eren en concepte de redacció de la memòria. D'aquestes, bona part han estat dedicades a la redacció del capítol d'anàlisi de l'estat de l'art.

En ser un projecte final de carrera, bona part d'aquestes hores han estat invertides en la recerca, lectura i síntesi de la informació. Per tant, si s'hagués contractat un analista professional amb coneixements adquirits sobre la matèria, és possible que aquesta quantitat s'hagués pogut reduir 50 hores quedant un total de 477 hores assignades a l'analista i disminuint així el seu cost de 31.620 a uns 28.620 € i, per tant, també el del projecte a 51.797 €.

1.6 Organització de la memòria

Aquest document s'ha dividit en les següents parts:

1. **Introducció**: en aquesta secció es defineix el projecte. Quina és la seva motivació, quins són els seus objectius i quin és el treball realitzat per aconseguir materialitzar-los. En aquest apartat analitzarem també tant la sostenibilitat econòmica del projecte com la planificació i distribució inicial de les tasques.

2. **Anàlisi del projecte i estat de l'art**: en aquest apartat es facilita la introducció a tots els conceptes que seran necessaris per comprendre aquest document. S'introduïran conceptes relacionats amb la computació distribuïda, les èines de treball utilitzades, i es farà també una repassada a altres projectes importants que comparteixen com a base del seu èxit el concepte de computació distribuïda.

3. **Desenvolupament del projecte**: aquesta secció descriu tot el desenvolupament realitzat, tant a nivell de disseny com d'implementació, experimentació final i extracció de resultats, explicant de forma acurada el procés de desenvolupament, modificació i adaptació del nou *runtime* de COMPSs a partir del punt de partida inicial.

4. **Conclusions i treball futur**: aquest apartat sintetitza tots els resultats obtinguts i extreu les conclusions que se'n deriven; a més presenta propostes de futures millores per tal d'aportar-hi més valor al projecte.

5. **Apèndix**: per acabar, podem trobar aquí tant la descripció com l'explicació de conceptes suplementaris que poden ajudar a comprendre certs ítems del document.

Part II

Anàlisi del projecte i estat de l'art

Capítol 2

Grid Computing

En el passat, gran part dels recursos de computació es trobaven reunits principalment en centres integrats. Actualment això ja no és així, l'augment en les necessitats de recursos de càlcul i en la complexitat dels problemes a abordar, ha provocat que sigui necessari el desenvolupament de noves tecnologies de computació distribuïda que permetin oferir serveis de computació a costos moderats, la computació basada en aquesta arquitectura és l'anomenada Grid Computing.

2.1 Què és Grid Computing

La tecnologia Grid es pot veure com un conjunt heterogeni de computadors o recursos de càlcul (de diferents arquitectures, supercomputadors, clústers...) distribuïts geogràficament i que permet compartir de forma global la capacitat de procés i d'emmagatzematge a través de xarxes.

Aquest model d'organització permet la integració i l'ús col·lectiu de computadors d'alt rendiment, xarxes, bases de dades i, en general, de diferents tipus de recursos que poden ser administrats per diverses institucions.

El terme Grid va ser proposat a meitat dels anys 90 per Ian Foster i Carl Kesselman [1], fent referència a una infraestructura de computació distribuïda capaç d'aportar capacitat de càlcul a baix cost, oferint així serveis de computació a diferents camps d'investigació científica [3]. Ian Foster defineix el Grid com un conjunt de recursos de computació no administrats centralment, basats en estàndards oberts i amb una qualitat de servei difícil d'assegurar.

L'última part d'aquesta definició és un dels punts més importants i cal tenir-lo especialment en compte. Garantir la qualitat de servei en un Grid és una de les tasques que poden resultar complexes. Aquest tema el tractarem en més profunditat en l'apartat 2.2.

Remuntant en el temps, els antecessors directes del concepte de computació en Grid se solen citar en el projecte SETI@Home [1].

Aquest projecte és ampliament conegut i una de les raons és el seu objectiu, la recerca de vida intel·ligent a l'espai (Search for ExtraTerrestrial Intelligence). Aquest projecte requeria una quantitat enorme de còmput i aquesta s'aconseguia compartint els cicles de CPU de milions de màquines cedides de forma voluntària.

[1] http://www.seti.org/

D'aquesta manera el propietari d'un computador cedia recursos de la seva màquina de manera que quan estava inactiva, executava tasques per al projecte. Aquesta iniciativa, pionera en el seu moment, va interconnectar milions de computadors a tot el món, permetent disposar d'una capacitat de càlcul molt superior a la dels supercomputadors de l'època.

La computació en Grid no només tracta de compartir cicles de CPU per realitzar tasques complexes, sinó que també tracta d'establir noves infraestructures de computació distribuïda. Aquesta tasca és complexa i obliga a realitzar tasques de definició de noves arquitectures, interconnexions de xarxes, definició d'estàndards, desenvolupament de models de programació, nous models de gestió de recursos, etc... Els models de computació distribuïda, més que una aplicació són i seran una revolució en el futur de la computació.

2.2 Avantatges i inconvenients del Grid

Al llarg d'aquest apartat analitzarem els punts favorables i desfavorables dels sistemes Grid.

Tal i com s'ha explicat anteriorment en la secció 2.1, podem definir el Grid com un sistema que coordina recursos que no estan subjectes a un control centralitzat, utilitzant protocols estàndards, oberts, de propòsit general i interfícies per donar unes qualitats de serveis no trivials.

Aquest sistema està creat amb la finalitat de solucionar determinats problemes que requereixen un gran nombre de cicles de processament i/o accés a grans quantitats de dades.

Disposar de *hardware* i *software* que permeti aquestes funcionalitats, planteja habitualment inconvenients a nivell de costos, seguretat i disponibilitat. En aquest sentit, en un Grid s'integren diferents tipus de màquines i recursos, per tant un Grid no quedarà obsolet mentre tots els recursos del què es disposa s'aprofitin; de la mateixa manera que gràcies a l'escalabilitat que ofereix aquesta arquitectura és possible anar-hi afegint recursos (habitualment de diverses característiques) segons el nivell de necessitats.

Aquesta tecnologia ofereix a la vegada un servei de computació de rendiment mig-alt amb uns costos de manteniment continguts, que suposen un avantatge a l'hora d'oferir serveis de computació a l'abast de la investigació.

Respecte a l'apartat de seguretat, aquesta anirà lligada a la seguretat que sigui capaç de garantir la xarxa sobre la qual és suportada la infraestructura. Això pot arribar a resultar un problema, especialment si parlem de la utilització de xarxes WAN o, més en concret, de les de tipus *best-efford* com és el cas d'Internet.

Habitualment seran necessàries connexions permanents de banda ampla les 24 hores i 365 dies de l'any, un bon nivell de seguretat, VPN's, *firewalls*, encriptació i túnels, comunicacions segures, polítiques de seguretat i altres característiques que assegurin tot el conjunt de dades que flueixen entre els nodes del Grid.

Un dels punts especialment importants quan parlem de computació Grid és la tolerància a fallades. Aquesta garanteix el funcionament de la infraestructura si alguna de les màquines que en formen part es col·lapsa o queda inoperativa. En aquest cas, el sistema ha de ser capaç de detectar-ho i prendre alguna decisió com pot ser, per exemple, replanificar la tasca de nou en una altra màquina que resti operativa. D'aquesta manera s'aconsegueix

crear infraestructures robustes i resistents, a la vegada que flexibles.

La figura 2.1 descriu el conjunt de capes que conformen un sistema de Grid Computing. L'estrat més baix correspon als servidors, és a dir, a la infraestructura de computació física. El segon estrat correspon al conjunt d'infraestructures de xarxa, d'emmagatzemament, etc...

La tercera tot el conjunt de serveis que garanteixen la seguretat de les dades que s'allotjen al Grid. La quarta capa correspondria al software encarregat de gestionar i monitoritzar els recursos de les capes inferiors. Finalment per sobre d'aquesta capa, es recolzen el conjunt d'aplicacions que exploten l'arquitectura de Grid Computing i el client o usuari de l'aplicació.

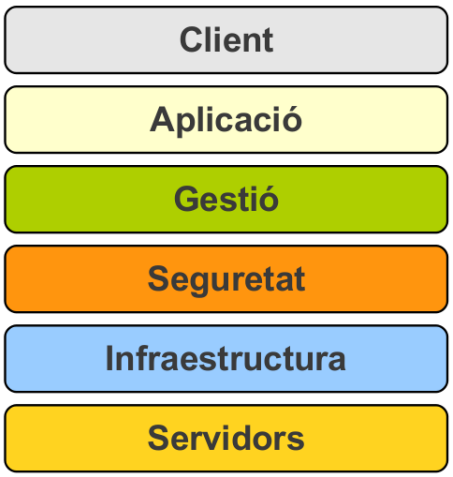


Figura 2.1: Components que conformen un sistema de Grid Computing.

2.3 Aplicacions del Grid al món real

Actualment hi ha 5 aplicacions generals i ben definides de la computació Grid:

1. **Supercomputació distribuïda:**

   Són aquelles aplicacions on les necessitats temporals per solucionar el problema no puguin ser satisfetes exclusivament per un sol node. Algunes d'aquestes necessitats poden ser generades en curts instants de temps consumint gran quantitat de recursos.

2. **Sistemes distribuïts en temps real:**

   Són aquelles aplicacions que generen fluxos de dades d'alta velocitat que han de ser analitzats i processats en temps real.

3. **Serveis puntuals:**

   En aquest tipus d'aplicacions pot no ser especialment important la potència de càlcul o la capacitat d'emmagatzemament, sinó els recursos que una organització considera com a no necessaris en un moment determinat. En aquest cas, el Grid pot presentar l'organització d'aquests recursos.

4. **Procés intensiu de tractament de dades *(Data-Crunching)*:**

   Són aquelles aplicacions que fan un gran ús de l'espai d'emmagatzemament i, en general, provoquen una gran càrrega als sistemes d'entrada/sortida. Aquest tipus d'aplicacions superen la capacitat d'emmagatzemament d'un únic node i les dades són distribuïdes a través de tot el Grid. A més de l'increment total en l'espai disponible, la distribució de les dades a través del Grid en permet l'accés de forma distribuïda.

5. **Entorns virtuals de col·laboració *(Còmput Voluntari)*:**

   Aquestes aplicacions utilitzen els recursos computacionals del Grid i la seva naturalesa per generar entorns virtuals 3D distribuïts.

   Existeixen aplicacions reals que encaixen perfectament en cadascuna de les classificacions descrites anteriorment.

Ara, al següent apartat presentarem els models de programació distribuïts, introduïnt al final de la secció, l'entorn de programació que ha motivat l'elaboració d'aquest projecte.

2.4 Models de programació distribuïts

A principis dels anys 90, les plataformes *software* monolítiques perdien força davant les distribuïdes. Els models de programació distribuïts sorgeixen de la necessitat de crear *software* més flexible, autònom, tolerant a fallades i, el més important, deslocalitzat. L'objectiu d'aquest capítol és realitzar una introducció a alguns d'ells, veient com a exemple final COMP Superscalar.

2.4.1 Model d'objectes distribuïts

El model d'objectes distribuïts permet al *software* dividir-se en mòduls, permetent que treballin en conjunt encara que resideixin en diferents computadors interconnectats a través d'una xarxa, o bé en diferents processsos dins un mateix equip. Un objecte envia un missatge a un altre que es troba allotjat en una màquina o procés remot per tal que realitzi una tasca. La informació, un cop processada, és retornada, en acabar aquest procés, a l'objecte que n'ha realitzat la crida.

Aquest procediment genèric és el *modus operandi* comú entre els models d'objectes distribuïts. En aquesta secció veurem de forma detallada *Grid Component Model*, la base de COMPSs.

Grid Component Model (GCM)

Un component és un paquet o mòdul *software* que encapsula un conjunt de funcions o dades relacionades. Cada un dels processos del sistema es distribueixen en diferents components de tal forma que totes les funcions i dades de cada component estiguin semànticament relacionades. Per aquest motiu, moltes vegades es diu que els components són *modulars* i *cohesius*.

Pel que fa la coordinació global del sistema, els components es comuniquen entre si a través d'interfícies. Quan un component vol comunicar-se amb la resta del sistema, ho fa a través d'una interfície en què hi especifica quins dels seus serveis poden ser utilitzats per altres components.

Aquesta interfície es pot veure com la signatura del component, permetent així ocultar-ne la implementació i encapsular-ne la funcionalitat.

Un dels factors més importants d'aquest model és que permet que els components siguin sustituïbles (ja sigui en temps de disseny o d'execució) si el component candidat manté els mateixos requeriments (expressats a través de les interfícies) que el component inicial. En conseqüència, els components poden ser reemplaçats per una versió alternativa o actualitzada sense haver de realitzar altres canvis en la resta del sistema.

GCM (Grid Component Model) és un model de componentització jeràrquica que permet que un component pugui estar format com una composició d'altres components ja existents. Aquesta propietat ja havia estat descrita en altres models de components com per exemple: Fractal [2], el model de components en què es basa GCM.

Fractal és un model de components abstracte, modular i altament extensible que pot ser utilitzat en diversos llenguatges de programació per tal de dissenyar, implementar,

[2] http://fractal.ow2.org

desplegar i configurar tant aplicacions com sistemes. L'objectiu de Fractal és reduir el desenvolupament i els costos de manteniment d'aquests.

Utilitza alguns dels patrons de disseny ben coneguts, com ara la separació entre interfícies i implementacions, promovent també la separació de competències.

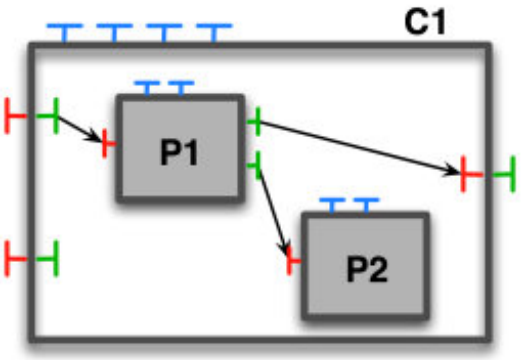


Figura 2.2: Interacció de components a Fractal.

Algunes de les implementacions més conegudes d'aquest model són: **Julia, Cecilia, AOKell, Think i ProActive**. Aquesta última serà la que analitzarem a continuació, ja que és el model sobre el que COMPSs està construït.

ProActive

ProActive [3] és la implementació de referència del model GCM. ProActive és un *middleware* Java orientat a computació paral·lela, distribuïda i *multithreaded* [4] (veure Figura 2.3) que ofereix un entorn de fàcil comprensió i un model de programació que en permet simplificar tant el desenvolupament com l'execució de les aplicacions que corren sobre sistemes multicore, distribuïts en xarxa local (LANs), clústers, datacenters o Grids. Aquest model de programació combina tant el disseny amb **objectes actius** com amb **objectes futurs**.

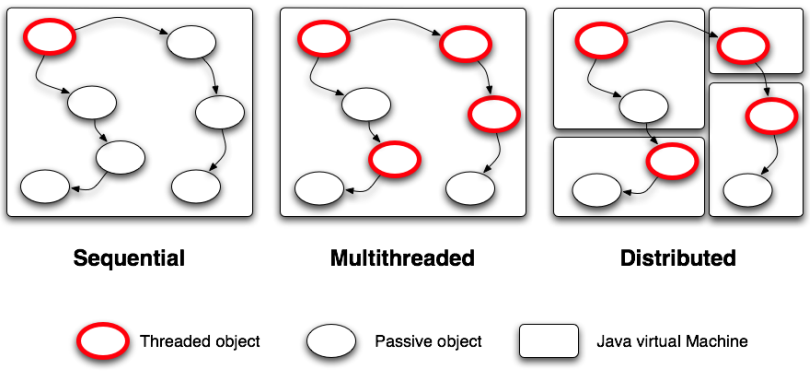


Figura 2.3: Exemple de funcionament seqüencial, multifil i distribuït de ProActive.

[3] http://proactive.inria.fr
[4] Amb vàris fils d'execució simultanis.

Objectes Actius:

Els objectes actius són les unitats bàsiques d'activitat i distribució utilitzades a ProActive per a la construcció aplicacions concurrents. Un objecte actiu s'executa en el seu propi fil. Aquest fil executa sobre l'objecte, mètodes invocats per altres objectes actius i també pels objectes passius del subsistema al que l'objecte actiu pertany. Amb Proactive, no cal manipular explícitament objectes de tipus Thread, a diferència del Java estàndard.

Aquest objecte es compon de dos altres objectes: un cos, i un objecte Java. El cos no és visible des de l'exterior de l'objecte actiu. S'encarrega de rebre crides (o peticions) en l'objecte actiu i els emmagatzema en una cua de crides en espera. Aquestes s'executaran segons l'ordre especificat en la política de sincronització. Si aquesta política no s'especifica, llavors es gestionaran segons la política FIFO. Posteriorment, el fil d'un objecte actiu tria mètodes de la cua de peticions pendents i els executa.

Per la banda del subsistema que envia crides a un objecte actiu, aquest es veu com un *proxy* [5], que genera objectes futurs per representar-ne valors futurs i transforma les crides en sol·licituds d'objectes.

Objectes Passius:

A ProActive els objectes passius no són compartits entre subsistemes. Quan un objecte qualsevol invoca un mètode d'un objecte actiu pot ser que els paràmetres que ha d'enviar siguin objectes passius. De totes maneres, com que no es poden compartir objectes passius, el que es fa és passar una còpia. D'altra banda, els objectes actius i els de retorn es passen per referència.

El fet de no compartir dades ens permet que l'aplicació no necessiti cap canvi estructural per executar-se tant en forma seqüencial, *multithread* o distribuïda.

Crides asíncrones i objectes futurs:

Una altra característica important de ProActive és que totes les crides entre subsistemes es realitzen mitjançant crides asíncrones. Això fa que les aplicacions no hagin d'esperar que l'altre subsistema els retorni el control. Aquest fet té certs desavantatges com no poder demanar objectes de retorn en aquestes comunicacions.

Per solucionar aquest problema s'afegeixen objectes futurs. Quan fem la petició al subsistema es retorna un objecte de tipus Future independentment de l'objecte que volíem rebre. Mentre que l'objecte és processat, el *thread* del subsistema que n'ha fet la crida pot continuar amb l'execució i en cas que necessiti operar amb l'objecte bloqueja el *thread* fins l'objecte que es tingui físicament. (Figura 2.4).

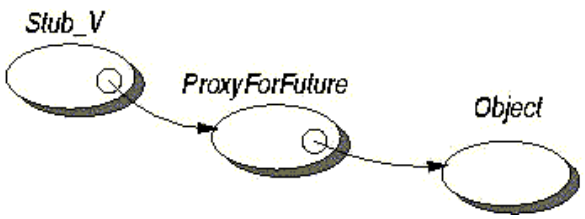


Figura 2.4: Exemple d'objecte Futur.

[5] El proxy desenvolupa la funció d'intermediari.

2.5 COMP Superscalar

COMPSs [14] [6] és un entorn de programació paral·lela que permet simplificar tant el desenvolupament com el desplegament d'aplicacions en Grid resultant transparent a l'usuari, de manera que aquest pugui traslladar una aplicació seqüencial tradicional a una aplicació paral·lela seleccionant només quines són les tasques que s'hauran d'executar al Grid.

COMPSs es troba compost d'una part estàtica i una de dinàmica. La part estàtica consta d'una API on es permet al programador seleccionar quines tasques de l'aplicació vol executar al Grid. La part dinàmica consta d'un *runtime* que gestiona, en temps d'execució, tot el comportament de l'aplicació, permetent a COMPSs ser eficient en la coordinació de recursos, escollint en cada moment quin és el millor recurs per enviar-hi una tasca.

Al llarg d'aquesta secció analitzarem una mica més l'API de COMPSs i el seu *runtime*.

2.5.1 Model de programació de COMPSs

En la introducció de COMPSs hem anunciat dues de les característiques més importants. En la primera d'aquestes, la fase estàtica, el model de programació de COMPSs transforma l'estructura de l'aplicació seqüencial a un patró Master-Worker (Figura 2.5). Aquest patró obliga a tenir dues entitats lògiques ben diferenciades: el Master, del qual només en tindrem una, i el Worker, del qual en podem tenir més d'una.

El seu funcionament és molt senzill, el Master inicia l'execució de l'aplicació i en duu el control, generant a partir de la definició de l'usuari un conjunt de tasques que s'aniran enviant a cada un dels recursos Worker disponibles, llavors el Master quedarà esperant-ne el resultat. Els Workers, en finalitzar cada una de les tasques encomanades, retornen els resultat al Master, que els rep i els processa per tal de generar el resultat final.

L'ús d'aquest patró ens imposa que el codi hagi d'estar dividit en dos grans blocs: **L'aplicació principal (Master)** i **les tasques (Workers)**. A causa d'aquesta separació, caldrà definir una interfície que ens permeti la comunicació entre ambdues parts.

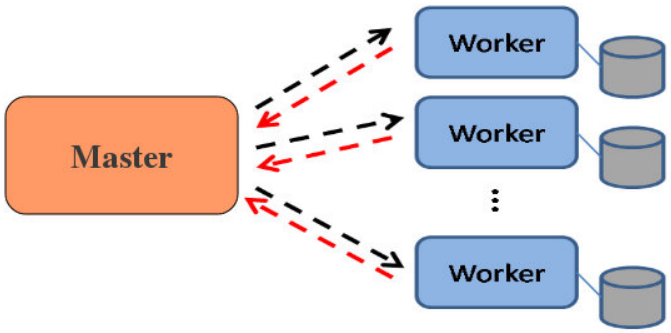


Figura 2.5: Representació del patró Master-Worker

La segona característica important és la transparència i facilitat que COMPSs aporta a l'usuari, de manera que l'aplicació codificada de forma seqüencial pugi ser executada també de forma concurrent i distribuïda al Grid sense necessitat de canvis estructurals.

[6] http://www.bsc.es/compss

Totes les restriccions imposades pel model de programació vénen donades pel fet d'utilitzar el patró Master-Worker i per la forma en què el *runtime* assigna les tasques.

Algunes d'aquestes restriccions són:

- Les tasques d'un programa Java remot hauran de ser de tipus *static*.
- Cap crida no podrà tenir tipus de retorn.
- S'accepten tipus String, File i també tipus bàsics com: **boolean, char, byte, short, int, long, float i double**.
- Actualment se suporten només llenguatges de programació com Java o C.

En resum, totes les funcions que podem invocar seguiran un esquema similar als següents:

```
public static void f1 (String,){......}
public static void f2 (File,){......}
public static void f3 (int, float, boolean){......}
```

En quasi qualsevol aplicació de còmput complex, el més usual és que es requereixin grans volums de dades d'entrada, ja siguin matrius, bases de dades, objectes complexos, etc...

Per això, COMPSs ofereix la possibilitat d'utilitzar els fitxers d'entrada de l'aplicació permetent enviar com a paràmetre d'entrada el nom del fitxer. D'aquesta manera es resol el problema d'enviar objectes o vectors com a paràmetres d'entrada.

El mateix, doncs, succeeix amb els tipus de retorn de les funcions. Podem fer que el Worker escrigui en un fitxer l'objecte que es desitja retornar. D'aquesta manera, tant el Master com els Workers poden llegir aquest objecte del fitxer. Un bon mètode per tal de realitzar el pas d'objectes a fitxers i viceversa és utilitzar la interfície *serializable* oferta per Java.

2.5.2 L'API de COMPSs

Tal i com hem mencionat a l'inici de l'apartat 2.5, COMPSs disposa d'una API que permet al programador indicar quines són les tasques que vol executar al Grid.

Durant el procés de càrrega de l'aplicació *JavaAssist* [16] [7], modifica les classes codificades pel programador a partir de les dades subministrades a la interfície anotada (secció 2.5.5), canviant-hi les crides a funcions per crides a COMPSs. Aquest s'encarregarà d'executar aquests mètodes en recursos Grid gestionats pel *runtime*, assegurant-se que es compleixin les restriccions imposades pel programador.

Així doncs, en temps de càrrega s'afegeixen les crides necessàries per tal d'arrencar i aturar el *runtime* a l'inici i final de l'aplicació. De totes maneres és possible també evitar tenir activat el *toolkit* al llarg de tota l'execució. Per aquest motiu COMPSs disposa d'una API que ofereix al programador la possibilitat d'indicar al *runtime* quan ha d'arrencar i aturar els components.

Com que el *runtime* pot ser aturat i tornat a arrencar més endavant durant la mateixa execució, el programador haurà d'indicar en quin moment s'està accedint a un fitxer que és resultat d'una tasca per tal que es tingui en compte la disponibilitat d'aquell fitxer.

[7] http://www.javassist.org

L'API de COMPSs ofereix, doncs, tres funcions:

- **startIT():** arrenca el *runtime* i comença tot el procés d'enviar les tasques als workers del Grid.
- **stopIT(terminate):** atura el *runtime* i executa l'aplicació únicament en el Master.
- **openFile(fileName, openMode):** obre el fitxer amb nom fileName de forma local en mode: Read, Write o Append. Abans d'accedir-hi comprovarà si el fitxer és al Master, si no el portarà del Worker en què es trobi.

2.5.3 *Runtime* componentitzat

Tal i com s'explica a l'inici de la secció 2.5.1, COMPSs consta de dues parts ben diferenciades; la part estàtica, explicada anteriorment, i la part dinàmica, formada pel *runtime*, que en temps d'execució gestiona el comportament de l'aplicació. Aquest s'encarrega de buscar-hi el paral·lelisme implícit, de gestionar-ne les dependències entre tasques, d'assignar els millors recursos del Grid a cada tasca i d'enviar els paràmetres d'execució als Workers.

El *runtime* de COMPSs explota la idea d'execució fora d'ordre inspirada en el concepte d'execució *Superscalar* proposada per Seymour Cray al voltant del 1965 [8], intentant fomentar el paral·lelisme a nivell de tasca. En el cas de COMPSs, això s'aconsegueix generant un graf de dependències entre les tasques de l'aplicació per tal d'aïllar-ne les que són independents. Aquestes es podran executar en diferents recursos del Grid de forma simultània tal i com il·lustra la figura 2.6.

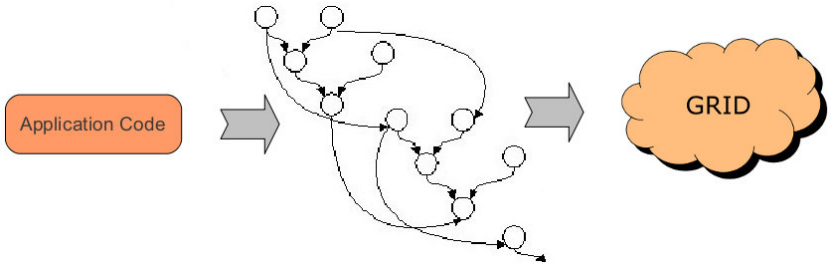


Figura 2.6: Anàlisi automàtica de dependències.

Components

Com hem vist, el *runtime* de COMPSs està implementat utilitzant el model de componentització GCM i més concretament la seva implementació, ProActive. Cada un dels seus components: **Task Analyzer**, **Task Scheduler**, **Job Manager** i **File Manager** treballen en fils d'execució diferents podent ser desplegats en diferents recursos de computació, repartint de forma més homogènia la càrrega computacional provocada pel *runtime*. Així doncs, la distribució dels components queda reflectida en el diagrama de la Figura 2.7.

Com veiem, cada component té el seu comportament, que en unir-se al de la resta dóna lloc a una funcionalitat completa. El **Task Analyzer** s'encarrega d'analitzar les dependències generant un graf de precedència que manté les relacions entre les diferents tasques del l'aplicació.

[8] http://en.wikipedia.org/wiki/Superscalar

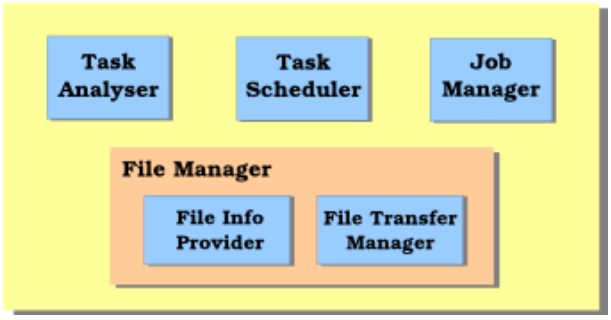


Figura 2.7: Interacció de components del *runtime*.

El **Task Scheduler** rep les tasques generades pel component anterior i en planifica l'execució al millor recurs disponible, generant així un Job o feina. El **Job manager** és l'encarregat d'executar-lo en el recurs assignat i de recollir-ne finalment els resultats.

El **File Manager** està format per dos subcomponents el **File Information Provider** i el **File Transfer Manager**. El primer s'encarrega de gestionar els accessos a fitxers i mantenir-ne una relació de versions; el segon, gestiona totes les transferències de fitxers entre nodes del Grid. Aquest, per tal d'actuar de forma transparent es recolza en un *middleware* que permet abstreure la gestió de fitxers del protocol de transferència utilitzat. Aquest *middleware* és JavaGAT i l'introduirem a continuació.

JavaGAT

JavaGAT (Java Grid Application Toolkit) [9] és un *middleware* que permet abstraure les aplicacions Grid dels *middlewares* de Grid tradicionals. Es col·loca entre les aplicacions desenvolupades i els *middlewares* de gestió de Grid com: Globus [10], Glite, SGE (Sun Grid Engine), etc... Permetent a l'aplicació mantenir-se deslligada de l'entorn de gestió del Grid.

D'aquesta manera, es permet desenvolupar aplicacions a través de l'API estàndard de SAGA [11], que proveeix d'una interfície uniforme que ofereix a la vegada la possibilitat de realitzar: operacions amb fitxers, lectura de *streams* [12] de dades, enviament de tasques (job submission), monitoritació, accés a serveis d'informació (information services), etc...

D'aquesta manera el programador només ha de desenvolupar a partir de l'API, aconseguint que l'accés als diversos components del *middleware* de Grid es faci de forma transparent. A més a més, JavaGAT manté una implementació modular que li permet estendre fàcilment el suport a altres *middlewares* a través d'adaptadors.

L'estructura de JavaGAT és la mostrada en la figura següent:

Com podem veure en la figura 2.8, l'aplicació interactuaria amb l'API de SAGA que es troba un nivell per sota, podent gestionar recursos, fitxers, monitoritzant recursos, etc... El motor de GAT faria doncs la traducció de les crides SAGA a les crides corresponents al *middleware* subjacent al Grid a través d'un adaptador. Com podem veure, hi ha múltiples adaptadors que permeten suportar: Globus, Unicore, SSH, etc...

[9] http://www.cs.vu.nl/ibis/javagat.html
[10] http://www.globus.org/toolkit
[11] http://saga.cct.lsu.edu
[12] La paraula *stream* fa referència al fet de tractar fluxos de dades continus (sense interrupcions).

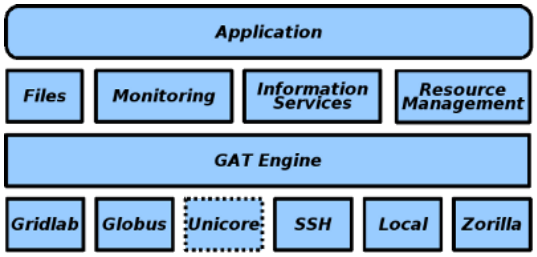


Figura 2.8: Estructura de JavaGAT distribuïda en capes.

El *runtime* de COMPSs està implementat sobre JavaGAT a nivell de la capa d'aplicació. Això permet a COMPSs mantenir-se completament desvinculat de qualsevol *middleware* de Grid guanyant polivalència i flexibilitat. D'aquesta manera, en cas de voler treballar amb algun d'ells, només caldria acoblar-hi el seu adaptador.

2.5.4 Interacció entre components

Com hem explicat en l'apartat 2.5.3, els diferents components del *runtime* gestionen tot el procés d'execució de les tasques.

Quan l'aplicació detecta que hi ha una nova tasca s'avisa al Task Analyzer. Aquest haurà de ser capaç de saber en quin moment es pot enviar a executar, és a dir, ha de ser capaç de detectar que totes les tasques de les quals depèn han finalitzat. Per poder-ho fer es construeix un graf de dependències on cada node representa una tasca i una aresta entre nodes significa una dependència. Una tasca, doncs, podrà ser enviada a executar quan no tingui cap aresta que hi apunti.

Quan el Task Analyzer rep una tasca, afegeix un node inconnex al graf i en comprova les dependències. Busca tots els paràmetres de tipus fitxer i anota la tasca que hi accedirà a través del FIP (File Information Provider).

Aquest s'encarregarà de mantenir un registre de les versions d'aquest fitxer. Cada vegada que la tasca hi escriu, es crea una nova versió amb un nou nom, aquest procediment és anomenat *renaming*. El FIP permet, així, fer la traducció entre els fitxers lògics i els reals que la tasca llegeix.

Quan una tasca es troba lliure de dependències, el TA (Task Analyzer) indica al TS (Task Scheduler) que ja pot ser planificada. Aquest intenta escollir el millor recurs per executar-hi la tasca. Actualment, el planificador original de COMPSs pren les decisions analitzant:

- Les restriccions pròpies de la tasca.
- Les tasques que pot assumir en aquell moment el node (segons el nombre de CPUs).
- Nombre de fitxers que requereix la tasca i que ja es troben al node candidat.

Quan arriba una tasca al TS, filtrarem tots els recursos que no compleixen els requeriments suficients per executar la tasca. Dels candidats restants, s'analitzarà quins d'ells tenen *slots* lliures, això significa que el nombre de tasques planificades en el recurs no superi el nombre màxim de tasques simultànies que pot executar la màquina (habitualment el nombre de CPUs).

D'entre els candidats restants d'aquest segon procés de filtratge, buscarem el que té el màxim de fitxers necessaris per executar la tasca. Això ho farem consultant al FIP (File Information Provider) quins nodes tenen el fitxer. En cas de no tenir cap recurs disponible per executar la tasca, quedaria en estat pendent al TS fins que algun node tingui *slots* lliures per executar-la.

Un cop la tasca ha estat planificada, s'avisa al JM (Job Manager) que ordena al FTM (File Transfer Manager) que transfereixi els fitxer necessaris al node planificat. Quan el FTM notifica el final de les transferències, el JM envia la feina a executar a través de JavaGAT, que quan acaba, retorna una notificació al JM amb l'estat de finalització de la feina. Si l'estat és correcte, s'avisa al TS i aquest al TA, que elimina el node de la tasca del graf.

D'altra banda, durant el procés d'execució d'una tasca poden produir-se tant errors en la transferència de fitxers, com errors en l'execució. COMPSs implementa alguns mecanismes de tolerància a fallades. Per exemple, en cas de fallada de l'execució d'una feina, el JM prova d'enviar-la a execució de nou; si torna a fallar, en demana al TS una replanificació a un altre recurs, llavors aquest provarà de planificar-la de nou en algun dels altres recursos disponibles del Grid.

La Figura 2.9 mostra el flux d'execució que segueix una tasca a través dels components del *runtime*.

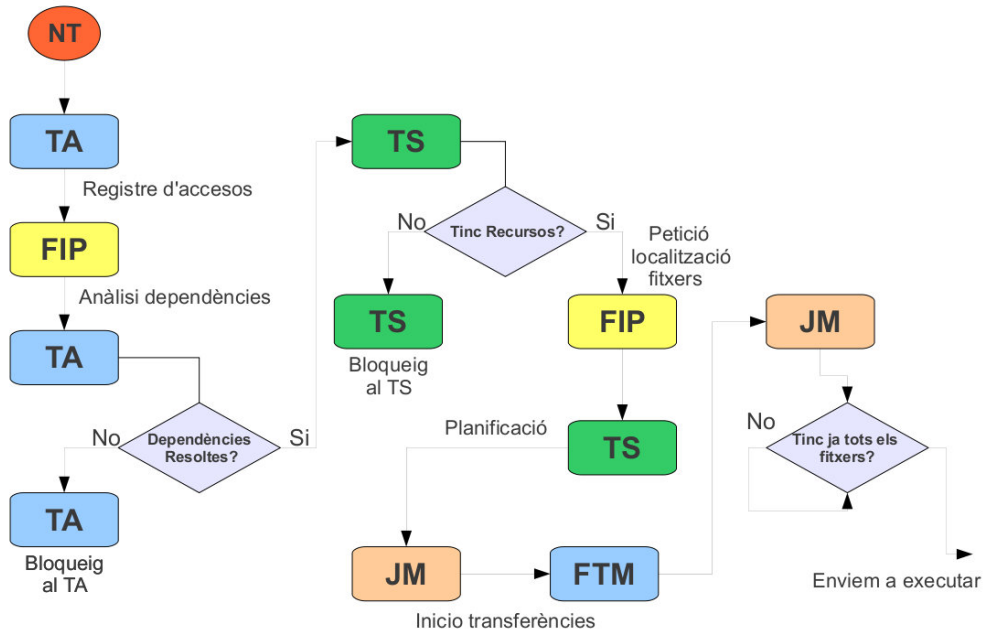


Figura 2.9: Flux d'execució d'una tasca a través del *runtime*.

2.5.5 Definició de la interfície i selecció de tasques

En la secció 2.5.1 fèiem referència a la necessitat de definir una interfície per tal de poder comunicar el Master amb els Workers.

En aquesta es defineixen totes aquelles tasques que es vol que els Workers puguin executar. L'estructura d'aquesta interfície serà com les habituals de Java, encara que a més hi realitzarem algunes anotacions extra a través de la *Java Annotation Interface* [13] per tal de permetre al *runtime* conèixer tant les restriccions de les tasques com els tipus dels seus paràmetres.

A la vegada, COMPSs necessita saber on es troba implementada cada una de les funcions i també el tipus i direcció dels seus paràmetres (entrada, sortida o entrada-sortida). Per cada paràmetre, doncs, crearem una anotació *@ParamMetadata* per tal d'especificar aquestes dades. La direcció del paràmetre vindrà indicada com: ***Direction.IN***, ***Direction.OUT*** o ***Direction.INOUT***.

D'altra banda, el programador pot indicar també al *runtime* quins requisits ha de complir un recurs per tal de poder ser candidat a executar una tasca. Això ho indicarem afegint a cada mètode de la interfície un conjunt d'anotacions del tipus *@MethodConstraints*. Un exemple el podem veure a continuació en la funció:

```
void f1(String a, String b, String c)
```

Aquesta té una cadena d'entrada a i els fitxers b (entrada) i c (sortida) que recordem, poden resultar útils per tal de transferir estructures complexes i prèviament serialitzades.

Per aquesta funció tindrem llavors la següent interfície:

```
public interface AppItf {

    @MethodConstraints(operatingSystemType = Linux,
                       processorCPUCount = 4,
                       appSoftware = "Xen")

    @ClassName(package.classA)
    void f1(
        @ParamMetadata(type = Type.STRING, direction = Direction.IN)
        String a,
        @ParamMetadata(type = Type.FILE, direction = Direction.IN)
        String b,
        @ParamMetadata(type = Type.FILE, direction = Direction.OUT)
        String c);
}
```

[13] Java Annotation Interface permet afegir metadades al codi font Java que poden ser també accedides pel programador en temps d'execució. Molts cops s'utilitza com una alternativa a la tecnologia XML.

Part III

Desenvolupament del projecte

Capítol 3

Disseny i implementació

El recorregut realitzat fins ara ens ha permès repassar *grosso modo* el món de la computació Grid i alguns dels models de programació existents. Ara doncs, és moment de centrar-se pròpiament en el desenvolupament del projecte. En aquest capítol tractarem de detallar cada una de les extensions realitzades a COMPSs. Per fer-ho, ens situarem al punt de partida i anirem presentant cada una de les propostes realitzades.

3.1 Punt de partida i extensions proposades

Tal i com s'explica en la secció d'objectius 1.2, la meta d'aquest projecte és estendre les funcionalitats de COMP Superscalar a partir del la seva primera versió.

Com a punt de partida trobem, doncs, un model de programació que disposa d'un *runtime* format per 5 components en els que als acoblarem noves funcionalitats intentant mantenir i modificar sempre el mínim possible l'arquitectura original del *runtime* (interfícies dels components) col·locant nous mètodes als components més adequats.

En l'anàlisi de la versió inicial es detectaren alguns dels punts millorables. Per exemple, cada vegada que es llençava una aplicació, el *runtime* en transferia els fitxers d'entrada cap als nodes del Grid; d'aquesta manera s'assegurava que cada vegada que s'executava una aplicació, aquesta disposava de l'última versió dels fitxers. Això resultava una solució correcta, però provocava, en cas de tenir fitxers grans, que l'execució es veies considerablement alentida.

Aquest fet portà a replantejar la forma en què es gestionaven els fitxers d'entrada, proposant una extensió per tal de millorar aquest punt. L'objectiu era permetre a COMPSs tenir *consciència* dels fitxers de l'aplicació transferits al Grid en execucions anteriors, emmagatzemant-ne la seva localització de tal forma que en cas d'haver estat transferits prèviament només calgués transferir-los de nou en cas que haguessin estat modificats entre les execucions.

A més a més, hi havia altres punts millorables. Com hem mencionat a l'apartat 1.1, COMPSs disposava de la primera versió del planificador, funcional però a la vegada bàsica. Aquest analitzava per cada conjunt de fitxers d'entrada d'una tasca quin era el recurs, d'entre els disponibles, que en tenia el màxim nombre de fitxers. D'aquesta manera s'aconseguia sempre transferir el mínim nombre possible de fitxers.

Això, de fet, no és una bona solució, ja que la decisió de transferir el mínim nombre de fitxers no garanteix minimitzar a la vegada el temps de transferència. Pot succeir que la mida de cada un d'aquests sigui gran i que, per tant, transferir el mínim de fitxers no aporti cap benefici sobre el temps de transferència, ja que pot resultar millor transferir 4 fitxers de 20KB (petits) que 1 d'1GB (gran).

Per aquest motiu es pensà a realitzar la segona extensió, que buscava solucionar aquesta situació permetent al planificador predir el temps que es trigaria a transferir el conjunt de fitxers permetent escollir com a millor recurs el que minimitzés el temps de transferència entre la font dels fitxers i el Worker escollit.

D'altra banda, el planificador tampoc no era conscient del rendiment de cada un dels recursos de què disposava, ja que treballava exclusivament amb el nombre de *slots* lliures [1]. Demanava al principi del procés de planificació totes les màquines que tenien com a mínim un *slot* lliure i que, per tant, podien executar la tasca. D'entre aquestes triava la que ja tenia el màxim nombre de fitxers.

Per aquest motiu, es va pensar també en la forma de millorar la manera en què el planificador gestionava els recursos per tal que no considerés tots els *slots* per igual i, per tant, fos capaç també de classificar les màquines segons el seu potencial.

Si recordem, COMPSs també implementava cert grau de tolerància a fallades. Aquest mecanisme també s'ha intentat millorar proposant, a més, mesurar la fiabilitat dels recursos tenint-la en compte a l'hora de planificar i poder donar, així, més confiança als Workers amb més grau de fiabilitat.

El desenvolupament d'aquestes tres propostes han marcat el rumb de treball d'aquest projecte. Al llarg d'aquest capítol explicarem tant el disseny com la implementació de cada una de les extensions presentades.

3.2 Gestió de rèpliques

Com hem comentat en l'apartat anterior, una de les extensions proposa incorporar a COMPSs la possibilitat de gestionar rèpliques dels fitxers generats durant les execucions d'una aplicació.

En aquesta secció explicarem el disseny i la implementació d'aquesta extensió, quines han estat les funcionalitats que s'han incorporat i com han estat acoblades dins el component que gestiona els fitxers, el **File Information Provider**.

3.2.1 Disseny del sistema de gestió de rèpliques

Anàlisi del model

A COMPSs el component que manté tota la informació relacionada amb els fitxers és el File Information Provider [2]. En la seva versió original, registra tots els accessos a fitxers que va detectant el Task Analyzer. Quan es detecta el primer accés a un fitxer, el FIP li assigna un identificador únic (enter) anomenat *fileId* i un objecte de tipus *FileInfo*. Aquest serà l'encarregat d'emmagatzemar tota la informació relacionada amb el fitxer com: *Locations*

[1] Nombre de processadors desocupats del recurs.
[2] Veure secció 2.5.3

o referències dels fitxers [3], nom original, les múltiples versions que el fitxer ha anat tenint, etc...

D'altra banda, si l'accés detectat pel Task Analyzer no és el primer accés realitzat sobre el fitxer, llavors comprova si el mode d'accés és READ, WRITE o READ/WRITE. Si el mode és un d'aquests 2 últims es genera automàticament una nova versió del fitxer.

Cada versió està composta per un conjunt de *locations* que indiquen on hi ha rèpliques. Cada vegada que es genera una nova versió del fitxer aquesta és totalment independent de l'anterior, deixant les *locations* de la versió anterior com a obsoletes. D'aquesta manera, les *locations* vàlides d'un fitxer seran sempre les de l'última versió existent.

A més, l'objecte FileInfo mencionat anteriorment, conté objectes de tipus *FileInstanceId*. Aquests objectes permeten realitzar *renaming* [4] a cada una de les versions dels fitxers, de manera que a partir d'un identificador de fitxer fileId, la seva versió renanomenarà el fitxer de la següent manera:

```
public FileInstanceId(int fileId, int versionId) {
  this.fileId = fileId;
  this.versionId = versionId;
  this.renaming = "f" + fileId + "v" + versionId + "\_" + tStamp + ".IT";
}
```

Aquest procediment permet que en l'accés a un fitxer en mode READ o WRITE es generin físicament també les noves versions. Per tant, quan transferim o sol·licitem el fitxer demanarem al seu FileInfo l'ultima versió i el seu *LastFileInstanceId* per tal de transferir-lo amb el nou nom, al qual podrem accedir a través del mètode *getRenaming* de l'objecte *FileInstanceId*.

A la figura 3.1 i 3.2 podem veure les classes originals del component anteriorment descrit.

Proposta de disseny - Requeriments funcionals

Una vegada detallada la forma en què COMPSs gestiona els fitxers, definirem l'estratègia i funcionalitats que caldrà implementar per tal de materialitzar l'extensió proposada. En aquest apartat intentarem establir una llista de necessitats que caldrà realitzar per tal de satisfer el conjunt de funcionalitats proposades.

Tal i com es veu en l'apartat anterior, COMPSs reanomena els fitxers recolzant-se en el benefici que això aporta a l'hora de gestionar múltiples versions de fitxers. Aquesta estratègia és bona, però suposa alguns problemes de cara al nostre desenvolupament.

En finalitzar l'execució d'una aplicació, COMPSs recull les últimes versions dels fitxers de sortida o entrada/sortida, d'allà on indiquen les seves *locations* i les porta al Master, recollint els resultats finals de l'aplicació. Després, realitza el *clean-up* de cada Worker, que consisteix a eliminar tots els fitxers reanomenats considerats com fitxers temporals o intermedis.

[3] Cadena de caràcters que identifica inequívocament un fitxer, habitualment una URI.
[4] Reanomenament aplicat als fitxers.

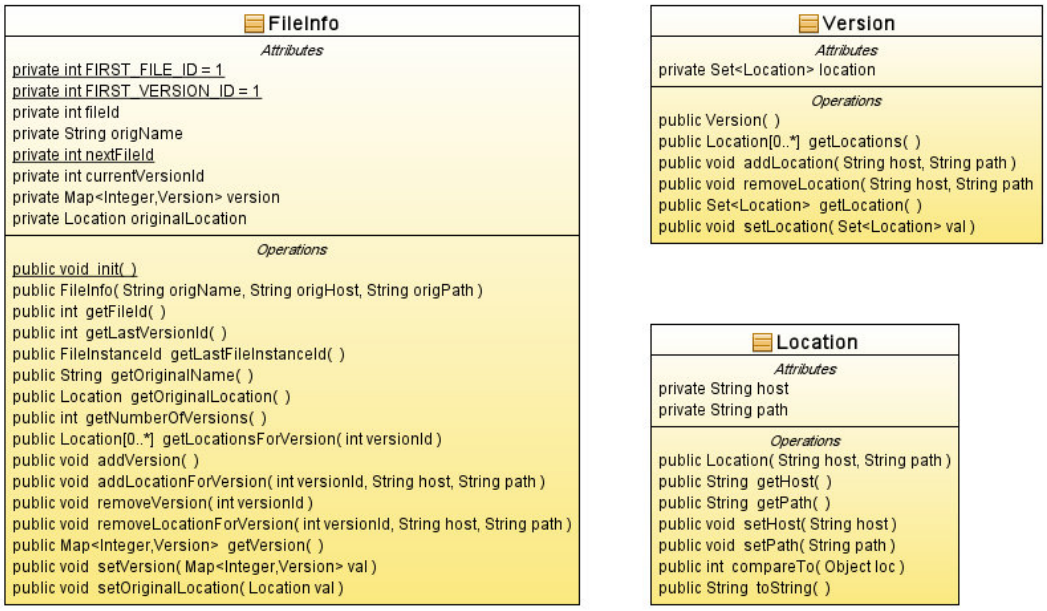


Figura 3.1: Classes dels objectes *FileInfo*, *Version* i *Location*.

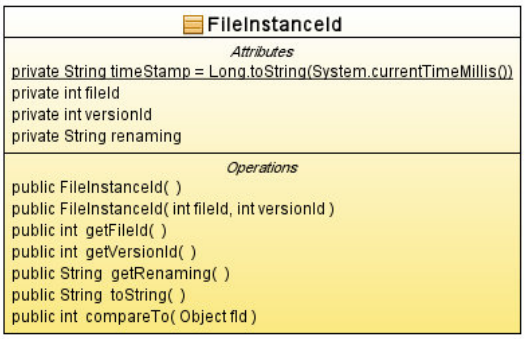


Figura 3.2: Classe de l'objecte *FileInstanceId*.

Per tant, a l'hora d'implementar la millora caldrà desenvolupar les següents subfuncionalitats:

1. Caldrà que dels fitxers d'entrada (en als que mai no s'escriurà) no se'n faci *renaming*, d'aquesta manera, els podrem localitzar amb més facilitat, ja que el seu nom serà sempre el mateix a tot arreu; així podrem evitar que el procés de neteja final aplicat a cada Worker els elimini.

2. S'haurà d'establir una sintaxi que permeti emmagatzemar les *locations*, de les quals en voldrem guardar: el nom del fitxer, l'última data de modificació d'aquest i el conjunt de *locations* on es troba el fitxer en format URI.

3. Per què el funcionament sigui transparent l'úsuari, caldrà implementar també mètodes per poder realitzar tant consultes sobre la mida dels fitxers [5] com de les dates de modificació d'aquests.

[5] Es veurà a la secció 3.4.1

4. S'haurà d'habilitar la possibilitat d'emmagatzemar, al final d'una execució, les localitzacions d'un fitxer i també la possibilitat de carregar-les a l'inici. Aquestes es guardaran al fitxer de configuració *project.xml* (que explicarem en detall a la secció 3.7.1) que és propi de cada aplicació.

5. Es desenvoluparà la possibilitat de poder gestionar fitxers d'entrada que no estiguin estrictament a la màquina Master, podent així treballar amb dades inicials externes al Grid.

6. Per últim, caldrà modificar el component principal, el FIP, per tal que a l'hora de registrar els primers accessos de cada fitxer de tipus READ, es busqui si aquest té altres rèpliques a més de l'original i les incorpori a l'objecte FileInfo de cada fitxer. A més, caldria incorporar-hi també la seva mida i data de modificació. També es comprovarà que el fitxer no hagi estat modificat des de l'última execució, ja que en cas de haver-ho estat no n'incorporaríem les rèpliques a causa de la possibilitat que no estiguin actualizades.

3.2.2 Implementació del sistema de gestió de rèpliques

Un cop vist quins seran els requeriments funcionals que haurà de complir l'extensió, només ens queda veure la implementació de les funcions que desenvolupen les subfuncionalitats esmentades. De totes maneres, no entrarem massa en detall en el codi, només en mostrarem en aquells casos en què pugui resultar interessant veure certs detalls d'implementació.

1. **Eliminació de *renaming* en fitxers de tipus IN:**

   Com hem dit, la primera de les modificacions anunciades era anul·lar el *renaming* dels fitxers de tipus IN. Així doncs, s'ha hagut de focalitzar el treball principalment en el File Transfer Manager.

   Quan el Task Analyzer detecta en el graf l'accés a un fitxer, ho notifica al FIP perquè el registri. Un cop fet, el TA envia les tasques que són lliures de dependències al Task Scheduler perquè les planifiqui. Un cop el TS ha decidit a quin recurs s'executarà la tasca, envia aquesta decisió al Job Manager que crea un *Job* o feina. Llavors, el Job Manager mira si s'ha de transferir algun fitxer al Worker seleccionat. Si és així, les transferències s'encarreguen al File Transfer Manager que a través del seu mètode *transferFiles*, en transfereix les còpies al Worker.

   En la versió original del FTM, és *transferFiles* qui transfereix la còpia reanomenada. Per tant, serà en aquest mètode on haurem de realitzar les modificacions esmentades.

   En el següent exemple veiem part del codi que s'ha hagut d'afegit al FIP per tal de poder accedir al nom original d'un fitxer des de fora del component. Aquest mètode accedeix a l'estructura *idToFile* de tipus *Map* de Java que permet relacionar l'identificador únic del fitxer amb el seu objecte de tipus FileInfo.

```
public String getOriginalName(FileInstanceId fId){
   FileInfo info = idToFile.get(fId.getFileId());
   return info.getOriginalName();
}
```

   El següent fragment és la funció *transferFiles* anteriorment mencionada, on podem veure la substitució de codi realitzada respecte al mètode original.

```
private int transferFiles(List<FileAccessId> fileAccesses, ...){
    for (FileAccessId faId : fileAccesses) { //Per cada access a fitxer
        if (faId instanceof RAccessId) {
            raId = (RAccessId)faId;
            sourceFile = raId.getReadFileInstance();
            //targetName = sourceFile.getRenaming();
            targetName = fileInformation.getOriginalName(sourceFile);
            ...
        }
        else {
            if (faId instanceof WAccessId) {
                //Es crea el fitxer reanomenat en al Worker.
            }
            else { //RW
                //Es còpia el fitxer reanomenat i versionat.
                ...
            }
            //Les noves versions de fitxer R i RW es reanomenaran.
            targetName = targetFile.getRenaming();
        }
    }
```

2. **La sintaxi de les *locations*:**

   Al segon ítem dels presentats en l'apartat de requeriments funcionals, es planteja crear una sintaxi per tal de tenir una estructura on emmagatzemar les *locations* dels fitxers. Aquesta estructura es trobarà en un fitxer de tipus XML i, per tant, la sintaxi adoptada serà la pròpia d'aquest tipus de documents.

   El codi que veiem a continuació és l'exemple de la implementació final de l'etiqueta que representarà les rèpliques.

```
<Locations>
    <File LastModDate="1292137021000" Name="file1">
        <Path>file://host1.foo.es/home/user/path1/</Path>
        <Path>file://host2.foo.es/path2/</Path>
    </File>
</Locations>
```

   Com veiem, esta format per: **el nom del fitxer**, **l'última data de modificació** i per cada una de les **referències URI** de cada una de les rèpliques. Aquesta etiqueta s'inclourà dins el fitxer *project.xml* gestionat pel ProjectManager, que s'explicarà de forma més detallada en la secció 3.7.1.

3. **Emmagatzemament del tamany dels fitxers:**

   El FIP emmagatzema en els objectes de tipus FileInfo (Figura 3.1) tota la informació referent al fitxer. En aquest apartat s'han implementat les funcions auxiliars necessàries per tal de poder emmagatzemar-hi la mida i la data de modificació. Com que cada versió pot tenir diferents valors d'aquestes dades, ha calgut afegir a la classe Versions, atributs i mètodes per tal de poder guardar-les.

Com hem dit, cada versió és dins l'objecte FileInfo i a causa d'això ha calgut també implementar mètodes que, donada una versió, permetessin tant afegir com obtenir mides i dates de modificació dels fitxers.

Modificacions fetes en la classe FileInfo:

- **getSizeForVersion(int versionId):** donada una versió, retorna la mida del fitxer.
- **addSizeForVersion(Version v, long Size):** donada una versió, afegeix la mida del fitxer a l'objecte de tipus Version.
- **getLastModForVersion(int versionId):** donada una versió, retorna la data de modificació del fitxer.
- **addLastModForVersion(Version v, long modDate):** donada una versió, afegeix la data de modificació a l'objecte de tipus Version.

Modificacions fetes en la classe Version:

- **addSize(long size):** afegeix la mida del fitxer.
- **getSize():** retorna la mida del fitxer.
- **addLastMod(long size):** afegeix la data de modificació del fitxer.
- **getLastMod():** retorna la data de modificació del fitxer.

Modificacions del component File Information Provider:

En el component FIP és on s'han implementat gran part de les modificacions que permeten gestionar les rèpliques. Per aquesta finalitat s'han implementat el següent conjunt de mètodes:

Mètodes afegits:

- **addSize(FileInstanceId fId,List<Long> sizeModDate):** afegeixen a l'objecte FileInfo del fitxer la seva mida i data de modificació.
- **getSize(FileInstanceId fId):** retorna la mida del fitxer a partir del seu objecte FileInstanceId.
- **getFileSizeAndLastMod(String fileName,String host,String path):** demana la mida i la data de modificació consultant directament el fitxer. Aquest mètode el veurem de forma detallada a la secció 3.4.1
- **storeLocations():** emmagatzema les *locations* dels fitxers al *project.xml* a través del ProjectManager.
- **getOriginalName(FileInstanceId fId):** retorna el nom original d'un fitxer. Explicada a l'apartat *Eliminació de renaming en fitxers IN.*

Gran part de les modificacions fetes han estat realitzades a *registerFileAccess*, el mètode del FIP que registra els fitxers. Al següent apartat desglossarem pas a pas l'ampliació realitzada a partint del mètode original.

```
public FileAccessId registerFileAccess(String fileName, ...){
    FileInfo fileInfo;
    String locationKey = fileName + ":"+ host + ":"+ path;
    Integer fileId = nameToId.get(locationKey);
    //Primer access al fitxer
    if (fileId == null){
        //Actualitzem els mappings
        fileInfo = new FileInfo(fileName, host, path);
        fileId = fileInfo.getFileId();
        nameToId.put(locationKey, fileId);
        idToFile.put(fileId, fileInfo);
        //Inserció del nou codi
        //S'informa al File Transfer Manager sobre el nou fitxer
        ...
    }//Si ja s'ha accedit prèviament al fitxer...
    else {
        fileInfo = idToFile.get(fileId);
    }
}
```

El mètode anterior crea una clau de localització a partir del nom, màquina i ruta on es troba el fitxer. Després intenta accedir al mapa *nameToId* que relaciona l'identificador únic de fitxer amb aquesta clau. Si no existeix és perquè és el primer accés que s'hi realitza. En aquest cas, es crea l'objecte FileInfo i s'actualitzen els mapes de dades del component. És just en aquest punt on incorporem el conjunt de noves funcionalitats.

El següent fragment serà l'encarregat de consultar la mida i data de modificació del fitxer, a la vegada que consulta també al fitxer *project.xml* l'última data de modificació del fitxer.

```
//Consulta la mida actual de fitxer i la seva data de modificació.
List<Long> sizeLastModSet = getFileSizeAndLastMod(fileInfo.getOriginalName(),...);
long size = sizeLastModSet.get(0);
long lastMod = sizeLastModSet.get(1);
//Data anterior de modificació del fitxer
long prev_lastMod = projManager.getFileLocLastMod(fileInfo.getOriginalName());
```

Posteriorment, si el fitxer té *locations* extra (rèpliques) a banda de l'original, llavors si la data de modificació del fitxer consultada és la mateixa que l'emmagatzemada en l'ultima execució, significa que el fitxer no ha estat modificat entre execucions. Només llavors es permet afegir aquestes rèpliques a l'objecte *fileInfo* del fitxer. A continuació podem veure el codi d'aquesta part:

```
if(projManager.getFileLocations(fileInfo.getOriginalName()) != null
    && (lastMod == prev_lastMod)){
    //Afegim les locations de les rèpliques
    String l = null;
    List<String> locat = projManager.getFileLocations(fileInfo.getOriginalName());
```

```
    Iterator<String> i = locat.iterator();
    ...
    while (i.hasNext()) {
        l = i.next();
        try{
            URI locURI = new URI(l);
            locations.add(new Location(locURI.getHost(),"/"+locURI.getPath()));
            //Afegim al FIP cada location de tipus IN
            fileInfo.addLocationForVersion(fileInfo.getLastVersionId(),
            locURI.getHost(),"/"+locURI.getPath());
        }
        catch (...) {
        ...
}
```

Per últim, actualitzarem la mida i la data del fitxer a l'objecte *fileInfo*, afegint-los a la seva última versió, que en aquest cas serà la primera, perquè estem registrant el primer accés d'un fitxer.

```
//Afegim al FIP la mida de la versió del fitxer
fileInfo.addSizeForVersion(fileInfo.getLastVersionId(),size);
//Afegim al FIP l'última data de modificació de la versió del fitxer.
fileInfo.addLastModForVersion(fileInfo.getLastVersionId(),lastMod);
```

4. **DataNodes:**

   Aquest últim punt de l'extensió busca dotar a COMPSs de la possibilitat de gestionar fitxers d'entrada que no es trobin inicialment a la màquina Master i que, per tant, funcionin com a *DataNodes*. [6]

   La majoria de les vegades, els protocols de transferència de fitxers verifiquen l'accés entre nodes a través d'un procés de *login* [7] que demana nom d'usuari i contrasenya. Aquest és el cas de COMPSs, que utilitza per defecte SCP [8] com a protocol de transferència a través de l'adaptador proporcionat per JavaGAT.

   En aquest apartat definirem la sintaxi utilitzada per declarar un DataNode. Aquest anirà també inclòs dins el fitxer *project.xml* a mode d'etiqueta. La sintaxi serà la següent:

```
<DataNode Name="host1.foo.es">
    <User>username</User>
</DataNode>
```

   La implementació i utilització d'aquesta funcionalitat dins el codi de COMPSs, es troba al mètode **getFileSizeAndLastMod** del FIP, que veurem en detall a la secció 3.4.1.

[6] Nodes que no són Workers i d'on s'agafaran dades d'entrada.
[7] Procés mitjançant el qual es controla l'accés individual a un sistema.
[8] Secure Copy Protocol.

3.3 Planificador

En aquesta secció presentarem les propostes per millorar el planificador original. Explicarem el nou planificador i en presentarem el disseny i la implementació duta a terme. Per acabar, analitzarem cada una de les funcionalitats auxiliars que ha calgut desenvolupar per poder proporcionar la informació necessària perquè pugui avaluar de forma correcta.

3.3.1 Anàlisi del planificador inicial

A la secció 3.1 hem vist que el planificador original de COMPSs tenia certs punt millorables. Entre d'altres, un d'ells era que analitzava per cada conjunt de fitxers d'entrada quin era el recurs, d'entre els disponibles, que tenia el màxim nombre de fitxers. D'aquesta forma en transferia el mínim nombre possible, encara que això no garantia transferir la mínima quantitat de dades i, per tant, minimitzar el temps de transferència.

L'estratègia per trobar el millor recurs, es troba en el mètode *assignTaskToBestResource* del Task Scheduler. En la versió inicial d'aquest mètode quedava separada en dues fases: ***Scoring*** [9] ***i cerca del recurs amb la puntuació més alta.***

Aquest mètode busca per cada un dels paràmetres de la tasca que siguin de tipus fitxer [10] i n'agafa el seu *FileInstanceId* corresponent, depenent de si és READ, WRITE o READ/WRITE. A través d'aquest identificador fa una petició per demanar totes les seves *locations* al FIP utilitzant la funció *getLocations(FileInstanceId fId)* publicada a la interfície *fileLocation*.

Per cada una d'aquestes *locations* (URIs) es comprova el seu *host*, si coincideix amb algun dels recursos dels quals es disposa, s'hi anota un punt. D'aquesta manera, al final del procés obtindrem una puntuació que indicarà quants dels fitxers necessaris per executar la tasca té cada un dels nostres recursos disponibles. A continuació podem veure un exemple del codi simplificat.

```
//Per cada paràmetre de la tasca
for (Parameter p : params) {
    //Si p és un fitxer
    if (p instanceof FileParameter) {
        FileParameter fp = (FileParameter)p;
        FileInstanceId fId = null;
        switch (fp.getDirection()) {
            case IN:
                fId = raId.getReadFileInstance();
                ...
            }
        //Si fId != null el fitxer és IN o INOUT
        if (fId != null) {
            Set<Location> locs = fileLocation.getLocations(fId);
```

[9] Procés de puntuació de recursos.
[10] Si són de tipus bàsic no s'han de transferir.

```
//Per cada location calculem la seva puntuació
for (Location l : locs) {
    String host = l.getHost();
    if ((score = hostToScore.get(host)) == null) {
        score = new Integer(0);
        hostToScore.put(host, score);
    }
    hostToScore.put(host, score + 1);
...
```

La segona fase busca el valor més gran del mapa de puntuacions *hostToScore* trobant el recurs que té el màxim nombre de fitxers i que, per tant, pot ser considerat com el millor recurs per enviar-hi la tasca.

```
//Seleccionem el recurs amb més puntuació
String bestResource = null;
int bestScore = 0;
    for (Map.Entry<String,Integer> e : hostToScore.entrySet()) {
        String host = e.getKey();
        Integer score = e.getValue();
        if (score > bestScore){
            bestResource = host;
            bestScore = score;
        }
    }
```

Després de veure que aquesta solució no és del tot correcta, es va decidir desenvolupar un planificador que avalués a partir de la informació extreta directament del Grid, de la forma més real i actualitzada possible.

3.3.2 Disseny i implementació del planificador

Com hem vist, el planificador original de COMPSs és un planificador estàtic i no és capaç de minimitzar el temps implicat en les transferències. Això és a causa de no ser un planificador retroalimentat amb dades directes del Grid.

És per aquest motiu que es planteja desenvolupar un planificador que sigui capaç de trobar el millor recurs buscant quin d'ells minimitza el temps total d'execució d'una tasca. Aquest planificador treballarà amb dades extretes del Grid, de la mateixa manera que ho fa el proposat a l'article: *Grid Superscalar and job mapping on the reliable grid resources* [17]. Per fer-ho, doncs, caldrà definir quins seran els paràmetres que determinaran aquest temps.

Temps de transferència

Hauria de ser possible poder realitzar la predicció del temps que es trigaria a transferir tot el conjunt de fitxers d'una tasca. Aquest paràmetre és important, ja que ens proporcionarà

la predicció del temps que hauríem d'esperar per poder començar l'execució en el cas d'escollir aquell node.

Per poder mesurar aquest temps de transferència necessitarem saber:

- La velocitat de la xarxa entre l'origen de cada fitxer i el candidat a destí.
- La mida de cada fitxer.

Aquestes necessitats obligaran a desenvolupar, a més, algunes subfuncionalitats suplementàries: els mètodes necessaris per calcular la mida dels fitxers i una forma per representar i mantenir actualitzades les velocitats de la xarxa. A aquest temps l'anomenarem ***TrfPrediction***, i el seu disseny i implementació es veurà als apartats 3.4.1 i 3.5.

Temps d'espera en cua

Per altra banda COMPSs, en la seva versió original, incorpora una funcionalitat anomenada *prescheduling*, que permet al planificador enviar tasques als recursos que en aquell moment no tenen *slots* disponibles per poder executar.

Quan el TS envia la decisió al Job Manager aquest inicia, en el cas de ser necessari, la transferència dels fitxers de la tasca. Si en acabar encara no hi ha *slots* disponibles al Worker, la feina queda esperant a una cua de tasques pendents del *host* anomenada *pending queue*. D'aquesta manera, quan una tasca acaba, despertarà un procés que mirarà si n'hi ha d'altres en espera pendents de disposar de processador.

Així doncs, al temps d'espera a causa de les transferències cap a un Worker, cal sumarhi també el temps d'espera en cua. Aquest temps serà el que haurà d'esperar una tasca des del moment en què entra a la cua d'espera per *slot* fins al moment en què se n'hi assigna un. Aquest temps l'anomenarem ***WaitTimeInQueue*** i el seu disseny i implementació els veurem a la secció 3.4.3.

Temps d'execució

Finalment ens quedaria per definir l'últim paràmetre de la fórmula que modela el planificador. Quan a una tasca se li ha assignat un *slot*, l'únic temps que queda saber per poder calcular el temps total d'execució, és el temps que trigarà aquella tasca a alliberar el *slot*, és a dir, en definitiva, el temps d'execució d'aquella tasca. Aquest temps l'anomenarem ***TypExecTime***.

Per tal de poder-lo predir s'anirà mesurant el temps d'execució de cada un dels mètodes d'una aplicació enviats a cada node del Grid i s'aniran emmagatzemant en un històric propi de cada aplicació. D'aquesta manera, a mesura que realitzem execucions, les prediccions seran cada vegada més acurades. El disseny i la implementació del càlcul d'aquest temps el veurem en l'apartat 3.4.2.

Per tant, tenint en compte tot això, la fórmula del planificador aplicada a cada un dels recursos seria:

Per cada tasca t
Per cada recurs r

$$ExecutionTime_r = TrfPrediction_{t,r} + WaitTimeInQueue_r + TypExecTime_{t,r}$$

Com podem veure, el tipus de tasca és necessari per poder calcular tant la predicció del temps de transferència com la del temps d'execució. Això és perquè cada tasca té el seu conjunt de fitxers propi i també el seu temps d'execució típic, que pot ser diferent en cada un dels recursos.

Fiabilitat dels recursos

Per altra banda es va voler que el planificador disposés també d'un paràmetre per tal de poder tenir en compte les màquines més fiables a l'hora de planificar. Encara que el càlcul d'aquest paràmetre no el veurem en aquest apartat [11], aquí en mostrarem la seva utilització.

Per poder ponderar la fiabilitat, es mantindrà un coeficient associat a cada un dels nodes, que s'emmagatzemarà al llarg de les execucions en un històric propi de cada aplicació.

Inicialment es pensà que el valor de fiabilitat, al qual anomenarem ***AvailRate***, es relacionés de forma directament proporcional a *ExecutionTime*; després es va veure que relacionar-los de forma directa no era una bona solució.

Això venia provocat perquè consideràvem com a millor recurs el que tenia el valor més petit de *ExecutionTime* i, per tant, si es multiplicava un valor alt de temps d'execució (dolent), per un valor de fiabilitat baix, aquest temps disminuïa i, com a conseqüència, l'anàlisi que en treia el planificador era que el temps millorava quan això no era cert.

Propostes de planificació

Per tant, això resultà totalment incoherent i calia trobar una manera de poder incorporar correctament el paràmetre de fiabilitat a la fórmula original. A continuació es presenta la primera proposta plantejada.

$$Score_r = ExecutionTime_r * AvailRate_r \;\; on \; 0 \leq AvailRate_r \leq 1 \rightarrow Incorrecte$$

Una altra opció va ser fer-ho de forma inversament proporcional tal i com s'exposa en la següent fórmula, on els recursos amb temps més baixos donen puntuacions més altes:

$$Score_r = \left(\frac{1}{ExecutionTime_r} \right) * AvailRate_r \;\; on \; 0 \leq AvailRate_r \leq 1$$

Aquesta formula és correcta en la teoria, però pot no resultar-ho a la pràctica, ja que per temps d'execució grans el quocient podria produir valors molt petits que podrien acabar-se arrodonint a zero i generant falsos empats entre recursos.

Com a solució final es va decidir abstraure dels recursos els valors d'*ExecutionTime* sotmetent-los a un rànquing de classificació ascendent [12]. D'aquesta manera es podien classificar els recursos de forma que el valor més baix del rànquing, l'1, indiqués el pitjor lloc i a partir d'aquí anés augmentant de forma ascendent.

[11] Es veurà a l'apartat 3.6.1, tolerància a fallades.
[12] Classificació per posicions ordenades de forma ascendent on el valor més gran és el millor element.

Això permetia evitar els problemes presentats en la segona fórmula i aconseguia situar tots els recursos en una equidistància mútua. D'aquesta manera, quan s'apliqués el coeficient de fiabilitat sobre la classificació de recursos, aquesta podria variar segons el % de fiabilitat de cada un d'ells.

A continuació podem veure el disseny final del planificador:

Per cada tasca t
  Per cada recurs r
    $ExecutionTime_r = TrfPrediction_{t,r} + WaitTimeInQueue_r + TypExecTime_{t,r}$
  Fi Per
  Ordenem de gran a petit $ExecutionTime_r$
  Per cada valor de ExecutionTime e
    $Rank_r$ = *S'assigna un rank a e*
  Fi Per
  Per cada recurs r
    $Score_r = Rank_r * AvailRate_r$
  Fi Per
  Ordenem de gran a petit $Score_r$
  $millorRecurs_t = max\{Score\}$
Fi Per

En cas d'empat, s'assignaria a tots els recursos el mateix valor de rànquing i es desempataria triant-ne un aleatòriament. D'aquesta manera s'obliga al planificador a provar entre els diferents recursos que es troben en igualtat de condicions.

Això és molt beneficiós, ja que fomenta l'execució en quantes màquines millor i, per tant, millora l'obtenció de dades per generar l'històric, millorant també la planificació en futures execucions.

A continuació, a la figura 3.1 podem veure alguns exemples de com actuaria el planificador en cas de no disposar d'històric previ [13]:

Recursos	MethodId	TP	WT	ET	AvailRate	Rank	Score
host1	**1**	**0.9s**	**0s**	**0s**	**1.0**	**1**	**1**
host2	1	0.9s	0s	0s	1.0	1	1
host3	1	0.9s	0s	0s	1.0	1	1
host4	1	0.9s	0s	0s	1.0	1	1

Taula 3.1: Exemple de planificació sense històric.

Com podem veure en aquest cas, l'únic valor diferent a zero serà el temps de transferència. Això és a causa que l'usuari haurà definit prèviament les velocitats inicials de xarxa per defecte de cada un dels recursos [14].

La resta de valors seran tots nuls, ja que no hi haurà valors previs ni de temps d'execució ni de generació de cues als recursos. Per tant, en cas de tenir, per exemple, un Grid amb una xarxa homogènia, podria donar-se empat entre recursos durant les decisions inicials preses pel planificador. A més, a base d'anar acumulant informació, s'aniran obtenint valors dels altres paràmetres restants que quedaran guardats a l'històric en finalitzar l'aplicació.

[13] On TP = TrfPrediction | WT = WaitTimeInQueue | ET = TypExecTime.
[14] Es veurà a la secció 3.7.2.

A la figura 3.2 podem veure un exemple de planificació partint, aquesta vegada, d'un històric previ on els recursos ressaltats són els finalment escollits pel planificador:

Recursos	MethodId	TP	WT	ET	AvailRate	Rank	Score
host1	1	0.11s	1504.1484s	0s	0.9	2	1.8
host2	**1**	**0.10s**	**1316.8849s**	**0s**	**1.0**	**3**	**3**
host3	1	0.15s	1254.6338s	0s	0.7	4	2.8
host4	1	0.09s	8451.267s	0s	1.0	1	1

Taula 3.2: Exemple de planificació amb històric previ.

Implementació del planificador

La implementació del planificador s'ha realitzat, pràcticament en la seva totalitat, dins el Task Scheduler. Tal com s'ha explicat a la secció 3.3.1, aquest component allotja el mètode *List assignTaskToBestResource(Task t, List<String> resources)* que és l'encarregat de realitzar l'avaluació de recursos.

Aquest mètode rep una tasca i un conjunt de recursos i retorna una llista de dos elements: una cadena i una llista d'objectes *FileInstanceId* que contenen el nom del millor recurs per enviar-hi la tasca i la llista de fitxers que s'hi haurà de transferir.

Aquestes dades s'envien al Job Manager a través de la seva funció *sendJob* publicada a la seva interfície i és l'encarregada d'iniciar la transferència dels fitxers, en el cas que sigui necessari, i d'enviar la feina al Worker.

La figura 3.3 mostra la classe final modificada del Task Scheduler.

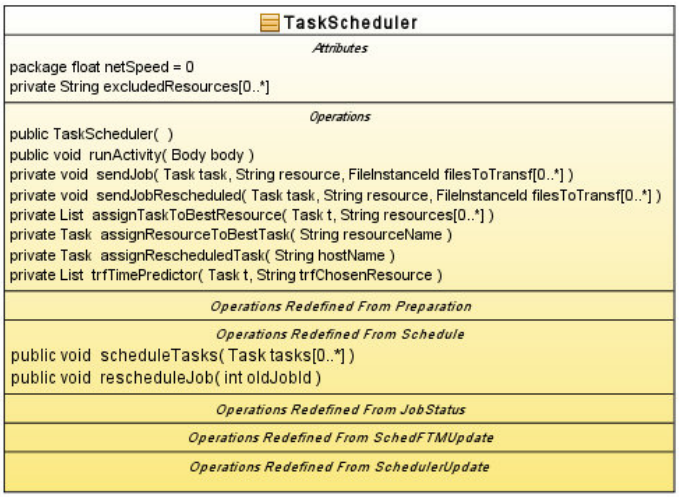


Figura 3.3: Classe simplificada del component Task Scheduler.

A causa que la implementació del mètode *assignTaskToBestResource* conté gran quantitat de codi, s'ha preferit transmetre'n només la seva essència, que queda explicada a l'apartat 3.3.2, propostes de planificació.

3.4 Extracció de dades necessàries per al planificador

Desenvolupar un planificador de les característiques esmentades implica haver de mantenir actualitzat en tot moment el conjunt de dades extretes del Grid per tal que les decisions siguin preses de la forma més acurada possible; és per aquest motiu que aquest és un dels punts crítics del desenvolupament, ja que en dependrà gran part del rendiment esperat del planificador.

En aquesta secció, doncs, veurem de forma detallada com s'ha realitzat l'extracció i mesura de les dades necessàries per alimentar el planificador.

3.4.1 Predicció del temps de transferència

Com hem vist, la predicció del temps de transferència és un dels elements clau. Com s'explica a la secció 3.3.2, s'utilitza per obtenir el valor del paràmetre *TrfPrediction*, el temps de transferència estimat per enviar els fitxers d'una tasca al recurs determinat. Aquest mètode l'anomenarem *trfTimePredictor* i serà utilitzat en la funció *assignTaskToBestResource*, que serà l'encarregada d'avaluar les decisions del planificador. Per aquest motiu anirà implementada al Task Scheduler, tal i com indica la Figura 3.3.

La invocació d'aquest mètode es farà dins d'*assignTaskToBestResource* passant-hi com a paràmetres: la tasca a planificar i el recurs candidat. Per a cada un dels paràmetes de la tasca se'n consultarà el tipus i es comprovarà si és un fitxer, ja que només tindrà sentit realitzar la predicció en cas que el paràmetre ho sigui.

Per tant, si és de tipus fitxer i és IN o INOUT llavors té una o més referències de localització (URI) que es demanaran al FIP, als fitxers de tipus sortida (OUT) tampoc no tindrà sentit fer-hi cap predicció, ja que es generaran directament als Workers com a resultat final d'una tasca. Per tant, a l'hora de realitzar la predicció ens interessarà fer-la només dels fitxers de tipus IN i INOUT.

En l'apartat de gestió de rèpliques, hem explicat que les de fitxers de tipus IN s'emmagatzemaran al *project.xml* [15]; per tant, si el fitxer és d'aquest tipus i té rèpliques definides, (rèpliques que hagin estat definides manualment o que ja existeixin a causa d'execucions prèvies de l'aplicació) es comprovarà per cada URI si el seu nom de host fa referència a la màquina candidata. Si és així, significa que el fitxer ja existeix a la màquina i no caldrà transferir-lo.

En cas de no existir-hi l'afegirem a la llista de fitxers a transferir *fitxersTransferir* i se'n farà la predicció. Per fer-la, es demanarà al FIP la mida del fitxer que, recordem, s'emmagatzema gràcies a l'extensió de la gestió de rèpliques. Per altra banda, es demanarà a la matriu de velocitats [16] la velocitat de l'enllaç entre l'origen del fitxer i el destí en el qual es vol transferir que, en aquest cas, serà la màquina candidata *chosenResource*.

Llavors es realitzarà la predicció a partir de la següent fórmula:

$$\text{Predicció} = \text{Predicció} + \left(\frac{tamanyFitxer\ (bytes) * 8}{velocitatEnllac\ (Mbits/s) * 1000000} \right) \quad (segons)$$

[15] Veure secció 3.7.1
[16] Veure secció 3.5

A continuació podem veure la versió simplificada del codi del mètode:

```
List trfTimePredictor(Task t, String chosenResource) {
  Per cada paràmetre p de la tasca
    Selector(p.Tipus){
      cas FITXER:
        Si (p != OUT)
          locs = fileLocation.obtenirLocations(...)
        Fi Si
        Si (p == IN) && Hi ha rèpliques definides
          Per cada element l de locs
            host = l.obtenirHost()
            path = l.obtenirPath()
            Si (host == chosenResource) && (path == workingDir)
              fitxerExisteix = true
              sortir bucle
            Fi Si
          Fi Per
        Fi Si
        Si (!fitxerExisteix) && (paràmetre != OUT)
          //Afegim fitxer a llista de fitxers a transferir
          fitxersTransferir.afegir(...)
          //Predicció del temps de transferència
          tamanyFitxer = fileLocation.getSize(...)
          //S'agafa el host de la primera de les rèpliques
          source = locs.get(0).obtenirHost()
          velocitatEnllaç = matriuVelocitats(source, chosenResource)
          predicció = predicció + (tamanyFitxer*8)/(velocitatEnllaç*1000000)
        Fi Si
      Fi cas FITXER
      cas DEFECTE:
        //El paràmetre és de tipus bàsic, no es fa res
      Fi cas DEFECTE
    Fi Selector
  Fi Per
  llistaRetorn.afegir(0,predicció)
  llistaRetorn.afegir(1,fitxersTransferir)
  retorn llistaRetorn
}
```

Com podem veure, la funció retorna una llista anomenada *llistaRetorn* on s'afegeixen la predicció i la llista d'objectes *FileInstanceId* corresponents a cada un dels fitxers que hauran de ser transferits.

Ara, a la següent secció veurem com s'ha mesurat el temps mig d'execució de les tasques.

3.4.2 Mesura del temps mig d'execució

Tal i com s'ha explicat a la secció 2.5.5, l'usuari és qui determina quines seran les funcions de l'aplicació que s'executaran al Grid. Cada una d'elles es definirà a l'interfície i tindrà el seu identificador de mètode únic o *methodId*, consultable com a atribut en cada objecte de tipus Task (tasca).

D'aquesta manera, per calcular el temps mig d'execució de cada tipus de tasca i recurs, calia capturar dos instants de temps. El primer, en iniciar la feina, és a dir, en el moment on el Job Manager l'envia al recurs seleccionat prèviament pel Task Scheduler i el segon en rebre la notificació, a través de JavaGAT, que la feina ha acabat correctament. En cas d'acabar amb errors, el temps no es tindrà en compte.

Per fer-ho, es mantindrà una relació de feines començades i es guardarà en una estructura que relaciona l'identificador *JobId* amb el temps d'inici de cada una d'elles. D'aquesta manera, la notificació o *callback* generada al final d'una feina retorna l'estat i identificador d'aquesta. Si l'estat és correcte podem calcular llavors el temps d'execució de la feina a través de la següent fórmula:

$$JobTime = JobTime_f + JobTime_i \ \ (segons)$$

A mesura que es van aconseguint valors, van servint com a mostra per anar calculant i afinant el temps mig d'execució. Per fer-ho, s'ha implementat al Task Scheduler un mètode anomenat *addHostExTimeMeanValue*, que pren com a paràmetres d'entrada la màquina on s'ha executat la feina, el temps d'execució i el seu identificador.

En la següent figura podem veure la dinàmica de funcionament del sistema.

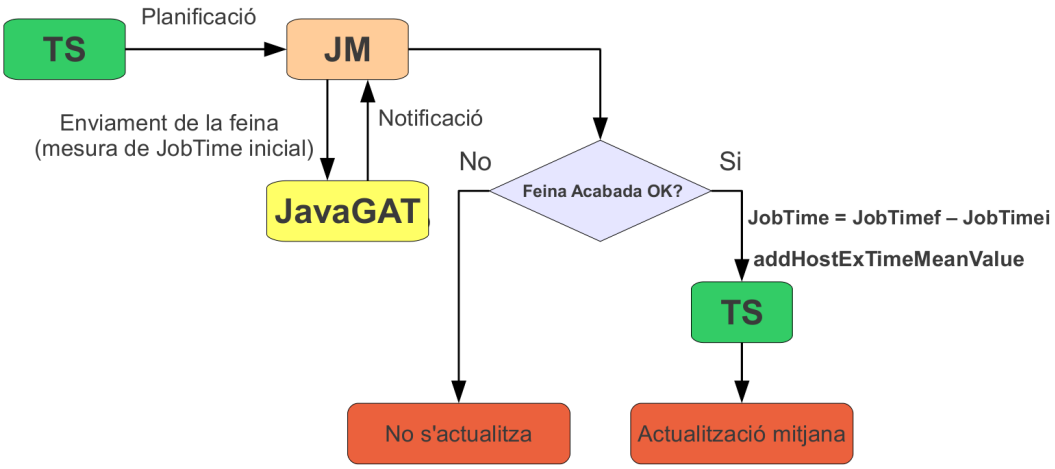


Figura 3.4: Dinàmica de la mesura del temps mig d'execució.

Així doncs, el mètode *addHostExTimeMeanValue* recalcula el temps mig de cada tasca i host, tal com van arribant noves mostres, a partir de la següent fórmula:

$$MitjanaActual = \frac{(\sum_{i=1}^{N} mostres_i) + mostraActual}{N+1} \ \ on\ N = \#mostres$$

Per acabar, en finalitzar l'execució de l'aplicació, aquestes dades quedaran emmagatzemades en el fitxer d'històric *historical.xml* propi de l'aplicació [17], mantenint una sintaxi de tipus XML com la següent:

```
<MeanExecTime>
    <TaskType Id="1">
        <Worker Name="host1.foo.es">10.5</Worker>
    </TaskType>
    <TaskType Id="2">
        <Worker Name="host2.foo.es">1.53</Worker>
    </TaskType>
    ...
</MeanExecTime>
```

3.4.3 Mesura del temps d'espera en cua

Per tal d'estimar el temps que s'haurà d'esperar una feina per poder accedir al processador d'un recurs que en aquell moment no disposa de *slots* lliures, ha calgut dissenyar i desenvolupar aquesta funcionalitat de la manera més eficient possible, ja que serà de vital importància que el càlcul d'aquest valor es realitzi de forma ràpida i pugui mantenir-se actualitzat. Això és important perquè el planificador pugui decidir amb les dades més reals possibles.

El temps d'espera en cua serà el temps previst d'execució de cada una de les feines que ja són a la cua del recurs, més el temps previst perquè el recurs alliberi un *slot*.

Per tal de calcular el temps d'espera en cua, s'ha implementat al Task Scheduler un mètode anomenat *updateWTCounters*, que pren com a paràmetres d'entrada la màquina on s'ha executat la feina, l'identificador de mètode i el valor de l'incrementador/decrementador.

D'aquesta manera, cada vegada que una feina no es pot executar a causa que al recurs on ha estat assignada no té *slot* lliure, entra a la cua d'espera del recurs i llavors es crida al mètode ***updateWTCounters(host,methodId,(+1))***.

Aquest incrementa el comptador que manté el registre del nombre de feines de cada un dels tipus de mètode que hi ha a la cua del recurs. D'aquesta manera es porta un recompte de quantes feines hi ha de cada tipus de mètode. La figura 3.5 n'il·lustra la dinàmica de funcionament.

D'altra banda, cada vegada que una feina surt de la cua per passar a executar-se a un *slot* que hagi quedat lliure gràcies a haver acabat alguna tasca, es crida de nou al mètode, però aquesta vegada amb els següents paràmetres ***updateWTCounters(host,methodId,(-1))***, eliminant la feina de la cua i dels comptadors.

Aquest mètode, a més de controlar els comptadors, actualitza els elements de la cua i recalcula el temps d'espera. Aquest temps es calcula a partir del temps d'execució típic de cada mètode, vist en l'apartat anterior.

D'aquesta manera, multiplicant el nombre de feines de cada tipus pels seus temps d'execució típics sabrem el temps aproximat que trigaran a executar-se cada un d'aquests mètodes i, per tant, sumant tots aquests temps, podrem saber el temps que haurà d'esperar

[17] Veure secció 3.7.3

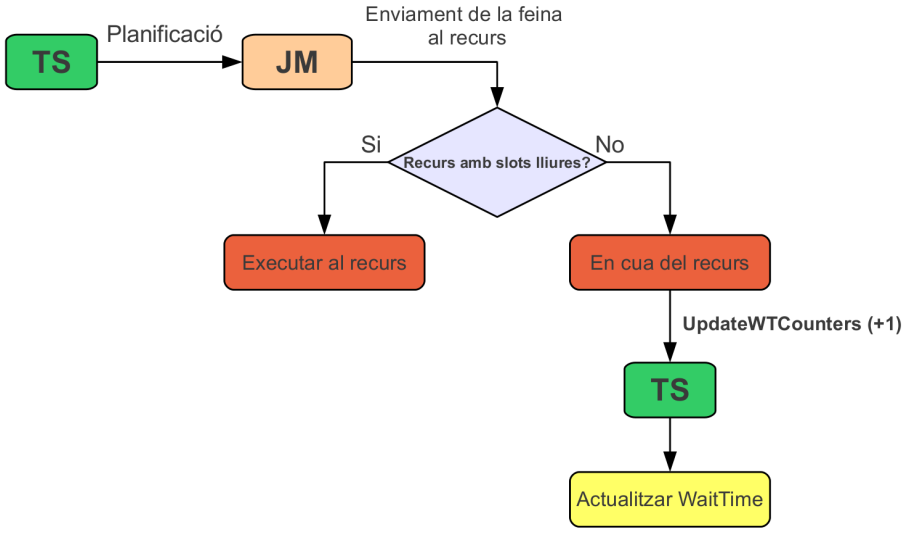


Figura 3.5: Dinàmica de la mesura del temps d'espera en cua.

a la cua una feina que just hi acaba d'entrar. L'algorisme seguit per aproximar aquest valor és el següent:

Per cada methodId m
Obtenim el comptador c de feines del mètode m
WaitTime = WaitTime + (c * obtenirTExec(recurs,m))
Fi Per

Com es pot veure, en aquest càlcul no s'ha afegit el terme que calcula el temps previst perquè el recurs alliberi un *slot*. Tot i que inicialment es plantejà i teòricament és correcte incloure'l; a la pràctica, implementar aquest càlcul provocava un increment en la complexitat del mètode i l'alentía considerablement.

Per poder calcular aquest valor calia mantenir en tot moment un registre de les feines que es trobaven en execució a cada un dels recursos, registrar el temps d'inici de cada una d'elles, fer la diferència amb el temps d'execució estimat de cada una i buscar el mínim d'aquests temps. És per aquest motiu que la implementació final s'ha realitzat tenint només en compte el temps d'espera en cua.

3.5 Mesura i actualització de la velocitat de xarxa

Tal i com s'ha presentat a la secció 3.3.2, per poder predir el temps de transferència ens cal saber, entre altres coses, la velocitat de xarxa entre l'origen de cada fitxer i el seu destí. Ara bé, per poder disposar d'aquestes velocitats, abans cal tenir alguna manera per poder representar-les. Aquest és l'aspecte que tractarem en aquesta secció.

Per permetre al planificador disposar d'aquestes dades, ha calgut dissenyar una forma perquè l'usuari pogués indicar les velocitats inicials de xarxa de cada un dels nodes a través del fitxer on es detallen les característiques d'aquests, el *resources.xml*. Per fer-ho s'ha aprofitat un dels camps no utilitzats d'aquest fitxer, el camp *<NetworkAdaptor>*, que veurem més endavant a la secció 3.7.2.

De totes maneres, cada un dels recursos del Grid tindrà en aquest fitxer una etiqueta XML definida com la següent, indicant la velocitat de l'adaptador de xarxa de cada node:

```
<NetworkAdaptor>
  <NetworkSpeed>547.2</NetworkSpeed>
</NetworkAdaptor>
```

D'aquesta manera, a partir d'implementar mètodes addicionals al ResourceManager es permet carregar aquestes dades en una matriu que es troba al Task Scheduler i que anomenarem **Matriu de velocitat de xarxa**. El mètode que més hi haurà d'accedir serà el *trfTimePredictor* que, recordem, és l'encarregat de predir el temps de transferència dels fitxers.

Un detall important a considerar és que a l'hora de carregar aquests valors, s'escull com a velocitat de l'enllaç la més petita d'entre els dos punts que connecta. Aquest detall ha de ser considerat en motiu de la possiblitat de trobar nodes interconnectats a través de xarxes asimètriques i que, per tant, mantenen diferents velocitats a l'hora de rebre i enviar informació cap a un node determinat.

D'aquesta manera i, a mode d'exemple, podríem trobar-nos amb una matriu inicial definida per l'usuari com la següent:

Recursos	host1	host2
host1	91.2	91.2
host2	78.8	504.3

Taula 3.3: Exemple de matriu inicial de velocitat de xarxa (Mbps).

Com podem observar, aquesta primera solució és realment bàsica, en tractar-se d'un model estàtic que no té en consideració cap de les possibles fluctuacions en la velocitat de la xarxa. Justament per aquest motiu es va decidir ampliar aquesta funcionalitat permetent l'actualització dinàmica de la matriu a mesura que es transfereixen fitxers entre nodes.

3.5.1 Actualització dinàmica de la velocitat de xarxa

Com diem en l'apartat anterior, la segona fase de desenvolupament es planteja com un model dinàmic que sigui capaç d'actualitzar les dades a mida que vagin sorgint transferències entre nodes del Grid. D'aquesta manera s'espera obtenir un comportament més realista del planificador permetent-lo disposar de dades com més real i actualitzades sigui possible.

Tot i que aquest planteig era inicialment correcte, es veié que calia decidir el moment en què s'actualitzarien les dades d'aquesta matriu. Cal tenir en compte la naturalesa de components de COMPSs i que l'intercanvi d'informació entre aquests requereix l'existència d'interfícies entre ells. A més, transferir dades entre components pot penalitzar força el rendiment global del sistema.

Recordem que, per altra banda, la màxima seguida durant el desenvolupament era no modificar l'arquitectura inicial de COMPSs, intentant mantenir sempre les interfícies inicials el més intactes possible.

És per aquest motiu que teníem un problema, les transferències es gestionen al FTM i la matriu s'utilitza al TS i entre ells no existia cap interfície definida.

Per aquest motiu es va decidir que el Job Manager mantingués també la matriu en memòria i que les actualitzacions de velocitats que fes el FTM, les fes sobre la matriu del JM, que en finalitzar l'aplicació quedaria actualitzada i s'emmagatzemaria al fitxer *historical.xml* [18], que és on es guarden totes les dades d'històric referents a l'aplicació.

D'aquesta manera s'aconsegueix cert grau de dinamisme entre execucions, però no durant la mateixa, ja que en llençar una aplicació, el TS carrega inicialment la matriu des del fitxer d'històric, que havia estat actualitzat al final de l'execució anterior pel JM i realitza tota l'execució amb aquella matriu, que tornarà a quedar actualitzada pel JM en acabar l'execució.

L'ús d'aquest sistema presenta els seus avantatges i inconvenients, que veurem descrits detalladament a l'apartat 3.5.1 d'aquesta secció.

Càlcul de la velocitat de xarxa

El càlcul de la velocitat de xarxa és un dels punts en què més ha calgut treballar. Com hem comentat en seccions anteriors, quan el TS dóna l'ordre de crear una feina al JM, aquest verifica a través del seu mètode *orderTransfers* si s'ha de transferir algun fitxer abans d'enviar a executar-la. Si és així, invoca el mètode *transferFiles*, al qual li passa la llista de fitxers a transferir i la localització del destí de la transferència.

Un cop el FTM executa aquesta funció, no en comença la còpia immediatament, sinó que les col·loca en una cua de peticions. Llavors, és l'objecte anomenat *TransferDispatcher*, que es troba dins el mateix FTM, l'encarregat de desencuar-les i atendre-les utilitzant cinc *threads* independents.

Inicialment, doncs, per calcular la velocitat de tranferència entre dos punts, s'ideà una primera solució que consistia a realitzar una marca d'inici cada vegada que comencés una transferència. D'aquesta manera, com que cada transferència és única, en acabar, es realitzaria una marca de temps final, i així, amb la diferència de temps final menys inicial i la mida del fitxer transferit, podríem saber fàcilment la velocitat a la qual s'havia transferit el fitxer.

Aquesta primera aproximació no va funcionar, ja que no es tenia en compte la possibilitat que mentre preníem la mesura de temps d'una transferència podia apareixe'n una altra entre el mateix node d'origen i destí, provocada per un dels fils del *TransferDispatcher* que desencués una nova transferència. Això dividia la capacitat de transferència de l'enllaç repartint-lo entre les dues transferències, que passaven a compartir l'enllaç i, per tant, a compartir la capacitat d'aquest.

Mesurar aquestes transferències de la forma anteriorment explicada provocava errors en les mesures, ja que sempre s'assumia la velocitat del canal com la d'una sola transferència.

Per tal de resoldre aquest problema es proposa una nova solució, que té en compte la utilització global de l'enllaç entre dos punts. El que s'ha realitzat és crear una estructura que manté un registre dels enllaços sobre els que hi ha transferències en curs.

D'aquesta manera quan s'inicia una transferència entre dos punts, es registra l'ús d'aquest enllaç a través d'una estructura que emmagatzema l'instant de temps en què s'ha començat a utilitzar, el nombre de fitxers simultanis que s'hi estan transferint i el volum total de dades transferit a través de l'enllaç. Així, si apareix una nova transferència,

[18] Veure secció 3.7.3

s'incrementarà el nombre de fitxers simultanis en curs i també el volum total de dades transferides per l'enllaç.

La marca de temps final la posarem quan el nombre de fitxers simultanis d'un enllaç es redueixi a zero. Just en aquest moment tindrem disponibles el nombre total de dades que s'han transferit i el temps en què s'ha fet, podent calcular així la velocitat mitjana de l'enllaç a partir de la següent fórmula:

$$VelocitatCanal = \frac{volumDades * 8}{tempsTrf * 1000000} \ (Mbps)$$

Un cop aplicada la fórmula, es veié que els valors de les mesures no eren sempre correctes, concretament en el cas de transferències de fitxers petits aquest càlcul no sempre donava bons resultats, a causa de les característiques de les xarxes TCP/IP.

Aquest protocol presenta un sistema anomenat arrencada lenta *(slow-start)* [19]. Com que ni l'emissor ni el receptor tenen forma de saber quin és el volum de dades màxim que poden arribar a transferir a la xarxa, per no saturar-la es comença a enviar i rebre de forma progressiva augmentant la velocitat de forma gradual fins que la xarxa arribi al seu cabal màxim o se saturi. En aquest cas, TCP reduirà la tasa d'enviament per disminuir-ne la saturació.

Per aquest motiu, necessitem mesurar fitxers suficientment grans perquè la velocitat de la xarxa hagi arribat a estabilitzar-se. Si no, pot passar que les velocitats dels enllaços mesurades siguin més baixes del que ho són en realitat. Aquest valor ha estat mesurat empíricament fins obtenir valors estables i s'ha deixat com a paràmetre configurable del sistema per tal de poder ser eliminat en cas d'utilitzar altres tipus de xarxes.

Dit això, la velocitat de xarxa s'actualitzarà només en cas que el volum de dades transferit pel canal sigui més gran que el del llindar establert per obtenir mesures estables.

A la següent figura 3.6 podem veure, de forma il·lustrativa, com es realitza el càlcul explicat. Es registraria la marca d'inici de transferències d'aquest enllaç (segon 0) i l'última transferència, que és la que faria disminuir el nombre de transferències simultànies a zero i generaria la marca final al cap de 45 segons.

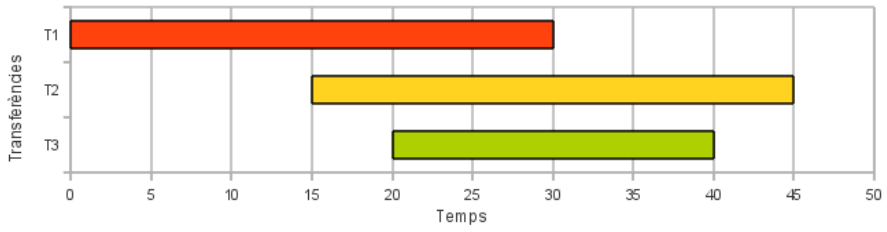


Figura 3.6: Diagrama amb solapament de transferències.

[19] http://es.wikipedia.org/wiki/Slow-start

Actualització de la matriu de velocitat de xarxa

Cada un d'aquests càlculs anteriors actualitzen la matriu que es troba al Job Manager a través del mètode *speedMatrixUpdate*. Per fer-ho, rep com a paràmetre un node origen, un node destí i el nou valor mesurat actualitzant la velocitat de l'enllaç. De totes maneres, aquest valor no és actualitzat reemplaçant-se, sinó que el valor que queda substituït a la matriu és en un 50% el valor nou i en l'altre 50% el valor anterior, tal i com indica la fórmula següent:

$$Velocitat = \frac{VelocitatAnterior + VelocitatActual}{2} \ (Mbps)$$

D'aquesta manera s'aconsegueix mantenir el resultat anterior, donant també una bona adaptabilitat als canvis sobtats en la xarxa.

Finalment, en acabar l'execució d'una aplicació, aquesta matriu s'emmagatzemarà al fitxer d'històric a través del mètode *setSpeedMatrix* implementat a l'HistoricalManager, que guardarà l'estructura de la matriu en format XML, tal i com es mostra a continuació.

Per seguir amb l'exemple d'aquest apartat, s'ha utilitzat com a referència la matriu de la figura 3.3.

```
<NetSpeed>
    <Link Src="host1.foo.es">
        <Speed Dst="host2.foo.es">91.2</Speed>
        <Speed Dst="host1.foo.es">91.2</Speed>
    </Link>
    <Link Src="host2.foo.es">
        <Speed Dst="host2.foo.es">504.3</Speed>
        <Speed Dst="host1.foo.es">78.8</Speed>
    </Link>
</NetSpeed>
```

Avantatges i inconvenients d'utilitzar aquest sistema

Tot sistema té els seus avantatges i els seus inconvenients i com és de suposar aquest no intenta ser-ne cap excepció. En aquest apartat tractarem d'analitzar quins són els principals avantatges i inconvenients d'utilitzar aquest sistema per mesurar la velocitat de la xarxa.

Inicialment i abans de desenvolupar el sistema, es pensà a utilitzar algun sistema de monitorització i recol·lecció d'informació per a recursos distribuïts (Grid Information System), per exemple: Globus Toolkit o Ganglia [18] [20], que és un sistema exclusiu per a monitorització de recursos distribuïts.

Realment, a l'hora de buscar precisió en la mesura, un sistema de monitorització és possiblement el més adequat, ara bé, utilitzar-lo implicava una sèrie de restriccions com, per exemple, haver d'estar sempre lligat al sistema o haver de desplegar tot el sistema al Grid utilitzat, cosa que podria provocar inconvenients sobretot si la persona que instal·la i

[20] http://ganglia.sourceforge.net

gestiona COMPSs, que habitualment és el propi usuari, no és l'administrador de la xarxa del Grid.

Per aquest motiu es va decidir incorporar el sistema de mesura dins de COMPSs a mode d'extensió; d'aquesta manera tota la gestió i mesura es realitza de forma transparent encara que això suposi alguns inconvenients.

El fet de mesurar la velocitat dins el File Transfer Manager pot provocar algunes desviacions en la mesura a causa de la càrrega del propi component. Recordem que per tal de gestionar les transferències, el FTM es recolza en JavaGAT, que és qui ordena les transferències i les execucions de les feines als Workers, les quals en acabar generen una notificació. Si a causa de la càrrega del FTM aquestes notificacions es processen més lentament poden provocar variacions en els resultats dels càlculs.

Bo i que és dificil que succeeixi, existeix la possibilitat i, per tant, ha de ser considerada.

3.6 Millora de la tolerància a fallades

Com s'ha explicat a l'apartat 3.3.2, el planificador requereix del paràmetre *AvailRate* per poder tenir en compte el % de fiabilitat dels recursos. D'aquesta manera, si un recurs és menys fiable que altres, perdrà crèdit i per tant el planificador l'escollirà menys vegades. En aquest apartat s'explicarà quin ha estat el mètode utilitzat per calcular aquest índex.

3.6.1 Disseny de mecanismes de tolerància a fallades

Quan el Job Manager envia una feina al Worker, aquesta pot acabar correcta o incorrectament; això se sap a partir de la notificació o *callback* generat per JavaGAT, que invoca al mètode *jobStatusNotification* del JM. Segons l'estat de finalització de la tasca, aquest mètode entrarà dins una etiqueta (case) OK o FAILED, executant diferents accions en cadascun dels casos.

Aprofitant aquesta estructura, es van implementar al Task Scheduler dos mètodes: un, que comptabilitzava el nombre de feines correctament executades a cada un dels Workers i l'altre, el nombre de feines que havien fallat. D'aquesta manera, en rebre una notificació es comptabilitzava la tasca en un dels dos anotadors.

Així el % de fiabilitat es calcula segons la següent fórmula:

$$RatioFiabilitat_r = \frac{FeinesOK_r}{FeinesEnviades_r} \quad on\ r = recurs$$

D'aquesta manera, el mètode *addOkSubmit* incrementarà el nombre de *FeinesOK* i *FeinesEnviades* d'un recurs determinat; en canvi *addFailedSubmit* n'incrementarà només el nombre de *FeinesEnviades*. D'aquesta manera, a mesura que van acabant feines s'obté de forma no gaire costosa el % de fiabilitat de cada recurs.

El codi simplificat corresponent a aquests mètodes és el següent:

```
void addOkSubmit(String r){
    feinesEnviades = Obtenir el nombre de feines enviades del recurs r
    feinesEnviades = feinesEnviades + 1

    feinesOK = Obtenir el nombre de feines ok del recurs r
    feinesOK = feinesOK + 1

    fiabilitat = feinesOK/feinesEnviades
    Guardar valor de fiabilitat a mapa: recurs -> fiabilitat
}
```

L'única diferència entre ambdues funcionalitats és que ***addOkSubmit*** incrementa el nombre de feines que han acabat correctament i ***addFailedSubmit*** no.

```
void addFailedSubmit(String r){
    feinesEnviades = Obtenir el nombre de feines enviades del recurs r
    feinesEnviades = feinesEnviades + 1

    feinesOK = Obtenir el nombre de feines ok del recurs r

    fiabilitat = feinesOK/feinesEnviades
    Guardar valor de fiabilitat a mapa: recurs -> fiabilitat
}
```

Per assegurar la propagació d'aquests valors entre execucions, s'han implementat una sèrie de funcionalitats que permeten guardar els valors de fiabilitat en format XML al fitxer d'històric seguint la sintaxi següent:

```
<Availability>
   <Worker Name="host01.foo.es">1.0</Worker>
   <Worker Name="host02.foo.es">0.75</Worker>
   <Worker Name="host03.foo.es">0.3</Worker>
</Availability>
```

En iniciar una aplicació, el planificador treballa amb les dades de disponibilitat de l'històric. Si inicialment no se'n tinguessin, es generen automàticament a l'inici amb valors de fiabilitat màxima per a cada un dels recursos.

Aquestes dades són sempre estàtiques al llarg de l'execució de l'aplicació, ja que el planificador treballa sempre amb les dades que ha carregat de l'històric anterior. Tal i com es va planificant a partir de les dades d'execucions anteriors, també se'n van generant de noves durant l'execució actual. Aquestes, en acabar passaran a fusionar-se amb les del nou històric fent servir una mitjana ponderada, de manera que estarà format pel **60%** dels nous valors i el **40%** dels antics. La configuració d'aquests pesos ha estat determinada empíricament i és facilment modificable.

3.6.2 Recuperació de fiabilitat

Com hem vist en l'apartat anterior, és possible que un recurs perdi la confiança del planificador, és a dir, que en fallar es redueixi el seu índex de fiabilitat i, per tant, el planificador l'esculli en menys ocasions.

Podria passar, en cas extrem, que alguns recursos patissin errors freqüents produïts per problemes de connectivitat. A causa d'això, obtindrien valors de fiabilitat tan baixos que és possible que el planificador no el tornés a escollir més. Justament per aquest motiu ha calgut desenvolupar un mètode anomenat *addConfidence* que permeti als recursos recuperar de forma progressiva la seva confiança.

El mètode desenvolupat és anomenat "mètode de confiança cega" i consisteix a donar, al final de l'execució, un petit % de confiança extra [21] als recursos que no tenen la fiabilitat màxima i que no han estat escollits durant aquella execució. D'aquesta manera és possible que recursos que han perdut tota la fiabilitat, a poc a poc la vagin recuperant fins al moment en què el planificador els torni a enviar feines. Si aquestes acaben correctament, aniran recuperant els seus valors de fiabilitat; si segueix fallant, els valors tornaran a baixar i el planificador els deixarà novament de banda.

3.6.3 Limitació en el nombre de reintents

L'última millora implementada en aquest apartat és limitar el nombre de vegades que es permetrà fallar a un recurs al llarg d'una execució. D'aquesta manera si un recurs falla tantes vegades com el límit especificat, quedarà descartat d'aquella execució. En cas que la seva fiabilitat hagi quedat afectada, el mètode de recuperació anterior s'encarregarà de restablir-la.

Aquest valor és especificable al fitxer de configuració de l'aplicació *project.xml* a través de la següent sintaxi:

<MaxRetries>3</MaxRetries>

En cas de no especificar res, el nombre d'errors permesos per defecte serà de 5 per host i execució.

3.7 Gestió de recursos

A la versió de COMPSs desenvolupada pel projecte, tota la gestió dels recursos es fa a través dels 3 gestors presentats a continuació:

- **ProjectManager:** conté totes les dades necessàries per poder accedir als Workers per executar-hi tasques i les localitzacions de fitxers.
- **ResourceManager:** conté les característiques dels recursos i en defineix les seves capacitats.
- **HistoricalManager:** manté l'històric d'informació acumulada per poder subministrar-la al planificador.

[21] Típicament en increments d'un 5%.

Aquests gestors emmagatzemen la seva informació en 3 fitxers que són clau a l'hora de realitzar la configuració d'una aplicació de COMPSs: el **project.xml, resources.xml i historical.xml**, que explicarem a continuació.

D'altra banda, cal mencionar que per poder obtenir una millor comprensió de l'estructura d'aquests fitxers, s'han inclòs a l'apèndix exemples reals d'aquests fitxers de configuració.

3.7.1 Modificacions al ProjectManager

El ProjectManager és el gestor que permet obtenir la informació continguda al fitxer de configuració de l'aplicació *project.xml*, un document escrit en llenguatge XML que conté:

- La definició de cada un dels workers que s'utilitzaran durant l'execució de l'aplicació (obligatori), d'on se'n definirà:
  - **WorkerName:** el nom del worker.
  - **InstallDir:** la ruta on es troben les classes Java o binaris dels codis de la part worker de l'aplicació.
  - **WorkingDir:** la ruta de treball on es trobaran els fitxers amb els quals treballarà el Worker.
  - **User:** el nom d'usuari amb el qual COMPSs realitzarà el procés de *login* cap a cada Worker.
- Els **DataNodes** que existeixin, en cas que n'hi hagi (opcional) [22].
- El nombre màxim de **reintents** abans d'excloure un Worker de l'execució (opcional) [23].
- Les **localitzacions** inicials dels fitxers, en cas de tenir-ne; si no, es generaran de forma automàtica al llarg de l'execució (opcional).

El *runtime* de COMPSs realitzarà consultes a aquest document per tal d'obtenir la informació per poder enviar feines als Workers o gestionar les funcionalitats anteriorment explicades.

Per fer-ho, es val de xPath [24], que permet construir expresions per processar dades en format XML tenint en compte l'estructura jeràrquica del document.

Per poder treballar amb les noves dades emmagatzemades en aquests fitxer, ha calgut incorporar al gestor algunes funcions addicionals:

- **getDataNodeProperty(String name):** en especificar el nom d'un recurs, retorna el nom d'usuari definit perquè COMPSs pugui accedir al node de dades.
- **getFileLocations(String filename):** en especificar el nom d'un fitxer, retorna una llista amb les URI's de les rèpliques d'aquell fitxer.
- **getFileLocationsLastMod(String filename):** en especificar el nom d'un fitxer, retorna l'última data de modificació del fitxer.

[22] Veure apartat 3.2.2.
[23] Veure apartat 3.6.3
[24] http://es.wikipedia.org/wiki/XPath

- **getFileLocationsSize(String filename):** en especificar el nom d'un fitxer, retorna la seva mida.
- **getMaxRetries():** retorna el nombre màxim de reintents que es permetran a un Worker abans de ser exclòs de l'execució.
- **setFileLocations(...):** permet guardar al fitxer *project.xml* cada una de les etiquetes de localització de fitxers definits a l'apartat 3.2.2
- **hasLocations():** retorna un booleà amb el valor corresponent depenent de si hi ha localitzacions addicionals definides al *project.xml*, o no.

3.7.2 Modificacions al ResourceManager

El ResourceManager és l'encarregat de gestionar totes les característiques de la màquina per saber si compleix amb les restriccions necessàries per poder executar una tasca. En el codi original de COMPSs, el funcionament era semblant al del ProjectManager, l'usuari introduïa al fitxer *resources.xml* les característiques que definien als seus recursos descrits a partir de l'estàndard *Grid Information and Data Modelling* de OGSA [4].

Aleshores, en iniciar l'execució d'una aplicació, el fitxer es carrega i s'hi fan consultes amb xPath retornant aquelles màquines que compleixen les especificacions per poder executar la tasca.

De totes maneres, bo i utilitzar aquest model de descripció, la majoria de les vegades a l'hora de definir els recursos no se solen utilitzar tots els seus camps; aquest és el cas del camp que defineix l'adaptador de xarxa, que no era utilitzat a la versió original de COMPSs, però sí a la nova. D'aquesta manera es facilita a l'usuari poder definir la velocitat de la interfície de xarxa de cada un dels nodes.

Per fer-ho ha calgut implementar un mètode que permet llegir aquest camp a partir del nom d'un recurs. El mètode *getNetworkSpeed* retorna la velocitat de l'adaptador de xarxa definida al fitxer *resources.xml*.

3.7.3 Implementació de l'HistoricalManager

L'HistoricalManager és l'encarregat de gestionar tot el conjunt de dades que s'emmagatzemen per poder mantenir l'històric necessari per al nou planificador. En aquest fitxer anomenat *historical.xml* s'hi guardaran dades de: fiabilitat dels recursos, temps mig d'execució de cada tipus de tasca i recurs disponible i la matriu de velocitat de xarxa, explicats en apartats anteriors.

De totes maneres és habitual realitzar la primera execució d'una aplicació sense històric previ; per tant, inicialment no tindrem dades en aquest fitxer. En aquests cas COMPSs el generarà automàticament i l'anirà completant amb dades extretes de les execucions.

Per tal de poder gestionar aquestes dades ha calgut implementar els següents mètodes:

- **getAvailability():** retorna una estructura de tipus mapa amb les dades de fiabilitat de cada recurs.
- **getMeanTimeStructure(String name):** En especificar-hi el nom del recurs, retorna una estructura amb el temps mig d'execució de cada tipus de tasca en cada un dels recursos del Grid.

- **getSpeedMatrix():** retorna una estructura amb la matriu de velocitat guardada a l'històric.
- **setAvailability(...):** permet guardar els coeficients de fiabilitat de cada un dels recursos.
- **setMeanExecTime(...):** permet guardar els temps mig d'execució de cada tipus de tasca al fitxer d'històric.
- **setSpeedMatrix(...):** permet guardar la matriu de velocitat de xarxa al fitxer d'històric.

Ara, en el següent capítol posarem a prova cada una d'aquestes extensions realitzades. Experimentarem i analitzarem els resultats per comprovar quines han estat les millores obtingudes.

Capítol 4

Tests, Experiments i Resultats

Un cop presentat el desenvolupament del projecte, en aquest capítol presentarem el conjunt de proves i experiments als quals s'ha sotmès el prototip, per verificar el bon funcionament de les millores finals. En primer lloc presentarem l'entorn sobre el qual s'han realitzat el seguit de proves, en segon lloc presentarem cada una de les aplicacions utilitzades en el procés d'experimentació. Finalment es presentarà el conjunt de proves realitzades sobre cada una de les extensions dissenyades.

4.1 Entorn de proves

Abans de presentar el conjunt de proves i experiments realitzats, introduirem l'entorn de proves en què s'han realitzat els experiments que es mostraran al llarg d'aquest capítol.

El Grid utilitzat per realitzar les proves consta de 5 servidors. Les màquines: **bscgrid02, bscgrid03, bscgrid04, bscgrid06 i tamariu.**

En tots els experiments, la mateixa màquina que executa l'aplicació principal serà la que suportarà tots els components del *runtime*. La màquina Master (bscgrid05) és un servidor equipat amb un processador Intel Q9300 Core 2 Quad a 2.5GHz amb 3MB de memòria *caché*, 4 GB de RAM i un disc de 220GB a 7200rpm.

La màquina bscgrid06 és exactament d'iguals característiques a la bscgrid05, en canvi les bscgrid02, 03 i 04 són servidors inferiors, que van equipats amb processadors Intel Pentium 4 Dual Core a 3.6Ghz, 2MB de memòria *caché*, 1 GB de RAM i un disc de 60GB a 7200rpm.

D'altra banda, la màquina tamariu, que és la més potent, disposa de 4 processadors Intel Xeon E7450 @ 2.40GHz de 6 nuclis amb 12MB de *caché*, 47 GB de RAM i dos discs, un de 550GB i un de 320GB.

A la següent figura podem veure la configuració d'aquest Grid.

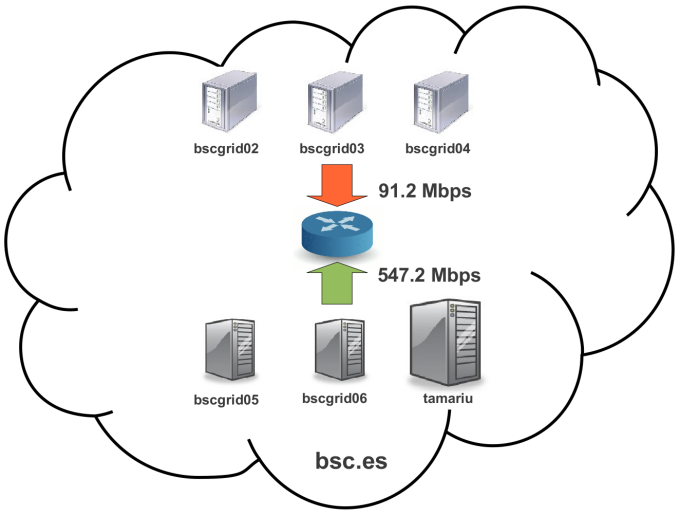


Figura 4.1: Topologia del Grid de proves.

La connexió entre els diferents nodes es realitza a través de xarxes que disposen de diferents velocitats. Les connexions entre les màquines bscgrid02, 03 i 04 disposen de 91.2 Mbps reals. En canvi, les connexions entre les bscgrid05, bscgrid06 i tamariu es realitzen a 547.2 Mbps reals, tal i com especifica la següent taula:

Recursos	bscgrid02	bscgrid03	bscgrid04	bscgrid05	bscgrid06	tamariu
bscgrid02	loop	91.2	91.2	91.2	91.2	91.2
bscgrid03	91.2	loop	91.2	91.2	91.2	91.2
bscgrid04	91.2	91.2	loop	91.2	91.2	91.2
bscgrid05	91.2	91.2	91.2	loop	547.2	547.2
bscgrid06	91.2	91.2	91.2	547.2	loop	547.2
tamariu	91.2	91.2	91.2	547.2	547.2	loop

Taula 4.1: Velocitats de connexió entre els nodes del Grid.

4.2 Aplicacions

4.2.1 HMMER

HMMER és una suite d'aplicacions utilitzada en l'àmbit de la bioinformàtica [1] que s'utilitza per analitzar models HMM (Hidden Markov Model) que representen famílies de proteïnes. Una de les aplicacions més importants és HMMPfam, que llegeix un conjunt de seqüències d'aminoàcids i els compara contra una base de dades buscant altres seqüències similars.

L'objectiu final de l'aplicació és trobar, per cada seqüència, el conjunt de famílies de la base de dades amb les que tenen més similituds. Aquest procés és anomenat alineament de seqüències proteiques. Aquest procés comporta un càlcul intens, però a la vegada altament paral·lelitzable.

[1] http://hmmer.janelia.org

Comparat amb altres suites com BLAST [2], que utilitza l'algorisme de *Smith-Waterman* per realitzar els alineaments, o FASTA [3], HMMER genera resultats més acurats.

La paral·lelització de l'aplicació es divideix en 3 fases:

- **Fragmentació:** els fitxers de seqüències i la base de dades es divideixen en funció del nombre de processadors i de memòria disponible, intentant generar fragments que càpiguen a la memòria del sistema, evitant així un accés intensiu a disc.
- **Execució:** s'executa el binari HMMPfam sobre cada parella de fragments de la base de dades i seqüències.
- **Reducció:** els resultats d'aquestes execucions s'uneixen de nou per generar el resultat final.

L'execució d'aquesta aplicació genera un graf com el que mostra la figura 4.2. L'execució de la fase de segmentació es realitza de forma seqüencial al Master. El nivell més alt de l'arbre correspon a la segona fase. Els nivells inferiors mostren com la fase de reducció uneix els fitxers resultants d'aplicar el binari a cada fragment de seqüència i base de dades.

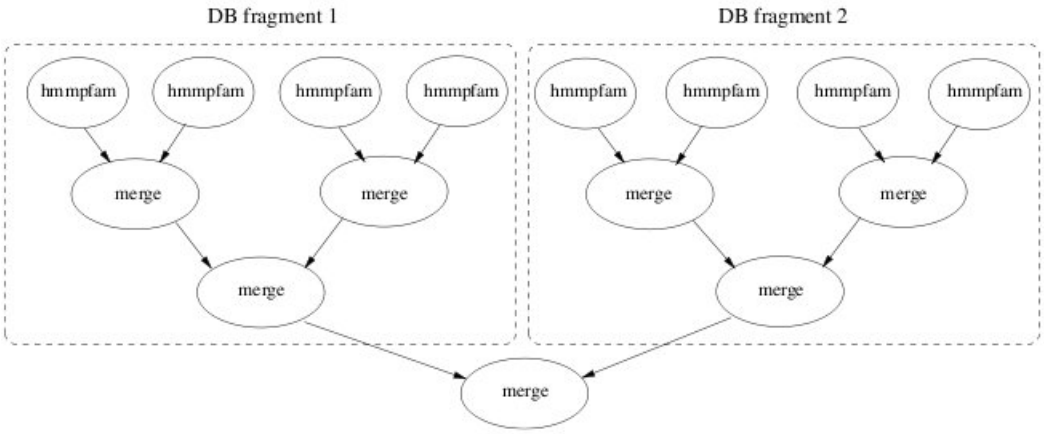


Figura 4.2: Graf de l'aplicació HMMER.

De fet, a la implementació existeixen 3 tipus de tasca de reducció en funció dels fragments que s'haguin d'unir: Si s'uneixen fragments corresponents a la mateixa base de dades s'utilitzarà el mètode *mergeSameDB*. En canvi, si s'uneixen fragments amb les mateixes seqüències, s'utilitzarà *mergeSameSeq*. En funció d'això el temps de cada procés d'unió serà més o menys gran. D'aquesta manera, l'aplicació queda dividida en 3 tipus de tasques diferents:

- **hmmpfam:** executa al Worker el binari HMMPfam amb un fragment de la base de dades i un del conjunt de seqüències. Generarà un resultat parcial.
- **mergeSameDB:** a partir de dos resultats parcials que provinguin del mateix fragment de la base de dades, els uneix i genera un nou resultat parcial.
- **mergeSameSeq:** uneix dos resultats parcials que provinguin del mateix conjunt de seqüències generant un nou resultat parcial.

[2] http://ca.wikipedia.org/wiki/BLAST
[3] http://en.wikipedia.org/wiki/FASTA

La unió dels dos últims resultats parcials es fa sempre al Master.

Tot el conjunt de proves es durà a terme utilitzant la base de dades SMART [4], que conté 725 models de longituds d'entre 11 i 971. En cada prova es variarà la mida del fitxer de seqüències d'entrada, agafant-ne 2048, 4096 i 8192 d'entre una base de dades de 100.000 provinent d'UniParc [5].

HMMER és una aplicació que depèn, en un alt grau, de la capacitat de càlcul del sistema, disposant de tasques de diferent duració i d'un alt nivell de paral·lelisme a l'inici de l'execució que va decreixent a mida que avança el graf.

Aquesta aplicació s'utilitzarà per realitzar les proves del nou planificador. Independentment del nombre de seqüències d'entrada, les proves es faran amb **2, 4, 6, 10, 14, 18 i 22** Workers, analitzant l'evolució del temps final d'execució de la nova versió de COMPSs vs COMPSs original.

4.2.2 JRA4

JRA4 és una aplicació que permet realitzar prediccions de temperatures de superfícies i treballa amb conjunts de models de temperatures anteriorment obtingudes sobre el terreny. Per fer-ho utilitza el mètode *Multimodel Ensemble Mean Forecasting* [6], que permet realitzar prediccions d'estats futurs en sistemes canviants.

L'aplicació treballa per mitjà d'un binari anomenat CDO (Climate Data Operators) que permet manipular, analitzar i aplicar més de 400 operadors a fitxers de dades de prediccions climàtiques.

En aquest cas CDO s'utilitza per computar la *Multimodel Ensemble Mean*. Per fer-ho es realitzen 4 passos:

1. **Selecció temporal (T1):** selecciona el mes i any dins el model agafant-ne un subconjunt de mostres.
2. **Normalització del sistema de referència (T2):** normalitza el sistema de referència agafant-ne un de comú per a tots els models.
3. **Computació de la mitjana temporal (T3):** realitza la mitjana temporal del model a partir de la interpolació bilineal [7].
4. **Computació de la mitjana del model (T4):** aquesta etapa es realitza al Master un cop s'han computat tots els passos anteriors; llavors, per cada un dels models d'entrada en realitza l'*ensemble mean* o mitjana del model.

L'execució d'aquesta aplicació genera un graf com el que es mostra a la figura 4.3. L'execució de la fase de selecció temporal, normalització del sistema de referència i computació de la mitjana temporal es realitzen al Grid, mentre que el procés final de la computació de la mitjana es realitza al Master, processant així tots els models de forma concurrent.

Aquesta aplicació serà especialment adequada per experimentar amb l'extensió de gestió de rèpliques, ja que els fitxers d'entrada són especialment grans. Utilitza fitxers de

[4] http://smart.embl-heidelberg.de
[5] http://www.ebi.ac.uk/uniparc
[6] http://en.wikipedia.org/wiki/Ensemble_forecasting
[7] http://en.wikipedia.org/wiki/Bilinear_interpolation

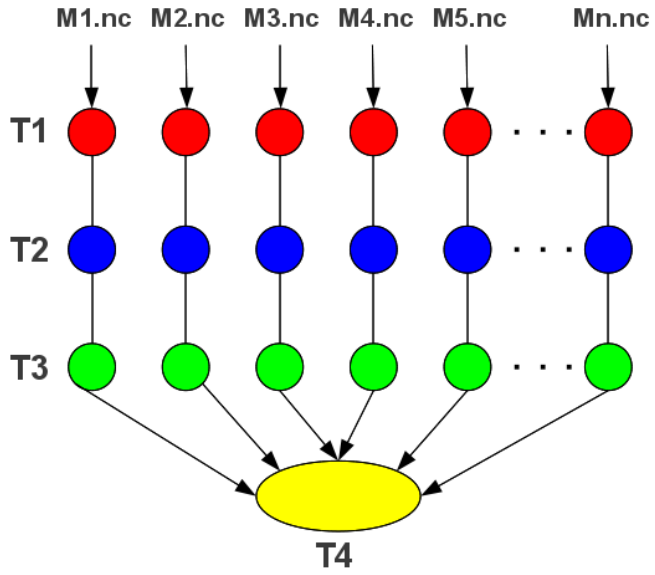


Figura 4.3: Graf de l'aplicació JRA4.

models de tipus netCDF (Network Common Data Form) [8] d'1GB cada un. Per tant, l'aplicació s'utilitzarà per realitzar les proves del sistema de gestió de rèpliques, on observarem com la nova versió de COMPSs redueix el volum de transferències respecte a la versió original agilitant-ne l'execució.

4.2.3 SparseLU

SparseLU és una aplicació que multiplica dues matrius mitjançant el mètode de factorització o descomposició LU, que consisteix a factoritzar una matriu com el producte d'una matriu triangular inferior i una superior [9].

La matriu queda dividida en blocs de NxN sobre els quals es van aplicant 4 tipus d'operacions que modifiquen cada un dels blocs: **lu0, fwd, bdiv i bmod.** A la implementació, aquestes 4 operacions coincideixen amb les tasques que executaran els Workers.

```
for (int k = 0; k < NB; k++) {
        lu0(A[k][k]);
        for (int j = k+1; j < NB; j++) {
                fwd(A[k][k], A[k][j]);
        }
        for (int i = k+1; i < NB; i++) {
                bdiv(A[k][k], A[i][k]);
                for (int j = k+1; j < NB; j++) {
                        bmod(A[i][k], A[k][j], A[i][j]);
}       }       }
```

[8] http://en.wikipedia.org/wiki/NetCDF
[9] http://es.wikipedia.org/wiki/Factorizacion_LU

La figura 4.4 mostra el graf de dependències que genera el codi de l'aplicació:

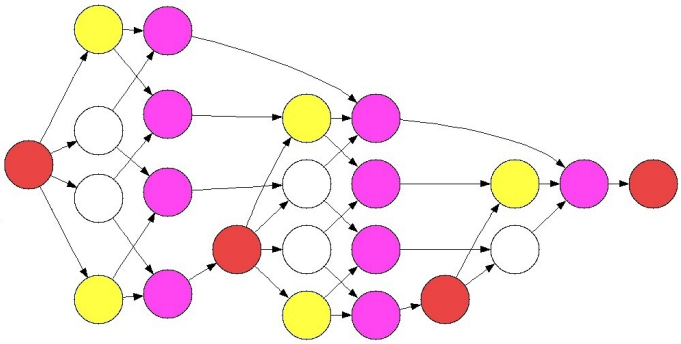


Figura 4.4: Graf de l'aplicació SparseLU.

Tal com es pot veure, només una tasca del mètode lu0 pot ser executada a la vegada. Les tasques bmod, fwd i bdiv de diverses iteracions poden solapar-se a mesura que les dependències se solucionen.

Aquesta aplicació serà utilitzada per experimentar amb la millora implementada en l'apartat de tolerància a fallades, provocant diversos tipus de fallades amb l'objectiu d'analitzar les reaccions del sistema i veient com varia el nombre final de tasques assignades a cada Worker entre execucions.

4.3 Anàlisi del rendiment del planificador

4.3.1 Anàlisi del temps d'execució

Com hem dit a l'apartat anterior, l'experiment que realitzarem determinarà la millora de rendiment del planificador de la nova versió de COMPSs respecte al de l'original.

Per fer-ho es planteja realitzar execucions de l'aplicació HMMER utilitzant 2048, 4096 i 8192 seqüències que permetran analitzar l'evolució dels dos *runtimes* treballant amb diferents volums de tasques. Tindrem per a 2048 seqüències un total de 639 tasques [10], per a 4096 un total 1279 i per a 8192 un total de 2559. D'altra banda, anirem variant també el nombre de Workers, cosa que permetrà determinar, de forma gràfica, l'escalabilitat de cada sistema.

A la següent taula (Taula 4.2) podem veure comparat, els temps d'execució de HMMER aplicat a 4096 i 8192 seqüències i la millora aconseguida respecte al *runtime* original. Els valors en verd representen els punts on s'ha obtingut millora respecte la versió original, les files en vermell representen pèrdua de rendiment.

Hi ha alguns punts on la nova versió de COMPSs perd rendiment respecte a l'original. El nou planificador desenvolupat és, a fi de comptes, computacionalment més costós que l'anterior, ja que cada vegada que aquest planifica sobre un recurs ha de realitzar més etapes que a la versió original (si recordem, ha de calcular el temps de transferència dels fitxers per cada un del recursos disponibles i elaborar posteriorment el rànquing); per tant aquest cost afegit comença a ser justificable quan el temps perdut escollint el millor recurs, beneficia pel fet de triar el millor disponible.

[10] Cada tasca correspondrà a cada node del graf creat per COMPSs a l'inici de l'execució.

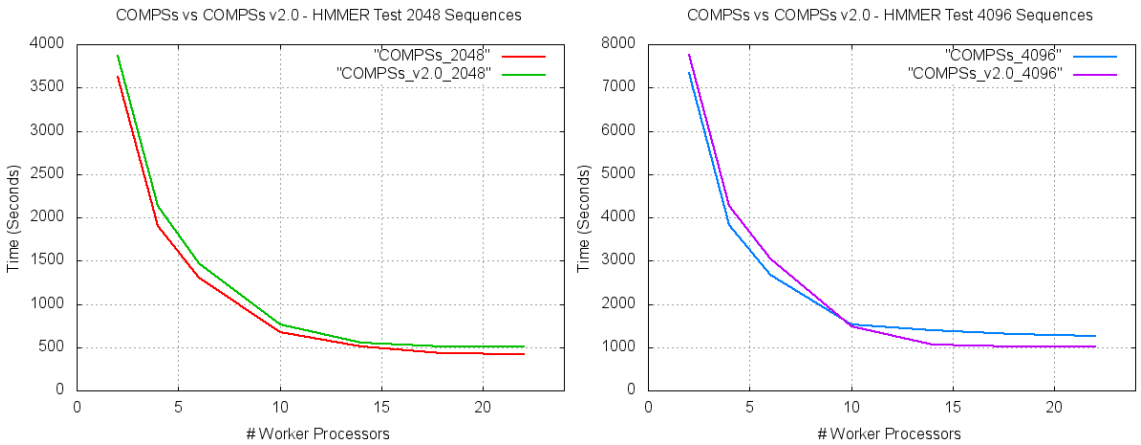
Ara bé, quan es disposa de pocs Workers sobre els que planificar, no hi ha massa on triar i per tant no hi ha possibilitat de determinar quin és el recurs que minimitza el temps de transferència, ja que, per exemple, només se'n pot triar un. En aquests casos el planificador original de COMPSs treu cert avantatge al nou.

D'altra banda, a mesura que augmenta el nombre de seqüències d'entrada augmenta també el nombre de tasques i amb elles, el nombre de fitxers a transferir i la mida d'aquests. Així que la mida del problema augmenta, el nou planificador treu més benefici de no haver de moure, a l'inici de cada execució, tot el conjunt de fitxers d'entrada al Grid i de moure els fitxers intermitjos de forma més eficient que a la versió original.

4096 Seqüències	**COMPSs**	**COMPSs v2.0**	**Millora**
2 Workers	7342.74	7762.90	-5.72%
4 Workers	3819.44	4264.4	-11.65%
6 Workers	2667.91	3059.05	-14.66%
10 Workers	1517.55	1481.44	2.38%
14 Workers	1395.19	1051.02	24.67%
18 Workers	1313.20	1021.93	22.18%
22 Workers	1250.30	1008.33	19.35%
8192 Seqüències	**COMPSs**	**COMPSs v2.0**	**Millora**
2 Workers	15480.51	15654.31	-1.12%
4 Workers	8058.07	8539.07	-5.97%
6 Workers	5931.41	5891.61	0.67%
10 Workers	5105.58	2867.39	43.84%
14 Workers	5007.11	2255.92	54.94%
18 Workers	4843.51	2050.09	57.67%
22 Workers	4780.31	1844.27	61.42%

Taula 4.2: Comparativa de temps d'execució de l'aplicació HMMER.

A la següent figura podem veure les gràfiques on es mostra els temps d'execució segons el nombre de seqüències i Workers.



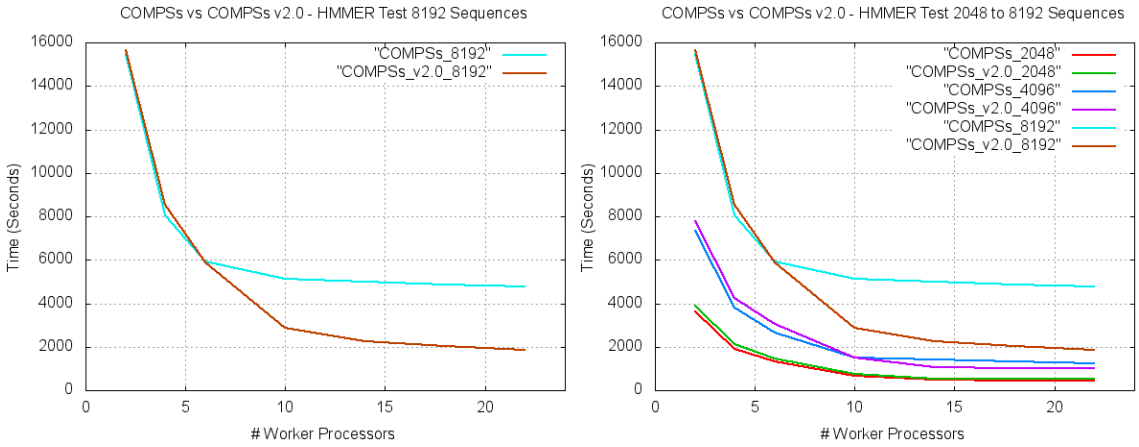


Figura 4.5: Gràfiques de temps d'execució de l'aplicació HMMER.

4.3.2 Anàlisi del balanceig de tasques

El segon experiment que realitzarem consisteix a analitzar com el nou planificador reparteix el volum de tasques entre els recursos disponibles. Per fer-ho, s'ha partit de les dades resultants de les execucions de l'apartat anterior.

La figura 4.7 mostra el comportament del sistema executant l'aplicació HMMER amb 8192 seqüències d'entrada i **6, 10, 14 i 22** Workers.

Per interpretar els gràfics s'utilitzarà la llegenda que es mostra a continuació:

Figura 4.6: Llegenda dels gràfics de balanceig de tasques.

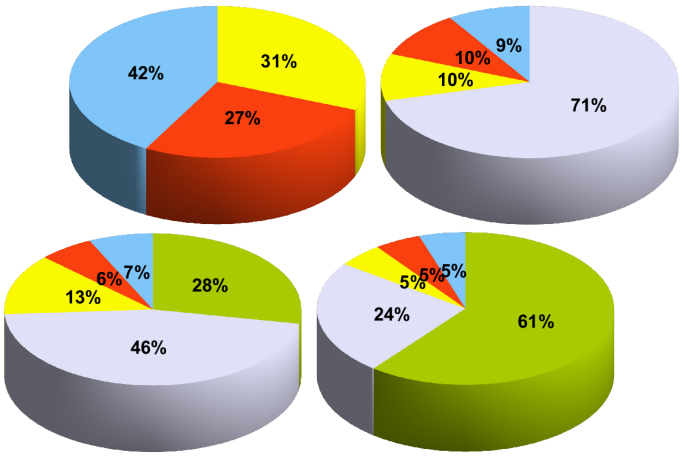


Figura 4.7: Gràfics de balanceig de tasques en un HMMER de 8192 seqüències amb **6, 10, 14 i 22** Workers.

La primera de les gràfiques mostra que el repartiment de tasques és força equitatiu, ja que les màquines utilitzades, la bscgrid02, bscgrid03 i bscgrid04, són d'iguals característiques i la velocitat de xarxa entre la màquina Master (bscgrid05) i cada una d'elles és igual (91.2 Mbps).

La segona de les gràfiques, en sentit horari, incorpora la màquina bscgrid06 al conjunt de recursos, passant de tenir 6 Workers a 10. Com podem veure, en aquesta ocasió s'assignen moltes més tasques a la nova màquina, que disposa de 4 CPUs i per tant és capaç d'absorbir una major quantitat de tasques; a més a més aquesta màquina disposa d'una connexió amb el Master a una velocitat superior a la resta.

La tercera de les gràfiques representa el cas que cal analitzar amb més detall. En aquesta ocasió s'incorpora la màquina tamariu utilitzant-hi 4 de les 24 CPUs disponibles. Aquesta màquina és molt semblant a bscgrid06, disposa de CPUs de rendiment similar i d'una interconnexió de xarxa igual que la de bscgrid06, 547.2 Mbps. De totes maneres el desequilibri que podem veure entre el % de tasques assignades a bscgrid06 i les de tamariu és degut a la càrrega habitual d'aquesta última.

La màquina bscgrid06 és una màquina exclusivament dedicada, per tant, en el moment de l'execució ningú més l'estava utilitzant. En canvi, tamariu és compartida per varis usuaris de diferents departaments del BSC, per tant degut a la durada de les execucions, va resultar impossible obtenir la màquina de forma exclusiva al llarg de les proves.

A la taula 4.3 podem veure algunes dades extretes de l'històric de l'execució, concretament es mostren els temps d'execució de cada un dels mètodes a cada un dels recursos disponibles.

Recursos	**CPUs**	**hmmpfam**	**mergeSameDB**	**mergeSameDB**
bscgrid02	2	15.79 s	1.18 s	8.96 s
bscgrid03	2	20.08 s	1.01 s	7.59 s
bscgrid04	2	21.05 s	1.04 s	9.91 s
bscgrid06	4	11.09 s	**0.92 s**	**14.05 s**
tamariu	4	10.91 s	**1.20 s**	**16.98 s**

Taula 4.3: Temps d'execució de cada tipus de tasca en un HMMER amb 14 Workers.

Com podem observar, tamariu triga més temps a completar els mètodes *mergeSameDB* i *mergeSameDB* que bscgrid06 al ser un recurs compartit. Al mètode *mergeSameDB* perd 2.93 segons cada cop; per tant, com que la mesura del temps d'espera en cua es calcula a partir del temps típic d'execució del mètode en aquell recurs, el planificador assigna més tasques a bscgrid06 per tal d'evitar que la cua de tamariu segueixi creixent.

A l'última de les gràfiques s'han incorporat 8 CPUs més a tamariu, cedint així un total de 12 CPUs (sempre considerant que la màquina disposa de suficient memòria principal per poder executar 12 Workers de forma simultànea sense haver de fer *Swapping* [11]). D'aquesta manera es converteix en la màquina capaç d'absorbir més tasques i, per tant, serà la que tindrà menys cua; per això el planificador hi assignarà moltes més tasques que en el cas anterior.

[11] Recorrer a l'espai d'intercanvi del disc al no disposar de prou memòria principal per a poder allotjar a tots els procesos del sistema.

4.4 Anàlisi del sistema de gestió de rèpliques

Aquest experiment busca demostrar l'efectivitat del sistema de gestió de rèpliques comparant el temps d'execució final de l'aplicació JRA4 entre la versió original de COMPSs i la nova.

Per fer-ho es planteja un escenari de proves amb 12 Workers i utilitzant les màquines bscgrid02, bscgrid03, bscgrid05 i bscgrid06 i 12 models d'entrada de 1GB cada un. D'aquesta manera, com que els fitxes es trobaran inicialment al Master (bscgrid04), a l'hora d'executar l'aplicació els 12GB d'entrada hauran de ser transferits al Grid.

La versió original de COMPSs els transferirà cada vegada al no disposar de cap mètode per representar l'existència de rèpliques. En canvi, a la nova versió es transferiran només una vegada ja que les localitzacions dels fitxers quedaran registrades al fitxer *project.xml*. D'aquesta manera només caldrà transferir-ne de nou en el cas que algun hagi patit modificacions.

Per poder observar aquests resultats s'han realitzat dues execucions de l'aplicació en cada una de les versions de COMPSs. A la figura següent podem veure'n els resultats observant que a COMPSs original les dues execucions amb 12 models triguen prop de 1150 segons, contra els 1133 i 39 segons de la nova versió.

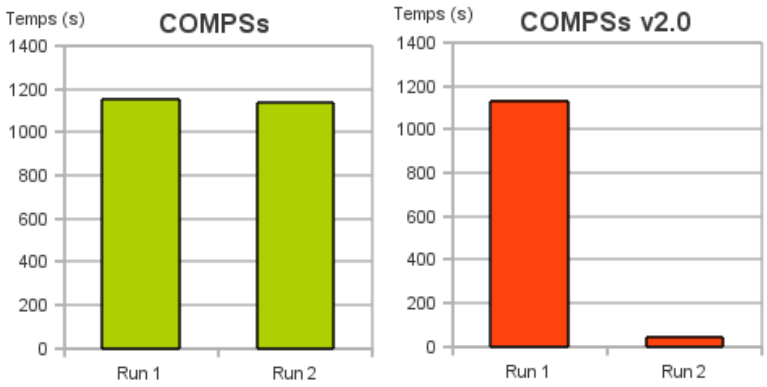


Figura 4.8: Comparativa de temps d'execució entre versions de COMPSs.

Com es pot veure, aquest aplicació te una càrrega alta en transferència i baixa en computació. Per aquest motiu, minimitzar el temps de transferència li aporta beneficis.

A la taula 4.4 es desglossa el temps d'execució d'un sol model, on podem veure el temps invertit a cada tasca de l'aplicació.

Les tres primeres files representen les tasques que s'executen al Grid, on s'observa que el temps de computació de l'aplicació és baix. La quarta fila representa el temps invertit en la transferència de fitxers corresponent al 91.52% del temps total. La cinquena fila representa el temps invertit pel *runtime* en la gestió de l'execució.

Per altra banda, recordem que l'extensió havia de permetre detectar també si entre execucions hi havia hagut algun canvi en el conjunt de fitxers d'entrada. Per tal de verificar-ne el bon funcionament s'ha modificat un d'aquests fitxers i s'ha tornat a executar l'aplicació per comprovar que efectivament el fitxer modificat era transferit de nou.

Tasca	Temps mig	% del total
T1	2.8 s	2.37 %
T2	0.86 s	0.73 %
T3	0.77 s	0.65 %
Trf	108 s	**91.52 %**
Runtime	5.57 s	4.72 %
Total	118 s	100 %

Taula 4.4: Temps mig d'execució inicial per un sol model.

A la següent figura podem observar-ne el funcionament: s'aprecia la transferència a la primera execució trigant un total de 125 segons, mentre que a la segona ja no hi ha cap transferència perquè tots els fitxers han estat actualitzats. En aquest cas l'execució triga només 39 segons.

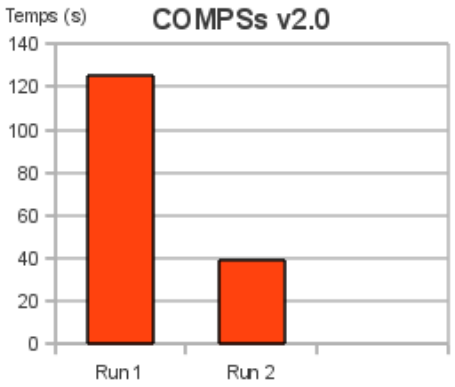


Figura 4.9: Exemple amb actualització dels models entre execucions.

4.5 Anàlisi de la tolerància a fallades

El tercer experiment que realitzarem consisteix a analitzar com el planificador és capaç de variar l'assignació de tasques segons el percentatge de fiabilitat dels recursos. Per fer-ho s'executarà l'aplicació sparseLU amb 10 Workers i un total de 20 blocs de 4x4 valors que formen una matriu quadrada de 320x320 valors.

4.5.1 Anàlisi sense limit de reintents

Per apreciar el funcionament del sistema, s'executarà l'aplicació amb tots els Workers amb valors de fiabilitat màxima; després s'anirà reduint la fiabilitat de la màquina bscgrid06 fins al 70%. Aquesta reducció es farà en aquesta màquina pel fet de ser la més potent i la que en perdre confiança perdrà també més pes en l'execució cedint part de les tasques a nodes amb menys potència.

A la següent taula podem veure els resultats d'aquest experiment:

Fiabilitat	**100%**	**90%**	**80%**	**75%**	**70%**
bscgrid02	14%	13%	54%	5%	30%
bscgrid03	12%	10%	5%	32%	51%
bscgrid04	7%	19%	3%	44%	19%
bscgrid06	**67%**	**58%**	**38%**	**20%**	**0%**
Tasques Ok	890	791	692	643	594
Fallades	0	99	198	247	296
Temps exec.	1011 s	1094 s	1512 s	1592 s	1610 s

Taula 4.5: Assignació de tasques segons el % de fiabilitat de la màquina bscgrid06.

Com podem veure, a mesura que el nombre de fallades augmenta, la fiabilitat de la màquina bscgrid06 va disminuint i, per tant, el planificador li assigna a la següent execució un nombre menor de tasques, descartant-la per complet per sota del 70% de fiabilitat.

Cal recalcar que en aquesta prova la limitació en el nombre de reintents ha estat desactivada per poder detectar el percentatge d'error al qual el recurs era descartat pel planificador de forma natural. Tal i com es veu a la taula 4.5 el planificador ha permès a bscgrid06 un error màxim del **30%**.

Per altra banda, el repartiment de tasques entre les màquines bscgrid02, 03 i 04 no ha estat gaire equitatiu, bo i ser màquines de característiques similars. Això és a causa que els temps d'execució de les tasques d'aquesta aplicació són molt baixos, entre 0.01s i 0.9s.

La mida dels fitxers és també reduïda, al tractar-se de blocs de només 16 valors. Per aquest motiu el temps de transferència és gairebé negligible, ja que la diferència en la velocitat de xarxa entre els recursos és practicament irrellevant. Per això, el fet d'haver-hi una lleugera variació en els temps d'execució dels mètodes ha fet variar també de forma considerable el repartiment de tasques.

4.5.2 Anàlisi amb límit de reintents

El segon experiment d'aquest apartat consisteix a comprovar el sistema de limitació de nombre màxim de reintents. Per fer-ho s'ha substituït la màquina bscgrid06 per una fictícia (bscgrid07) amb l'objectiu d'aconseguir un domini que no retornés mai cap resposta. D'aquesta manera es volia simular la fallada de la màquina al llarg de tota l'execució, tal i com podria succeir en el cas que, per exemple, una de les màquines quedés sense connectivitat de forma temporal.

Per fer-ho, s'ha establert el límit a 3 reintents, així quan el *runtime* detecti que s'ha assolit el nombre màxim, descartarà el recurs de l'execució.

Com veiem a la taula 4.6, el nombre de feines enviades a la màquina fictícia és de 21, això és perquè el Job Manager les ha enviat a executar abans de poder rebre les 3 notificacions d'error. De totes maneres, un cop rebudes s'elimina el recurs del Task Scheduler aconseguint que no s'hi planifiquin més tasques, bo i tenir-ne 21 prèviament planificades i que caldrà replanificar.

Com veiem, tant el nombre de tasques que han acabat correctament com la seva disponibilitat són zero, per tant, s'ha hagut de replanificar a altres recursos les 21 que s'hi han enviat inicialment.

Recursos	T.enviades	T.Ok	Replanificades	Disponibilitat
bscgrid02	183	183	0	100%
bscgrid03	442	442	0	100%
bscgrid04	265	265	0	100%
bscgrid07 (Inactiva)	**21**	**0**	**21**	**0%**

Taula 4.6: Eliminació de bscgrid07 en excedir el nombre màxim d'errors permesos.

4.6 Anàlisi del consum de recursos

En aquest últim apartat comprovarem quin ha estat l'impacte de les extensions realitzades, quant a recursos consumits del Master. Per fer-ho ens centrarem principalment en 2 aspectes, el consum de CPU i el de memòria al llarg de l'execució.

L'experiment constarà a executar l'aplicació HMMER amb 10 Workers aplicada a 8192 seqüències, tant al *runtime* original com al nou. El primer aspecte a analitzar serà el consum de la CPU.

La figura 4.10 mostra l'execució per les dues versions de *runtime*. El gràfic de dalt correspon al percentatge utilitzat per la versió original, el de sota correspon al de la nova versió.

L'ocupació de memòria, en canvi, és un punt complicat d'avaluar. Existeixen dos problemes principals que en dificulten l'avaluació: ProActive i l'alliberament d'espai al *heap*.

Com s'explica a la secció 2.4.1, per poder enviar una estructura de dades entre dos components, ProActive necessita fer una còpia de l'objecte de manera que com més comunicacions es generin entre components, més espai de memòria utilitzarà. A Java l'encarregat d'alliberar dades del *heap* és el Garbage Collector. Per poder netejar totes les estructures que ha hagut de copiar ProActive en la comunicació dels components, cal esperar la seva actuació periòdica.

Com es pot veure, tant el consum de CPU com el de memòria és menor a la nova versió. Al *runtime* original el consum de CPU es troba sobre un 20% al llarg de l'execució generant pics de demanda d'un 43% i donant un consum mig del 24% [12]; en canvi, a la nova versió el consum de CPU oscil·la entre el 10% i el 39% donant un consum mig de l'11% **que suposa un 54% menys.**

En quant al consum de memòria (figura 4.11), la versió original consumeix entre 80 i 288MB donant un consum mig de 184MB, mentre que trobem el de la nova entre 63 i 218MB, generant un consum mig de 140.5MB **que suposa un 24% menys.**

Analitzant la desviació dels valors extrems respecte la mitjana de cada paràmetre mesurat, s'observa que el consum de CPU del *runtime* original pateix una desviació respecte la mitjana d'un 19% contra un 28% del nou, com es pot veure al gràfic el consum de CPU del *runtime* original és més constant donant un valor de mitjana superior.

Respecte al consum de memòria, el *runtime* original presenta una desviació de 104MB contra 77.5MB del nou, és a dir, el nou *runtime* a més de consumir menys memòria manté el consum més constant al llarg de l'execució.

[12] Indicat a la figura 4.11 i 4.10 per la línia de color vermell.

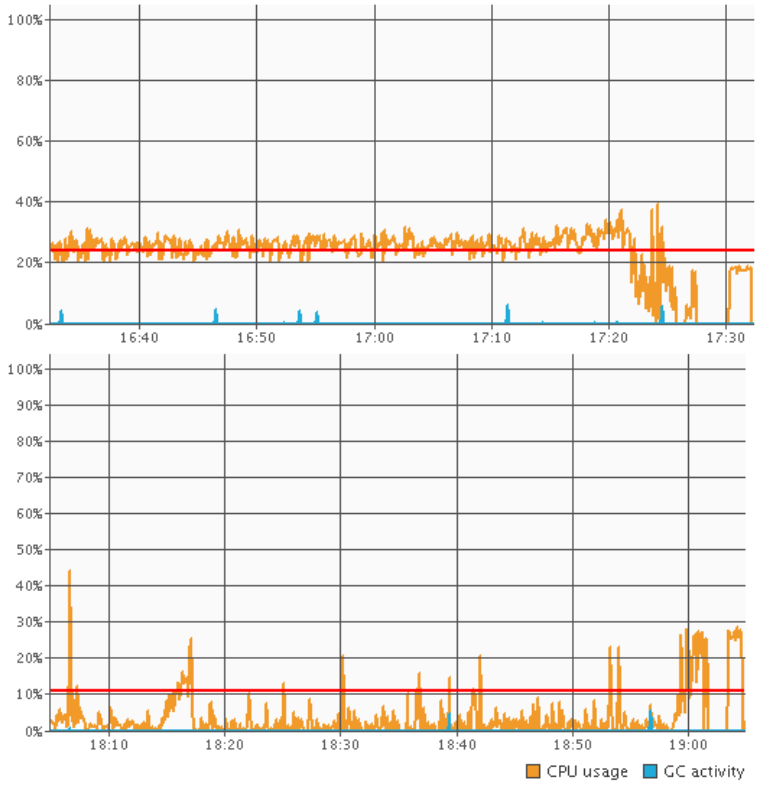


Figura 4.10: Comparació del consum de CPU entre versions de COMPSs.

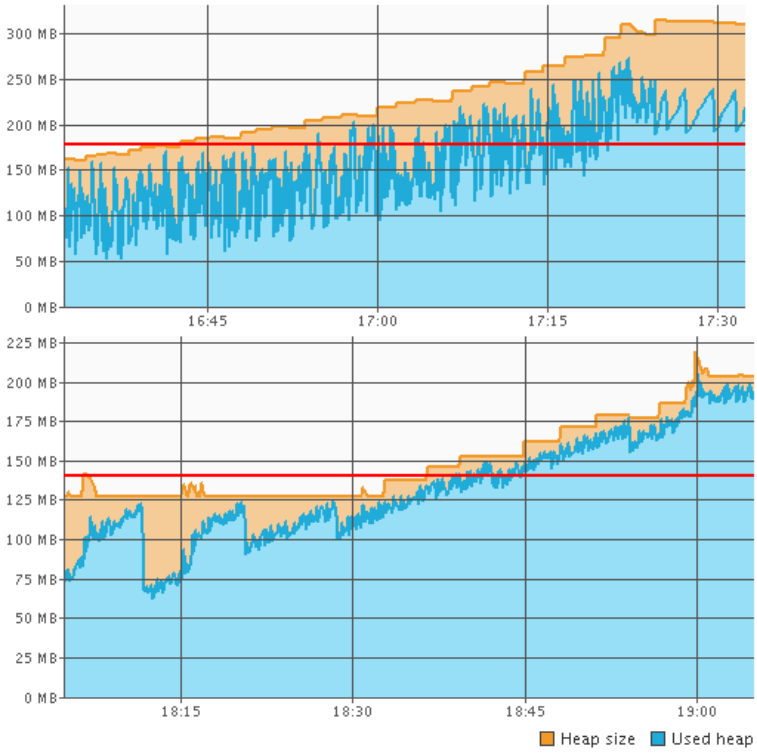


Figura 4.11: Comparació del consum de memòria entre versions de COMPSs.

Part IV

Conclusions i treball futur

Capítol 5

Conclusions del projecte

5.1 Conclusions

L'objectiu d'aquest projecte era desenvolupar un conjunt d'extensions per COMPS Superscalar amb l'objectiu de millorar-ne la planificació de recursos, habilitar un sistema de gestió i localització de rèpliques i millorar la gestió de fallades. Es pot afirmar que s'ha assolit la meta definida inicialment, ja que tot el conjunt d'extensions proposades milloren el rendiment global del sistema convertint COMPSs en un entorn encara més robust i potent.

Tal i com mostra el capítol anterior, els resultats obtinguts són força positius. El disseny del nou planificador ha aportat a COMPSs la capacitat de tenir més *feedback* del Grid, capacitant-lo per decidir a partir de més paràmetres que el planificador original, aconseguint així decisions més precises.

Per altra banda, s'ha demostrat també l'efectivitat del sistema de gestió de rèpliques, que ha aportat a COMPSs la possibilitat de representar còpies dels fitxers d'entrada que es troben als nodes del Grid. D'aquesta manera s'evita haver de transferir cada vegada aquests fitxers, movent-los només en cas d'haver patit modificacions. D'aquesta manera, com més gran és la mida dels fitxers d'entrada, més beneficis aporta aquesta extensió.

A l'apartat de gestió de fallades s'ha mostrat com el sistema és capaç de variar de forma satisfactòria l'assignació de tasques segons el % de fiabilitat dels recursos retirant de l'execució, si cal, els recursos que superin el nombre màxim de fallades permès.

D'altra banda, l'últim aspecte a destacar és la reducció del consum de recursos. Com s'ha vist, tant el consum de CPU com el de memòria és menor en la nova versió. Aquesta, bo i implementar noves funcionalitats, consumeix entre un 21% i 24% menys de memòria gràcies al conjunt d'optimitzacions realitzades, que han permès reduir substancialment el nombre de transferències entre components.

Finalment, deixant de banda els aspectes més pràctics, és important ressaltar el caràcter heterogeni del projecte, abordant diverses àrees de l'enginyeria, des de mètodes d'obtenció d'informació, gestió d'històrics, anàlisi de rendiment, aplicant també conceptes de diversos camps com sistemes operatius, enginyeria del *software* o la programació en diversos llenguatges. Tot plegat fa que el desenvolupament d'aquest projecte hagi resultat molt enriquidor.

5.2 Treball futur

El prototip desenvolupat demostra que les millores plantejades inicialment funcionen i efectivament milloren el rendiment global de la versió original de COMPSs. De totes maneres, tot i haver superat els objectius, el prototip no és més que una versió preliminar, quedant encara força camí per explorar.

Aquestes són algunes de les línies d'evolució proposades per poder continuar el desenvolupament d'aquesta nova versió:

Tal i com s'ha mencionat a la secció 2.5, el *runtime* de COMPSs es troba implementat sobre ProActive que, recordem, dóna la possibilitat de distribuir els components a través de diversos recursos per poder repartir la càrrega de treball dels components.

Recordem també que la transferència de fitxers es realitza través de JavaGAT, que es val d'un adaptador per transferir-los a través de SSH. Si hi ha moltes transferències, la càrrega del File Transfer Manager pot arribar a ser alta, podent saturar la màquina on es troba el *runtime*, en cas de no trobar-se distribuït.

De totes maneres, la realitat és més aviat una altra. Poques vegades s'arriba a aquest punt. Analitzant ProActive s'observa que consumeix una gran quantitat de recursos en haver de copiar a memòria els objectes que transfereix entre els components, tal i com es veu a l'últim apartat de *Tests, experiments i resultats*. Per aquest motiu caldria treballar per desenvolupar un *runtime* que no utilitzés cap *middleware* de componentització mantenint cada component diferenciat i fent que cada un s'executés en un *thread* independent sobre una màquina amb 5 o 6 nuclis per poder tenir-ne almenys un per component.

Un altre pas a seguir seria prescindir de l'extensió de mesura i actualització de la velocitat de xarxa passant a utilitzar algun tipus de *Grid Information Service* com Ganglia i provar d'obtenir dades d'estat i velocitat de la xarxa a través d'ell, verificant-ne posteriorment el resultat.

Per seguir afinant el model, caldria provar-lo també en un entorn més gran com per exemple un supercomputador; així es podria observar l'escalabilitat del sistema en un entorn més real.

A nivell d'optimitzacions, recordem que cada vegada que es planifica una tasca, abans d'aplicar al conjunt de recursos la funció d'avaluació es fa una preselecció dels recursos que compleixen les característiques per poder executar aquella aplicació. L'avaluació d'aquestes característiques es fa comparant la descripció de cada recurs, especificada al fitxer *resources.xml*, amb les restriccions programades per l'usuari a la interfície de definició dels mètodes de l'aplicació que s'executaran al Grid. Actualment això es fa comparant aquestes restriccions contra l'arbre XML de definició de cada recurs carregat en memòria en iniciar el *runtime*.

Aquestes comparacions es fan utilitzant Xalan i el llenguatge XPath consumint bastants recursos. La manera d'evitar-ho podria ser carregant, a l'inici, totes les dades proporcionades pels fitxers XML en una estructura de dades de més ràpid accés per tal d'agilitar la planificació de tasques.

Com es veu, queda encara treball per fer, però en aquest cas no era abarcable dins el marc del projecte. De totes maneres, els resultats obtinguts són més que satisfactoris i superan tots els objectius inicialment plantejats.

5.3 Anàlisi final de la planificació

Al llarg de la realització del projecte han anat sorgint tot un seguit d'inconvenients que han fet variar-ne la planificació inicial presentada en l'apartat 1.4.

L'apartat de familiarització es va planificar per poder ser desenvolupat en 100 hores, encara que només en van caldre 76. Això és perquè l'apartat de comprensió inicial es veié retallat en 4 dies per poder passar a treballar com més aviat millor en l'apartat de gestió de rèpliques. Aquesta millora havia de servir també com a aportació al projecte europeu IS-ENES [1] en el qual el BSC participa. Gràcies a això es va poder avançar la planificació del projecte 6 dies més.

El procés d'implementació va ser elaborat dins els marges previstos. En un principi es planificaren els temps de forma que no n'hagués de faltar. De totes maneres, en l'única part on es van trobar més problemes va ser en la gestió de l'històric. Es va haver de perdre més temps del planificat fent proves per poder assegurar que els valors que s'hi guardaven eren els correctes i necessaris per poder obtenir el millor rendiment del planificador. D'altra banda i per aquest mateix motiu, es van aplicar més esforços dels previstos per poder implementar de la forma més acurada possible la mesura dels temps d'execució i d'espera en cua.

L'apartat optimitzacions va prendre finalment 8 dies quan s'havia planificat per durar-ne 12. Això és perquè es va voler realitzar l'avaluació de rendiment com més aviat possible per saber com de lluny o a prop s'estava de la versió original de COMPSs. Per ordre de prioritats es va decidir comprovar abans d'iniciar un procés d'optimització que de ben segur s'hagués pogut allargar molt.

A més, durant els tests d'avaluació de rendiment es detectaren també alguns problemes d'estabilitat que van haver de ser resolts consumint 2 dies més dels inicialment previstos. De totes maneres, aquesta dilatació temporal va poder ser mitigada sense problemes en l'apartat de redacció de la memòria, on inicialment es van preveure alguns dies més dels necessaris, que han permès concloure el projecte en menys temps de l'esperat.

A la taula 5.1 es mostra la variació d'hores finals, ressaltant en vermell aquelles etapes en què s'ha trigat més temps del planificat i en verd aquelles que s'han pogut completar en menys.

[1] https://is.enes.org

Etapa	Planificació inicial	Planificació real
Familiarització	**100**	**76**
Definició del projecte	12	12
Anàlisi i documentació inicial	32	24
Comprensió de COMPSs	56	40
Implementació	**432**	**444**
Gestió de rèpliques	56	52
Test gestió de rèpliques	36	40
Disseny del planificador	56	56
Gestió de l'històric	28	32
Tolerància a fallades	52	52
Càlcul de velocitat de xarxa	60	60
Mesura del temps d'espera en cua	48	52
Mesura del temps d'execució	28	32
Test planificador	68	68
Optimitzacions	**40**	**32**
Avaluació	**120**	**100**
Correcions finals	**24**	**32**
Memòria	**244**	**200**
Total	**960**	**890**

Taula 5.1: Hores finals dedicades a cada una de les etapes del projecte.

Part V

Apèndix

Tal i com s'ha mencionat en l'apartat 3.7, Gestió de recursos, es mostren aquí alguns exemples reals dels fitxers de configuració: ***project.xml, resources.xml i historical.xml.***

El següent exemple mostra el fitxer *project.xml*, on es pot apreciar la definició dels Workers de l'aplicació, datanodes, el màxim de reintents permesos en cas de fallada i el conjunt de localització de rèpliques dels fitxers.

```
<?xml version="1.0" encoding="UTF-8"?>

<Project>
  <Worker Name="bscgrid02.bsc.es">
     <InstallDir>/home/username/IT_worker/</InstallDir>
     <WorkingDir>/home/username/IT_worker/files/</WorkingDir>
     <User>username</User>
  </Worker>

  <Worker Name="bscgrid03.bsc.es">
     <InstallDir>/home/username/IT_worker/</InstallDir>
     <WorkingDir>/home/username/IT_worker/files/</WorkingDir>
     <User>username</User>
  </Worker>

  <DataNode Name="bscgrid04.bsc.es">
     <User>username</User>
  </DataNode>

  <MaxRetries>2</MaxRetries>

  <Locations>
    <File LastModDate="1292140891000" Name="dbF0">
      <Path>file://bscgrid05.bsc.es/home/username/appfiles/</Path>
      <Path>file://bscgrid02.bsc.es/home/username/IT_worker/files/</Path>
      <Path>file://bscgrid03.bsc.es/home/username/IT_worker/files/</Path>
    </File>
    <File LastModDate="1292140891000" Name="dbF1">
      <Path>file://bscgrid05.bsc.es/home/username/appfiles/</Path>
      <Path>file://bscgrid02.bsc.es/home/username/IT_worker/files/</Path>
      <Path>file://bscgrid03.bsc.es/home/username/IT_worker/files/</Path>
    </File>
    <File LastModDate="1292140891000" Name="dbF10">
      <Path>file://bscgrid05.bsc.es/home/username/appfiles/</Path>
      <Path>file://bscgrid03.bsc.es/home/username/IT_worker/files/</Path>
      <Path>file://bscgrid02.bsc.es/home/username/IT_worker/files/</Path>
    </File>
    ...
  </Locations>
</Project>
```

El següent exemple mostra el fitxer *resources.xml*, on es pot observar la definició de recursos seguint l'estàndard d'OGSA anomenat *Grid Information and Data Modelling*.

```
<?xml version="1.0" encoding="UTF-8"?>

<ResourceList>
   <Resource Name="bscgrid02.bsc.es">
           <Capabilities>
                   <Host>
                           <TaskCount>0</TaskCount>
                           <Queue>short</Queue>
                           <Queue/>
                   </Host>
                   <Processor>
                           <Architecture>Intel</Architecture>
                           <Speed>3.6</Speed>
                           <CPUCount>2</CPUCount>
                    </Processor>
                    <OS>
                            <OSType>Linux</OSType>
                            <MaxProcessesPerUser>32</MaxProcessesPerUser>
                    </OS>
                    <StorageElement>
                            <Size>60</Size>
                    </StorageElement>
                    <Memory>
                            <PhysicalSize>0.5</PhysicalSize>
                            <VirtualSize>8</VirtualSize>
                    </Memory>
                    <ApplicationSoftware>
                            <Software>Xerces</Software>
                            <Software>Xalan</Software>
                    </ApplicationSoftware>
                    <Service/>
                    <VO/>
                    <Cluster/>
                    <FileSystem/>
                    <NetworkAdaptor>
                            <NetworkSpeed>91.2</NetworkSpeed>
                    </NetworkAdaptor>
                    <JobPolicy/>
                    <AccessControlPolicy/>
            </Capabilities>
           <Requirements/>
   </Resource>
...
</ResourceList>
```

El següent exemple mostra el fitxer *historical.xml*, on es poden apreciar les dades de fiabilitat dels recursos, temps mig d'execució de cada tipus de tasca i recurs, i la matriu de velocitat de xarxa.

```
<?xml version="1.0" encoding="UTF-8"?>

<RunStatistics>
 <Availability>
     <Worker Name="bscgrid03.bsc.es">1.0</Worker>
     <Worker Name="bscgrid02.bsc.es">1.0</Worker>
 </Availability>

 <MeanExecTime>
     <TaskType Id="2">
       <Worker Name="bscgrid03.bsc.es">8.587157</Worker>
       <Worker Name="bscgrid02.bsc.es">6.7536664</Worker>
     </TaskType>
     <TaskType Id="1">
       <Worker Name="bscgrid03.bsc.es">0.9715135</Worker>
       <Worker Name="bscgrid02.bsc.es">1.2075663</Worker>
     </TaskType>
     <TaskType Id="0">
       <Worker Name="bscgrid03.bsc.es">19.917158</Worker>
       <Worker Name="bscgrid02.bsc.es">20.031963</Worker>
     </TaskType>
 </MeanExecTime>

 <NetSpeed>
   <Link Src="bscgrid03.bsc.es">
      <Speed Dst="bscgrid02.bsc.es">82.550415</Speed>
      <Speed Dst="bscgrid05.bsc.es">73.84643</Speed>
      <Speed Dst="bscgrid03.bsc.es">181.67133</Speed>
   </Link>
   <Link Src="bscgrid05.bsc.es">
      <Speed Dst="bscgrid02.bsc.es">91.2</Speed>
      <Speed Dst="bscgrid05.bsc.es">547.2</Speed>
      <Speed Dst="bscgrid03.bsc.es">91.2</Speed>
   </Link>
   <Link Src="bscgrid02.bsc.es">
      <Speed Dst="bscgrid02.bsc.es">91.2</Speed>
      <Speed Dst="bscgrid05.bsc.es">73.417908</Speed>
      <Speed Dst="bscgrid03.bsc.es">63.571274</Speed>
   </Link>
 </NetSpeed>
</RunStatistics>
```

Bibliografia

[1] I. Foster. What is the Grid? - A three point checklist. GRIDtoday, (6), Juliol 2002. (Disponible a http://www.mcs.anl.gov/ itf/Articles/WhatIsTheGrid.pdf)

[2] I. Foster. The Physiology of the Grid - An Open Grid Services Architecture for Distributed Systems Integration. Juny 2002 (Disponible a http://www.mcs.anl.gov/uploads/cels/papers/P1249.pdf)

[3] I. Foster, "The Grid: A New Infrastructure for 21st Century Science,"Preprint ANL/MCS-P937-0202, Febrer 2002. (Disponible a http://www.mcs.anl.gov/uploads/cels/papers/P937.pdf)

[4] E.J. Stokes, S.Andreozzi, M.Drescher, A.Savva. Information and Data Modeling in OGSA Grids (GFD-I.137), Juliol 08. (Disponible a: www.gridforum.org/documents/GFD.137.pdf)

[5] Luis F. G. Sarmenta. Volunteer Computing. Ph.D. thesis. Dept. of Electrical Engineering and Computer Science, MIT, Març 2001. (Disponible a http://bayanihancomputing.net/sarmenta-phd-mit2001.pdf)

[6] Luis F. G. Sarmenta. Sabotage-Tolerance Mechanisms for Volunteer Computing Systems. ACM/IEEE International Symposium on Cluster Computing and the Grid (CCGrid'01), Brisbane, Australia, May 15-18, 2001. (Best Paper Finalist) (Disponible a http://www.cag.lcs.mit.edu/bayanihan/papers/ccgrid01/ccgrid01.pdf)

[7] R. Buyya, C. S. Yeo i S. Venugopal. Market-oriented cloud computing: Vision, hype, and reality for delivering it services as computing utilities. In HPCC '08: Proceedings of the 2008 10th IEEE International Conference on High Performance Computing and Communications. Washington, DC, USA: IEEE Computer Society, 2008.

[8] An EGEE Comparative Study, Grids and Clouds - Evolution or Revolution?, Maig 2005. (Disponible a https://edms.cern.ch/file/925013/3/EGEE-Grid-Cloud.pdf)

[9] Judith M. Myerson, Cloud Computing versus Grid Computing, Març 2009. (Disponible a http://www.ibm.com/developerworks/web/library/wa-cloudgrid/)

[10] CoreGRID, Programming Model Institute: Basic features of the grid component model (assessed), Deliverable D.PM.04 (2006), (Disponible a: http://www.coregrid.net/mambo/images/stories/Deliverables/d.pm.04.pdf)

[11] A.Cansado, D.Caromel, L.Henrio, E.Madelaine, M.Rivera, and E.Salageanu, "A Specification Language for Distributed Components Implemented in GCM/ProActive", 2008 (Disponible a: http://www.springerlink.com/index/g551w852k05pp307.pdf)

[12] J.Dean i S.Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. In OSDI 04, 6th Symposium on Operating Systems Design and Implementation, Desembre 2004(Disponible a http://labs.google.com/papers/mapreduce-osdi04.pdf)

[13] R.Lämmel. Google's MapReduce Programming Model, Octubre 2007 (Disponible a http://portal.acm.org/citation.cfm?id=1290812)

[14] E.Tejedor, R.M Badia. COMP Superscalar: Bringing GRID superscalar and GCM together, Maig 2008

[15] COMP Superscalar User Guide, Abril 2010 (Disponible a http://www.bsc.es/media/3934.pdf).

[16] Shigeru Chiba. ECOOP 2000, Object-Oriented Programming, LNCS 1850, Springer Verlag, page 313-336, 2000. (Disponible a http://www.csg.is.titech.ac.jp/ chiba/pub/chiba-ecoop00.pdf)

[17] A.Anciaux, R.M Badia. R.Sirvent, J.M Pérez, Grid Superscalar and job mapping on the reliable grid resources, 2008 (Disponible a http://www.cs.vu.nl/ kielmann/papers/ani-reliable.pdf).

[18] M.L.Massie, B.N.Chun, D.E.Culler, The ganglia distributed monitoring system: design, implementation and experience, 2004 (Disponible a http://ganglia.info/papers/science.pdf).

[19] Wikepedia (http://www.wikipedia.org)

Glossari

API Application Program Interface o interfície de programació d'aplicacions.

Datacenter Es denomina Datacenter o centre de processament de dades a aquella ubicació on es concentren tots els recursos necessaris per al processament de la informació d'una organització determinada.

FIFO És l'acrònim en anglès de First In, First Out (primer en entrar, primer en sortir). És un mètode utilitzat en estructures de dades i teoria de cues. Guarda analogia amb les persones que esperen en una cua i que són ateses en l'ordre en què han arribat, és a dir, la primera persona que hi entra és la primera que en surt.

Firewall Un firewall o tallafocs és la part de la xarxa dissenyada per bloquejar l'accés no autoritzat permetent al mateix temps les comunicacions autoritzades.

Grid Sistema que coordina recursos que no estan subjectes a un control centralitzat, utilitzant protocols estàndards, oberts, de propòsit general i interfícies per donar unes qualitats de serveis no trivials [2].

Runtime Es tracta habitualment de *software* dissenyat per donar suport i ajudar a computadors en l'execució d'aplicacions. És el cas, per exemple, d'aplicacions paral·leles.

Stream Fa referència a un flux continu de dades (sense interrupció).

TCP/IP El model TCP/IP descriu un conjunt de guies de disseny i implementacions de protocols de xarxa específics que permeten a un computador comunicar-se en una xarxa. TCP/IP proveeix de connectivitat extrem a extrem especificant com les dades han de ser formatejades, direccionades, transferides i rebudes per part de l'origen i el destinatari.

Thread Fil d'execució d'una aplicació. La unitat més petita que pot ser planificada pel sistema operatiu.

URI Uniform Resource Identifier és una cadena curta de caràcters que identifica inequívocament un recurs (servei, pàgina, document, adreça de correu electrònic, etc...) Normalment accessible a través de la xarxa o sistema.

VPN Una xarxa privada virtual o VPN (en anglès Virtual Private Network), és una tecnologia de xarxa que permet estendre una xarxa local sobre una xarxa pública no controlada, com per exemple Internet.

Signat,

Bellaterra, ... de ..... de 2010

Resums

Català:

COMPS és un entorn de programació paral·lela desenvolupat per BSC-CNS. Aquest projecte busca estendre aquest entorn per tal de dotar-lo de funcionalitats inicialment no suportades. Aquest conjunt d'extensions radiquen principalment en la implementació de mecanismes que permetin incrementar la flexibilitat, robustesa i polivalència del sistema.

Castellano:

COMPSs es un entorno de programación paralela desarrollado por BSC-CNS. Este proyecto busca extender el entorno para dotarlo de funcionalidades inicialmente no soportadas. Este conjunto de extensiones radican principalmente en la implementación de mecanismos que permitan incrementar la flexibilidad, robustez y polivalencia del sistema.

English:

COMPSs is a parallel programming environment developed by BSC-CNS. The project aim is extend this environment giving it some features that were not initially supported. This set of extensions lies mainly in the implementation of mechanisms that allow to increase the flexibility, robustness and system versatility.