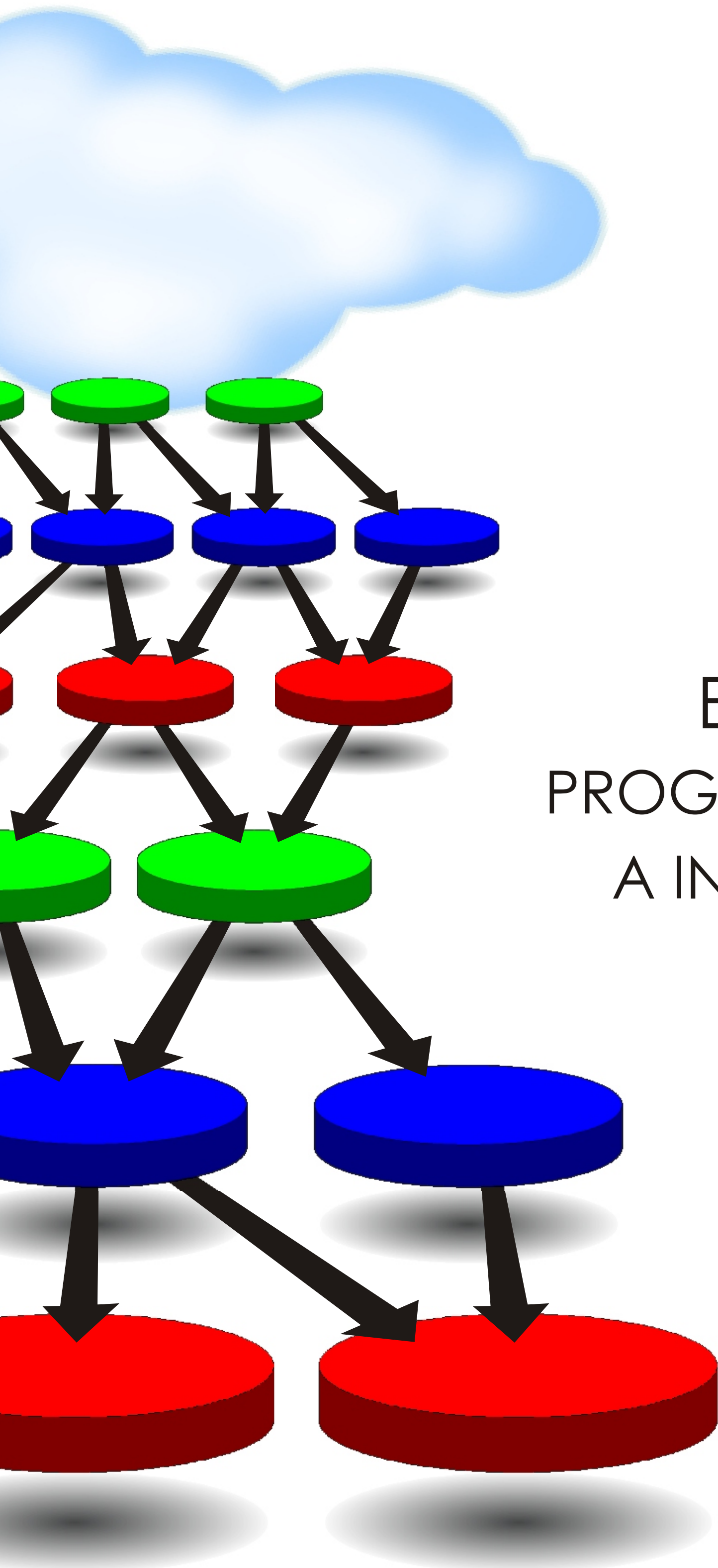

PROJECTE FINAL DE CARRERA

ENGINYERIA EN INFORMÀTICA

EXTENSIONS AL MODEL DE PROGRAMACIÓ COMPSs PER A INFRAESTRUCTURES CLOUD

Alumne: FRANCESC-JOSEP LORDAN GOMIS
Directora: DRA. ROSA MARIA BADIA SALA

Volum: 1/1

Data: 25 de Gener de 2011

DADES DEL PROJECTE

Títol del Projecte: Extensions al model de programació COMPSs per a infraestructures Cloud

Nom de l'estudiant: Francesc-Josep Lordan Gomis

Titulació: Enginyeria en Informàtica

Crèdits: 37,5

Director/Ponent: Rosa Maria Badia Sala

Departament: Arquitectura de Computadors

MEMBRES DEL TRIBUNAL (nom i signatura)

President: Antonio Cortés Rosselló

Vocal: Narcís Nabona Francisco

Secretari: Jesus Jose Labarta Mancho

QUALIFICACIÓ

Qualificació numèrica:

Qualificació descriptiva:

Data:

Agraïments

En aquestes línies voldria mostrar el meu agraïment a totes les persones que m'han ajudat a fer possible aquest projecte. Molt em temo que me'n deixaré alguna i no de forma premeditada sinó per la gran quantitat que n'hi ha.

En primer lloc, vull agrair sincerament a la Rosa Badia, directora del projecte, no només el seu suport i dedicació al llarg de tot el projecte sinó també la confiança que va donar-me en el moment en que li vaig demanar fer un projecte relacionat amb la computació per Grid.

Seguidament vull donar les gràcies a tota la gent del BSC-CNS que m'ha ajudat i donat suport al llarg de tot el projecte. Per un costat, a tota la gent de l'equip de Grid Computing and Clusters per la paciència d'explicar-me tot el funcionament de COMPSs, les idees que m'han anat donant per resoldre cada un dels problemes que han aparegut i el bon ambient de treball al llarg de tot el projecte. Per l'altre, a la gent que durant aquest temps ha format part de l'equip d'Autonomic Systems and e-Business Platforms per tot el suport i ajuda amb els problemes que he tingut amb EMOTIVE Cloud. Sense tots ells, aquest projecte hagués estat una tasca molt més complicada.

Per acabar vull agrair a tota la meva família i als meus amics i companys l'interès, paciència i comprensió que han mostrat durant el que ha durat el projecte. El seu suport ha estat inestimable especialment aquests últims mesos de projecte.

De tot cor, moltes gràcies a tots.

Francesc-Josep Lordan Gomis

Índex

Introducció 1

1 Introducció 3
- 1.1 Motivació . . . 3
- 1.2 Objectius . . . 4
- 1.3 Metodologia i planificació inicial . . . 5
- 1.4 Organització de la memòria . . . 8

Entorn del projecte i estat de l'art 9

2 Cloud Computing 11
- 2.1 Què és Cloud Computing . . . 11
- 2.2 Avantatges i Inconvenients del Cloud . . . 13
- 2.3 Tipus de serveis que ofereix l'Utility Computing . . . 14
- 2.4 Anàlisi d'alguns proveïdors d'Utility Computing . . . 15
  - 2.4.1 Amazon Elastic Cloud Computing "EC2" . . . 16
  - 2.4.2 RackSpace . . . 17
    - Cloud Servers . . . 17
    - Cloud Sites . . . 18
    - Cloud Files . . . 18
  - 2.4.3 GoGrid . . . 18
    - Cloud Servers . . . 18
    - Cloud Storage . . . 19
    - Load Balancer . . . 19
  - 2.4.4 AppNexus . . . 19
  - 2.4.5 IBM Smart Business . . . 20
  - 2.4.6 Microsoft Azure . . . 21
    - Storage . . . 22
    - Compute . . . 23
    - Fabric . . . 24
    - Preus . . . 24
  - 2.4.7 Google AppEngine . . . 25
    - Preus . . . 25
- 2.5 Actors d'un sistema d'Utility Computing . . . 26
- 2.6 Cloud privats . . . 27
  - 2.6.1 EUCALYPTUS . . . 28
  - 2.6.2 OpenNebula.org . . . 28
  - 2.6.3 EMOTIVE Cloud . . . 29
  - 2.6.4 IBM CloudBurst . . . 30

3 Models de Programació 31
3.1 Patró Master-worker 31
3.2 Grid Component Model(GCM) i ProActive 32
3.3 Microsoft Dryad 35
3.4 Google MapReduce 36

4 COMP Superscalar 39
4.1 Model de Programació 40
4.1.1 Selecció de tasques i definició de la interfície 41
4.1.2 API de COMPSs 43
4.2 Runtime 44
4.2.1 Arquitectura de Components 44
4.2.2 Integració dels Components 45
Camí d'una tasca 45
Camí d'un openFile 47
4.3 Execució de tasques remotes 48
4.4 Fitxers de configuració 48
Definció dels recursos 49
Definció de l'entorn en el worker 49

Desenvolupament del Projecte 51

5 Desenvolupament del Projecte 53
5.1 Directrius del disseny 53
5.2 Arquitectura del Projecte 55
5.2.1 Preparació dels workers virtuals 57
5.2.2 Implementació del Connector 58
APIs del Connector 58
Implementació del Connector prototip 59
5.2.3 Instanciació del Connector 64
5.3 Gestio de Recursos 64
5.3.1 ProjectManager 65
5.3.2 ResourceManager 66
5.3.3 QueueManager 67
5.4 Planificació de tasques 68
5.4.1 Canvis en l'entrada d'una tasca al runtime 69
5.4.2 Alliberament de les dependències d'una tasca 71
5.4.3 Una tasca acaba correctament l'execució 72
5.4.4 Una tasca falla l'execució 75
5.5 Procés de creació i destrucció de màquines 76
5.5.1 Procés de creació 76
5.5.2 Procés de destrucció 78
5.6 Demanar nous recursos al Cloud 84
5.6.1 Política de recursos inicials 84
5.6.2 Implementació de la política de recursos inicials 85
5.6.3 Política de demanda periòdica 87
5.6.4 Implementació de la política de demanda periòdica 89
5.7 Alliberar recursos del Cloud 90
5.7.1 Política de destrucció a l'indicar que no hi haurà noves tasques . . . 90

5.7.2 Política de destrucció al moment de la creació 92
5.7.3 Polítiques de destrucció al final de l'execució 93
5.7.4 Política de destrucció al completar una tasca 94
5.7.5 Polítiques de destrucció periòdica 95
5.7.6 Implementació de la política de destrucció al llarg de l'execució . . . 98
5.8 Exemple d'aplicació de les polítiques de creació i alliberament de recursos . 101

Anàlisi del Prototip 105

6 Anàlisi del Prototip 107
6.1 Entorn de proves 107
6.2 Aplicacions 108
6.2.1 SparseLU 108
6.2.2 Hmmer 110
6.2.3 Stream 111
6.3 Rendiment de la planificació 112
6.4 Dependència del Cloud Provider 114
6.5 Escalar aplicacions 117
6.5.1 Hmmer 118
1 tipus de tasca 118
2 tipus de tasca 120
6.5.2 SparseLU 122
1 tipus de tasca 123
2 tipus de tasca 123
3 tipus de tasca 124
6.5.3 Stream 125
1 tipus de tasca 125
2 tipus de tasca 127
6.6 Consum de recursos del Master 128
6.7 Comparació de costos entre Grid i Cloud 131

Conclusions del Projecte 135

7 Conclusions del Projecte 137
7.1 Conclusions 137
7.1.1 Conclusions del Prototip 137
7.1.2 Conclusions del Cloud Computing 139
7.2 Planificació 140
7.3 Anàlisi econòmic 144
Recursos Humans 144
Hardware utilitzat 144
Software utilitzat per desenvolupar el projecte 145
Software utilitzat per escriure la memòria i fer la defensa 145
Anàlisi del cost total 146
7.4 Treball futur 146
7.5 Valoració personal 148

Apèndixs **151**

A Organització de l'entorn de treball de COMPSs **153**

B Contingut del fitxer adjunt a la memòria **155**

C Instal·lació de COMPSs i execució d'una aplicació **157**

C.1 Preparació del master . . . 157

C.1.1 Java . . . 157

C.1.2 Apache Ant . . . 158

C.1.3 ProActive . . . 158

C.1.4 JavaGAT . . . 158

C.1.5 COMPSs . . . 158

C.2 Preparació d'EMOTIVE Cloud . . . 159

C.2.1 Extensió del Scheduler . . . 159

C.2.2 Afegir la imatge dels workers . . . 160

C.3 Compilació i execució d'aplicacions . . . 160

Bibliografia **163**

Índex de figures

1.1 Diagrama de Gantt de la planificació inicial del projecte 7

2.1 Estructura de serveis 15
2.2 Diferents tipus d'api segons el servei que oferim 15
2.3 Windows Azure té tres parts principals: el servei de Compute, el de Storage i el Fabric 22
2.4 Azure permet emmagatzemmar dades en blobs, tables i Queues accessibles mitjançant HTTP 22
2.5 Una aplicació de Microsoft Azure està formada per instàncies Web Role i/o instàncies Worker Role, cada una de les cuals té la seva pròpia màquina virtual amb Windows 23
2.6 Actors que trobem al llarg de l'oferiment d'un SaaS que està en un Cloud . 27
2.7 Jerarquia lògica d'EMOTIVE Cloud 29

3.1 Flux d'execució d'una aplicació amb el patró master-worker 32
3.2 Component compost definit segons Fractal 33
3.3 Organització del sistema de Microsoft Dryad 35
3.4 Operadors per crear el graf acíclic dirigit d'una aplicació a Microsoft Dryad 36
3.5 Exemple de definició d'una graf de flux d'aplicació en Microsoft Dryad . . . 36
3.6 Flux d'execució d'una crida a MapReduce 38

4.1 Arquitectura lògica de COMPSs 40
4.2 Components del runtime de COMPSs 45
4.3 Diagrama de flux bàsic d'una tasca dintre del runtime 47

5.1 Arquitectura lògica del projecte 57
5.2 Diagrama de seqüència del la creació de noves màquines 62
5.3 Estructures de dades de gestió que trobem en el TaskScheduler 65
5.4 Diagrama de classes resumit del FileInfoProvider 70
5.5 Diagrama de seqüència a l'entrar una nova tasca al TaskScheduler 72
5.6 Diagrama de seqüència de l'actualització de les estructures al acabar una tasca 73
5.7 Diagrama de seqüència de l'elecció d'una tasca a re-planificar al acabar una tasca 74
5.8 Diagrama de seqüència al rebre una notificació de tasca fallida 76
5.9 Diagrama de seqüència al final de la creació d'una màquina que ja no serà útil 77
5.10 Diagrama de seqüència al final de la creació d'una màquina que serà útil . . 78
5.11 Diagrama de seqüència de manar la destrucció d'una màquina (TaskScheduler) 80

5.12 Diagrames de classe (parcials) del FileTransferManager de les dues versions de COMPSs . . . 82
5.13 Graf de dependències que pot portar a errors a la destrucció de màquines a l'acabar una tasca . . . 95
5.14 Graf de dependències marcant les tasques que es faran servir per la política de destrucció periòdica . . . 96
5.15 Graf de flux de l'aplicació d'exemple d'aplicació de les polítiques de creació i alliberament . . . 102
5.16 Llegenda de l'histograma d'exemple 5.17 . . . 102
5.17 Traça d'exemple d'aplicació de les diferents polítiques de creacií alliberament de recursos . . . 102

6.1 Esquema de la xarxa utilitzada com a testbed . . . 107
6.2 Graf de dependències del SparseLU . . . 109
6.3 Graf de dependències de l'aplicació HMMER . . . 111
6.4 Graf de dependències de l'aplicació Stream . . . 112
6.5 Diagrama de Gantt per una execució del Hmmer amb 512 seqüències i 8 workers . . . 115
6.6 Comparació entre número de workers solicitats a l'inici i el temps utilitzats per executar Hmmers de 512, 1024, 2048 i 4096 seqüències amb un cloud d'un i dos nodes i un grid. . . . 117
6.7 Llegenda pels gràfics d'execucions Hmmer amb 1 sol tipus . . . 118
6.8 Evolució dels recursos en un hmmer de 512 seqs. i 1 tipus de tasca . . . 119
6.9 Evolució dels recursos en un hmmer de 1024 seqs. i 1 tipus de tasca . . . 119
6.10 Evolució dels recursos en un hmmer de 2048 seqs. i 1 tipus de tasca . . . 119
6.11 Evolució dels recursos en un hmmer de 4096 seqs. i 1 tipus de tasca . . . 119
6.12 Evolució del temps de servir una petició en un node on s'acumulen peticions 120
6.13 Evolució del temps de servir una petició en un Cloud amb 2 nodes . . . 121
6.14 Llegendes pels gràfics d'execucions Hmmer amb 2 tipus . . . 121
6.15 Evolució dels recursos en un hmmer de 512 seqs. i dos tipus de tasca . . . 121
6.16 Evolució dels recursos en un hmmer de 1024 seqs. i dos tipus de tasca . . . 122
6.17 Evolució dels recursos en un hmmer de 2048 seqs. i dos tipus de tasca . . . 122
6.18 Evolució dels recursos en un hmmer de 4096 seqs. i dos tipus de tasca . . . 122
6.19 Llegenda pels gràfics d'execucions SparseLU amb 1 sol tipus de màquina . . 123
6.20 Evolució dels recursos en un SparseLU amb 1 sol tipus . . . 123
6.21 Llegendes pels gràfics d'execucions SparseLU amb 2 tipus . . . 124
6.22 Evolució dels recursos en un SparseLU amb 2 tipus . . . 124
6.23 Llegendes pels gràfics d'execucions SparseLU amb 3 tipus . . . 124
6.24 Evolució dels recursos en un SparseLU amb 3 tipus . . . 125
6.25 Llegenda pels gràfics d'execucions Stream amb 1 sol tipus de màquina . . . 126
6.26 Evolució dels recursos en un Stream amb 1 sol tipus . . . 126
6.27 Llegendes pels gràfics d'execucions Stream amb 2 tipus de màquina . . . 127
6.28 Evolució dels recursos en un Stream amb 2 tipus . . . 127
6.29 Comparació de l'ocupació de CPU entre les versions Grid i Cloud . . . 129
6.30 Comparació del consum de memòria entre les versions Grid i Cloud . . . 130
6.31 Relació entre els costos executant 8 recursos en el Cloud i fent executant un cluster de 2 nodes . . . 132

7.1 Diagrama de Gantt de la planificació real . . . 143
7.2 Percentatges del cost total i RRHH . . . 146

Índex de taules

2.1 Característiques i preus de les màquines ofertes per Amazon 16
2.2 Preus d'Amazon per les tranferències de sortida del Cloud 16
2.3 Tipus de màquines i preus disponibles pels Cloud Servers de RackSpace . . 17
2.4 Tipus de màquines i preus disponibles pels Cloud Servers de GoGRID . . . 19
2.5 Característiques de les màquines ofertes per AppNexus 20
2.6 Característiques de les màquines ofertes per IBM 21
2.7 Preus(€) de les instàncies pel Cloud d'IBM sense reserva 21
2.8 Característiques de les màquines ofertes per Windows Azure 24
2.9 Límits de consum de recursos acceptats per Google AppEngine 26
2.10 Cost dels recursos informàtics utilitzant AppEngine 26

6.1 Temps (segons) d'execució aïllant la planificació 113
6.2 Temps (segons) d'execució per HMMER de 512, 1024, 2048 i 4096 sequències en la versió Grid 115
6.3 Temps (segons) d'execució per HMMER de 512, 1024, 2048 i 4096 sequències en un Cloud d'un node 115
6.4 Temps (segons) d'execució per HMMER de 512, 1024, 2048 i 4096 sequències en un Cloud de dos nodes 116
6.5 Quantitat d'hores de computació necessàries perquè recòrrer al Cloud deixi de ser útil. 132

7.1 Canvis en el nombre d'hores dedicades a cada etapa 142
7.2 Quadre resum dels costos en recursos humans 144
7.3 Quadre resum dels costos en recursos hardware 145
7.4 Quadre resum dels costos en software per desenvolupar 145
7.5 Quadre resum dels costos en software per escriure la memòria 145
7.6 Quadre amb els costos totals del projecte agrupats per concepte 146

Introducció

Capítol 1

Introducció

Aquest projecte persegueix estendre el model de pregramació COMP Superscalar per tal que aquest sigui capaç d'aprofitar el paral·lelisme inherent a nivell de tasca de les aplicacions, executant-les no només en un Grid, sinó també en recursos demanats a alguna infraestructura Cloud. Al llarg d'aquest primer capítol es detalla què busca aquest projecte i com pretenem complir amb els objectius marcats.

1.1 Motivació

La computació en paral·lel és un dels factors accelerants del progrés tecnològic que vivim avui en dia. Aquesta tecnologia aporta una gran potència de càlcul que permet resoldre en molt menys temps problemes complexes i costosos, com són per exemple: l'anàlisi de models climàtics, la interacció de proteïnes o simulacions d'efectes físics com els túnels de vent o la física del plasma.

Un dels grans problemes que té és l'elevat cost dels grans clusters que permetin l'execució paral·lela de forma massiva. Conscients d'això, els governs que volen apostar per la investigació i el desenvolupament financen la construcció i el manteniment de grans supercomputadors, com és el cas del MareNostrum, que es troba al BSC-CNS (Barcelona Supercomputing Center - Centro Nacional de Supercomputación, `http://www.bsc.es`).

L'accés a aquests supercomputadors és difícil, és per això que moltes empreses i centres d'investigació han d'optar per la compra de clusters més petits limitant en part la seva potència de càlcul.

A principis dels anys 90, va aparèixer una solució al problema del cost, el Grid. Aquest permet l'execució paral·lela d'aplicacions en una infraestructura heterogènia de recursos distribuïts geogràficament compartida entre diferents organitzacions.

Els avenços en les tecnologies de telecomunicacions i de virtualització dels darrers anys han donat lloc a l'aparició d'una nova realitat comercial, la computació en Cloud. Aquest nou model de negoci permet a tots aquests centres i empreses accedir a recursos informàtics sense necessitat de pagar ni per l'adquisició ni pel manteniment, simplement pagar per l'ús dels recursos, donant així solució al gran problema del cost.

S'obre davant dels nostres ulls un dels somnis tecnològics més esperats, la computació com una utilitat. L'únic problema que hi ha actualment és que no hi ha cap model de programació capaç d'executar-se utilitzant recursos que es trobin tant en el Grid com en el

Cloud d'una manera fàcil i sense la necessitat de ser conscient de tot l'entorn d'execució.

La raó principal d'aquest projecte és crear una eina que permeti una programació fàcil i que amagui totes les característiques que estan implícites tant en el Grid com en el Cloud.

1.2 Objectius

Vista la necessitat d'execució en paral·lel a un baix cost, l'objectiu del projecte serà arribar a tenir un model de programació capaç d'executar aplicacions en un Grid i en el Cloud, podent-los combinar tots dos, amagant al programador tot detall tant del Grid com de la interfície encarregada de demanar més recursos al proveïdor del Cloud.

El BSC-CNS disposa d'un model de programació, COMP Superscalar[1], COMPSs, amb el seu corresponent runtime que permet l'execució d'aplicacions Java en un Grid. El mateix codi seqüencial serveix per executar de forma paral·lela sobre el Grid, de manera que ofereix una programació fàcil i que no requereix tenir consciència de l'entorn en el que l'aplicació s'executarà.

La meta del projecte serà ampliar el model de programació ja existent, COMP Superscalar, per tal que algunes o totes les tasques d'una aplicació puguin executar-se en el Cloud. En altres paraules, l'objectiu del projecte és ampliar COMP Superscalar per tal de que aquest sigui capaç de reservar i alliberar recursos del Cloud, ajustant la quantitat de recursos utilitzats a les necessitats de l'aplicació en temps real.

Entra en els objectius del projecte conèixer i aprofundir en el nou model que ens ofereix el Cloud. Tenir un punt de vista com a usuari de què és: veure les característiques de la tecnologia, els seus avantatges i els seus inconvenients. També caldrà analitzar la vessant més comercial, és a dir, veure quins serveis i a quins preus dóna el mercat, a quin tipus d'usuari arriba i a quins usos s'està aplicant. No serà necessari entendre tota la tecnologia que s'amaga darrera, veure detalls de la seva implementació ni quines eines o tecnologies utilitza. Únicament serà necessari veure quines possibilitats ofereix i de quina manera es pot utilitzar per tal de trobar la millor implementació possible del projecte.

Existeixen diferents proveïdors de recursos Cloud, i cada un té la seves particularitats, no només en la forma d'oferir els recursos, sinó també en la forma que es poden utilitzar. Degut a la varietat de l'oferta actual i a tota la que pot sortir en un futur, no cal que la solució plantejada pugui executar en totes elles. Es busca la preparar tota l'arquitectura necessària per utilitzar un servei qualsevol de Cloud Computing i implementar-ne un que serveixi de guia a l'hora d'ampliar-lo amb algun altre servei.

El prototip que sortirà d'aquest projecte únicament utilitzarà amb el middleware EMOTIVE Cloud. EMOTIVE Cloud és el software desenvolupat pel BSC-CNS per tal d'oferir solucions Cloud. Aquest s'encarrega de gestionar una sèrie de recursos físics per tal d'oferir als usuaris entorns virtualitzats i executar-hi tasques d'una forma senzilla i eficient.

L'elecció d'aquest software com a eina de gestió es deu principalment al fet de que el BSC-CNS disposa ja d'un petit cloud amb EMOTIVE Cloud que permetrà fer proves i comprovar que el nou runtime funcioni correctament. A més a més, COMP Superscalar també ha estat desenvolupat pel BSC-CNS. Així doncs, el resultat del projecte seria una solució corporativa al problema que planteja la programació utilitzant el Cloud.

A l'hora de gestionar recursos, les decisions es prenen en funció d'una sèrie de polítiques

[1] Queda descrit amb més detall a l'apartat 4

que caldrà definir al llarg del disseny del projecte. Per tal de valorar una política cal tenir en compte dos grans factors: el cost i el temps d'execució. Prendre la millor decisió en cada moment és un problema computacionalment costós. L'objectiu del projecte passa per determinar quines dades són necessàries per prendre aquestes decisions, ser capaç de calcular-les i definir algunes polítiques que tinguin un rendiment acceptable. No és l'objectiu del projecte definir una política òptima en quant a rendiment, cost o utilització dels recursos.

L'últim objectiu del projecte és avaluar el prototip resultant i amb ell la tecnologia Cloud; i així, veure quins són els punts fort del software implementat i de la tecnologia i en quins casos apareixen necessitats que provoquen pèrdues de rendiment o funcionalitat. Tant en uns casos com en altres, s'intentarà trobar una explicació a l'efecte obtingut i, en cas de ser negatiu, com podria millorar-se.

1.3 Metodologia i planificació inicial

L'apartat anterior defineix què és el que es vol desenvolupar al llarg d'aquest projecte. El primer apartat es descriu el perquè es vol desenvolupar i quina serà la seva utilitat. Per acabar d'explicar el projecte queda explicar com, quin procés es seguirà per desenvolupar-lo, i quan es desenvoluparà, és a dir, la planificació inicial d'aquest projecte.

El primer pas per poder començar a desenvolupar qualsevol projecte és detectar un problema o una oportunitat i a partir d'aquí definir el projecte, trobar exactament quins són els requisits que haurà de complir la solució.

Un cop ja està clar quins objectius persegueix el projecte, caldrà preparar tot l'entorn de treball i familiaritzar-se amb les eines que s'han de fer servir. En el cas concret d'aquest projecte això significava instal·lar tots els programes necessaris per desenvolupar el projecte: un IDE, un client de vpn, apache-ant i totes les dependències de software que tingués COMPSs i configurar-ho tot per tal de permetre les comunicacions entre les màquines de forma automàtica: vpn, generar i copiar les claus ssh. Per altra banda, també calia entendre el funcionament i el codi de COMPSs i conèixer les diferents funcionalitats que ofereix EMOTIVE Cloud.

Tenint clar què es vol fer i tenint tot el que es necessita per desenvolupar el projecte comença la resolució. En aquest projecte, per fer la resolució es segueix un procés iteratiu basat en sis etapes:

1. Creació de les màquines virtuals de forma manual i configuració de tot l'entorn necessari per tal de que COMPSs sigui capaç d'executar-hi tasques sense modificar el runtime ni el model de programació.

2. Creació de noves màquines virtuals de forma automàtica a l'inici de l'execució en funció dels requisits de les tasques i la conseqüent reconfiguració en temps d'execució del runtime en el moment en que s'acaba de tractar la petició. S'esperarà a començar a executar a que tinguem totes les màquines que s'han demanat.

3. Destrucció automàtica de les màquines virtuals un cop ja ha acabat tota l'execució.

4. Considerar el compromís entre els punts 1 i 2. Si pèrdua de temps degut a crear les màquines de manera automàtica és important caldrà buscar algun sistema que permeti solapar el temps de creació d'algunes màquines amb temps de computació.

5. Afegir dinamisme al runtime. El runtime serà capaç de cridar als mecanismes de creació i destrucció en qualsevol moment de l'aplicació. És en aquest moment on entraran les polítiques més bàsiques, es demanarà una nova màquina quan es tingui un nombre de tasques sense recurs on executar i s'apagarà quan no tingui cap tasca que executar. Aquesta iteració es divideix en 2 subiteracions diferents: una per solucionar el problema de la creació dinàmica de màquines i l'altra per destruir dinàmicament les màquines.

6. Creació de polítiques més complexes que permetin veure quin tipus de màquines és convenient demanar i detectar si se'n pot treure profit més endavant per no eliminar-la.

Cada una d'aquestes iteracions està dividida en 4 passos:

- Definir que busca la iteració i com es podria fer
- Fer el disseny corresponent a la iteració tenint en compte el que ja està implementat
- Implementar el nou disseny
- Revisar el correcte funcionament del runtime

Arribats a aquest punt el primer prototip ja està creat, només queda avaluar el rendiment que té. Cal trobar-ne els punts forts i febles. Detectar quina és la causa i intentar millorar els punts que fan perdre rendiment, com per exemple alguns paràmetres de les polítiques de creació i destrucció.

Un cop explicat com es desenvoluparà el projecte, falta veure la correspondència de la metodologia amb el temps del que es disposa. La normativa de la facultat marca que un projecte ha de desenvolupar-se en aproximadament de 600 hores. Les primeres 75 hores anirien dedicades a preparar i familiaritzar-se amb l'entorn.

En aquest moment, s'entra en el procés iteratiu del desenvolupament del prototip, al que es dedicarà un total de 375 hores, repartides de la següent manera: 25 hores per encendre manualment les màquines i configurar el runtime perquè sigui capaç d'executar-hi tasques. 75 hores per crear les màquines virtuals a l'inici de l'execució en funció dels requisits de tasca. Les següents 50 es dedicaran a implementar la destrucció de màquines virtuals únicament quan s'acaba tota l'execució. Solapar els temps de creació de màquines amb el de computació ocuparà al voltant de 50 hores. S'haurà de dedicar al voltant de 100 hores per tal d'aconseguir tot el tema del dinamisme en la gestió de les màquines virtuals, d'aquestes 40 seran dedicades a la creació i 60 a la destrucció. Per acabar fer polítiques més complexes comportarà unes 75 hores més de dedicació.

Per acabar el que s'ha descrit com a metodologia, només queda l'anàlisi del prototip que es realitzarà en 75 hores. Completar tota la metodologia ocuparà al voltant de 525 hores.

Cal tenir en compte a l'hora de planificar que a part d'implementar el prototip també cal escriure aquest document. La seva redacció es farà de manera paral·lela a la metodologia i se li atorguen unes 100 hores de dedicació. En total el projecte comportarà 625 hores de dedicació que es repartiran entre el mes de Febrer de 2010 i Desembre del mateix any tal i com mostra el diagrama de Gantt de la figura 1.1.

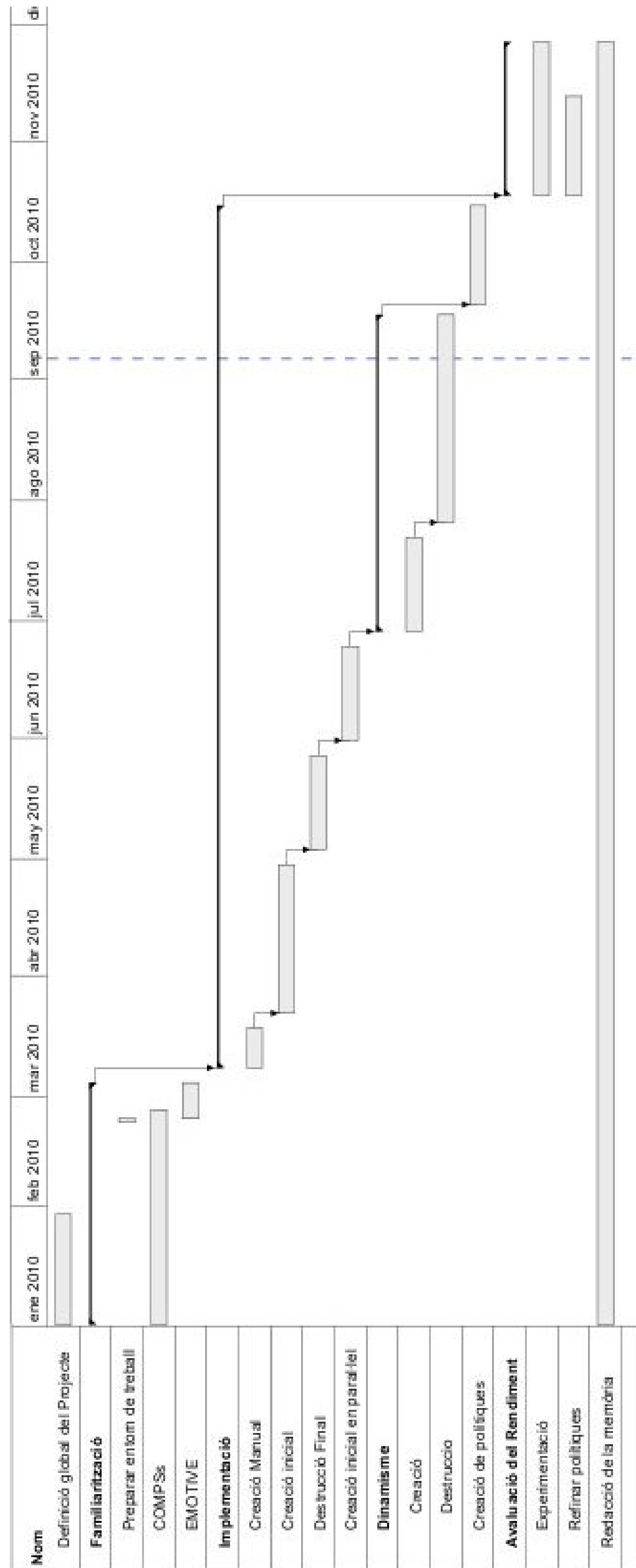

Figura 1.1: Diagrama de Gantt de la planificació inicial del projecte

1.4 Organització de la memòria

El que queda de document està dividit en les següents parts:

Entorn del projecte i estat de l'art: Es dóna una visió de cada una de les parts que componen aquest projecte per tal de veure d'una manera més global què significa. Per una banda s'introdueixen el concepte Cloud Computing i es descriu quines opcions dóna avui en dia. Per l'altra banda s'expliquen diferents models de programació donant una importància especial a COMPSs.

Desenvolupament del projecte: Aquest capítol parteix d'una descripció de l'escenari inicial que s'ha detallat en l'anterior. Discutieix les diferents decisions de disseny s'han pres a l'hora de realitzar el projecte i resumeix de quina forma s'han implementat per tal d'obtenir un resultat eficient.

Anàlisi del prototip: Un cop explicat com funciona el nou runtime, serà necessari avaluar-ne el rendiment mitjançant una sèrie de tests per detectar quins són els punts forts i febles que té l'extensió realitzada. Per fer-ho es definirà quin és l'entorn de les proves, quina metodologia s'ha seguit per fer-les, els resultats obtinguts i s'intentarà explicar quins són els motius dels problemes o punts forts que pot tenir el sistema.

Conclusions del projecte: L'objectiu d'aquest capítol és fer una síntesi de tot el projecte per extreure'n unes conclusions del prototip, de la tecnologia i altres aspectes importants a l'hora de desenvolupar el projecte: canvis respecte la planificació inicial que es va fer i un anàlisi del cost econòmic que suposa desenvolupar el projecte i implementar-lo. Al llarg d'aquesta part, també s'intentarà descriure a quins altres possibles camps d'investigació obre la porta aquest projecte i que no han estat tractats degut a que no formaven part dels seus objectius.

A *Apèndix A*: Explica l'organització de fitxers que segueix l'entorn de treball de COMPSs

B *Apèndix B*: Descriu el contingut del fitxer adjunt a la memòria

C *Apèndix C*: Descriu els passos necessaris per instal·lar el runtime de COMPSs i executar una aplicació

Entorn del projecte i estat de l’art

Capítol 2

Cloud Computing

Moltes organitzacions, grups de recerca i experts han intentat descriure el concepte Cloud Computing [5, 2, 1, 4]. Tot i els esforços, encara no hi ha cap definició que ens aclareixi que és. Al llarg d'aquest apartat repassarem la bibliografia del tema per tal de buscar els punts que tenen en comú i intentar donar una definició que englobi les característiques d'aquesta tecnologia.

2.1 Què és Cloud Computing

A inicis dels anys 60, John McCarthy, un dels pares de la IA, proposava com un objectiu a assolir en un futur que la computació fos una utilitat pública (Utility Computing), és a dir, hi hauria d'haver una organització que mantingués una infrastructura que fos capaç d'oferir computació a qualsevol persona. Al 1966, Douglas Parkhill publica el llibre "The Challenge of the Computer Utility", en que feia una comparació del que podia ser aquesta utilitat de computació amb el model de negoci de les xarxes elèctriques.

Al mateix temps que apareixia aquesta idea, l'agència americana Advanced Research Projects Agency, ARPA promocionava les investigacions relacionades amb les comunicacions entre ordinadors, les xarxes. Aquestes amb el temps van acabar transformant-se en el que avui es coneix com Internet. L'evolució de les xarxes i la de l'Utility Computing han anat donades de la mà. Al llarg dels anys 70 van començar a apareixer els primers clusters d'ordinadors a les universitats que poc a poc van anar creixent i depurant-se fins arribar als supercomputadors d'avui en dia.

L'any 90, Tim Berners Lee va desenvolupar el World Wide Web. Aquesta creació va suposar la popularització d'Internet. Tot el món tenia accés a Internet, creant així una xarxa de computadors a nivell mundial. L'any 1999, Ian Foster i Carl Kesselman van veure en Internet la possibilitat de que la gent pogués accedir a la computació tal i com Parkhill havia definit 25 anys abans. Van veure Internet com un element d'unió entre ordinadors als que es tenia accés i es podien utilitzar per fer càlculs de manera distribuïda, el Grid. L'exemple més extès és SETI@home (més tard evolucionat a BOINC, Berkeley Open Infrastructure for Network Computing) en que qualsevol ordinador connectat a Internet podia adscriure's al projecte SETI i permetre que l'ordinador realitzés càlculs pel projecte mentre no es feia servir el processador.

També es va veure com un recurs capaç d'oferir al públic uns serveis, webServices. A nivell de càlcul, un exemple pot ser Neos Server (Network-Enabled Optimization System),

actiu des de 1998. Neos Server és un sistema encara actiu avui en dia que permet a qualsevol usuari del món solucionar un problema d'optimització. L'usuari només ha de donar el model del problema que es vol resoldre i els paràmetres d'entrada. Neos enllaç el problema concret amb el seus resolutors d'optimització i l'encua en algun dels ordinadors que el té disponible. L'usuari en tot moment pot consultar quin és l'estat de la seva petició, igual que si fos la batch queue d'un cluster. Quan acaba el càlcul, Neos Server dóna els resultats per email i per l'eina que s'hagués utilitzat per enviar el problema (Web, Email o un programa especial).

L'any 2006, Amazon veu que té els seus servidors molt desaprofitats, només utilitzava un 10% de les màquines, i tenia el 90% reservades pels pics de demanda. Decideix actualitzar el seu datacenter creant una nova arquitectura basada en els serveis de VPN que s'oferien en l'àmbit de la telefonia, batejats amb el nom de Cloud. Veient el bon rendiment intern comença a comercialitzar els Amazon WebServices.

Des d'aquest moment, molta gent ha intentat definir el que és el Cloud Computing, però encara no hi ha un acord en la seva definició. Una de les definicions que ha pres més força és la que dóna la Universitat de Berkeley a [1], ja que queda bastant oberta a les possibilitats que poden sorgir en el futur.

> "Cloud Computing és el conjunt d'aplicacions ofertes com un servei a través d'Internet i el hardware i sistemes de software dels datacenters que proveeixen aquests serveis. Els serveis oferts s'han anomenat des de fa temps SaaS, Software as a Service. Anomenem Cloud al hardware i software del datacenter".

De totes maneres en aquest cas, el concepte de Cloud queda molt obert ja que qualsevol datacenter pot ser considerat Cloud. [4] restringeix una mica més el concepte argumentant que:

> "Un cloud és un tipus de sistema paral·lel i distribuït consistent en un conjunt de computadors interconnectats que es presenten com a un o més recursos de computació basant-se en un SLA[1] establerts entre un proveïdor del servei i els seus clients".

Una definició del Cloud més des de'l punt de vista de l'usuari pot ser la que es troba a [2].

> "El Cloud és una gran reserva de recursos (com pot ser hardware, serveis o plataformes de desenvolupament) virtualitzats de fàcil ús i accés a través de la xarxa. Aquest recursos poden reconfigurar-se dinàmicament per ajustar-se a la càrrega (escalar) i així permetre un ús òptim dels recursos".

Tal i com va fer Amazon amb el primer Cloud aquests poden ser utilitzats internament dins d'una empresa o bé poden oferir algun tipus de recurs al públic oferint un servei d'Utility Computing. Aquests serveis s'ofereixen generalment basant-se en tarifes en funció del temps d'ús i del tamany del recurs demanat.

En resum, es pot dir dir que el Cloud Computing dóna als clients:

- La sensació d'infinitat de recursos virtuals disponibles per ser demanats.

[1]Service Level Agreement

- L'habilitat de pagar en funció del consum de recursos que produeixen les pròpies necessitats i que aquests recursos poden ser alliberats quan ja no facin falta.
- La possibilitat d'eliminar la barrera del cost de montar tot un cluster per tal de poder començar a oferir un servei i la possibilitat d'escalar-lo segons les necessitats de cada moment.

Al parlar de Cloud Computing, sempre cal tenir present que tot i haver comportat el desenvolupament d'una tecnologia i d'un tipus de software concret, és un model de negoci i no una tendència tècnica.

2.2 Avantatges i Inconvenients del Cloud

Com ja es diu al llarg del primer capítol, el Cloud es presenta com una evolució del Grid pel que fa al camp de la computació. Al llarg d'aquest apartat s'analitzen aquells punts en que els dos paradigmes divergeixen o convergeixen.

Es pot 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.

Pel que fa a l'heterogeneïtat dels recursos tots dos permeten l'ús de recursos de característiques diferents a nivell hardware i software. En un Grid pot quedar amagada darrera d'una API, permetent virtualitzar la suma de diverses parts d'una WAN en una sola reserva de recursos . La virtualització cobreix dades (fitxers plans i bases de dades) i recursos computacionals. El Cloud afegeix un nivell més permetent-nos virtualitzar el hardware.

Gràcies a aquest nivell addicional de virtualització la forma en que es comparteixen els recursos és diferent. Mentre que el Grid suporta una compartició justa dels recursos per tots els usuaris, el Cloud proveeix recursos virtualitzats sota demanda. L'aïllament, que s'obté mitjançant la virtualització, provoca la impresió de que es disposa d'un únic recurs dedicat. Així doncs, no es comparteixen recursos.

Aquest nivell extra de virtualització, també comporta canvis en molts altres punts, per exemple, la programació. El Cloud permet treballar en un entorn totalment virtualitzat que aïlli l'execució de l'usuari evitant que apareixin problemes de compatibilitat o depedència amb certs softwares o hardwares, es poden crear els recursos virtuals que es necessitin independentment de la màquina física sobre la que estigui executant-se. Mentre que el Grid no disposa de mecanismes per evitar aquest tipus d'errors en l'execució d'un usuari final.

Un altre aspecte que apareix amb la virtualització és seguretat en l'accés als recursos de l'usuari. Avui en dia, és vista com un dels principals problemes que pot tenir el Cloud Computing i al que encara s'ha de dedicar moltes hores d'estudi. El Grid, ja disposa de serveis de seguretat o la possibilitat d'utilitzar credencials per accedir a qualsevol dels recursos o dades que puguin trobar-se en l'entorn virtual.

Un altre punt que és font de diferències és l'origen com a model de negoci del Cloud Computing. Això provoca diferències en els pagaments. El Grid normalment era finançat amb diners públics, el Cloud té arrels comercials i, per tant, els diners els posen les empreses que volen invertir-hi. Al moment de facturar, mentre que els Grids es paguen en funció d'una tarifa fixa pel servei o repartida entre les diferents organitzacions que utilitzen els

recursos, el Cloud proposa un cost per l'usuari en funció de la quantitat de recursos que s'utilitzen i el temps dels que l'usuari n'ha disposat.

Una altre fet que ha aportat l'origen comercial del Cloud és una estandardització. Les tecnologies i aplicacions necessàries per accedir-hi estan basades en altres tecnologies estàndard. Així el que s'aconsegueix és que hi hagi una interfície comuna darrera la que cada empresa pugui implementar el que s'adapti millor a les seves necessitats sense necessitat de donar detalls. De fet, una de les propostes interessants en l'àmbit de l'estandardització és la formació d'una federació d'abast mundial de Clouds.

2.3 Tipus de serveis que ofereix l'Utility Computing

Segons els serveis que el Proveïdor del Cloud decideixi oferir als seus clients, l'estructura del Cloud pot tenir més o menys capes. Es pot oferir des del control total sobre una màquina virtual fins la simple consulta d'una web o WebService.

El nivell més bàsic que es pot oferir és Infraestructura, Infrastructures-as-a-Service (IaaS). S'anomena capa BackEnd. El que s'ofereix és computació (CPU i memòria) i espai de disc. Són dos conceptes complementaris i que normalment es donen en un sol paquet en forma de màquina virtual sobre la que el client en té un control total accedint per exemple per ssh. L'exemple comercial més reconegut és l'Amazon WebServices (EC2 i ES3), però altres proveïdors són GoGrid i RackSpace.

En un segon nivell, capa Middle es troben aquells proveïdors que no ofereixen el control de la màquina sinó que aquesta queda abstreta en una Plataforma. Una plataforma és un conjunt hardware i un marc de software que permet executar software desenvolupat per altres. Normalment estan formades per un conjunt de recursos computacionals, un sistema operatiu, un conjunt de llenguatges de programació amb les respectives llibreries de runtime i d'interfície gràfica. La Plataforma és el servei que s'ofereix, Platform-as-a-Service (PaaS). Els dos grans líders comercials en aquest aspecte són Windows Azure i Google App Engine. Com un cas curiós, cal destacar la xarxa social Facebook, que ofereix una plataforma per desenvolupar aplicacions i consultant les dades que tenen dins a les seves bases de dades. Existeixen algunes empreses que ofereixen la seva plataforma utilitzant una infraestructura oferida com a IaaS, per exemple: Scalr, basada en Amazon EC2, o empreses amb el seu propi Cloud que ofereixen accés únicament a la plataforma com Google App Engine, Microsoft Azure, Facebook o OpenSocial.

El nivell més restrictiu que existeix és el que ofereix únicament software que pot ser executat, la capa FrontEnd. Algú pot desenvolupar una aplicació sobre qualsevol de dues capes anteriors i voler oferir-la. El Cloud també pot donar accés a aquestes aplicacions i serveis, Software-as-a-Service. Hi ha aplicacions realitzades amb les plataformes anteriors: aplicacions de Facebook, Google i OpenSocial o serveis oferits directament com podrien ser les noticies de NY Times.

La figura 2.1 reflecteix com queda organitzada aquesta estructura, des dels serveis que ofereixen només la Infraestructura, el nucli de tota la tecnologia; passant per la capa intermitja on trobem les PaaS, fins arribar a la capa més externa i menys conscient de la màquina, els serveis de software.

Finalment, per tal de poder accedir a qualsevol dels tres nivells, cal una API que permeti al client accedir als serveis que ha contractat. Per accedir al BackEnd s'ha estandaritzat l'ús de tecnologies web com són els Serveis SOAP o REST per demanar els

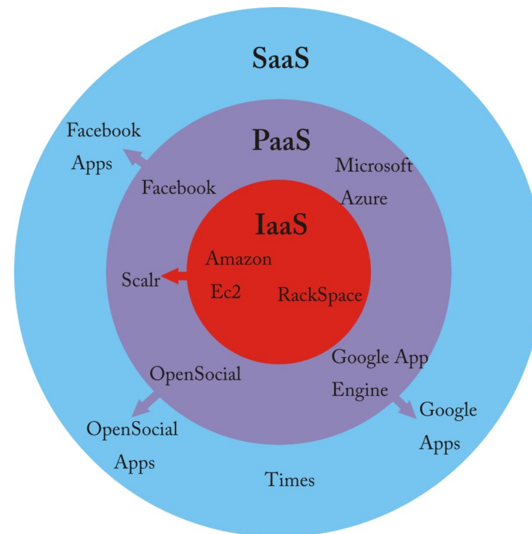

Figura 2.1: Estructura de serveis

recursos i l'accés com a root per poder controlar la màquina virtual. Les capes superiors utilitzen APIs específiques, cada plataforma té el seu propi SDK i per accedir als serveis existeix una gran varietat d'APIs específiques per cada software tot i que sovint s'utilitzen els serveis REST. La figura 2.2 ens il·lustra aquestes diferències.

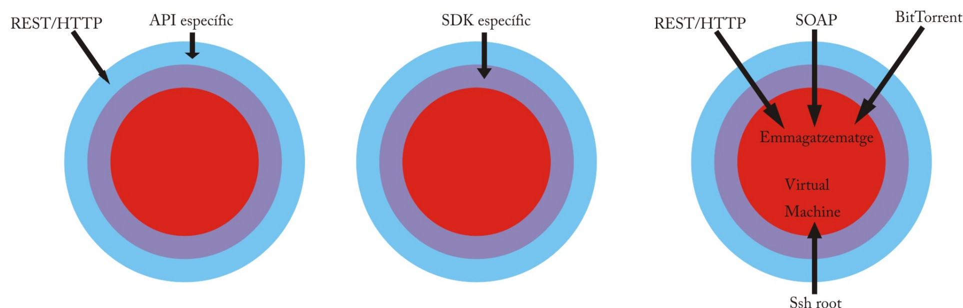

Figura 2.2: Diferents tipus d'api segons el servei que oferim

2.4 Anàlisi d'alguns proveïdors d'Utility Computing

Al llarg d'aquest apartat es fa un estudi de les diferents opcions que ofereix el mercat. La majoria de productes que es poden trobar són SaaS. N'existeix una gran varietat, i cada un té unes característiques úniques depenent de la funció que ha de realitzar: existeixen des de serveis de noticies en temps real, a consultes de dades personals en bases de dades. En l'apartat anterior ja es detalla com s'accedeix a aquests serveis, per tant, no cal veure'n exemples.

Pel que fa als PaaS existeix molta més diversitat en la forma d'accedir-hi, això fa més interessant l'anàlisi d'alguns exemples, per poder veure quines característiques destacables té cada un dels proveïdors. Els que més destaquen en aquest camp són Google AppEngine i Microsoft Azure.

Per la banda dels IaaS, l'accés es fa d'una manera estandarditzada, i el producte que s'ofereix és molt similar a tots els proveïdors. Com que el IaaS és la base del projecte que presenta aquesta memòria, es donarà un cop d'ull a diferents proveïdors, per tal de veure quines són les característiques comunes que es poden prendre com a base per fer el desenvolupament.

2.4.1 Amazon Elastic Cloud Computing "EC2"

Amazon va ser la primera companyia en oferir Cloud Computing. Ofereix un WebService que dóna un entorn computacional virtual que permet arrancar instàncies de múltiples sistemes operatius, carregar-hi un entorn d'aplicació personalitzat i gestionar els permisos d'accés per xarxa i així poder executar una imatge en tantes instàncies com sigui necessari.

El sistema ofereix control complet sobre les instàncies de les màquines que s'han demanat i s'interactua amb elles de la mateixa manera que es faria amb una màquina física remota qualsevol. El sistema s'executa sobre l'estructura de xarxa d'Amazon i es garanteix una disponibilitat del 99,95%. A més a més, el sistema incorpora mecanismes de seguretat per evitar atacs o pèrdues d'informació.

Es permet al client escollir quina configuració es vol per cada instància, des del hardware fins a paquets de software (servidors de Base de Dades, Servidors Web, Servidors d'aplicacions, eines de desenvolupament, ...) passant per múltiples sistemes operatius: Windows Server i algunes distribucions de Linux per servidors com Redhat o Oracle Linux i distribucions pel públic com Ubuntu i OpenSUSE. Sigui quina sigui l'elecció del client, els recursos hardware que ofereix el servei estan estandarditzats, en la taula 2.1 podem veure les característiques de cada una d'elles i el seu preu per hora d'ús.

	memòria	CPU	Disc	Linux	Windows
Small	1,7 GB	1 CPU de 2 cores	160 GB	0.095$	0.12$
Large	7,5 GB	2 CPUs de 2 cores	850 GB	0.38$	0.48$
Extra Large	15 GB	2 CPUs de 4 cores	1690 GB	0.76$	0.96$

Taula 2.1: Característiques i preus de les màquines ofertes per Amazon

Aquestes són les instàncies estàndard. A part d'aquestes es també s'ofereixen màquines per usos més específics: com màquines que requereixen molta memòria, màquines que facin un treball intens de computació o, fins i tot, màquines per fer computació d'alt rendiment.

A part de l'ús que es fa de cada màquina, també es cobra per les transferències de sortida tal i com mostra la taula 2.2

Quantitat	Cost per GB transferit
Primer GB al mes	0.00$
Fins a 10 TB al mes	0.15$
Entre 10 i 40 TB al mes	0.11$
Entre 40 i 100 TB al mes	0.09$
Per sobre de 150 TB al mes	0.08$

Taula 2.2: Preus d'Amazon per les tranferències de sortida del Cloud

A més a més, del servei de creació d'instàncies, s'ofereix una sèrie d'eines que permeten

simplificar la feina al programador i millorar el rendiment de les aplicacions. Amazon CloudWatch permet a l'usuari monitoritzar les instàncies que s'han demanat i extreure'n patrons sobre la demanda de memòria, ús de CPU o transferència de dades. Un altre complement, molt relacionat amb aquestes mètriques és Amazon AutoScale, que permet a l'usuari definir una sèrie de condicions que al complir-se augmentin o disminueixin la quantitat de recursos demanats per tal d'ajustar-lo a les necessitats de l'aplicació.

Un altre tema pel que destaca el producte d'Amazon és la capacitat de configuració de xarxa que ofereix. Permet que les instàncies que hi ha demanades al Cloud formin part d'una VPN externa al Cloud o formar-ne una únicament amb instàncies que haguem demanat. També ofereixen Elastic Load Balance, és a dir, balancejar la càrrega entre totes les instàncies que tingui el client en el moment i Elastic IP Address, assignar una adreça IP al compte de l'usuari i dirigir les peticions sobre aquesta adreça, després ja es redirigirà sobre alguna instància concreta.

A més a més, ofereix espai de disc a part del que es dóna a la màquina i la possibilitat d'escollir en quin dels seus datacenters (Nord de Virgínia, Nord de Califòrnia, Irlanda i Singapur) instanciar les màquines per tal de poder millorar la connexió amb els usuaris finals de l'aplicació.

2.4.2 RackSpace

RackSpace ofereix principalment tres serveis: Cloud Server, Cloud Sites i Cloud Files únicament a Estats Units.

Cloud Servers

El servei de Cloud servers és similar a l'EC2 d'Amazon. Ofereix accés root a màquines virtuals amb una imatge Linux o Windows, tot i que, s'està treballant per permetre que l'usuari pugui utilitzar les seves pròpies imatges. En el cas de RackSpace, el producte està pensat per que les màquines siguin utilitzades com a servidor web i no màquines de propòsit general, com és el cas d'Amazon.

Igual que en Amazon, el client pot escollir entre diferents tipus de màquines, únicament en funció de la quantitat de memòria que desitja. El preu de la màquina és en funció de la seva mida i de la imatge escollida, tal i com mostra la taula 2.3. A més a més, es paga per transferència tant d'entrada, 0.08$/ GB, com de sortida, 0.22$/ GB.

RAM	Disc	Preu Linux	Preu Windows
256 MB	10 GB	0.015$/hora	-
512 MB	20 GB	0.03$/hora	-
1024 MB	40 GB	0.06$/hora	0.08$/hora
2048 MB	80 GB	0.12$/hora	0.16$/hora
4096 MB	160 GB	0.24$/hora	0.32$/hora
8192 MB	320 GB	0.48$/hora	0.58$/hora
15872 MB	620 GB	0.96$/hora	1.08$/hora

Taula 2.3: Tipus de màquines i preus disponibles pels Cloud Servers de RackSpace

Les màquines físiques que suporten els cloud Servers són quad core. En el cas d'utilitzar una distribució Linux l'usuari disposarà dels 4 processadors a la vegada i hi estarà

executant un número de cicles en funció de la quantitat de memòria que tingui la màquina definida. En el cas de les màquines Windows, el nombre de processadors dels que disposa la màquina estarà vinculat a la quantitat de memòria: per 1024 MB es dóna 1 CPU, per 2048 i 4096 se'n donen 2 i per 8192 i 15872 s'utilitzen les 4 CPUs.

Cloud Sites

Ofereix un servei de Hosting en el Cloud. Quan es dóna un pic de demanda, automàticament és capaç d'escalar per donar resposta balancejant la càrrega entre diferents nodes. En aquest cas el preu es fa en funció dels cicles de CPU consumits més una tarifa base.

Cloud Files

Cloud Files permet als usuaris emmagatzemar fitxer de fins a 5 GB via un webService REST. Els fitxers es guarden en un sistema distribuït que replica per evitar pèrdues d'informació i ofereix la possibilitat d'actuar com a CDN[2]. El cost del GB de dades és de 0.15$.

2.4.3 GoGrid

GoGrid també és un proveïdor d'Infraestructures. El producte és similar al de RackSpace, l'objectiu és crear Servidors Webs capaços d'escalar. Per això, ofereixen tres serveis diferents: Cloud Servers, Cloud Storage i Load Balancing.

Cloud Servers

Ofereix accés root a màquines virtuals amb una imatge Linux o Windows que ja incorporen software per poder actuar com a servidor Web i SQL. En el cas de GoGRID permet que l'usuari tingui imatges preparades amb el software que desitja, ofereix dues maneres de crear aquestes imatges. La primera és mitjançant la seva web, on es pot crear una màquina virtual en el Cloud amb un dels sistemes operatius que ofereixen i instal·lant-hi el software necessari. Un cop es té la màquina ja creada, la mateixa interfície web permet guardar la màquina com una imatge. La segona opció és importar directament la màquina d'un altre proveïdor. Algunes empreses ofereixen la possibilitat de crear imatges que continguin tot l'entorn necessari per executar el seu software de manera segura i ràpida, per exemple: BitNami DocuWiki, BitNami Rubystack o OpenVPN Access Server.

En quant a les característiques de les màquines opten per un sistema similar al de RackSpace, la quantitat de memòria RAM de la màquina porta associada una quantitat de processadors i es factura en funció de la quantitat de GB de RAM que tingui la màquina i el temps que estigui encesa. En la taula 2.4 es poden veure els costos de les màquines i les característiques. També es facturen les transferències de sortida, per cada GB de sortida es paga a 0.29$.

[2]Content Distribuition Network

RAM	CPUs	Disc	Sistema Operatiu	Preu
0.5 GB	0.5	25 GB	Linux/Windows	0.095$/hora
1 GB	1	50 GB	Linux/Windows	0.19$/hora
2 GB	2	100 GB	Linux/Windows	0.38$/hora
4 GB	4	200 GB	Linux/Windows	0.76$/hora
8 GB	8	400 GB	Linux/Windows	1.52$/hora
16 GB	8	800 GB	Windows	3.04$/hora
16 GB	16	800 GB	Linux	3.04$/hora

Taula 2.4: Tipus de màquines i preus disponibles pels Cloud Servers de GoGRID

Cloud Storage

S'Ofereix també la possibilitat d'emmagatzemar dades en el Cloud de manera que l'espai que s'ocupa pugui ser accedit des de les màquines virtuals Linux i Windows utilitzant protocols com SCP, FTP, SAMBA/CIFS i RSYNC. L'accés a aquestes dades des dels Cloud Servers és gratuït. Els 10 primers GB d'espai ocupat són gratuïts, cada GB extra ocupat al llarg d'un mes es cobra a 0.15$.

Load Balancer

L'últim servei que ofereix és la possibilitat d'utilitzar un aparell hardware específic per fer el balanceig de càrrega. Pot escollir-se entre dues opcions a l'hora de fer-ho: per Round Robin, cada petició s'assigna a un node diferent seguint un ordre específic, o Least Connect, en que l'aparell que té menys càrrega és el que serveix la petició. L'ús d'aquests aparells s'ofereix de forma gratuïta.

2.4.4 AppNexus

L'empresa proveïdora d'AppNexus disposa de grans datacenters i t'ofereix la possibilitat de llogar porcions de les màquines. A través d'AppNexus s'ofereix una abstracció de la línia de comandes que permet encendre i apagar servidors escollint la localització d'aquest dintre del datacenter. Si es fa servir el mateix rack per tenir enceses varies instàncies, les comunicacions de les aplicacions tindran una latència menor. Si per contra s'utilitzen diferents racks l'usuari tindrà més redundància i, per tant, millor tolerància a fallades. La mateixa línia de comandes permet gestionar el sistema de fitxers.

Les principals funcions d'AppNexus són similars a les que ofereix Amazon amb EC2. A través de la línia de comandes, l'usuari s'identifica i encén imatges de distribucions Linux. Poden fer-se servir imatges d'altres proveïdors com Amazon. També amb pocs clicks l'usuari pot configurar el balanceig de càrrega entre les màquines virtuals que té enceses.

El sistema de fitxers està muntat sobre un sistema Isilon IQ X-Series, això permet a l'usuari compartir les dades entre els servidors del cluster, enlloc de treballar amb claus abstractes es treballa directament sobre el nom del fitxer, el cluster s'encarrega de tota la resta.

Una altra solució que ofereix és el que anomenen CDN, Content Delivery Network. Els sistema d'emmagatzematge disposa també del seu propi conjunt de servidors HTTP i

deixant els fitxers a la carpeta /cdn, AppNexus distribuirà els fitxers a diferents datacenters perquè es pugui accedir al fitxer des del més proper.

AppNexus ofereix màquines de 3 categories diferents. La taula 2.5 mostra les característiques de les màquines físiques que l'usuari pot escollir com a servidor per muntar el seu cluster.

GrupI		
	Web	Database
Processador Memòria Disc	1 Quad-core Xeon L5420 16 GB 2x146 GB en RAID-1	2 Quad-core Xeon L5420 32GB 8x146 GB en RAID-10
GrupII		
	Web	
Processador Memòria Disc	2 Quad-core Xeon L 5520 24 GB 2x500GB en RAID-1	
GrupIII		
	Database	
Processador Memòria Disc	2 Quad-core Xeon L 5520 48 GB 6x500GB en RAID-1E	

Taula 2.5: Característiques de les màquines ofertes per AppNexus

El preu estàndard per utilitzar un servidor del grup I durant un dia és 12.50$ per core, si s'utilitza durant un mes sencer el preu és de 247$ per core. L'espai ocupat també entra dintre dels paràmetres de configuració a 1$ el GB, sempre arrodonint als 100GB superiors.

2.4.5 IBM Smart Business

IBM també ha pujat al carro d'oferir Cloud computing. El fet de comercialitzar tant hardware com software els permet crear un producte complet. Ofereixen màquines virtuals amb paquets del seu software ja incorporats que permeten altres formes de fer servir les màquines virtuals. La gestió es fa mitjançant un portal web que facilita l'elecció dels paquets de software, característiques de les màquines que es volen fer servir i permet a l'administrador monitoritzar l'ús que s'està fent dels recursos i predir costos.

Les màquines virtuals poden treballar amb 32 o 64 bits amb sistema operatiu Linux: Redhad Enterprise Linux 5.4 o OpenSuse Enterprise Server 11.0. A part del sistema operatiu també s'ha de seleccionar el paquet de software que es vol que porti la instància amb aplicacions com Rational, Tivoli, Lotus, WebSphere, DB2 o Informix. Cada instància disposa d'un espai d'emmagatzematge reservat que desapareix en el moment que s'apagui. A més a més, IBM ofereix la possibilitat d'utilitzar un sistema d'emmagatzematge persistent organitzat en blocs de 256, 512 i 2048 GB.

Per accedir a les màquines per xarxa s'ofereixen les opcions d'assignar una adreça IP pública estàtica o a través d'una vpn. Les característiques de les instàncies estan predeterminades, l'usuari pot escollir entre els 9 tipus de màquines que es mostren a la taula 2.6.

32 bits					
	Cobre	Bronce	Plata	Oro	-
CPUs virtuals a 1.25GHz	1	1	2	4	-
Memòria virtual (GB)	2	2	4	4	-
Disc de la instància (GB)	60	175	350	350	-
64 bits					
	Cobre	Bronce	Plata	Oro	Platino
CPUs virtuals a 1.25GHz	2	2	4	8	16
Memòria virtual (GB)	4	4	8	16	16
Disc de la instància (GB)	60	850	1024	1024	2048

Taula 2.6: Característiques de les màquines ofertes per IBM

A l'hora de facturar, IBM permet dues opcions: la primera és pagar en funció del temps que s'ha estat utilitzant una instància, amb els preus que mostra la taula 2.7 i l'altre és mitjançant paquets de preu fix en que l'usuari pot disposar d'un nombre de CPUs virtuals, memòria i disc durant un període de 6 o 12 mesos. Per poder ampliar l'espai d'emmagatzematge és necessari utilitzar l'emmagatzematge persistent en blocs de 256, 512 i 2048 GB amb uns preus de 26€, 51€ i 201€ respectivament. També es cobra per opcions d'accés a les màquines a través de la xarxa, com són per exemple: tenir una IP fixa o configurar una VPN.

32 bits					
	Cobre	Bronce	Plata	Oro	-
OpenSuse 11.0	0.133	0.151	0.235	0.364	-
RedHat 5.4	0.169	0.186	0.275	0.408	-
64 bits					
	Cobre	Bronce	Plata	Oro	Platino
OpenSuse 11.0	0.311	0.399	0.284	0.444	0.710
RedHat 5.4	0.355	0.444	0.337	0.595	0.976

Taula 2.7: Preus(€) de les instàncies pel Cloud d'IBM sense reserva

2.4.6 Microsoft Azure

La solució que ofereix Microsoft al Cloud Computing és Windows Azure. Azure ofereix una plataforma on poder executar programes que utilitzin C#, el Framework .NET, C++ i l'API de Win32. També poden executar-se programes fets en altres llengutges de programació sempre que s'utilitzi un d'aquests per iniciar l'execució.

Microsoft defineix 3 parts destacables del projecte: Compute, Storage i Fabric; tal i com mostra la figura 2.3.

Tal i com suggereixen els seus noms, el servei Compute executa les aplicacions que l'usuari defineix mentre que els servei de Storage emmagatzema les dades. El tercer component , Fabric, ofereix una forma comuna de gestionar i monitoritzar les aplicacions que estan utilitzant la plataforma. Al llarg de la secció veurem els trets més característics de cada un d'ells.

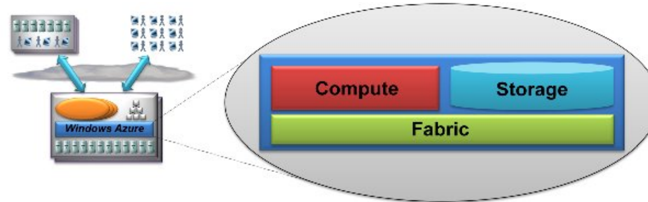

Figura 2.3: Windows Azure té tres parts principals: el servei de Compute, el de Storage i el Fabric

Storage

El servei de Storage permet a l'usuari emmagatzemar dades en el Cloud i accedir-hi des de les aplicacions que s'estiguin executant dintre de la plataforma com des de les aplicacions executant-se fora d'ella. Ofereix quatre maneres diferents de guardar les dades: dintre d'una base de dades relacional SQL, oferida a través de Microsoft SQL Azure, o dintre de blobs, tables i queues. Els blobs, tables i queues són tres estructures de dades que permeten emmagatzemar informació de diferents maneres.

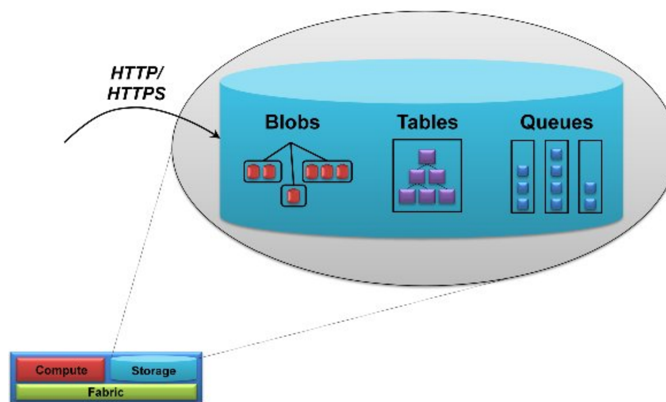

Figura 2.4: Azure permet emmagatzemmar dades en blobs, tables i Queues accessibles mitjançant HTTP

La forma més bàsica de guardar-nos les dades són els blobs, Binary Long OBjectS. Un blob conté dades en format binari, tot i que permeten afegir-hi metadades per descriure'n el contingut. A més a més, permeten veure el seu contingut de la mateixa manera que si es tractes trobés en un disc local.

Sovint els blobs no són suficientment estructurats per guardar-hi dades certes dades. La solució són els fitxers de tipus table. Aquestes no tenen res a veure amb les taules de les bases de dades, tot i que el concepte és similar. Cada table conté un conjunt d'entitats amb una sèrie de camps que la descriuen. Sobre aquest tipus de fitxer es poden llençar consultes que retornin els valors de les propietats referents a les entitats que conté.

Aquest dos tipus de fitxers serveixen per emmagatzemar les dades. L'últim tipus d'emmagatzematge, les Queues, tenen un objectiu diferent. Tal com indica el seu nom permeten encuar-hi diferents objectes i que puguin ser consultats en el mateix ordre en el que han estat introduïts. El propòsit principal d'aquestes és permetre l'escalabilitat. Així es poden tenir diferents instàncies d'aplicacions enceses i que una afegeixi a la cua un objecte que descrigui la feina que s'ha de fer, de manera que una altra instància sense feina pugui agafar-la i tractar-la.

Independentment de la manera en que es guarden les dades aquestes es repliquen tres vegades per tolerància a fallades i són accessibles tant des de dintre del Cloud com des de fora: ja sigui des d'un host o des d'un altre Cloud, mitjançant una API REST accessible per HTTP.

Compute

L'objectiu principal d'Azure és donar suport a un gran nombre d'usuaris de manera simultania per tal de que puguin executar les seves aplicacions i que aquestes escalin en funció de les seves necessitats. Per permetre-ho, cada aplicació ha de poder executar-se en varies màquines virtuals.

Quan l'usuari programa una nova aplicació pot escollir dos tipus d'instància de màquina en la que aquesta s'executarà: Web Role i Worker Role.

Les aplicacions Web Role s'executaran en màquines que per defecte accepten peticions HTTP i HTTPS i que poden executar aplicacions amb tecnologia .NET que funcioni amb IIS o codi natiu en qualsevol altre llenguatge, PHP o Java. La tecnologia Cloud s'encarrega de fer el balanceig de càrrega de totes les peticions que arribin entre les diferents instàncies que donin aquest tipus de serveis. Això fa que no es pugui assumir que dues peticions van al mateix node. Les aplicacions Web Role no poden tenir estat, qualsevol dada compartida ha de guardar-se en un fitxer (blob, table o queue) o en base de dades i ser consultada cada vegada que s'hi faci un accés.

Les instàncies de tipus Worker Role no tenen IIS configurat, per tant, no accepten peticions HTTP ni HTTPS. Si es vol que la instància n'accepti es pot executar un Web Server, fins i tot es podria instal·lar un servidor Apache. De totes maneres l'objectiu de les instàncies de tipus Worker és executar-se d'una manera similar a un procés en background.

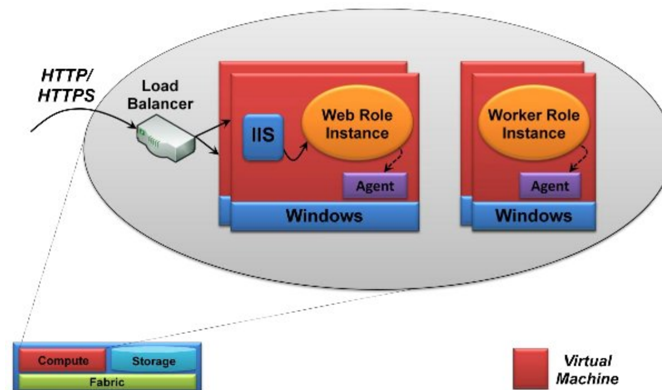

Figura 2.5: Una aplicació de Microsoft Azure està formada per instàncies Web Role i/o instàncies Worker Role, cada una de les cuals té la seva pròpia màquina virtual amb Windows

El que es recomana és tenir una o varies instàncies de tipus Web role que s'encarreguin de gestionar les comunicacions amb l'exterior i tenir varies instàncies de Worker role que s'encarreguin de processar les peticions. La forma de comunicació entre les instàncies Web Role i les Worker Role és tenir un fitxer Queue compartit. Quan un Web Role rep una petició l'acumula a la Queue i quan un Worker Role estigui disponible llegirà del fitxer la tasca i la processarà.

L'usuari pot gestionar el nombre d'instàncies que vol per tal de donar resposta mitjançant el mateix compte LiveID que Microsoft posa a disposició del públic per utilitzar algun

altre del serveis que ofereix per Internet. A través d'aquest pot crear-se un compte per fer hosting de les aplicacions, d'emmagatzematge o de les dues. En el seu compte pot crear fitxers i carregar aplicacions que s'executaran quan es demanin el nombre d'instàncies i tipus amb les que vol ser desplegada.

Fabric

Totes les aplicacions i dades que es pugen a Azure es troben en un datacenter de Microsoft. La forma d'organitzar aquest datacenter s'anomena Fabric. El datacenter està gestionat per un grups de 5 a 7 màquines que s'anomena Fabric Controller. Aquest sap quina aplicació s'executa en cada un dels nodes (també veu el Storage com una aplicació més). Quan l'usuari demana nova màquina virtual, el Fabric Controller decideix en quins node del datacenter es crearà la màquina virtual demanada. Les màquines que pot demanar l'usuari estan estandaritzades:

	memòria	CPU	Disc
Small	1,75 GB	1 CPU de 1 cores	225 GB
Medium	3.5 GB	1 CPU de 2 cores	490 GB
Large	7 GB	1 CPU de 4 cores	1000 GB
Extra Large	14 GB	1 CPU de 8 cores	2040 GB

Taula 2.8: Característiques de les màquines ofertes per Windows Azure

Preus

Al llarg de tota l'explicació d'Azure, s'ha fet un resum de l'arquitectura que tenen en el Cloud i destacar els serveis més simples que donen per tal de veure com funciona la plataforma. A part del que s'ha explicat s'ofereixen altres serveis com AppFabric. En aquesta última part es mostren els preus per els serveis bàsics sense entrar en detalls de llicències, tipus de servidors de SQL, o altres serveis més complexes.

- Compute
  - Small: 0,12 $/hora
  - Medium: 0,24 $/hora
  - Large: 0,48 $/hora
  - Extra Large: 0,96 $/hora
- Storage
  - 0,15 $/GB emmagatzemat al mes
  - 0,01 $/ 10.000 transaccions
- Transferència de dades
  - 0,10 $ per GB d'entrada al Cloud i 0,15 $ per GB de sortida a Europa i Nord Amèrica
  - 0,30 $ per GB d'entrada al Cloud i 0,45 $ per GB de sortida a Asia

2.4.7 Google AppEngine

Igual que Microsoft, Google també ofereix una plataforma on poder executar aplicacions, tot i que està enfocat exclusivament a aplicacions web tradicionals. Està preparat per aplicacions basades en una petició amb resposta i permet limitar els recursos utilitzats mentre es serveix la petició. Força una estructura amb una separació molt clara entre el tractament de la petició, computació sense estat, i l'emmagatzematge. AppEngine té mecanismes per tenir una alta disponibilitat, escalar automàticament.

AppEngine proporciona accés a un servei d'emmagatzematge de dades distribuït amb motor de cerca i transaccions, MegaStore. L'emmagatzematge de dades no és una base de dades relacional, sinó que els objectes de dades disposen d'un tipus i d'un conjunt de propietats definides i gestionades pel programador. Les consultes permeten recuperar entitats d'un tipus determinat passant filtres o ordenant-les segons el valor de les seves propietats. El sistema té un control de concurrència que fa que només s'actualitzin les entitats en el moment en que s'hi realitza un cert nombre de transaccions o quan existeixi més d'un procés accedint-hi simultàniament sobre el mateix objecte.

Mitjançant una API d'usuari, permet al programador integrar l'ús de comptes d'usuaris de Google a l'aplicació per poder autenticar els usuaris o fer ús de les comptes de correu GMail. D'aquesta manera es poden identificar automàticament diferents nivells d'usuari dins l'aplicació, sense haver d'implementar tot el sistema de comptes només per l'aplicació.

AppEngine ofereix dos possibles entorns d'execució: Java i Python. Això vol dir que es poden crear aplicacions amb Python i la seva biblioteca estàndard o amb tecnologies Java estàndard, servlets, etc. ; també accepta qualsevol aplicació escrita en un llenguatge que utilitzi un intèrpret o compilador basat en JVM com Javascript o Ruby. Sigui quina sigui l'elecció del client, aquests entorns disposen d'un servidor web dinàmic compatible amb les tecnologies web més comunes.

Aquests entorns d'execució proporcionen un accés limitat al sistema operatiu subjacent. Aquestes limitacions són les que permeten donar a l'aplicació l'entorn segur independent del hardware. Alguns exemples d'aquestes limitacions són:

- Només es pot accedir a altres equips d'Internet a través dels serveis de correu electrònic i extracció d'URL (servei que ens permet recuperar contingut web de qualsevol punt d'Internet). Els altres equips només poden connectar-s'hi enviant solicitud HTTP o HTTPS en els ports estàndards.

- Una aplicació no pot escriure en el sistema de fitxers. L'únic moment en que es permet crear nous fitxers és en el desplegament de l'aplicació.

- El codi de l'aplicació s'utilitza únicament en resposta a una sol·licitud o si és una tasca programada, l'execució d'aquesta ha de fer-se en un temps màxim de 30 segons. Qualsevol dada que vulgui guardar-se haurà de fer-ho mitjançant MegaStore, MemCache (servei de memòria cache de valors-claus que és accessible des de diferents instàncies de l'aplicació) o qualsevol altre servei d'emmagatzematge extern a la plataforma. De cap manera, es pot tenir un procés executant-se en background.

Preus

Google ofereix AppEngine de manera gratuïta dins d'uns límits de consum, quan l'aplicació té un consum de recursos important, es cobra tot allò que s'excedeixi dels límits. La taula

2.9 ens mostra els límits en el consum (gratuits i facturats) que posa Google i en la taula 2.10 es poden veure les tarifes de Google pels aspectes més bàsics (temps d'execució i Transferència de Dades). A part d'això, Google també cobra per les crides a les diferents APIs (inclosa la d'emmagatzematge).

	Cuota gratuïta		Cuota de facturació	
	Límit diari	Màxim	Límit diari	Màxim
Sol·licituds	1.300.000	7.400/min	43.000.000	30.000 /min
Bandwith sortida	10GB	56MB/min	1046 GB	740 MB/min
Bandwith entrada	10GB	56MB/min	1046 GB	740 MB/min
Temps de CPU	46 hores	15 mins/min	1729 hores	72 mins/min

Taula 2.9: Límits de consum de recursos acceptats per Google AppEngine

	Cost
Bandwith sortida	0,12 $/GB
Bandwith entrada	0,10 $/GB
Temps de CPU	0,10 $/hora
Dades emmagatzemades	0.15 $/GB al mes
Correu electrònic	0.0001$/destinatari

Taula 2.10: Cost dels recursos informàtics utilitzant AppEngine

2.5 Actors d'un sistema d'Utility Computing

La facilitat d'accés al Cloud per Internet, l'escalabilitat de recursos i el bon balanceig de càrrega fa que un de les usos més comuns que es fan d'aquest Utility Computing sigui oferir serveis Web, Software as a Service (SaaS). L'avantatge que aquests ofereixen davant del software tradicional és que l'usuari no té una còpia del software instalada en les seves màquines, sinó que l'usuari només disposa d'una API (SOAP, REST, Web, ...) generada pel proveïdor del servei que en ser invocada realitza una acció on consulta o guarda dades a través Internet. D'aquesta manera, l'usuari final no ha de preocupar-se de mantenir una infraestructura capaç de suportar el software. Només ha de realitzar una invocació, simplificant així la seva part de feina.

També és un model que afavoreix al proveïdor d'aquest SaaS, ja que permet un sistema on s'elimina la necessitat de donar llicències per cada còpia del software, només s'ha de controlar l'accés al servei i comprometre's a fer el manteniment perquè el servei estigui disponible. El fet de tenir el software executant-se sobre una màquina coneguda simplifica el procés de programació i permet realitzar optimitzacions conscients de la màquina sobre la que s'executarà.

Fins aquí la teoria és la mateixa que si tenim el nostre servidor Web propi. L'avantatge que ofereix l'Utility Computing no és de cara a l'usuari final, sinó al proveïdor del SaaS. Una de les obligacions del SaaS és la de tenir una infraestructura capaç d'oferir el servei i mantenir-la on-line tot el temps possible. Això implica dedicar molts recursos econòmics a comprar aquesta estructura, al seu manteniment i a la connexió necessària per donar resposta a totes les peticions. L'Utility Computing permet als proveïdors oblidar-se de tota aquesta capa, apareixent així una nova relació Client-Proveïdor basada en el Cloud.

El Client del Cloud és a la vegada Proveïdor del servei. La seva feina es veu reduida a programar i mantenir el software, mentre que el Proveïdor del Cloud s'encarregarà de tota la infraestructura, del seu manteniment i de tenir-la connectada a Internet. Quan el desenvolupador creï el servei, farà el deployment sobre el Cloud i l'oferirà al client final des d'aquest, tal i com podem observar a la figura 2.6.

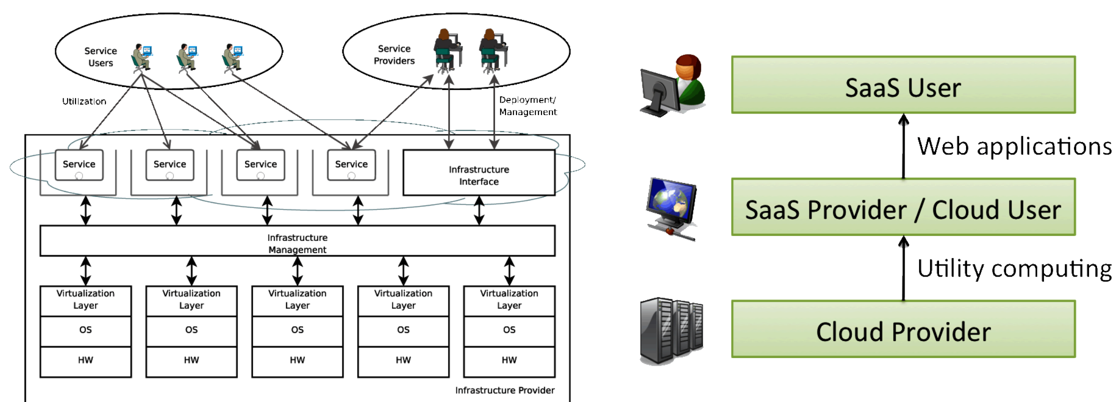

Figura 2.6: Actors que trobem al llarg de l'oferiment d'un SaaS que està en un Cloud

2.6 Cloud privats

La part anterior del capítol descriu i analitza els Clouds públics, on els proveïdors de serveis posen, a través d'Internet, recursos computacionals a disposició d'individuals i organitzacions: acadèmiques, empresarials o fins i tot governamentals. Els Clouds públics són una bona solució per aplicacions web i sistemes de recovery. S'utilitzen sobretot per les PIMES que no es poden permetre comprar i configurar un servidor amb garanties o per aquelles empreses que necessiten un conjunt de màquines durant un període curt de temps.

Aquests Clouds estan situats fora del firewall corporatiu i són gestionats per una empresa externa. El fet d'estar fora del firewall corporatiu fa difícil el compliment i millora dels SLAs, ja que depenen d'un proveïdor extern. Hi ha temes com la seguretat, l'accés a dades i la privacitat que complica complir els requisits a les auditories.

Són plataformes que estan dissenyades per donar resposta a demandes genèriques que limiten els tipus de màquines virtuals que se'ls hi pot demanar, tal com s'explica al llarg de l'apartat 2.4. Aquestes màquines poden ser insuficients per certes aplicacions i usuaris, de manera que els clouds públics no s'adapten als seus problemes.

La solució per aquests usuaris és un Cloud privat. Els Clouds privats no només tenen les mateixes prestacions que els anteriors, sinó que a més a més, incorporen prestacions de software per gestionar el Cloud. D'aquesta manera es pot donar resposta a les necessitats dels SLAs que anteriorment no es podien complir.

Els Clouds privats, també són utilitzats per les organitzacions que volen tenir una reserva de recursos IT disponible pels seus treballadors. La gestió i manteniment del Cloud la porta a terme un l'equip d'IT de l'organització d'aquesta manera es solucionen

els problemes de control i seguretat. S'estableix amb el altres departaments unes condicions de preu, ús, software i característiques de les màquines que es pot utilitzar.

Una altra característica important dels Clouds privats és que permeten recórrer als proveïdors de Clouds públics per tal de poder donar resposta als pics demanda que el Cluster subjacent no pot suportar. Així l'empresa s'estalvia haver de comprar recursos per tal de poder respondre a aquests moments de més demanda.

2.6.1 EUCALYPTUS

EUCALYPTUS és un software open-source que implementa cloud privats o híbrids que ofereixen IaaS, és a dir, màquines i espai d'emmagatzematge virtualitzats. L'estrutura del software està basada en 5 components: NC (Node Controller), que gestiona l'execució de màquines virtuals en un host; CC (Cluster Controller), que s'encarrega d'obtenir dades dels NCs per poder planificar les màquines virtuals en un Cluster; CLC (CLoud Controller), encarregat de comunicar-se amb els nodes gestors per obtenir informació dels recursos, prendre decisions d'scheduling d'alt nivell o gestionar la creació de màquines; SC (Storage Controller) implementa la gestió de l'emmagatzematge vinculat a cada VM; i , finalment, Walrus, que permetrà als usuaris guardar dades de manera persistent.

Tota la interacció amb l'usuari es fa mitjançant webServices. De fet l'interfície que dóna EUCALYPTUS implementa la interfície estàndard AWS[3], que permet la interoperabilitat amb altres clouds, i així gestionar no només les màquines virtuals del cluster, sinó també les d'altres cloud providers.

A part de la versió open-source existeix una versió Enterprise que ofereix suport per altres aspectes com la gestió tant imatge Linux com Windows o planificació basada en SLA.

2.6.2 OpenNebula.org

OpenNebula.org és un toolkit open source que orquestra tecnologies d'emmagatzematge, xarxes, virtualització, monitorització i seguretat per desplegar dinàmicament màquines virtuals en un conjunt de màquines físiques, utilitzant les plataformes KVM i Xen.

A més a més de gestionar un cluster privat, OpenNebula és capaç d'interactuar amb altres proveïdors de cloud. Això es deu a la seva arquitectura basada en 3 capes: Eines, Core i Drivers. La capa d'eines està formada per les aplicacions que permetran la interacció amb l'usuari: una interfície en forma de línia de comandes i un scheduler que permet definir un conjunt de màquines virtuals o polítiques de balanceig de càrrega.

La segona capa és el Core. Està formada per un conjunt de components que permetran a OpenNebula gestionar i monitoritzar màquines virtuals, xarxes virtuals, l'emmagatzematge i els hosts. Per actuar delegarà tota la seva feina sobre la tercera capa, els Drivers. Aquest no són res més que un conjunt de mòduls, que poden canviar-se i que actuen sobre un middleware específic per poder executar les peticions dels usuaris.

[3] Amazon WebService

2.6.3 EMOTIVE Cloud

EMOTIVE (Elastic Management Of Tasks In Virtualized Environments) és un gestor d'entorns virtualitzats desenvolupat pel BSC-CNS. El seu objectiu és proveir màquines virtuals que compleixin amb els requeriments d'un usuari en termes de software i recursos del sistema i que aquest pugui executar-hi tasques d'una forma eficient. Mitjançant una API REST, l'usuari pot treballar amb màquines virtuals basades en imatges generades per Xen, KVM o VirtualBox dintre del cluster o utilitzar recursos externs amb Amazon EC2.

La jerarquia de capes d'EMOTIVE Cloud queda reflectida a la figura 2.7. La capa més baixa, és l'anomenada Resource Fabric i Data Infrastructure. S'encarrega d'abstraure tota la part física de les màquines i espai d'emmagatzematge. La capa agrupa una sèrie de protocols que permeten distribuir les dades a través dels diferents nodes físics que formen el Cloud i permetre les migrar les màquines virtuals o fer-ne checkpoints. Qualsevol màquina pot accedir als fitxers que contenen la informació utilitzant NFS o Hadoop. També permet habilitar espais compartits, de manera que una màquina virtual pugui deixar-hi un fitxer i que la resta de màquines puguin accedir-hi o donar persistència a les dades.

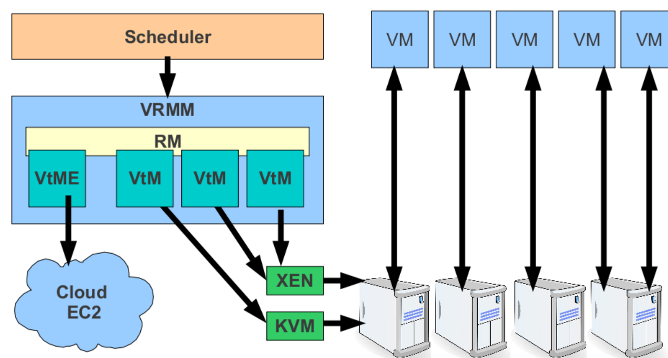

Figura 2.7: Jerarquia lògica d'EMOTIVE Cloud

La segona capa és l'anomenada Virtualized Resource Management and Monitoring (VRRM). La primera competència que té és crear i mantenir la màquina virtual, delega tota aquesta gestió en el Virtualitzation Manager (VtM). També dóna una interfície per monitoritzar l'ús de recursos de cada VM, s'anomena Resource Monitor (RM). Aquest component fa eficient la utilització dels proveïdors i suporta una varietat de nivells de QoS segons el SLA. S'encarrega de treure profit dels recursos del cluster permetent una gestió dinàmica d'aquests.

La segona prestació destacable que dóna aquesta capa és el suport a l'execució. Aquest mecanisme fa que les dades requerides per executar sigui accessibles dintre de la VM, és a dir, que pugui emmagatzemar dades d'usuari i que aquestes siguin recuperables en diferent sessions d'usuari. La qual cosa permetrà executar a l'inici del cicle de vida de la VM i recollir-ne l'output de les aplicacions.

La tercera i última capa, el Scheduler, és la que implementa l'API REST que fa servir l'usuari per interactuar amb EMOTIVE Cloud. La capa s'encarregarà de la gestió global dels recursos. Escull quin node executarà cada tasca o en quin moment és bo migrar una VM per tal d'optimitzar els recursos del sistema. Ofereix la possibilitat d'utilitzar polítiques diferents: SERA (Semantically Enhanced Resource Allocator); EERM (Economically Enhanced Resource Manager); GCS (la màquina aprèn i es decideix de forma distribuïda);

o que l'usuari crei les seves pròpies polítiques implementant l'API.

2.6.4 IBM CloudBurst

Anteriorment s'ha vist que IBM també tenia la seva solució com a Cloud Provider. A part d'oferir aquest servei IBM també entra en el mercat dels Clouds Privats, tot i que d'una manera una mica diferent dels altres casos presentats. IBM CloudBurst™ ofereix tot un sistema precarregat i preintegrat amb software, server i emmagatzematge. CloudBurst s'executa sobre la plataforma IBM System x® aportant al cloud els avantatges del IBM BladeCenter® permetent escalar ràpidament i una capacitat de xarxa alta. El tamany del cluster pot variar segons les necessitats del client.

Capítol 3

Models de Programació

La forma d'organitzar, gestionar i obtenir recursos per executar aplicacions és un dels àmbits en el que s'està innovant avui en dia. De totes maneres, no és l'únic camp en el que s'està investigant relacionat amb aquest projecte. La manera en que les aplicacions fan servir aquests recursos també és important i aquí entren en joc els models de programació.

L'objectiu del projecte és estendre el model de programació COMPSs que queda descrit en el proper capítol. Al llarg d'aquest capítol, per un costat es donarà un cop d'ull a aquells patrons i models en els que COMPSs està basat; per l'altre, s'expliquen altres projectes coneguts que ofereixen una programació distribuida per poder comparar-la amb el que ofereix COMPSs.

3.1 Patró Master-worker

El patró master-worker obliga a tenir dues entitats lògiques diferents: el master, del que només n'hi ha una única instància, i el worker, del que en pot haver-hi una o més.

El master és el que inicia i porta el flux de la computació. Genera un conjunt de tasques i n'espera el resultat per poder continuar el càlcul. El pes del càlcul d'aquestes tasques recau en el workers. El funcionament és molt senzill i queda molt clar en la figura 3.1. El master genera les tasques i es posa en un espai comú a tots els workers. Els workers consulten aquest espai esperant trobar-hi alguna tasca. Si no hi ha cap tasca, no fan res. En el moment, en que una tasca entra en aquest espai comú, un dels worker l'agafa i comença a calcular-ne el resultat, mentre que la resta esperen a tenir més feina. Quan el worker acaba la tasca que estava executant deixa el resultat en un espai comú per que el master o qualsevol altre worker pugui llegir-lo i saber que aquella tasca ja ha acabat.

Existeixen moltes variants d'aquest patró. Per exemple: no cal que el master deixi la tasca en un espai comú, directament pot indicar-li a un worker que comenci a resoldre una tasca, així no cal que tots els workers consultin l'espai comú. Una altra possibilitat és que el worker indiqui al master que ja ha acabat d'executar la tasca i es quedi el resultat de la tasca. El que s'ha de ressaltar és l'existència d'aquestes dues figures: un master que coordina i una sèrie de workers que resolen les tasques.

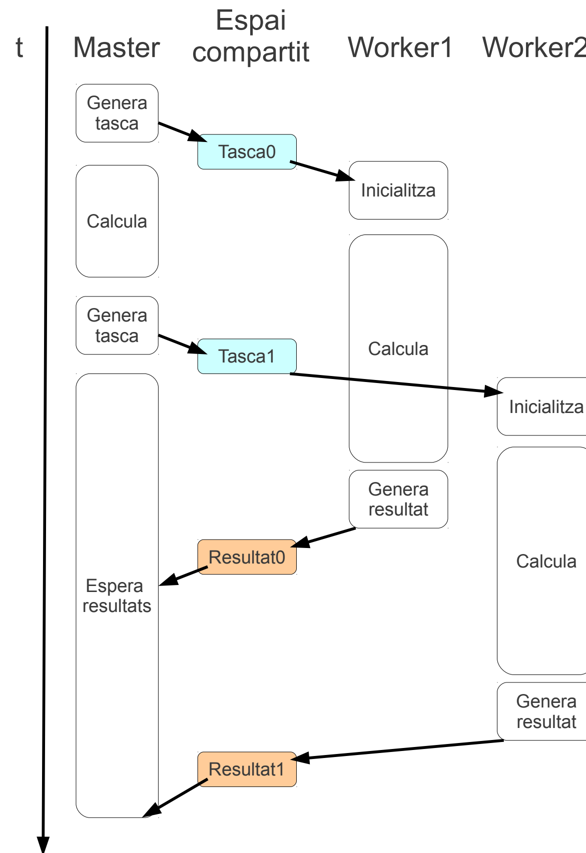

Figura 3.1: Flux d'execució d'una aplicació amb el patró master-worker

3.2 Grid Component Model(GCM) i ProActive

Abans de començar a descriure un model de components cal tenir clar què és un component. Un component és un mòdul de software, amb una descripció estàndard del que necessita i del que ofereix, que pot ser modificat per eines de composició i desplegament.

GCM és un model de components jeràrquic, és a dir, que un component GCM pot ser programat com una composició d'altres components GCM ja existents. No cal que l'usuari del component sàpiga que és una composició, excepte en el cas que vulgui conèixer el seu contingut. Aquesta propietat ja havia estat implementada en molts altres models, per exemple: el Fractal. Per aquest motiu GCM té el model de components Fractal com a referència.

Fractal és un model de components abstracte; té una especificació que pot ser instanciada en diferents llenguatges. Fractal defineix un model de components altament extensible que promou la separació de responsabilitats i la separació entre interfície i implementació. A Fractal tots els components estan formats per dues parts: el Content i el Controller o Membrane. El Content és una entitat abstracta que està formada recursivament per subcomponents i binding (enllaços).El Controller o membrane és una entitat abstracta que representa el comportament de control associat a un component. Un controller pot controlar arbitràriament el content del component del que és part (intercepta les invocacions d'operacions d'entrada i sortida per cada instància).

Cada component pot tenir una sèrie de interfícies:

- Server interface: interfície que rep invocacions (p.e: (a,I) a la Figura 3.2)
- Client interface: interfície que envia invocacions (p.e: (b,J) a la Figura 3.2)
- Control interface: interfície que té el component per gestionar aspectes no funcionals del component, per exemple: introspecció, configuració o reconfiguració
- Functional interface: interfície que correspon a una funcionalitat oferta o requerida. És l'oposat a una control interface.
- Component interface: interfície estàndard per tots els components de Fractal. Té dues funcions: getFCInterfaces i getFCInterface(interfaceName). Són dues operacions que permeten obtenir qualsevol altra interfície del component i així poder invocar operacions sobre el component.

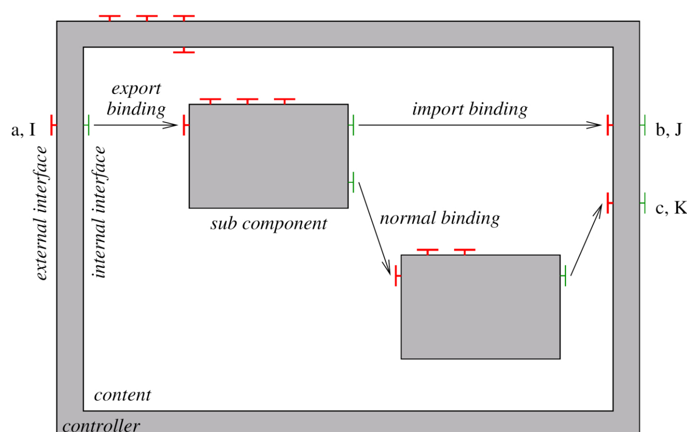

Figura 3.2: Component compost definit segons Fractal

La figura 3.2 mostra un component compost, el seu content són 2 subcomponents i la seva membrane conté els controllers. Podem veure també els enllaços que s'hi formen.

Fractal defineix 4 nivells de conformitat, principalment depenent del nivell de control que s'exerceix sobre els components. Cada nivell exigeix el mateix que tots els anteriors i afegeix algun requeriment addicional.

- Nivell 0: no hi ha res obligatori. Els components són simples objectes normals del llenguatge.
- Nivell 1: afegeix que tots els components han de tenir almenys implementada la Component interface
- Nivell 2: afegeix la restricció de que totes les interfícies del component han de poder fer cast amb Interface.
- Nivell 3: tots els components han d'utilitzar el sistema de tipus (extensible) definit a l'especificació de Fractal

L'altre pilar bàsic del model GCM és que els diferents components GCM permeten qualsevol tipus de semàntica de comunicació: punt a punt, streaming, basada en events, ... ja sigui síncrona o asíncrona.

L'última característica que cal de destacar del model GCM és l'habilitat dels components d'adaptar-se a situacions sense haver de dependre del que l'envolta. Com menys depenguin de l'exterior més autònoms són i, per tant, com més abstractes siguin les peticions i dades que poden rebre de l'exterior més autònoms són. L'autonomia es basa en que el component sigui capaç de configurar-se, buscar la manera en que rendirà millor, protegir-se i recuperar-se d'errors per si mateix; en resum, que sigui capaç d'autogestionar-se.

Com que aquests 4 aspectes fan referència a aspectes no funcionals del components s'implementen dintre dels controllers del components. Les interfícies que permeten consultar i modificar els aspectes no funcionals del comportament, sovint van lligats als autonomic controllers.

ProActive és l'aplicació de referència a l'hora d'implementar software que es basi en el model GCM. ProActive és un middleware que ofereix un framework comprensiu i un model de programació en paral·lel per tal de simplificar la programació i execució d'aplicacions que corren en processadors multicore, distribuïdes en xarxes locals (LANs), en clusters, en datacenters o en Grids, ja siguin en una intranet o a través d'Internet.

Els objectes actius són les unitats bàsiques d'activitat i distribució per tal de construir aplicacions concurrents utilitzant ProActive. Un objecte actiu és un objecte que té el seu propi thread de control i és capaç d'iniciar activitat. Els objectes passius son aquells que no són capaços d'iniciar activitat. L'objecte actiu gestiona totes crides i events que s'envien a ell mateix a qualsevol altre objecte passiu que tingui associat.

El model de programació que presenta ProActive es basa en dos premises:

- L'aplicació està estructurada en subsistemes. Cada subsistema té un objecte actiu i cada objecte actiu té el seu propi subsistema. Per tant, l'objecte actiu gestionarà totes les crides i events que es produeixin en el subsistema.
- No hi ha objectes passius compartits entre subsistemes

Aquestes dues característiques tenen moltes conseqüències en molts camps. Una de les més importants és la semàntica de les comunicacions. Quan un objecte qualsevol invoca un mètode d'un objecte actiu, pot ser que els paràmetres que ha d'enviar sigui objectes passius. Com que no es poden compartir objectes passius el que es fa es passar una còpia. Els objectes actius es passen per referència. De la mateixa manera s'actua amb els objectes de retorn.

El fet de no compartir dades permet que l'aplicació no necessiti cap canvi estructural per executar-se de forma seqüencial, multi-thread o distribuïda.

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 a que l'altre subsistema li retorni el control. De totes maneres, té un inconvenient: no es pot demanar objectes de retorn en aquestes comunicacions. Per solucionar aquest problema ProActive afegeix Future objects. Quan es fa la petició al subsistema es retorna un objecte Future independentment de l'objecte que s'esperés rebre. Mentre que l'objecte és calculat, el thread del subsistema que ha fet la crida pot continuar l'execució i quan necessiti operar amb aquest objecte, l'objecte Future bloquejarà el Thread fins que tingui l'objecte real.

La forma d'englobar GCM en ProActive és simple, cada subsistema serà un dels components GCM.

3.3 Microsoft Dryad

Dryad és un model de programació distribuïda de propòsit general per aplicacions amb una paral·lelisme de dades de gra gros. Una aplicació de Dryad és un graf acíclic dirigit que representa el flux de dades de l'aplicació, on els vèrtex són programes i les arestes canals de comunicació. Dryad executa l'aplicació resolent els vèrtex d'aquest graf en un conjunt de màquines disponibles, comunicant-los mitjançant fitxers, pipes de TCP o cues en memòria compartida segons el tipus de canal que hagi especificat l'usuari.

La figura 3.3 mostra un esquema de l'organització del sistema Dryad. La responsabilitat de coordinar l'execució recau sobre un procés anomenat Job Manager (JM) que pot ser executat en el cluster o en una màquina de l'usuari amb accés al cluster. El Job Manager conté un codi que genera el graf específic de l'aplicació i codi que planifica les tasques entre els diferents nodes disponibles. En cap cas s'encarrega de gestionar les comunicacions entre els diferents nodes. El cluster té un servidor de noms (NS) que pot fer-se servir per

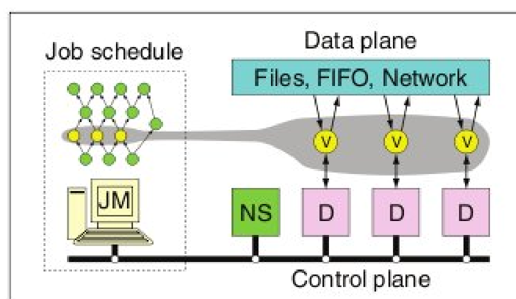

Figura 3.3: Organització del sistema de Microsoft Dryad

obtenir tots els recursos disponibles. A més a més, indica també la posició de la màquina dintre de la xarxa, així es pot utilitzar la topologia de la xarxa per prendre decisions durant l'scheduling i tenir en compte la localitat de les dades. Cada màquina del cluster té un daemon(D) que és responsable de crear els processos en nom del Job Manager. La primera vegada que un programa d'un vèrtex s'executa en un node, es copia tot el binari des del Job Manager, la resta de vegades que s'invoqui el programa s'executarà des d'una memòria cache de programes. El daemon actua com a proxy per tal de que el JobManager pugui comunicar-se amb els vèrtexs remots i monitoritzar l'estat de la computació.

El graf es construeix utilitzant la combinació de subgrafs més simples. Per definir-lo disposem dels operadors de la figura 3.4. En tots ells el graf resultat conserva la propietat d'acíclic.

La figura 3.5 ens mostra un exemple de com es defineix un graf

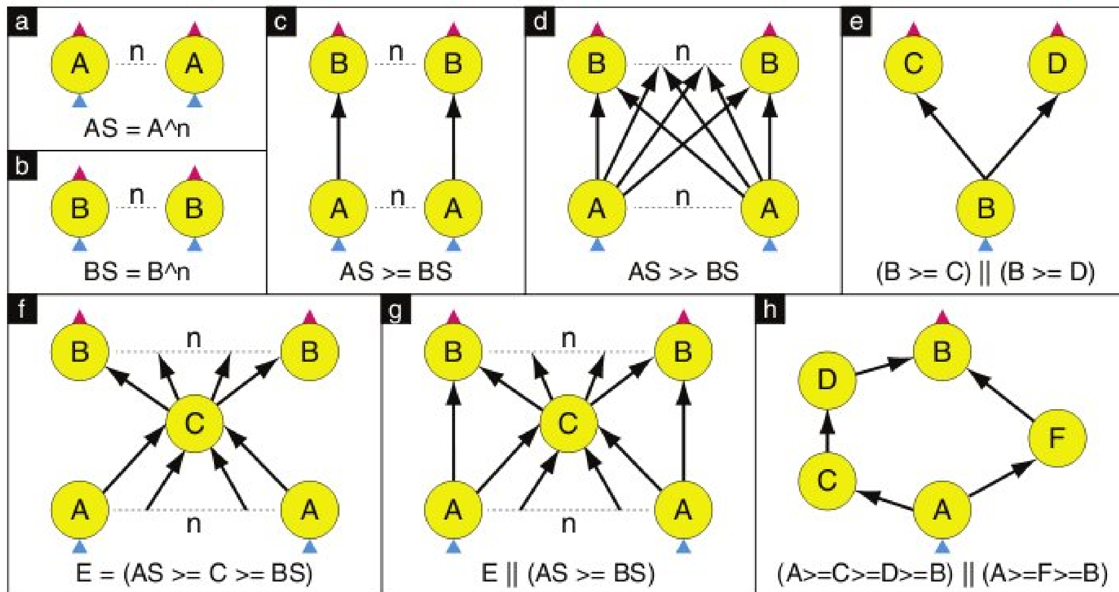

Figura 3.4: Operadors per crear el graf acíclic dirigit d'una aplicació a Microsoft Dryad

```
GraphBuilder XSet = moduleX^N;
GraphBuilder DSet = moduleD^N;
GraphBuilder MSet = moduleM^(N*4);
GraphBuilder SSet = moduleS^(N*4);
GraphBuilder YSet = moduleY^N;
GraphBuilder HSet = moduleH^1;

GraphBuilder XInputs = (ugriz1 >= XSet) || (neighbor >= XSet);
GraphBuilder YInputs = ugriz2 >= YSet;

GraphBuilder XToY = XSet >= DSet >> MSet >= SSet;
for (i = 0; i < N*4; ++i)
{
    XToY = XToY || (SSet.GetVertex(i) >= YSet.GetVertex(i/4));
}

GraphBuilder YToH = YSet >= HSet;
GraphBuilder HOutputs = HSet >= output;

GraphBuilder final = XInputs || YInputs || XToY || YToH || HOutputs;
```

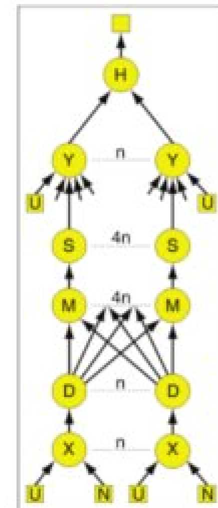

Figura 3.5: Exemple de definició d'una graf de flux d'aplicació en Microsoft Dryad

3.4 Google MapReduce

MapReduce és un model de programació per processar i generar grans quantitats de dades basat el les funcions map i reduce de LISP. El paper escrit per Jeffrey Dean i Sanjay Ghemawat [15] explica perfectament la filosofia d'aquest model de programació:

> *"La computació pren un conjunt de parells clau-valor d'entrada i produeix un conjunt de parells clau-valor de sortida. L'usuari de la llibreria MapReduce expressa la computació amb dues funcions: map i reduce.*
>
> *Map, definida per l'usuari, agafa un dels parells clau-valor d'entrada i produeix un conjunt de parells clau-valor intermitjos. La llibreria MapReduce*

agrupa tots els valors associats a una mateixa clau i els envia a una funció Reduce.

La funció Reduce, també definida per l'usuari, accepta una clau i el conjunt de valors que té associada. Uneix tots els valors per formar un conjunt de valors el més reduït possible. Normalment, una invocació a reduce produeix un valor de sortida o no en produeix cap. Els valors intermitjos es donen a la funció Reduce de l'usuari mitjançant un iterador. D'aquesta manera permet gestionar llistes de valors que són massa grans per entrar en memòria."

L'aplicació es basa en les funcions:

- $map(k1, v1) \rightarrow conjunt(k2, v2)$
- $reduce(k2, conjunt(v2)) \rightarrow conjunt(v3)$

Les dades d'entrada es divideixen automàticament en M parts a ser tractades per multiples màquines de forma paral·lela. Les divisions es fan d'una mida determinada per l'usuari, normalment s'utilitzen mides d'entre 16 i 64 MegaBytes. Les invocacions a reduce també es distribueixen en varies màquines. En aquest cas el que es fa és particionar l'espai de claus intermitges en R trams utilitzant una funció de partició. Igual que la mida de les divisions d'entrada, R pot ser definit per l'usuari.

Quan l'aplicació crida la funció MapReduce, la llibreria divideix les dades d'entrada en les M parts i s'inicien diverses còpies del programa en les màquines del cluster. Una d'elles és especial, el Master. Aquest escollirà una de les altres còpies que estigui en idle i li assignarà una de les M tasques de Map o una de les R de reduce.

Els workers que tinguin assignada una tasca Map llegeixen el contingut de la fracció d'entrada que li correspon. Parseja els parells clau-valor de l'entrada i passa cada parell per la funció Map definida per l'usuari. Els parells clau-valor intermitjos es guarden en un buffer en memòria.

Periòdicament, tots els parells guardats en el buffer s'escriuen en disc local repartits en funció de les R particions. Les ubicacions d'aquests parells en disc local s'envien al Master que és responsable de transmetre'ls al worker encarregat de tractar-los.

Quan el master notifica les ubicacions a un worker encarregat de fer un reduce rep les ubicacions, utilitza mètodes remots per poder llegir el contingut del buffer del disc local del worker que l'ha generat. Quan ha llegit totes les dades intermitges, les ordena en funció de la clau per tal de que tots els parells de la mateixa clau quedin agrupats. L'ordenació es fa perque moltes claus poden quedar mapejades en una mateixa tasca. Si tenim masses dades intermitges, no hi cabrien totes en memòria.

El worker itera sobre les dades intermitges ja ordenades i per cada clau, fa una sola crida a la funció Reduce amb tots els valors que li corresponen. El resultat d'aplicar Reduce s'adjunta al final del fitxer final de sortida per la partició.

Quan totes les tasques Map i Reduce han acabat, el master desperta el programa de l'usuari. En aquest moment la funció MapReduce retorna el control al codi d'usuari.

Al acabar tot el procés, l'usuari disposarà dels R fitxers, que ha demanat a l'inici, amb les dades de sortida; un per cada tasca de Reduce. Normalment l'usuari no necessita ajuntar tots aquests fitxers en un de sol, sinó que els utilitza per llançar una altra execució

de MapReduce o per qualsevol altra aplicació distribuïda capaç de treballar amb particions d'un mateix fitxer.

Tot el procés queda il·lustrat a la figura 3.6

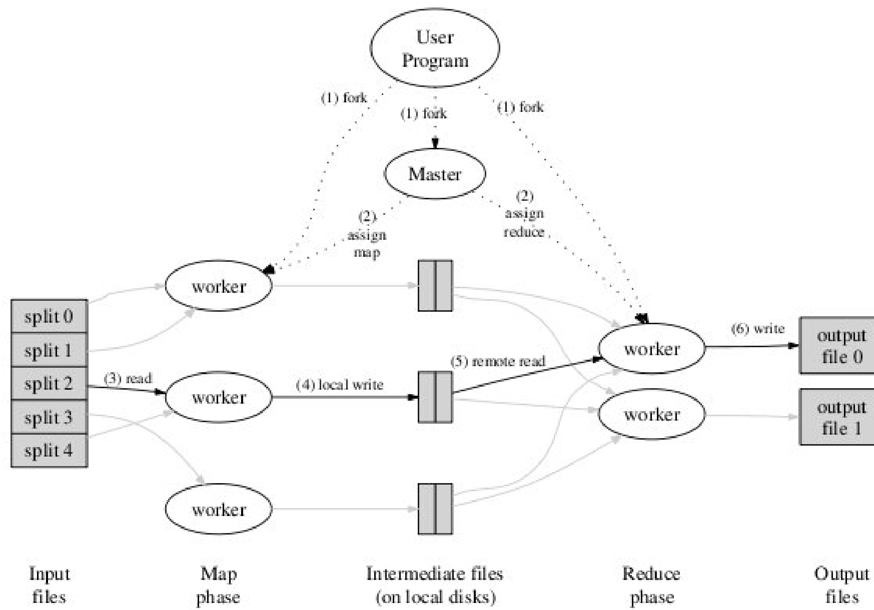

Figura 3.6: Flux d'execució d'una crida a MapReduce

Capítol 4

COMP Superscalar

COMP Superscalar, també anomenat COMPSs, és un model de programació que busca explotar el paral·lelisme inherent a nivell de tasca d'un codi seqüencial. Executa les aplicacions en els recursos que formen un Grid seguint un sistema similar al Superscalar utilitzat en el hardware, juntament amb l'execució fora d'ordre.

A partir d'un codi Java seqüencial, COMPSs genera en temps d'execució un graf de dependències entre les tasques que definieix el programador. Per tal de poder reduïr el nombre de dependències que s'han de controlar es fa un renaming de les dades de sortida, eliminant així totes les dependències WaR [1]i WaW[2] que poguessin tenir les tasques. D'aquesta manera, el graf només mostrarà les dependències de tipus RaW [3]. Una tasca queda lliure de dependències quan totes les tasques que generin els seus fitxers d'entrada s'hagin completat. En aquest moment, la tasca passa a formar part del conjunt de tasques que el planificador de COMPSs pot executar en un dels recursos del Grid. Un cop se li ha assignat un node a la tasca, s'envien totes les dades necessàries per calcular el resultat al node escollit i es llença l'execució. Mentrestant, altres tasques poden ser planificades i executades en els altres recursos sense necessitat d'esperar a que aquesta acabi. Un cop acaba la tasca, s'alliberen totes les seves connexions del graf permetent que les tasques subjacents puguin ser executades.

La interfície de programació que ofereix COMPSs és molt simple i permet mantenir el Grid transparent a l'usuari. Aquest pot programar les seves aplicacions sense tenir-lo en compte; només ha de seleccionar quines tasques han d'executar-se en ell sense haver de realitzar cap tipus de crida que hi estigui relacionada.

El procés de generació i execució d'una aplicació en COMPSs està format per dos parts: una estàtica i l'altre dinàmica. En la primera, l'estàtica, el programador utilitza l'API de COMPSs per programar l'aplicació i selecciona quines operacions vol executar en el Grid.

Per obtenir un bon rendiment i fer un bon ús dels recursos, les decisions sobre en quin recurs s'executarà cada tasca s'hauràn de pendre en temps d'execució. Això comporta la necessitat d'un runtime que sigui capaç de gestionar els recursos disponibles i decidir en cada moment quina és la millor decisió que es pot prendre a l'hora d'assignar les tasques. A l'inici d'aquesta segona part, l'usuari de l'aplicació indica els recursos que es poden fer servir durant l'execució i en iniciar-la es fa la transformació de les crides del codi seqüencial per invocacions a l'API d'aquest runtime COMPSs. La figura 4.1 mostra

[1] Write after Read
[2] Write after Write
[3] Read after Write

les diferents capes per les que passa l'aplicació abans d'executar-se. El runtime acabarà transformant aquesta invocació a la seva API en una execució en un dels nodes remots. L'estructura interna d'aquest runtime queda detallada a l'apartat 4.2.1 i la forma en que evolucionen les diverses crides s'expliquen a la secció 4.2.2.

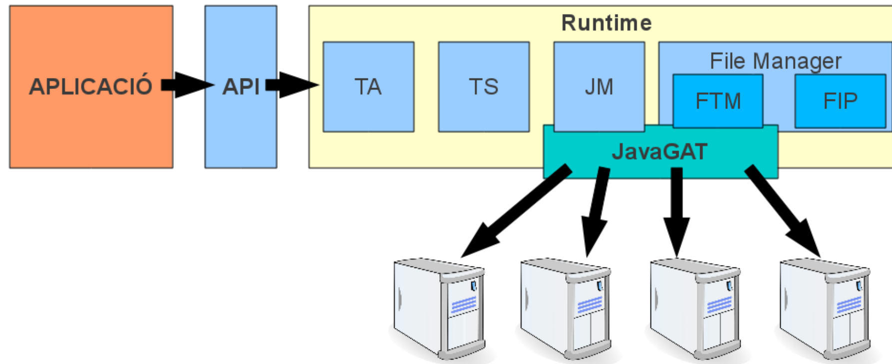

Figura 4.1: Arquitectura lògica de COMPSs

Al llarg d'aquest capítol es descriu en detall COMP Superscalar. De totes maneres, abans d'entrar en detall en el runtime cal veure quina és la forma de programar per COMPSs i quines operacions ofereix l'API.

4.1 Model de Programació

En la introducció de COMP Superscalar s'exposen dues de les característiques més importants que té. La primera és que al canviar les crides local per crides remotes, es passa d'una programació lineal a un paradigma master-worker. L'ús d'aquest fa que el codi seqüencial hagi d'estar estructurat en dos grans blocs:

- L'aplicació principal: la part del programa que s'executarà en el master i que dirigeix el flux d'execució.
- Les tasques: el codi que executaran els diferents workers per tal de calcular el resultat final de l'aplicació.

La segona característica és la facilitat de programar per tal de poder desplegar les aplicacions en el Grid. El fet de definir els recursos i seleccionar-los en la fase dinàmica implica que el programador no pugui ser conscient del Grid en el que s'executarà. El fet de que el runtime de COMPSs transformi les crides seqüencials en crides remotes fa que el programador no hagi de ser conscient de que executa en un Grid.

Executar tasques com un programa Java remot obliga a que siguin aquestes les que iniciin l'execució. Els paràmetres d'entrada poden ser string o tipus bàsics: boolean, char, byte, short, int, long, float i double. I no podràn tenir cap paràmetre de retorn. És a dir, que totes les funcions que puguin invocar-se segueixen un esquema similar a algun dels següents

```
public static void foo (String, String){......}
public static void foo (String, int, float){......}
public static void foo (boolean){......}
```

Sembla bastant contradictori que s'enviin tasques a executar a un node remot i no es pugui obtenir un resultat d'aquesta tasca. Si l'objectiu és enviar a fer càlculs complexos per aprofitar el paral·lelisme de l'aplicació, el més normal és enviar una gran quantitat de dades: matrius d'enters o objectes complexos amb diferents tipus de dades, per exemple, la descripció d'una galàxia.

La manera que COMPSs ofereix per tal d'entrar grans volums de dades o objectes és guardar les dades en fitxers i enviar com a paràmetre d'entrada el nom del fitxer. Així per enviar dades d'entrada només cal llegir el fitxer al començar l'execució de la tasca i es pot operar com si el worker tingués una còpia de l'objecte. De la mateixa manera, es poden treure resultats si el master indica al worker quin és el fitxer on ha d'escriure l'objecte resultant. Al final de l'execució de la tasca el master o qualsevol altre worker pot llegir l'objecte resultat llegint el fitxer.

COMPSs és un model de programació que vol amagar tot el que està relacionat amb el Grid, per tant, haurà d'amagar no només la gestió de versions d'aquests fitxers, sinó també les transferències entre workers.

4.1.1 Selecció de tasques i definició de la interfície

Degut a la separació del codi, caldrà definir una interfície (Java Annotated Interface) que indiqui quins mètodes tenen els workers implementats. L'estructura d'aquesta interfície serà la típica de Java, una interfície pública que conté les signatures de totes les funcions que el runtime haurà de canviar per crides remotes.

Per poder executar el programa Java remot, COMPSs necessita saber per cada funció en quina classe trobarà la seva implementació. Per tal de que el runtime pugui saber aquestes dades, cal anotar-ho; passant d'una interfície estàndard de Java a una interfície anotada (Java Annotated Interface). La classe que conté el mètode s'indicarà mitjançant l'anotació ClassName abans de la signatura de la funció.

Com que el runtime també ha de gestionar les versions dels diferents fitxers, ha de ser conscient de com modifica el mètode els objectes que li arriben com a paràmetre. Necessita que se li indiqui tipus (bàsic, string o fitxer) i direcció (entrada, sortida o entrada-sortida) per cada paràmetre del mètode. A l'hora de parlar dels tipus que pot tenir un paràmetre d'un mètode i les seves direccions cal tenir en compte el que diu la definició de Java sobre la manera en que es passen els paràmetres:

- Tipus bàsics i Strings: es passen per valor.
- Tipus complexes (objectes): passen per referència.

Els paràmetres d'entrada, in, podran ser tipus bàsics, Strings o tipus complexos emmagatzemats en un fitxer. Degut a que s'executa un programa Java remot, qualsevol paràmetre de sortida haurà d'estar en un fitxer ja sigui un tipus bàsic o un objecte. Per acabar de completar totes les opcions: el cas d'enviar un objecte complex que sigui modificat a l'interior del mètode. Com ja es diu abans, els tipus complexos s'hauràn de passar en un fitxer i la direcció haurà de ser inout.

Tota aquesta informació s'afegeix amb una anotació @ParamMetadata per cada paràmetre del mètode. Un exemple d'una interfície que fa servir COMPSs és el següent:

```
public interface SimpleItf {
```

```
    @ClassName(package.classA)
    void foo(
        @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);

    @ClassName(package.classB)
    void bar (
        @ParamMetadata(type = Type.INTEGER, direction = Direction.IN)
        int i,
        @ParamMetadata(type = Type.FILE, direction = Direction.OUT)
        String d);
}
```

Amb aquestes dades el runtime ja té tota la informació que necessita per gestionar el flux de l'aplicació. Per un costat té l'aplicació que li dicta quin flux ha de seguir, quines funcions han de ser executades en un worker i, a partir d'aquí, pot deduir-ne quines tasques pot executar de forma paral·lela.

Una de les característiques bàsiques del Grid és l'heterogeneïtat. Per tal de seleccionar les característiques del recurs en que s'executarà un mètode, el programador ha d'indicar-li al runtime quins requisits ha de complir. Això també es fa a la interfície, per cada mètode es pot afegir un conjunt d'anotacions, MethodConstraints. L'usuari pot definir les següents característiques:

- Host:
  - hostQueue: nom de la cua on la tasca serà enviada.
- Processador:
  - processorCPUCount: mínim nombre de processadors de la màquina.
  - processorSpeed: freqüència de rellotge mínima.
  - processorArchitecture: l'arquitectura de la màquina (definit per l'usuari).
- Memòria:
  - memoryPhysicalSize: quantitat de GB de memòria física.
  - memoryVirtualSize: quantitat de GB de memòria virtual.
  - memoryAccesTime: màxim de nanosegons que tardem a accedir a dades.
  - memorySTR: mínima velocitat de transmissió en GB/s.
- Emmagatzematge:
  - storageElemSize: quantitat de GB d'emmagatzematge.
  - storageElemAccessTime: màxim de nanosegons per accedir a dades emmagatzemades.
  - storageElemSTR: mínima velocitat de transmissió en GB/s.

- Sistema Operatiu:
  - operatingSystemType: s'ha d'escollir Linux o Windows.
- Software:
  - appSoftware: Cadena lliure amb totes les aplicacions que necessitarà separades per coma.

El següent codi mostra com quedaria un mètode completament anotat amb algunes restriccions:

```
@MethodConstraints(operatingSystemType = Linux,
                   processorCPUCount = 8,
                   appSoftware = "Xalan,Xerces")
@ClassName(package.classA)
void foo(
    @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
);
```

4.1.2 API de COMPSs

COMPSs permet executar qualsevol aplicació en un Grid sense necessitat de canviar res respecte la versió sequencial, sempre i quan compleixi amb el que s'ha explicat fins aquest punt del capítol. A l'iniciar l'aplicació es dóna l'ordre d'encendre el runtime i aquest es manté encès durant tota l'execució donant suport qualsevol crida a l'interfície que es trobi en l'aplicació durant l'execució.

De totes maneres, per a programadors avançats es dóna l'opció decidir quan està el runtime en funcionament. Per això, COMPSs ofereix una petita API que permet al programador indicar al runtime quan ha d'encendre i aturar la transformació de crides. Com que el toolkit pot apagar-se i tornar-se a encendre més endavant en la mateixa execució, el programador haurà d'indicar en quin moment està accedint a un fitxer que és resultat d'una tasca per tal de que es vigili que aquell fitxer ja hagi estat generat i, per tant, s'hi pot accedir.

L'API de COMPSs ofereix només tres funcions:

- startIT(): encèn el toolkit i envia tasques als workers
- stopIT(terminate): atura el toolkit i executarà el programa únicament en el master. El booleà terminate serveix per indicar-li al runtime si el toolkit tornarà a encendre's o no. En cas de no tornar a encendre's permet alliberar la memòria de tots els seus components ja que no necessitarà tornar a consultar-los.
- openFile(fileName, openMode): obre el fitxer amb nom fileName de forma local en el master en mode openMode (OpenMode.READ, OpenMode.WRITE, OpenMode.APPEND). Abans de la lectura comprovarà que es satisfan totes les dependències

i, si el fitxer encara no es troba en el master, el portarà d'algun dels workers que el tingui.

4.2 Runtime

Fins ara s'ha estat veient la primera part del procés de generació i execució d'una aplicació, la part estàtica, com el programador pot crear una aplicació: quines restriccions imposa el model, l'ús de l'API de COMPSs, com seleccionar i definir les tasques que executaran els workers i com seleccionar les caracterísitques del node.

El que falta veure és tota la part d'execució de l'aplicació. A l'inici d'aquesta, l'usuari enlloc d'executar directament l'aplicació, invoca un Loader que modifica el bytecode de l'aplicació feta pel programador per tal de que que incorpori les invocacions a l'API del runtime per fer les crides remotes i inicii l'execució d'aquesta. En funció del Loader escollit per l'usuari aquest bytecode es modificarà d'una manera o d'una altra. L'elecció del Loader serà en funció de si el programador ha utilitzat l'API o no.

Com ja s'argumenta abans les decisions de planificació s'han de prendre en temps d'execució, per tant, es farà ús d'una eina que porti el pes de tota l'execució, que s'encarregui de buscar el paral·lelisme d'aquesta aplicació, gestionar totes les dependències entre tasques, donar els recursos necessaris per executar les tasques i avisar als workers quina feina han de fer assegurant-se de que tindran tots els valors necessaris per que l'aplicació acabi amb èxit. Aquesta eina és el runtime de COMPSs.

El runtime vol donar llibertat en dos aspectes:

- La distribució del seu cost computacional entre diferents hosts. ProActive és una aplicació que permet programar el runtime seguint el model GCM (Grid Component Model). També permet deplegar els components en host diferents i així repartir les responsabilitats del runtime.
- La independència dels protocols i middlewares de comunicació que utilitzi el Grid. JavaGAT aporta una interfície uniforme per enviar tasques i transferir fitxers utilitzant middlewares i protocols com Globus, UNICORE o ssh. D'aquesta manera obté independència per operar sobre qualsevol Grid sempre i quan es tinguin adaptadors de JavaGAT que implementin aquests protocols.

4.2.1 Arquitectura de Components

El runtime de COMPSs, està implementat sobre ProActive. Això vol dir que l'aplicació s'obté de la unió de diferents subsistemes. Tenim quatre subsistemes: TaskAnalyser, TaskScheduler, JobManager i FileManager. Cada un d'aquest dóna lloc a un component GCM. L'últim component de tots està format per dos subcomponents : FileInfoProvider i FileTransferManager. El resultat és una jerarquia GCM com la que podem veure a la figura 4.2.

Cada component té unes responsabilitats que al unir-se amb la resta dóna lloc a les funcionalitats de l'API que suporta el runtime. Quan es detecta una nova tasca el component que rep la petició és el TaskAnalyser (TA). Aquest afegeix la tasca al graf de dependències que li permet detectar quan tots els valors necessaris per poder executar una tasca han estat generats. El TaskScheduler (TS) és capaç de buscar en quin recurs

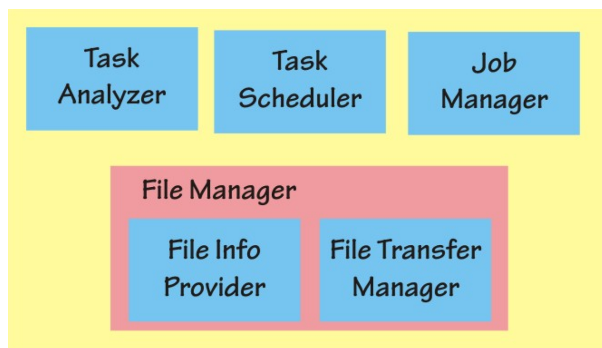

Figura 4.2: Components del runtime de COMPSs

la tasca pot executar-se. El JobManager (JM) és l'encarregat de que la tasca s'executi en un recurs remot i de recollir-ne el resultat, per fer-ho ha d'assegurar-se que tots els valors necessaris es troben a la màquina. L'últim component que encara falta per veure el FileManager. Tal com indica el seu nom s'encarrega de gestionar tots els fitxers, això significa saber quins fitxers té cada node i encarregar-se de les transferències d'aquests. Per simplificar el disseny, disposa de dos subcomponents que s'encarregaran de controlar-ho. FileInfoProvider (FIP) es dedica a gestionar els accessos a fitxers, ha de saber quins fitxers té cada una de les màquines, quina versió i quins tipus d'accessos i s'estan fent i es faran. El segon subcomponent és el FileTrasferManager (FTM). La seva funció és gestionar totes les transferències de fitxers entre els nodes.

4.2.2 Integració dels Components

Al llarg d'aquest apartat es veuen com els diferents components descrits a l'apartat anterior es relacionen entre ells per tal que el runtime doni les funcionalitats ha de ser capaç d'oferir.

Entre les competències del runtime es troba la capacitat de controlar en quin moment es pot executar una tasca, escollir en quin worker ho farà, i vigilar que el node tingui tots els valors necessaris. A més a més, també ha d'implementar l'API que s'ofereix al programador; és a dir, els mètodes:

- startIT(): permet l'entrada de noves tasques i si és la primera vegada que es dóna aquesta crida en l'execució s'han d'inicialitzar els components
- stopIT(): bloqueja l'entrada de noves tasques al runtime i, en cas que no es tornés a encendre el runtime, aturar els components.
- openFile(): registra un accés a un fitxer i l'obre de forma local en el master.

Només tractarà l'openFile ja que els altres dos són trivials.

Camí d'una tasca

Després de fer la transformació inicial del bytecode, al llarg de l'execució del codi principal apareixen crides a l'API del runtime perquè executi tasques remotament. El tractament d'aquestes crides comença a fer-se al TaskAnalyser. Aquest ha de detectar en quin moment pot enviar a executar la tasca, és a dir, ha de poder saber quan totes les tasques anteriors

de les que depèn han acabat. Per fer-ho construeix un graf dirigit de dependències RAW. Cada node representa una tasca i una aresta entre dos nodes significa que una tasca depèn de l'altre. Una tasca pot ser enviada a executar quan no té cap aresta de la que sigui destí.

Tal com el TaskAnalyser rep una nova tasca, afegeix un nou node inconnex al graf i es posa a comprovar quines dependències té. A partir dels paràmetres de tipus fitxer i registra a quins fitxers accedeix la tasca amb l'ajuda del FIP. El FIP té tota la informació de totes les versions del fitxer que s'han anat creant o es crearan al llarg de l'execució. En el moment en que una tasca escriu un fitxer, crea una nova versió del fitxer a la que se li assigna un nou nom. El FIP li retornarà al TaskAnalyser quins són els fitxers reals dels que ha de llegir.

A part de fer aquest registre, el TA manté una estructura pròpia que li indica quina serà la tasca que generarà la nova versió. Per tots els fitxers de lectura es consulta si el fitxer ja ha estat creat, és a dir, si la tasca que l'havia de generar ja ha acabat. Si encara no ha acabat, afegeix una aresta al graf cap a la tasca escriptora.

Per acabar, s'ha d'actualitza l'estructura on es guarden els últims escriptors. Per cada paràmetre que genera una nova versió es guarda que la nova tasca és l'últim escriptor. Totes les tasques que vulguin llegir la nova versió hauran d'esperar a que aquesta acabi.

Les arestes del graf representen totes les dependències que la tasca té. Si per una tasca no hi ha cap aresta que l'apunti, no cal esperar, el TA indica al TaskScheduler que la tasca ja està llesta per ser executada.

La funcionalitat del TS és escollir quin dels recursos disponibles és el millor per executar-hi la tasca en aquell moment. Aquesta decisió es pren quan la tasca ja pot ser llançada, sempre tenint en compte tres aspectes:

- Restriccions de la tasca
- Número de tasques simultànies que pot assumir la màquina i les que ja té assignades
- Quantitat de fitxers necessaris que ja estan en el node.

Quan arriba la tasca, el TaskScheduler mira quin dels mètodes que hi ha a la Interfície Anotada és el que es vol executar per la tasca i selecciona els recursos que poden executar-la en funció de les restriccions del mètode. El segon pas, és filtrar tots els recursos que no han arribat al límit de tasques en funció de les seves característiques.

Si encara queda més d'un recurs candidat, es mira quin de tots ells és el que té més fitxers dels que necessita la tasca. Per fer-ho, es consulta al FileInfoProvider quins nodes tenen cada un dels fitxers necessaris per executar la tasca. El que més en tingui és l'escollit, s'actualitza el nombre de tasques que està executant el recurs i es delega l'execució de la tasca al JobManager.

En el cas de no disposar de cap recurs on la tasca pugui executar-se en el moment, la tasca queda bloquejada al TaskScheduler fins que el nivell de càrrega d'alguna màquina que compleixi les restriccions hagi baixat.

Quan la tasca arriba al JobManager, el primer que aquest ha de fer és assegurar-se que tots els fitxers que necessita la tasca estiguin en el node. Delega aquesta responsabilitat al FileTransferManager i esperarà a que totes les transferències necessàries estiguin fetes. En el moment en que el FTM notifiqui al JobManager que ja han acabat les transferències que faltaven (pot ser en el mateix moment de la crida si no en faltava cap), envia la tasca

a executar-se al node mitjançant JavaGAT i espera a que hi hagi un canvi d'estat en la tasca. El diagrama de flux de la figura 4.3 il·lustra el camí que segueixen les tasques per arribar a executar-se.

Figura 4.3: Diagrama de flux bàsic d'una tasca dintre del runtime

Quan la tasca acaba, el primer component en saber-ho és el JobManager. Aquest indica al FileManager que el fitxer ja existeix i avisa al TaskScheduler que ja ha acabat l'execució remota. El TaskScheduler allibera el recurs de la tasca que ha acabat i mira si hi ha alguna tasca que no hagi pogut assignar que pugui ser llançada en aquell recurs continuant el procés de la mateixa manera que si el recurs se li hagués otorgat en la primera consulta. Per acabar avisa al TaskAnalyser de que la tasca ha acabat l'execució.

Al rebre l'avís de que la tasca ha acabat correctament, TaskAnalyser actualitza el graf de dependències, esborra totes les arestes que tenia la tasca, comprova si algun node del graf queda lliure de dependències i, si és el cas, envia les corresponents tasques al TaskScheduler per tal de que aquestes s'executin.

És possible que durant el procés d'una tasca sorgeixi algun error, per exemple, un error de transferència o un error durant l'execució. COMPSs disposa d'un mecanisme de tolerància a fallades. Quan detecta que hi ha hagut un error en l'execució d'una tasca, JobManager intenta repetir l'execució un cop. Si també falla, la tasca retorna al TaskScheduler on es busca un nou recurs on poder executar-la. Si torna a produir-se un error en l'execució no es torna a enviar la tasca, s'atura l'execució i es retorna l'error a l'usuari. En aquest casos, o altres errors que puguin produir-se eventualment, COMPSs cancel·la tota l'execució per tal d'avisar a l'usuari el més aviat possible.

Camí d'un openFile

L'openFile que ofereix l'API de COMPSs permet operar amb un fitxer de la màquina local de l'usuari. Una crida a openFile el que en realitat ha de fer és registrar l'accés al fitxer

de la mateixa manera que es fa amb els paràmetres, invocant al FIP. Si l'accés al fitxer és per ser llegit o modificat és necessari que aquest fitxer hagi estat generat. Quan l'API troba una crida a openFile es fa una petició al TaskAnalyser, perquè aquest l'avisi quan el fitxer hagi estat generat. La tasca que crea el fitxer s'haurà executat en un altre recurs, per tant, el fitxer pot haver estat generat en una altra màquina. Per poder treballar de forma local, el FileTransferManager ha de transferir el fitxer cap al master. En acabar la transferència, la última versió del fitxer es troba en el master i, per tant, aquest ja pot accedir-hi amb els valors correctes i donar un resultat correcte.

4.3 Execució de tasques remotes

Per executar una tasca remota, el runtime de COMPSs fa servir el middleware JavaGAT. Aquest ofereix una API que permet enviar comandes o fitxers a través del Grid. JavaGAT està format per un conjunt d'adaptadors, els encarregats de fer desaparèixer els detalls de les comunicacions entre els diferents nodes, així, a nivell d'aplicació, és igual si les ordres les han de donar-se per un sistema que utilitza Globus, gLite, Unicore 6, PBS, SGE o si s'ha de fer per ssh.

Perquè un node del Grid pugui executar una tasca que li ha estat assignada és necessari que prèviament si hagin copiat:

- l'aplicació worker
- les classes que contenen el codi de la tasca, és a dir, les que pertanyen al package worker.

L'aplicació worker és una única classe que dóna l'opció d'executar qualsevol mètode de qualsevol classe que es trobi en el CLASSPATH que tingui configurat el node remot. Quan el JobManager vol que un node executi una tasca, ha d'enviar-li una comanda a través de JavaGAT que invoqui aquesta aplicació worker i proporcionant-li com a paràmetre la classe i el mètode que vol executar.

Un altre detall important és que per executar la classe és necessari encendre una màquina virtual de Java. Cada worker pot tenir varies màquines virtuals de Java instal-lades. És bo donar l'oportunitat a l'usuari d'escollir quina màquina es vol utilitzar. La solució per la que opta COMPSs és que JavaGAT executi un script que té el worker que fa deixar tots els fitxers temporals de l'execució en el directori desitjat i que encèn la màquina virtual de Java que vulgui l'usuari.

4.4 Fitxers de configuració

Al llarg dels dos punts anteriors es veu com la combinació de la part estàtica i la part dinàmica aconsegueix un model de programació que permet executar en el Grid i fer la programació sense necessitat de conèixer els seus detalls. De totes maneres encara queden dos aspectes importants d'aquesta execució:

- com indicar els recursos dels que el runtime pot fer-ne ús
- com definir l'entorn d'execució de la tasca en el worker

Un cop s'ha definit un recurs, les seves característiques no acostumen a canviar. Normalment es guarden diferents plantilles amb conjunts de nodes del Grid i serveixen per a tots els usuaris. L'usuari només n'ha de triar una, en funció dels nodes on vol executar. Si el que es vol és poder observar les diferències entre dues execucions, que dos usuaris diferents utilitzin la mateixa aplicació o executar dues instàncies de COMPSs que utilitzen els mateixos nodes; el que s'ha de fer és modificar l'entorn de la màquina. Així tots els fitxers temporals es deixaran en carpetes diferents. Per aquest motiu, el segon resulta molt més variable que el primer. En aquest breu apartat s'explica com s'indiquen a COMPSs aquestes dades.

Definció dels recursos

La descripció de cada un dels recursos disponibles la dóna l'usuari a l'iniciar l'execució en un fitxer xml que els conté tots. El format del fitxer s'ha obtingut a partir de l'Information Modeling, que ha estandaritzat l'Open Grid Services Architecture WG de l'Open Grid Forum.

En aquest document es poden detallar aspectes com:

- Processador: Número de CPUs, la seva freqüència i arquitectura.
- Sistema Operatiu: tipus (Windows o Linux) i el màxim nombre de processos per usuari.
- Elements d'Emmagatzematge: Espai global i temps d'accés.
- Memòria: Espai físic, virtual i temps d'accés.
- Software instal·lat: quines aplicacions poden executar els seus processos.

Definció de l'entorn en el worker

El segon aspecte que queda per definir és com han d'executar-se les tasques en el worker. Per cada worker cal indicar a COMPSs quatre conceptes que són necessaris per poder executar les tasques:

- Directori d'instal·lació: on es troba l'aplicació worker i totes les classes de les tasques.
- Directori de treball: on es deixen tots els fitxers temporals de l'execució.
- Usuari: que permetrà executar les tasques en la màquina remota.
- Limit de tasques: número màxim de tasques que poden executar-se simultàniament en el node.

Igual que amb els recursos, aquesta informació es dóna en forma de fitxer XML, s'anomena fitxer de projecte.

Desenvolupament del Projecte

Capítol 5

Desenvolupament del Projecte

Feta ja la introducció necessària, aquest capítol se centra en el projecte realitzat. S'aborden els aspectes més rellevants de la realització de les diferents parts que el componen, tot indicant el desenvolupament realitzat i les decisions de disseny preses en cada punt. De tota manera, abans d'entrar en detall a cada paquet de treball, cal primer establir els principis que regiran les decisions de disseny.

5.1 Directrius del disseny

Com ja queda descrit a l'apartat 1.2 l'objectiu principal del projecte és estendre el model de programació COMP Superscalar per tal que pugui utilitzar infraestructures Cloud, la qual cosa limita el seu ús a proveïdors d'infraestructura, IaaS. El prototip utilitzarà com a base EMOTIVE Cloud, però haurà d'estar preparat per poder adaptar-se a qualsevol altra de les infraestructures vistes, ja sigui una demanant les màquines a un proveïdor comercial o un Cloud Privat.

Evidentment, les necessitats del mercat fan que les infraestructures ofertes públicament siguin més restrictives que no pas les opcions que ofereix un Cloud gestionat per la pròpia empresa. Repassant el que es comenta a l'apartat 2.4, es pot observar que tot i que cada proveïdor imposa les seves restriccions, en general, hi ha una sèrie de característiques que són comunes que ens permeten definir algunes de les característiques del software que s'ha de desenvolupar.

Una de les principals restriccions que apareix de manera recurrent i que no es troba a EMOTIVE Cloud és que les característiques de les màquines corresponen a uns patrons predeterminats pel proveïdor de Cloud que es vulgui fer servir. El mateix passa amb el software que conté la màquina virtual encesa. Alguns proveïdors permeten crear màquines virtuals utilitzant una imatge base feta per l'usuari, altres permeten utilitzar imatges amb paquets de software concrets i altres ofereixen simplement un sistema operatiu en el que l'usuari pugui fer el que vulgui.

Degut a la diferent oferta de serveis al voltant del Cloud apareix un altre tema en el que hi ha molta disparitat, l'elecció dels paràmetres que configuren el preu. En general el preu depèn de les característiques de les màquines que es demanen; per tant, també el càlcul del cost ha de ser canviable ja que dependrà de cada proveïdor.

Un tercer aspecte que hi ha és quin espai es s'utilitza a l'hora de treballar amb la màquina. La majoria de proveïdors ofereixen dos tipus d'espai: un disc virtual que té la

màquina i un espai de fitxers comuns en el Cloud. A l'haver de treballar sobre un Cloud, el runtime pot estar executant-se tant en un sistema en que el disc virtual de la màquina es troba en la mateixa màquina física o pot trobar-se en un sistema de fitxers distribuït. Aquest nivell d'abstracció que dóna el Cloud provoca que no es pugui determinar com ha de ser la gestió de dades dels fitxers. Per defecte, el runtime treballarà en el supòsit de que l'espai de disc virtual de la màquina es troba sobre el mateix node físic del Cloud i, per tant, caldrà mantenir tota la gestió de dades i transferències.

Tal com es diu a l'apartat 2.3, sí que hi ha acord pel que fa a la manera d'accedir al Cloud. Els proveïdors acostumen a oferir una API per gestionar les instàncies i permeten l'accés com a root a aquestes per tal de poder utilitzar-les com es vulgui. Partint d'aquesta base, l'arquitectura ha de permetre adaptar el runtime per la part de gestió de recursos, ja que cada proveïdor ofereix la seva API. De totes maneres, les comunicacions amb les instàncies es poden fer sempre de la mateixa manera.

Un cop vistes quines són les restriccions que el mercat imposa sobre el prototip cal veure també quines són les decisions que s'han pres com a desenvolupador.

Llegint l'apartat 4.2 es veu com COMPSs té una arquitectura definida en 5 components amb uns objectius clars. Un dels criteris a l'hora d'avaluar les diferents opcions de disseny serà optar per modificar el mínim possible tota aquesta arquitectura i afegir la nova informació o nous procediments al component que sembli més lògic posar-ho en funció del significat que té, tot i perdre rendiment.

Un altre aspecte en el que cal pensar és en l'usuari de COMPSs, el programador. Si es compara la interfície i la manera de programar en la versió per Grid de COMPSs amb la d'altres models de programació distribuïda com Dryad o MapReduce, es veu com té un programabilitat molt senzilla i ofereix una manera de resoldre utilitzable en molts processos de càlcul. No cal definir explícitament el graf de dependències de les tasques com a Microsoft Dryad ni ens fixa un graf de dependències concret com a MapReduce. El mateix model de programació crea el graf en funció del codi seqüencial d'una aplicació. Possiblement, aquest sigui l'aspecte en que COMPSs destaca comparat amb altres models de programació. Per tant, les decisions de disseny tendiran sempre a intentar no modificar la programació.

L'últim tema que queda per tractar no té res a veure ni amb les infraestructures Cloud ni amb el model de programació. Les polítiques de creació i destrucció de màquines poden provocar grans canvis tant en el rendiment del runtime com en el cost de l'execució. Com es deia en els objectius, el que es busca és que el runtime s'adapti a les necessitats de l'aplicació per poder aplicar tot el paral·lelisme possible. Això vol dir que el runtime anirà prenent decisions a mesura que l'aplicació avanci. Una condició bàsica que s'imposarà a les polítiques serà que només pugui fer un canvi per cada vegada que s'apliqui. Si es decideix que és un bon moment per apagar màquines, només se n'apagarà una; si decideix que es necessiten més màquines se'n demanarà només una al Cloud Provider.

Es farà una excepció a aquesta norma, permetent demanar o apagar vàries màquines a la vegada, en dos casos concrets. El primer cas es dóna a l'inici de l'aplicació, en aquest moment s'han de demanar totes les màquines necessàries per poder executar l'aplicació o el nombre de màquines que l'usuari hagi especificat. Totes aquestes màquines es poden encendre a la vegada.

El segon cas en que excepcionalment es permetrà modificar en més d'un recurs la quantitat de workers dels que disposa el runtime és quan s'avisi el runtime que no rebrà noves tasques. En aquest moment el runtime ja pot saber que hi ha màquines que no

tornaran a fer-se servir i no són necessàries. Per tant, pot apagar-les totes a la vegada.

Una altra condició que s'imposa a les polítiques que utilitzades durant aquest projecte és que, des del moment en que s'encén el runtime fins que arribi l'ordre d'aturar-lo i apagar els components, per cada mètode que hagi d'executar-se remotament existirà almenys una màquina capaç d'executar-lo.

5.2 Arquitectura del Projecte

Un cop assentades ja les bases sobre les quals es prendran les decisions de disseny, es pot explicar els canvis fets al llarg del projecte. El primer canvi que s'explica és la manera en que s'han ajuntat les dues parts d'aquest projecte: el model de programació COMPSs, i la infraestructura Cloud.

En els capítols que fan referència a COMPSs i a EMOTIVE Cloud, 4 i 2.6.3 respectivament, està explicada quina és l'estructura de cada un, i queda il·lustrada en les figures 4.1 i 2.7. Al llarg de l'apartat es discuteixen diverses opcions per resoldre el projecte, quins nivells extres aporten i quins motius fan que la implementació es decanti per un d'ells.

Acoblar els dos sistemes requereix que COMPSs afegeixi dues variacions importants en la seva arquitectura lògica:

- Afegir l'ús de les màquines virtuals generades per EMOTIVE Cloud a l'ús de les físiques.
- Permetre la comunicació del runtime amb el scheduler d'EMOTIVE.

La primera variació, no implica gaires canvis en quant a arquitectura. COMPSs només accedeix a les màquines físiques per enviar a executar tasques. La forma de fer-ho és mitjançant el JavaGAT que permet tenir diferents adaptadors que assegurin la comunicació entre diferents nodes de la plataforma distribuïda, entre ells, el ssh.

Per poder enviar a executar una tasca a màquines virtuals d'EMOTIVE Cloud existeixen dues opcions possibles. La primera és utilitzar l'API d'EMOTIVE Cloud per que aquesta delegui la tasca a una màquina concreta. La segona opció que ofereixen les màquines d'EMOTIVE és accés a través del protocol ssh.

Donat que l'adaptador que COMPSs utilitza per defecte de JavaGAT és per ssh, sembla que l'opció més encertada per enviar les tasques remotes es mantenir el protocol ssh per enviar les tasques a executar.

La segona variació és la que comportarà canvis en l'estructura. COMPSs està format per cinc objectes actius: TaskAnalyser, TaskScheduler, JobManager, FileInfoProvider i FileTransferManager. D'aquests cinc components, un ha d'encarregar-se de la comunicació amb el Scheduler d'EMOTIVE Cloud per demanar o alliberar recursos.

Si es repassen les funcionalitats de cada un dels components, el que sembla més adequat per realitzar aquesta tasca és el TaskScheduler. La seva funcionalitat és determinar en quin recurs ha d'executar-se una tasca. Això vol dir que ha de tenir un objecte passiu que contingui la informació de quins nodes té disponibles i tota la informació del seu estat. Aquest objecte és el ResourceManager. Una opció pot ser aprofitar aquest objecte perquè quan el TaskScheduler detecti que hi ha escassetat de recursos demani els recursos que necessita i que quan en tingui excés n'alliberi.

L'altre opció que es pot plantejar és crear un nou component que s'encarregui de portar a terme tot aquesta gestió dels recursos i que faci una monitorització de l'estat de les màquines. D'aquesta manera tota la part de computació que comporta la gestió de recursos s'allibera del TaskScheduler i pot anar en un altre node.

El problema d'aquesta opció és que ProActive no permet compartir objectes passius entre diversos objectes actius. Aquest nou component GCM es transformaria en un nou objecte actiu en el moment de la implementació. Cada cop que s'hagués de tornar un objecte complex o enviar-lo com a paràmetre en alguna crida, per exemple, les restriccions de la tasca que es vol executar; s'hauria d'esperar a que es fes la còpia d'aquest objecte i es fes la crida amb aquest nou objecte.

Un segon motiu per descartar aquesta opció és que els components de COMPSs poden estar desplegats en diferents nodes del Grid. Això vol dir que cada cop que es fa una petició sobre un recurs s'ha de fer la còpia dels objectes d'entrada i enviar un missatge utilitzant ProActive a través de la xarxa per fer una petició.

Com s'ha explicat al capítol 3.2, les crides entre objectes actius a ProActive són asíncrones fins al moment en que l'aplicació ha de llegir l'objecte retornat, que llavors el thread queda bloquejat fins a rebre la dada. Com que la majoria d'accions que es realitzarien sobre aquest component són consultes preguntant quin recurs pot executar una tasca, o directament quin és el millor recurs per fer-ho, el thread quedaria bloquejat immediatament i ,a més a més, impediria el tractament de noves peticions al TaskScheduler. Una possible solució seria utilitzar un sistema de senyals que permeti acumular peticions al nou component i a mesura que les resolgués, s'avises al TaskScheduler de que ja té la resposta perque aquest pogués acabar de planificar la tasca.

Sembla, doncs, que la opció de mantenir-ho tot en un sol component és millor en quant a rendiment i que, a més a més, és més senzilla de dissenyar i implementar.

La segona cosa que cal discutir és com serà la comunicació entre el component de COMPSs i el scheduler d'EMOTIVE Cloud. La primera opció és que el codi del TaskScheduler contingui les crides a l'API del Scheduler. És l'opció més senzilla d'implementar i, a més a més, no varia en res l'arquitectura.

De totes maneres, entre les diferents virtuts que es valoren del runtime en l'apartat 5.1 pot trobar-se una que diu que es busca fer un sistema que es pugui adaptar fàcilment a canvis en quant a la comunicació. Pot ser interessant que en diferents moments s'utilitzin diferents schedulers d'EMOTIVE Cloud o, fins i tot, diferents proveïdors. Aquesta opció no s'ajusta a aquest requisit.

La solució per la que s'opta és crear un nou nivell entre el component i l'API del Scheduler, anomenat Connector. Està inclòs dintre del codi del runtime, com un objecte passiu dintre del TaskScheduler. El seu principal objectiu és oferir al component una API per gestionar els recursos Cloud independent del proveïdor i donar informació sobre els costos de l'execució. És a dir, ha de ser capaç de gestionar la creació i destrucció de màquines en qualsevol cloud sigui públic o privat independentment de l'API que faci servir el proveïdor. A més a més, proporciona informació al runtime sobre quan costa mantenir les màquines que té enceses en aquell moment, quin és el cost total que porta acumulat al llarg de l'execució i quin és el cost de crear una màquina de certes característiques.

La figura 5.1 il·lustra com queda l'arquitectura lògica global del sistema. En la part superior de la imatge es pot veure l'arquitectura lògica de COMPSs, en la part inferior es pot veure l'arquitectura lògica d'EMOTIVE. L'objectiu del projecte és que COMPSs

sigui capaç d'utilitzar les màquines virtuals creades per EMOTIVE com a workers. Tal i com es diu al llarg del raonament anterior, la comunicació entre el runtime i els workers es seguirà fent per ssh mitjançant JavaGAT. L'altre canvi que es pot observar en la figura és l'aparició d'aquest connector que farà servir el TaskScheduler per poder crear les màquines comunicant-se amb la capa Scheduler d'EMOTIVE Cloud.

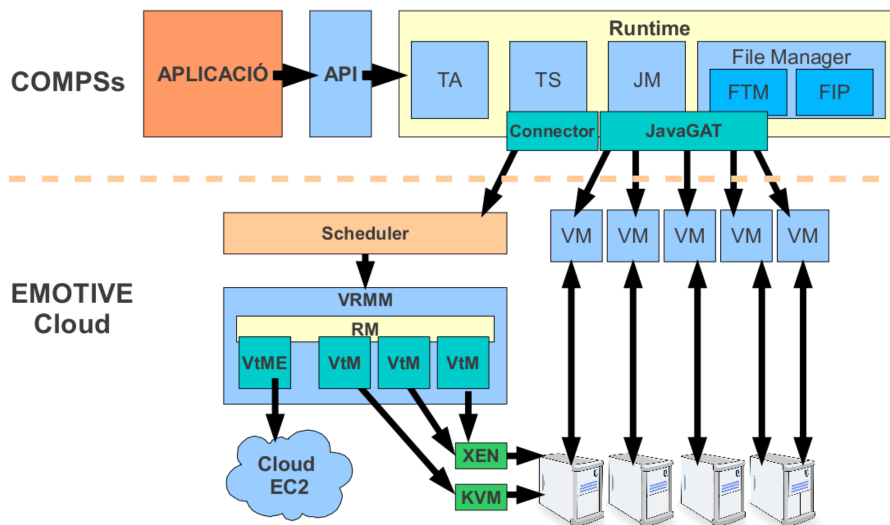

Figura 5.1: Arquitectura lògica del projecte

5.2.1 Preparació dels workers virtuals

Abans d'entrar en detall en el connector cal veure amb detall com han de ser aquestes màquines que demani a EMOTIVE Cloud i com s'han de preparar per tal de poder utilitzar-les com a un worker. En la versió per Grid de COMPSs abans de començar l'execució és necessari que es donin dos requisits per tal de que es puguis executar una tasca en un node remot:

- que hi hagin les classes de les tasques i l'aplicació worker.
- que el master pugui accedir-hi per ssh sense necessitat de password.

Al fer una extensió del model capaç d'utilitzar recursos Cloud i demanar-los de forma dinàmica a mitja execució no es pot configurar l'accés ssh ni fer el desplegament de l'aplicació worker abans de començar l'execució. Existeixen dues maneres de solucionar el problema: encendre les màquines virtuals i enviar les dades i les configuracions durant el procés d'engegada de la màquina o fer que les màquines s'encenguin directament amb el que necessiten utilitzant una imatge concreta.

Les aplicacions no es canvien sovint; normalment, quan es té una versió del codi de les tasques es manté durant un període llarg de temps. En el cas de les classes i el worker, l'opció de carregar les màquines virtuals amb una imatge preparada sembla la més lògica. Si al temps d'encendre una màquina virtual i assignar-li una adreça IP s'hi afegeix el temps de transferir els fitxers, el temps total d'encendre la màquina creix innecessàriament. Sempre i quan el proveïdor de Cloud permeti tenir diferents imatges, és millor estalviar-se

aquesta transferència. Si el proveïdor no permet crear noves màquines que ja tinguin el software necessari, s'haurà d'enviar en el moment d'encendre la màquina.

El segon aspecte que s'ha de valorar és com es fa la configuració del protocol ssh. Així com les aplicacions normalment es mantenen i són comunes per tots els usuaris, la configuració del ssh haurà de ser diferent cada cop que es canvii el node i usuari que executa el master. Si la opció per defecte fos que la imatge contingués la clau, la primera vegada que un node actués com a master, la imatge hauria de canviar-se. A diferència d'abans, la configuració per ssh és ràpida. Només cal enviar a executar una comanda a la nova màquina virtual per que aquesta accepti la clau del master sense necessitar password. Per tant, l'opció de copiar les claus serà més còmode per l'usuari ja que no haurà de canviar totes les imatges cada cop que tingui un nou master.

5.2.2 Implementació del Connector

La funcionalitat principal d'aquest Connector és oferir una API per tal de que el programador pugui executar les crides sobre qualsevol scheduler. Per això aquest apartat comença donant un cop d'ull a la les APIs que han d'oferir els Connectors que es vulguin implementar.

APIs del Connector

Com també es diu a la primera part de l'apartat 5.2 ha de poder donar dues funcionalitats ben diferenciades: permetre gestionar les màquines virtuals i mantenir informació sobre els costos que té l'execució utilitzant una plataforma Cloud.

L'API per gestionar les màquines virtuals ofereix les següents operacions:

- getId(): retorna un identificador del scheduler que es fa servir.
- getDefaultUser(): dóna quin és l'usuari necessari per accedir a les màquines virtuals.
- getDefaultWDir(): indica el directori per defecte on s'han de deixar tots els fitxers temporals de l'execució en la màquina virtual remota.
- getDefaultIDir(): indica el directori on es troben totes les classes necessàries per executar el worker.
- getNextCreationTime(): retorna el temps que preveu que es tardarà en poder tenir llesta una màquina que es demani en aquell moment.
- turnON(String name, String diskImage, ResourceRequest rR): permet encendre de manera asíncrona una màquina virtual a partir de la imatge diskImage i amb les característiques descrites amb el ResourceRequest sempre i quan aquesta es trobi dintre dels límits definits al Connector. En el cas de no entrar en els límits, es retorna que la màquina no pot ser creada. Alguns proveïdors no permeten demanar qualsevol tipus de màquina, el connector haurà d'encarregar-se de fer la transformació que més s'ajusti entre el que li ha demanat COMPSs i el que li ofereix el proveïdor. També es pot donar el cas que el proveïdor del cloud no permeti utilitzar imatges personalitzades, la funció ha de fer la transferència de fitxers per tal de que s'hi puguin executar les tasques.

- terminate(String workerName): dóna l'ordre d'apagar una màquina virtual. Sense opció a recuperar-la, ni a ella ni els fitxers que contenia.
- terminateALL(): apaga totes les màquines virtuals.
- executeTask(String vmId, String user, String command, boolean deleteJSDLFile): permet executar una comanda qualsevol en una màquina concreta del cloud. Aquesta funció es farà servir per copiar les claus ssh i permetre l'execució de tasques a traves de JavaGAT.
- stopReached(): indica al Connector que l'aplicació ja no farà ús del runtime i no entraràn noves tasques.
- reInitialize(): indica al Connector que l'aplicació torna a fer servir el runtime i, per tant, poden entrar noves tasques.

Cada proveïdor té els seus preus i la seva manera de reservar i comptabilitzar l'espai de disc. Per aquest motiu, no es pot fer una implementació que funcioni per qualsevol proveïdor. L'únic que es pot fer és definir una interfície comuna i que cada connector sigui capaç de saber-ne el cost. La segona API definida ofereix les següents funcions:

- getTotalCost(): retorna tot el cost acumulat al llarg del temps que ha durat l'execució.
- currentCostPerHour(): retorna el cost de totes les màquines que el connector té demanades en aquest moment
- getMachineCostPerHour(int procs, float mem, float disk): permet conèixer el cost que té una màquina de les característiques indicades.

Implementació del Connector prototip

Un cop explicat què ofereix el Connector, falta saber com s'implementen les funcions de les seves APIs. No cal entrar en un nivell de detall com per veure tot el codi ni la implementació de totes les funcions, només aquells casos en que és interessant conèixer internament de quina manera es fa la comunicació amb EMOTIVE Cloud per tal de donar aquesta asincronia en les comunicacions.

Un dels problemes que hi ha a l'hora d'utilitzar EMOTIVE és que la crida per demanar una màquina tarda uns segons a donar resposta. És el temps en que l'scheduler tarda en tractar les sol·licituds anteriors, reunir tots els permisos i dispositius per crear la màquina i iniciar una còpia de la imatge base i donar-li un identificador. A mesura que el sistema té més càrrega, més tarda el Scheduler en respondre.

Si el Thread que ha fet la crida ha d'esperar a que aquest pugui donar-li resposta cap petició pot entrar al TaskScheduler. Això vol dir que fins que el scheduler no retorni la resposta no pot entrar-hi cap tasca per ser planificada i que cap de les que acaba pot informar que ja ha acabat i pugui enviar-se'n una altra. Una solució a aquest problema és fer que quan s'invoqui al connector, aquest no comuniqui el ResourceManager directament amb el Scheduler, sinó que es creï un nou thread i es delegui en aquest la part de comunicació. Així el fil de control pot retornar directament al ResourceManager i el TaskScheduler pot continuar tractant les peticions i avisos que li arribin. Quan la màquina hagi estat creada i estigui ja llesta per executar el Thread avisarà al ResourceManager de que ja ha acabat.

El mateix problema apareix per la destrucció, tot i que l'efecte és menor ja que en aquest cas només s'ha d'aturar la màquina i no iniciar cap còpia.

Per tal d'ajudar i simplificar la feina del programador del connector, dintre del mateix package que conté les APIs (integratedtoolkit.connectors), es pot trobar un subpackage amb altres classes que s'encarreguen de l'asincronia de les comunicacions (integratedtoolkit.connectors.utils) fent que el programador només s'hagi de preocupar de gestionar les comunicacions i dades específiques de l'api del seu proveïdor. El programador pot trobar-hi dues eines a la seva disposició:

- CreationThread: és un thread que fa asíncrona la creació. Inicia la comunicació amb l'API i prepara una màquina amb les característiques demanades i la deixa llesta per poder accedir-hi per ssh. Quan la màquina està totalment preparada invoca un mètode del TaskScheduler per tal que la màquina s'afegeixi en totes les estructures del runtime que ho necessitin, siguin del TaskScheduler o de qualsevol altre component. (En l'apartat 5.6 es detallen tots els detalls)

- DeletionThread: és un thread que fa asíncrona la destrucció. Inicia la comunicació amb l'API per eliminar la màquina. (El procés es detalla en el corresponent apartat:5.7)

D'aquesta manera quan el programador del Connector vulgui crear una nova màquina o destruir-la només ha de crear i iniciar un thread d'aquests. Pel que s'ha dit fins ara d'aquests threads, sembla que continguin tota la feina a fer per qualsevol proveïdor. La qual cosa contradiu el que s'ha estat buscant amb el Connector. No és així.

Els thread contenen l'esquema que ha de seguir el procés de comunicació. Els detalls de la comunicació és deixen al programador del connector. El Connector ha d'implementar una tercera API que conté les diferents funcions que s'encarregaran de la comunicació específica amb el proveïdor de recursos. Aquesta API (integratedtoolkit.connector.utils.Operations) té totes les operacions necessàries per generalitzar el procés de comunicació:

- poweron(String name, String diskImage, ResourceConstraints constraintsRequested, ResourceConstraints constraintsGranted): és la petició de crear una màquina a partir d'un nom extern concret, el programador ha de transformar les seves característiques per tal d'adaptarla a l'oferta del proveïdor. Un cop feta aquesta transformació, el programador pot comprobar si la creació de la màquina entra dintre dels límits acceptats. Si entra dintre dels límits, demana la creació d'una nova màquina virtual, comença a comptabilitzar el cost d'aquesta i retorna les característiques de la màquina que finalment s'ha demanat al Cloud Provider. Si la máquina que es demana no pot ser creada perquè viola algun dels límits que s'ha imposat a la creació de màquines, aquesta funció retorna null.

- waitCreation(String vmId): El programador ha de fer esperar el Thread fins que la màquina estigui llesta per poder-hi executar qualsevol comanda. Això implica que si el proveïdor no permet utilitzar varies imatges, en aquesta funció haurà de realitzar-se la transferència de totes les classes que contenen les tasques i de l'aplicació worker.

- poweroff(String workerId): El programador ha d'especificar el codi necessari per destruir totalment la màquina virtual i parar de comptabilitzar el seu cost, però sempre mantenir-lo en el cost acumulat al llarg de tota l'execució.

- addCreationTimeToMean(float time): permet tenir estadístiques de quin temps es tarda a crear la màquina o guardar qualsevol dada interessant sobre la creació segons l'interès del programador.
- getTerminate(): indica si ha arribat al Connector la senyal per apagar totes les màquines enceses, terminateAll.
- getCheck(): serveix per dir si al final de la creació de la màquina aquesta ha de comprovar que sigui útil. És necessari per un cas que es pot donar degut al desfàs temporal amb la petició (explicat a l'apartat 5.6)
- getIP(String vmId): retorna l'IP de la màquina virtual que respon al nom vmID.

Totes aquestes funcions utilitzen dades importants del connector. Per tal d'assegurar que els valors que es llegeixen i escriuen són els correctes serà necessari que les operacions es sincronitzin abans d'accedir al conjunt de les dades. S'ha de vigilar al fer aquesta sincronització. Si un dels threads demana crear una màquina i bloqueja l'accés a totes les dades als altres, tots els threads hauran d'esperar a que la comunicació amb EMOTIVE Cloud acabi. Això provocaria un baix rendiment. Per poder evitar aquest problema, a l'hora de sincronitzar se separa el que és la petició a l'API del Scheduler del que és gestió de dades. Només se sincronitza la gestió de dades, permetent que altres thread accedeixin a elles i puguin completar les seves peticions.

Independentment del proveïdor que utilitzi el connector, aquest sistema serà capaç de comunicar-se amb l'API del proveïdor i portar a terme tota la gestió de recursos. La figura 5.2 ens mostra els diagrames de seqüència que esquematitzen la comunicació necessària per crear una màquina. El diagrama superior a l'esquerra mostra les classes i invocacions per les que es passa per començar a executar el CreationThread. El superior a la dreta, que passa quan aquest CreationThread recén invocat es troba que ja no es poden crear noves màquines perque es superaria el límit imposat. El diagrama de seqüència inferior mostra els diferents passos que el thread de creació segueix.

Vista la implementació d'un connector qualsevol, queda per detallar els aspectes concrets del connector que porta el prototip utilitzant el SimpleScheduler.

Una de les característiques que té el SimpleScheduler és que tota la gestió de la màquina es fa a partir d'un identificador que et retorna al fer la creació. COMPSs no té perquè coneixer aquest identificador únic per cada màquina, el ResourceManager ha de poder referir-se pel nom que coneix, l'adreça IP. Per poder fer aquesta conversió serà necessari que el prototip guardi quin identificador correspon a cada nom amb el que COMPSs identificarà la màquina.

Com ja s'ha dit les funcions que implementen l'API que farà servir COMPSs per comunicar-se amb l'exterior es basaran en les utilitats explicades anteriorment. Per les funcions turnON i terminate només serà necessari crear el corresponent Thread indicant en nom de COMPSs per cada màquina. Per poder implementar el terminateALL es demana a l'estructura tots els noms de COMPSs per poder encendre un thread per cada una de les màquines a apagar.

Només falten les dues funcions que mantenen la informació sobre l'estat del runtime de COMPSs. És a dir, si ha arribat un stopIT al runtime o si s'ha tornat a fer un startIT. Aquesta informació serà important a l'hora de tenir en compte si val la pena haver creat la màquina.

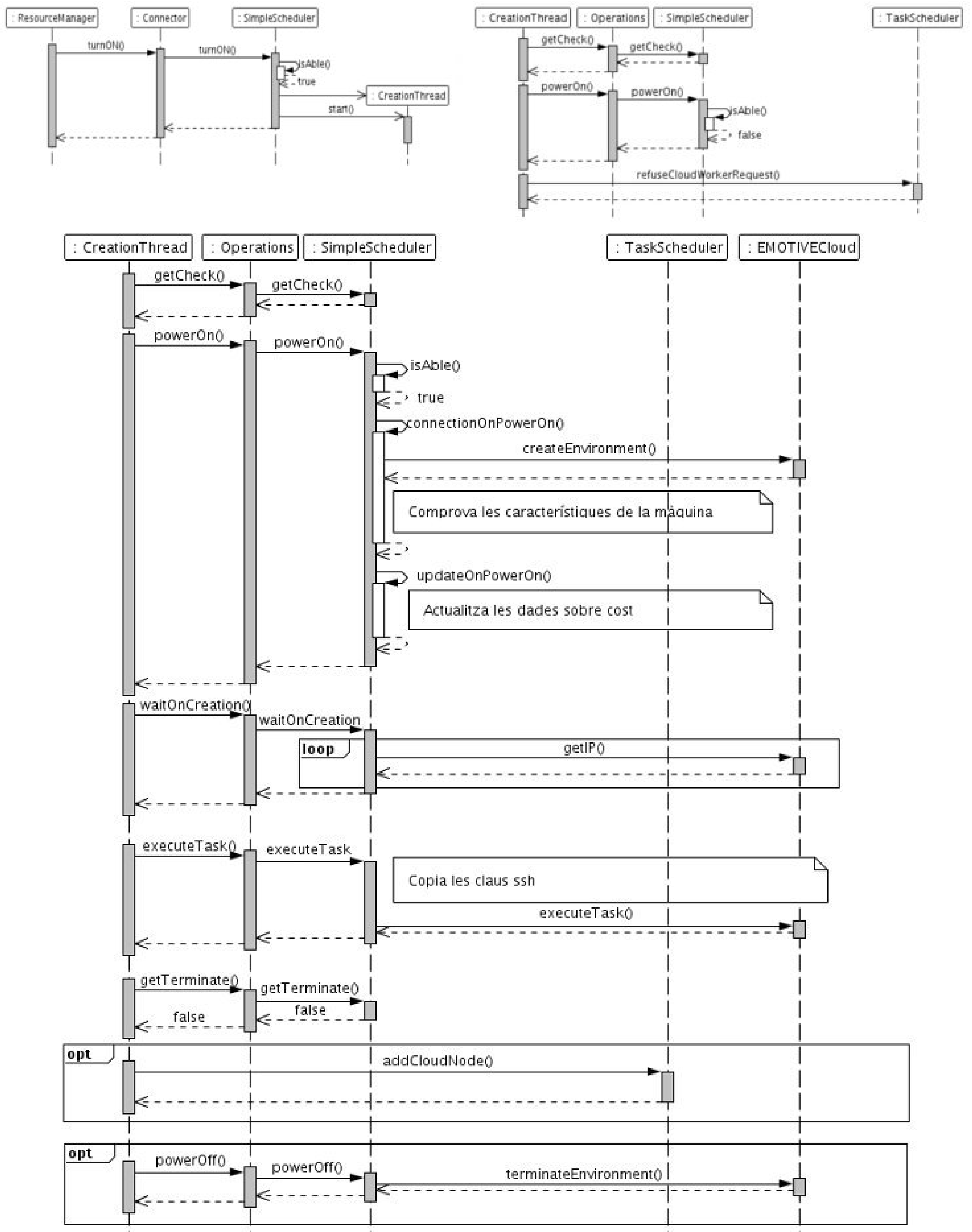

Figura 5.2: Diagrama de seqüència del la creació de noves màquines

Per la banda de l'API del cost, cal que el connector comptabilitzi els recursos que es tenen en compte al fer el càlcul del cost: número de processadors, quantitat de memòria i espai de disc utilitzat. Les funcions l'únic que fan és consultar variables i fer alguna operació aritmètica simple. Tot el càlcul del cost acumulat es fa únicament en moments en que varien els recursos dels que disposa el connector, és a dir, en la creació i destrucció de màquines. Quan el connector rep una petició calcula quin és el cost acumulat fins aquell moment, quin serà el preu de mantenir les màquines que es faran servir després de la petició i es guarda el moment en que s'ha fet la petició. D'aquesta manera, per calcular el cost acumulat només ha de fer la següent operació abans d'actualitzar el preu:

$$Cost_i = Cost_{i-1} + Preu_{i-1} * (Temps_i - Temps_{i-1})$$

El diagrama inferior de la figura 5.2 mostra l'ordre en que els threads invoquen les operacions del connector en que s'estableixen les comunicacions. Pel thread de creació se segueixen els següents passos:

1. Invoca poweron que fa la petició a EMOTIVE Cloud. Abans de demanar cap recurs, comprova que el recurs que es vol demanar entri dintre dels límits establerts. Si la màquina demanada quedés fora dels límits, tot el procés s'anul·la i s'avisa al TaskScheduler que la petició no pot ser tractada. Si la màquina entra en els límits, es crea un objecte Compute que descriu la màquina desitjada de la manera que ho fa EMOTIVE Cloud. Amb aquest objecte es demana que es creï una màquina virtual a la que assigna un identificador. En aquest moment el procés queda aturat fins que EMOTIVE Cloud notifica que la màquina corresponent a la petició ja s'ha començat a crear. Les característiques que s'han demanat poden variar, per exemple: l'espai de memòria o de disc. Finalment s'actualitza totes les dades que es tenen guardades de la petició i es comença a comptabilitzar el cost.

2. S'invoca la funció waitCreation, que espera a que la màquina pugui operar. En el cas d'EMOTIVE això vol dir que aquesta estigui totalment encesa i amb una adreça IP. Per saber si la màquina està llesta, es fa servir un bucle per enquesta, que dorm 0.5 segons entre peticions.

3. addCreationTimeToMean guarda el temps que s'ha tardat a encendre-la per si es vol fer algun tipus d'aproximació en funció del que es tarda normalment i així poder calcular el temps de creació si no el dóna el Cloud Provider.

4. executeTask: envia una tasca a través de l'API del scheduler i espera a que acabi. En aquest cas s'utilitza per copiar les claus ssh i obrir les comunicacions amb el master.

5. L'últim pas només ha de fer-se si el resultat de comprovar si la màquina és útil és negatiu i la màquina ha d'apagar-se. Per fer-ho no és necessari encendre un Thread, ja que el thread de creació pot encarregar-se d'eliminar-la invocant la mateixa funció: poweroff. Aquesta funció elimina la màquina de totes les estructures que té el connector. Després es comunica mitjançant l'API amb el Scheduler per donar l'ordre d'apagar-la i quan EMOTIVE Cloud confirma la destrucció s'actualitzen totes les dades de cost.

El thread de destrucció és molt simple només ha de fer la crida al poweroff. I ja ha quedat descrita en l'últim pas del de creació.

5.2.3 Instanciació del Connector

Un altre aspecte que cal veure és com s'inicien les crides en el Resource Manager i com s'indica quin connector es vol fer servir. Per encendre un connector qualsevol són necessàries com a mínim 3 dades:

- el nom complet de la classe que conté el Connector.
- el nom del servidor que conté el Scheduler(pot ser l'adreça IP).
- el port al que s'ha de connectar.

Quan s'inicialitza el ResourceManager, també s'inicialitza el connector. La primera de les dades que es fa servir és el nom de la classe. Amb el següent codi, el ResourceManager agafa qualsevol classe que es trobi dins del classpath com a connector i crear-ne una instància. El constructor d'aquest connector pot ser diferent segons la implementació. Per tal de poder oferir una API comuna i no definir uns paràmetres estàndars, el constructor farà servir com a paràmetre un hashMap que contingui tots els paràmetres que necessitarà. En el cas d'aquest prototip: el nom del servidor i port . A partir d'aquest moment, si s'encasta aquest objecte en una interfície (cost o connector) el ResourceManager pot accedir a les funcions indicades en aquestes APIs.

```
HashMap<String, String> h= new HashMap();
h.put("Server", SCHEDULER\_NAME);
h.put("Port", SCHEDULER\_PORT);
Class conClass= Class.forName(CONNECTOR\_PATH);
Constructor ctor = conClass.getDeclaredConstructors()[0];
connector=(Connector)conector;
cost=(Cost)conector;
```

5.3 Gestio de Recursos

Fins a aquest punt del capítol 5, queda detallat els dos nivells més baixos de l'arquitectura del sistema: les màquines que crea EMOTIVE Cloud i de quina manera s'aconsegueix comunicar de forma asíncrona el runtime amb el scheduler. El següent nivell és el runtime de COMPSs, que tal com mostrava la figura 5.1 serà qui utilitzarà el Connector descrit a l'apartat anterior i JavaGAT per comunicar-se amb les màquines.

La comunicació entre les màquines i el runtime no ha canviat respecte la versió original, per tant, no cal entrar a comentar com es fa. El que sí que canvia és la forma en que es gestionen els recursos. En la versió per Grid, els recursos es mantenen al llarg de tota l'aplicació i és l'usuari qui els indica en el moment d'iniciar l'execució. El fet d'estendre el model per que sigui capaç d'utilitzar els recursos Cloud canvia aquesta premissa i obliga a mantenir la informació d'aquestes estructures de dades actualitzades segons els canvis que es produiran al llarg de l'execució, ja sigui registrar nous recursos creats o esborrar els que s'alliberen.

Tota la gestió de recursos es fa al component TaskScheduler, que conté tres estructures que mantenen les dades relacionades amb els recursos:

- ResourceManager: que conté totes les característiques dels recursos i controla la quantitat de feina que tenen acumulada.
- ProjectManager: que conté totes les dades necessàries per poder accedir al node i executar-hi les tasques.
- QueueManager: que manté la infomació necessària per tal de gestionar totes les tasques pendents.

Tal com es diu a l'inici del capítol, tota la gestió dels nous recursos es produeix dintre del TaskScheduler, per tant, és aquest qui s'encarrega de gestionar les tres estructures. En la figura 5.3 es pot veure com es relacionen totes les parts per tal de poder obtenir la informació que cada una necessita per que el TaskScheduler pugui fer la seva funció.

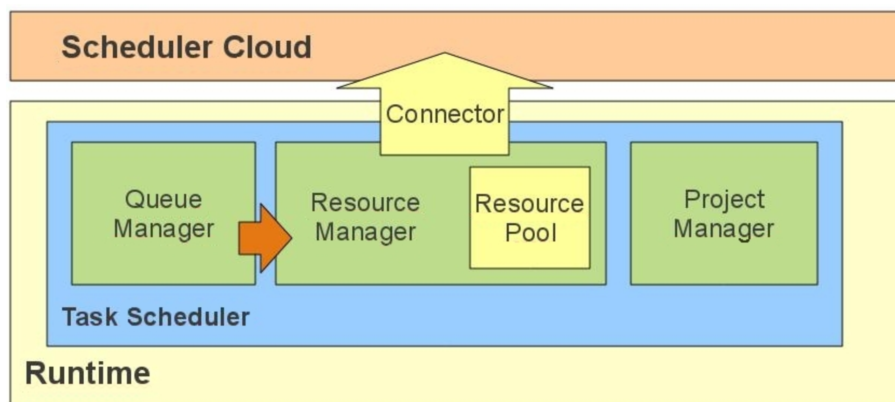

Figura 5.3: Estructures de dades de gestió que trobem en el TaskScheduler

A continuació, s'explica com estan implementades aquestes tres estructures i quins canvis han patit respecte les estructures ja existents en la versió del runtime per Grid (ResourceManager i ProjectManager) per tal d'adaptar-les o millorar-ne el rendiment.

5.3.1 ProjectManager

El ProjectManager no ha tingut cap canvi pel que fa a estructura. Està format per un document XML sobre el que es fan les consultes utilitzant la llibreria de Java xPath. En aquest aspecte no s'ha introduït cap canvi, permet accedir ràpidament a qualsevol de les propietats del node. A més a més, de cara a afegir noves propietats, només caldrà indicar el nom de la propietat i del node per tal de poder consultar-la de la mateixa manera que fa en l'altra versió.

Quan s'inicialitza l'objecte s'indica quin és el fitxer project.xml que s'ha de fer servir i el carrega en memòria permetent fer-hi consultes de forma eficient.

El canvi que s'ha fet el aquest objecte es troba en les operacions que ofereix. Ara ofereix dues noves funcions:

- addProjectWorker: que permet afegir un nou worker al document XML. Només cal d'indicar-li el nom que es vol donar al recurs i el valor de les propietats que cal que tingui.
- removeProjectWorker: a partir del nom d'un worker, l'elimina de tota l'estructura.

5.3.2 ResourceManager

El ResourceManager és la classe que més canvis ha patit. La funcionalitat que té el ResourceManager és la de gestionar totes les característiques de les màquines per tal de saber quines compleixen les restriccions d'una tasca. També monitoritza la càrrega de treball que el runtime li ha donat a cada worker per que no es sobrepassi el límit que definit per l'usuari.

En el codi de COMPSs original el funcionament era el mateix que el del ProjectManager. L'usuari introduïa el resources.xml, a l'iniciar l'execució es carrega el fitxer i les consultes es fan utilitzant xPath. Cada cop que el runtime envia a executar una tasca s'incrementa el valor que hi havia en el XML controlant el nivell de càrrega del node i quan acaba una tasca el TaskScheduler l'avisa per que redueixi el nivell del node que l'havia executat. D'aquesta manera, al fer la consulta a xPath per saber quines màquines poden executar una tasca, es donen tots els requisits de la tasca i aquest retorna únicament aquelles màquines que poden executar-la tenint en compte els respectius nivells de càrrega.

A part del canvi per tal d'oferir dinamisme als recursos disponibles, s'ha de tenir en compte que el ResourceManager serà l'encarregat de decidir quin és el moment de demanar noves màquines o d'alliberar-les. Per tal de separar el que és la política de gestió i el que són les màquines en si el que es fa és crear una nova estructura dintre del ResourceManager, anomenada ResourcePool. Aquesta conté tota la informació referent a les màquines, quines són les seves característiques, el seu nivell de càrrega, quin tipus de recurs són (físic o cloud).

És important tenir en compte que la principal funcionalitat que té el ResourceManager és donar quines màquines són capaces d'executar una tasca concreta. Els mètodes de les tasques i les seves restriccions ja estan definits a l'inici de l'execució. Quan una màquina es crea o és llegeix del fitxer, es pot saber quins tipus pot executar si té un nivell de càrrega suficientment baix. Per no haver de fer la consulta cada vegada sobre el mateix fitxer, es guarda aquesta relació de manera bidireccional: entre màquines que poden executar un mètode i mètodes que poden ser executats per una màquina. A més a més, a l'hora d'aplicar les polítiques siguin quines siguin en aquest ResourcesPool es poden tenir consultes molt més específiques que simplifiquin la llegibilitat del codi de la política i fer consultes que aparentment semblen costoses d'una manera més eficient.

En qualsevol cas, a l'inici de l'execució s'ha de llegir el fitxer XML per saber les dades, guardar quines són les característiques de cada màquina i mirar quins tipus de tasca pot executar per guardar les relacions amb els mètodes. Tota màquina que es llegeixi del fitxer XML serà considerada física.

Durant l'execució poden crear-se noves màquines. Si és així, el ResourceManager ha d'indicar al ResourcePool les característiques de la màquina i els mètodes que pot executar per així poder assignar-li les tasques que arribin.

En el cas d'apagar una màquina que està en el Cloud el que ha de fer el ResourceManager és demanar que s'apagui el node al connector i esborrar-lo del ResourcePool. El procés sencer per crear o destruir les màquines s'explica en els pròxims apartats del capítol.

Tal i com es diu en les directrius del disseny, una constant durant l'execució del runtime és que per cada mètode que pot ser executat remotament, existirà almenys una màquina on pugui ser executat. L'única manera de complir-la és creant a l'inici de l'execució tots els nodes necessaris i controlar quins recursos no poden ser apagats en un moment determinat.

El responsable de controlar que la constant es compleixi és el ResourceManager, ja que és qui aplica les polítiques de creació i destrucció. De totes maneres, l'única estructura de dades que coneix tots els detalls dels recursos disponibles és el ResourcePool, per tant, al final serà aquest qui podrà decidir si una màquina pot aturar-se o no.

Una opció per comprovar-ho és vigilar que un subconjunt de les màquines disponibles pot executar els mètodes de la màquina que es vol eliminar cada cop que es comprobi si una màquina pot ser eliminada. Si es fa la pregunta molt sovint, aquest càlcul pot fer-se molt pesat. La solució per la que s'ha optat és classificar els recursos del ResourcePool en tres conjunts:

- Físics: que són aquells que se li han indicat al runtime en iniciar l'aplicació.
- Crítics: són un subconjunt de les màquines virtuals demanades al Cloud Provider. Són aquelles màquines que permeten complir la condició. Si una d'aquestes màquines intenta ser apagada, el ResourcePool indicarà que no es pot.
- No crítics: són la resta de màquines virtuals demanades al Cloud Provider.

Cada cop que es crea una nova màquina cal recalcular aquest conjunt crític, ja que pot haver reduit el cost. L'avantatge que té aquest sistema és que cada vegada que es vulgui eliminar una màquina no cal comprovar que hi hagi una altra màquina per cada mètode; n'hi ha prou amb comprovar que pertanyi al conjunt de no crítics.

Determinar el subconjunt de màquines capaces d'executar tots els mètodes que tingui el cost més baix és un problema NP-complet. Com que el que es busca és un algoritme capaç d'executar-se ràpidament, s'utilitza un algoritme voraç que determini un bon subconjunt, tot i no ser òptim.

5.3.3 QueueManager

Aquesta és una nova estructura que en la versió per Grid de COMPSs no existia. S'han explicat les dues estructures que mantenen una informació actualitzada sobre les característiques dels recursos dels que disposa el runtime i com accedir-hi. L'objectiu d'aquesta estructura és monitoritzar totes les tasques que ja estan llestes per executar, però que encara no han acabat. Amb la informació que conté es podrà determinar el nombre de màquines que realment requereix l'aplicació per tal d'obtenir un resultat el més ràpid possible.

Una tasca qualsevol pot estar en tres estats:

- en espera: quan no hi ha cap recurs on aquesta tasca pugui executar en un moment determinat.
- assignada: quan ja ha estat planificada per executar-se en un recurs.
- per reschedular: quan la tasca ja havia estat assignada a un recurs, però l'execució ha fallat i en aquell moment no es disposa de cap recurs disponible on pugui ser executada.

Per tal de poder gestionar els diferents estats, les tasques es classifiquen en alguna de les següents estructures de dades:

- noResourceTasks: una llista que agrupa totes aquelles tasques que no existeix cap recurs on puguin ser executades.
- nodeToTask: és una estructura que guarda per cada recurs aquelles tasques que té assignades.
- tasksToReschedule: és una llista amb totes les tasques per reschedular.

Les polítiques que decideixen si és necessari o no crear o destruir màquines es basen en la informació que aquestes tres estructures els ofereixen. Per exemple, si hi ha tasques que no poden ser executades en cap recurs, es mira si hi ha necessitat de demanar una nova màquina capaç d'executar-les. Per tal de poder demanar més recursos per tasques que ja poden ser executades, el que s'ha de mirar són les llistes d'espera de cada un dels nodes i les tasques que estan executant, responsabilitat del nodeToTask.

NodeToTask és l'estructura de dades que alimenta les polítiques més crítiques del sistema: les de gestió de recursos i planificació de tasques. Per tant, ha de ser capaç de donar una informació el més precisa possible, per tal d'evitar que en aquestes es produexin errors. La millor manera és monitoritzar al detall quins nivells de càrrega té la màquina i intentar pre-planificar l'ordre i el moment en que s'executarà cada una de les tasques. Això implica intentar preveure quines tasques s'executaran en un moment determinat. Cal separar clarament els dos estats que poden tenir les tasques: executant i esperant. Per això es defineix un nou objecte: ResourceQueue, format per un nombre de slots. Els slots representen les tasques que poden ser executades de forma paral·lela en un node. Cada recurs tindrà tants slots com tasques puguin ser executades de forma paral·lela en ell. Quan una tasca comenci a executar-se ocupa un d'aquests slots, i quan acabi, l'allibera per que una altra tasca pugui utilitzar-lo.

Per cada slot es guarda quina tasca està executant-se i una llista de tasques que estan esperant a que quedi lliure. Una tasca només podrà estar esperant per un slot. Amb aquesta implementació, si es guarda el temps mig d'execució d'una tasca per cada un dels mètodes, el número de tasques que estan esperant i en quin moment ha començat a executar la tasca que ocupa l'slot, pot aproximar-se el temps que es tarda a tenir el slot lliure amb seguint l'algoritme presentat a continuació, on TE vol dir Temps d'espera, TM significa temps mig, TI i TA són els TimeStamps Inicial i Actual, respectivament, i el subíndex Exe indica Mètode en Execució.

$TE := TM_{Exe} + (TA - TI_{Exe})$
Per cada mètode m **fer**
$TE := TE + \#tasques_m * TM_m$

Al llarg de l'apartat 5.4 s'explica com aquestes estructures canvien al llarg del temps per tal de gestionar les dades de forma eficient.

5.4 Planificació de tasques

Aquesta secció pretèn fer veure quins canvis ha patit el procés des que una tasca entra en el TaskAnalyser fins que s'executa. Explicar com es mouen les tasques en els components i les estructures creades per gestionar-ho. La visió es dóna des del punt de vista d'uns recursos estàtics, no s'entra en com un nou node rep dades de la planificació, això s'explica a l'apartat 5.5.1.

5.4.1 Canvis en l'entrada d'una tasca al runtime

Igual que en la versió per Grid de COMPSs, les tasques entren en el runtime a través del TaskAnalyser. El TaskAnalyser no ha canviat gaire, tot i que, s'han afegit un parell de funcionalitats per facilitar l'aplicació de les polítiques.

La primera funcionalitat que s'ha afegit és una monitorització del nombre de tasques que queden en el graf i de quin tipus són. Per fer-ho s'ha creat l'estructura actualTaskCount, que indica quantes tasques de cada mètode hi ha en el graf. La forma d'actualització és molt senzilla: cada cop que entra una nova tasca al graf s'incrementa el valor del mètode de la tasca. El comptador es decrementa en el moment que una tasca del mètode abandona el graf, és a dir, en el moment en que s'envia al TaskScheduler.

L'actualització d'aquests comptadors és senzilla, de totes maneres apareixen dos problemes: la inicialització i l'accessibilitat.

La inicialització és un problema perquè no pot saber el número de mètodes que hi ha des d'un principi. Per saber-ho, s'ha de consultar el ConstraintManager, que només es troba al TaskScheduler. La solució per la que s'opta és utilitzar un Map, de manera que per cada nou mètode que es tingui s'afegeix dinàmicament un comptador. Una altra solució hagués estat posar una instància del ConstraintManager al TaskAnalyser.

El segon problema és l'accessibilitat de les dades. Cal tenir en compte que aquestes dades, les farà servir el TaskScheduler per aplicar les polítiques de recursos, per tant, han d'estar en un format que pugui entendre i de ràpid accés. Una manera d'identificar el mètode és a través de la seva signatura, el problema que té és que per accedir al map, ja sigui per lectura o escriptura, s'haurà de calcular la signatura del mètode i passar per una funció de hash per saber en quina posició es troba. Això fa que l'accés no sigui ràpid. Una altra forma de fer-ho és associar a cada signatura d'un mètode a un nombre enter. Així, a l'entrar la tasca al TaskAnalyser, calcula la seva signatura i assignar a la tasca l'identificador del mètode associat. L'accés al Map ara serà més ràpid ja que amb un enter no ens cal passar la funció de hash tan complexa. La relació Signatura-Identificador queda guardada a l'estructura methodSignatureToId.

El nou problema que es planteja és el següent: existeix una relació Identificador-nombre de tasques. Però aquesta relació ha de ser interpretada tant per TaskAnalyser com per TaskScheduler, o sigui, que tots dos han de tenir la mateixa relació Signatura-Identificador. Es pot pensar que una bona forma de coordinar-ho és utilitzar dues instàncies del ConstraintManager, una per cada component i assignar-lo segons l'ordre d'aparició en la interfície anotada. Però fer-ho així seria incorrecte. S'ha de tenir en compte que els components poden estar executant-se en dos hosts diferents, per tant, el programa pot estar en dues implementacions de la JVM diferents. Això suposa que a l'hora de llegir de la interfície, cada un dels ConstraintsManagers pot fer-ho en un ordre diferent i, per tant, la interpretació de l'estructura serà diferent a cada component. Per aquest motiu, també ha quedat descartada per saber el nombre de mètodes que hi havia a la interfície.

La solució per la que s'ha optat és que, al crear el ContraintManager durant la inicialització del TaskScheduler, s'assigni a cada un dels mètodes un identificador. Quan la inicialització del runtime acabi i s'enviï la primera tasca el TaskAnalyser llençarà una primera petició sobre el TaskScheduler demanant-li que li envii les relacions Signatura-Identificador. Aquesta estructura no canvia al llarg de l'execució, per tant, a partir de la primera tasca els valors seran coherents.

La segona funcionalitat que s'afegirà ha de permetre poder gestionar correctament els

fitxers que contenen les màquines virtuals. És necessari organitzar tot un sistema que permeti conèixer el nombre de lectors d'una versió d'un fitxer i quins mètodes executaran aquests lectors. Com que la informació està relacionada amb fitxers, tota la responsabilitat recau sobre el FileManager, concretament sobre el FileInfoProvider. El diagrama de classes del FileInfoProvider és el que mostra la figura 5.4

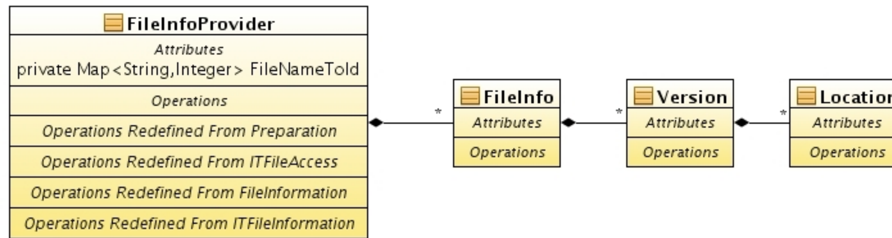

Figura 5.4: Diagrama de classes resumit del FileInfoProvider

Per tenir el compte de lectors de cada una de les versions, només s'ha hagut de modificar l'estructura de la classe Version i afegir mètodes al FIP i a FileInfo per tal de poder accedir no només a les Locations de la versió sinó també als seus lectors. S'ha afegit a Version:

- un Map amb el nombre de lectors per cada un dels mètodes. Es fa servir un Map pel mateix motiu que abans, no es coneix prèviament el nombre de mètodes.

- un comptador del nombre total de lectors, per evitar haver de calcular cada vegada el nombre de lectors totals.

Vistes ja les estructures on es guarden les dades, falta conèixer la manera en que aquesta informació s'actualitzarà. Quan el TaskAnalyser analitzi la nova tasca registra els accessos igual que ho feia, amb la funció registerFileAccess; però, a més a més, s'indica quin mètode vol fer l'accés. Si aquest accés implica lectura (R o RW), s'incrementa el nombre de lectors del mètode de la última versió, sinó s'actuarà de la meteixa manera que en la versió inicial.

Una altra manera que hi ha d'incrementar el nombre de lectors és que el master llegeixi directament el fitxer, amb la funció OpenFile, cridada des de l'API, o getFile, si el runtime demana portar un fitxer al master. Independentment del cas es crida la funció registerFileAccess, amb la diferència de que aquest cop la crida es farà amb un -1 com a identificador del mètode.

És important decrementar els comptadors tan aviat com sigui possible. El moment més immediat que hi ha es pot detectar al FileTransferManager, quan detecta que el fitxer ja s'ha copiat, en aquell moment el fitxer ja s'ha llegit ja que pot llançar-se l'execució de la tasca. Tot i ser el més immediat, no és el millor moment. Si la tasca falla, ha de llançar-se en un altre node i, per tant, s'ha de reincrementar el nombre de lectors i es pot haver perdut la versió del fitxer. Per tant, per poder decrementar el comptador, cal esperar a que la tasca acabi correctament. El primer en detectar que una tasca ha acabat és el JobManager. De la mateixa manera que informa al FileInfoProvider que s'han creat els fitxers de sortida amb els valors finals, avisa de que els fitxers d'entrada ja no s'hauran de llegir i es decrementa els comptadors.

5.4.2 Alliberament de les dependències d'una tasca

Quan el TaskAnalyser detecta que una tasca queda lliure de dependències i que ja pot ser executada, aquesta s'envia al TaskScheduler que ha de determinar en quin dels recursos pot executar-se i en quin moment es pot enviar. En la versió per Grid de COMPSs, el TaskScheduler demana al ResourceManager una llista amb els recursos que poden executar la tasca i que encara no tenen cobert el nombre de tasques que es s'hi poden enviar simultàniament. Aquest fa una consulta sobre el fitxer XML de recursos i dóna la resposta.

Si aquesta llista que retorna està buida, la tasca passa a esperar-se en una llista on estan totes les tasques pendents de ser executades. Sinó, es busca el millor d'aquests recursos, és a dir, aquell que té més fitxers requerits per la tasca i s'envia a executar directament indicant al ResourceManager el recurs escollit per que n'ocupi un espai.

En la versió per Cloud, aquest procés és diferent. Quan arriba una nova tasca al TaskScheduler, aquest fa la mateix pregunta al ResourceManager: quines màquines poden executar en aquell moment la tasca; i el ResourceManager retorna la mateixa llista que abans. En el moment en que es retorna aquesta llista és quan el procés comença a canviar ja que entra en joc en QueueManager.

Si la llista té algun recurs, s'escull el millor de la mateixa que en el cas anterior, s'envia la tasca a l'escollit i s'indica al ResourceManager que té un espai ocupat. A més a més, també cal indicar al QueueManager que aquell recurs ha començat a executar la tasca. El QueueManager afegeix la tasca com que està executant-se en el seu ResourceQueue ocupant un dels slots i en guarda també el moment en que es comença a executar la tasca. Finalment el TaskScheduler envia la tasca al JobManager, perque aquest faci tots els tràmits necessaris per que la màquina executi la tasca i guarda l'identificador del Job que se li dóna a la tasca. És el procés que segueix el primer condicional de la figura 5.5.

En cas contrari, si la llista torna buida, significa que no existeix cap node que en aquell moment pugui executar la tasca. En la versió original, s'hagués afegit a la cua de tasques pendents a executar i s'hagués acabat el procés. En la versió per Cloud, queda per determinar si hi ha alguna màquina que pugui executar-la. Per tant, es torna a preguntar al ResourceManager si existeix algun recurs capaç d'executar-la encara que no sigui en aquest moment. Evidentment, si la llista torna buida és que encara no existeix un recurs capaç d'executar-la i, per tant, la tasca s'encua a la llista noResource del QueueManager. En la figura 5.5, correspon a entrar en els dos condicionals per l'else.

Si per contra la llista té algun recurs, vol dir que passaria a estar pendent d'execució esperant tenir un slot lliure. Queda per decidir a quin dels slots de tots els recursos disponibles s'assigna. De la llista de recursos ha retornat el ResourceManager es busca el aquell que tingui un slot lliure el més aviat possible. El QueueManager demana per cada recurs quin és el slot que acabarà abans, n'escull el menor, és a dir, aquell on començaria a executar-se abans si es mantingués una sola llista de tasques pendents i indica al ResourceManager en quin recursos és millor executar.

El TaskScheduler rep quin és el millor recurs i confirma al QueueManager que la tasca s'espera en aquell node. Quan aquest rep la confirmació l'encua a l'slot amb menys temps d'espera, afegint-lo a la cua d'espera de l'slot corresponent. Tot aquest procés queda il·lustrat a la figura 5.5 en el cas que encara no s'ha explicat.

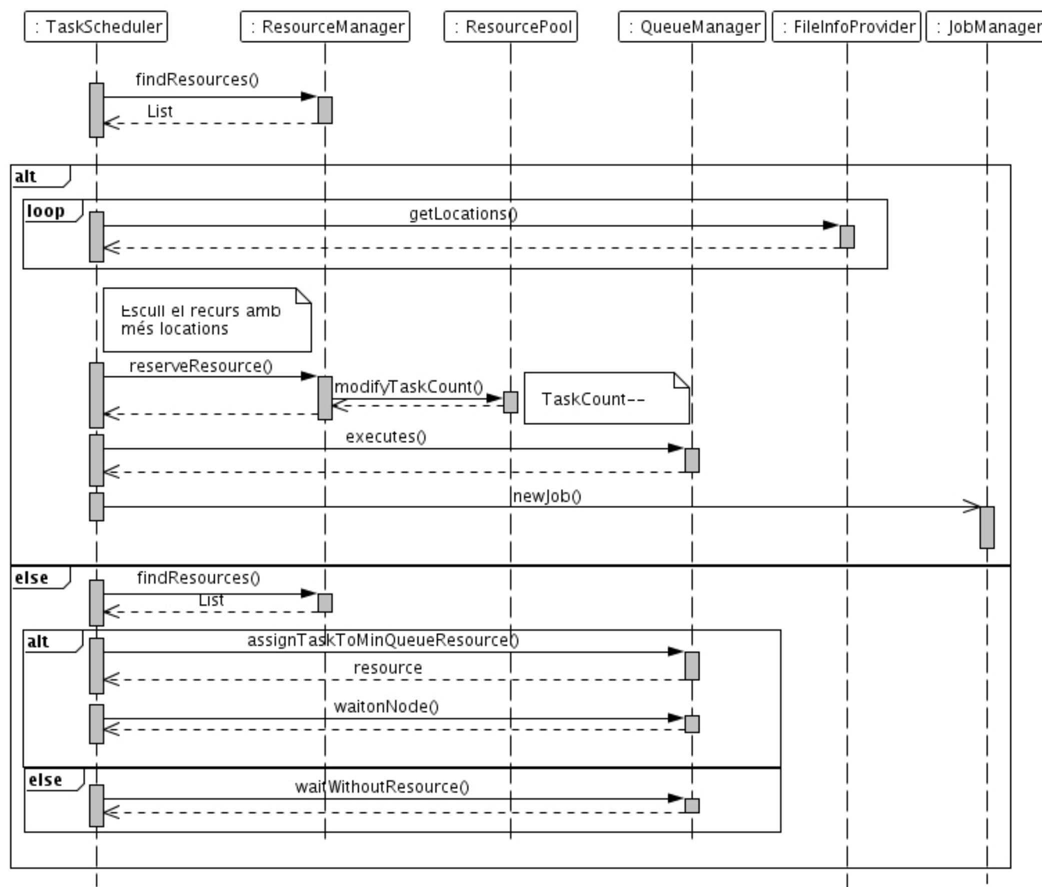

Figura 5.5: Diagrama de seqüència a l'entrar una nova tasca al TaskScheduler

5.4.3 Una tasca acaba correctament l'execució

El procés és similar que en la versió per Grid de COMPSs, tot i que, s'han de tenir en compte les noves estructures de dades. El procés comença en el JobManager esperant rebre una notificació conforme un Job ha acabat correctament. Al rebre-la, avisa al FileManager que els fitxer resultats ja s'han generat perque puguin ser detectats com una font per les noves còpies i avisa al TaskScheduler indicant l'identificador del Job que ha acabat.

Quan el TaskScheduler rep l'avís, mira quina tasca correspon al Job, actualitza les estructures de dades, mira si té alguna tasca pendent que es pugui actualitzar en el node i avisa al TaskAnalyser de que la tasca ha acabat perquè aquest pugui alliberar les dependències. Això ja es fa en la versió de COMPSs pel Grid, la diferència es troba en l'actualització de les estructures i com se seleccionen les tasques que s'envien.

Acabar una tasca significa que una altra tasca pot entrar a executar-se simultàniament en el recurs que l'estava executant, per tant, s'haurà d'avisar al ResourceManager que pot acceptar una nova tasca. Aquest es guardarà dintre del ResourcePool que el recurs que accepta una tasca. Per altra banda, també cal actualitzar el QueueManager. El que cal fer és alliberar l'slot que ocupava en el recurs en la ResourceQueue de la màquina, acumular el temps que ha tardat en les mitjanes del mètode i avisar al TaskAnalyser que la tasca ja s'ha acabat. Tal i com es mostra al diagrama de seqüència de la figura 5.6.

Un cop fet això, totes les estructures ja saben que la tasca ha acabat, només queda

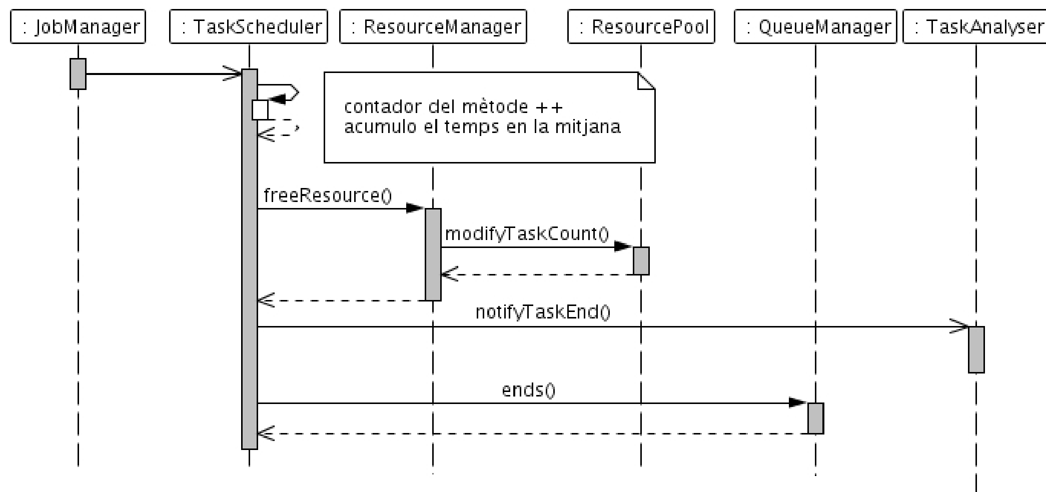

Figura 5.6: Diagrama de seqüència de l'actualització de les estructures al acabar una tasca

assignar una nova tasca al recurs. La decisió estarà basada en un sistema de prioritats ordenats de la següent manera:

- Tasques per re-planificar: com més fitxers de la tasca tingui el recurs millor
- Tasques pendents esperant per l'slot: com més antiga més prioritat
- Tasques d'altres slots del mateix recurs: agafa la primera tasca del recurs amb més càrrega.
- Tasques d'un altre recurs: la que requereixi menys transferències.

El procés comença en el TaskScheduler que comprova si hi ha alguna tasca que hagi fallat en el node que se l'hi havia assignat. Si és així, el tractament d'aquestes serà prioritari. El primer pas és escollir quina de totes les tasques ha d'enviar-se al recurs. Per cada tasca a re-planificar, es mira que no sigui el mateix recurs en el que ha fallat i pregunta si aquella màquina pot executar la tasca. De totes les tasques que compleixin aquestes restriccions es seleccionarà aquella que tingui més fitxers en el node. S'indica a QueueManager que la tasca s'executarà en el slot alliberat i al ResourceManager que el recurs té una tasca més executant-se i es delega en el JobManager l'execució aquesta tasca. Tot aquest procés queda il·lustrat en la figura 5.7.

Si per contra no hi ha cap tasca a re-planificar que pugui executar-s'hi, l'encarregat de buscar la nova tasca és el QueueManager. Primer consulta la ResourceQueue del recurs. Si hi ha alguna tasca esperant per aquell slot, aquest l'ocupa directament. En cas de que n'hi hagi més d'un passa a ocupar-lo la tasca més antiga, és a dir, la primera de la cua. S'esborra de la llista de tasques pendents i passa a estar executant-se guardant el moment en que s'ordena el tractament. Un cop ja s'ha fet tot el tractament a les estructures, s'indica TaskScheduler que aquella tasca pot ser llançada en el recurs. Es guarda al ResourceManager que s'està executant una nova tasca i s'envia al JobManager.

Si no hi ha cap tasca esperant per aquell slot, es miren les cues de tasques pendents de la resta de slots de la mateixa màquina. Com que tots els slots del mateix recurs poden executar els mateixos mètodes, es pot escollir qualsevol de les tasques que tinguin. La decisió és pren en funció del temps d'espera de cada un dels slots. S'escull la primera

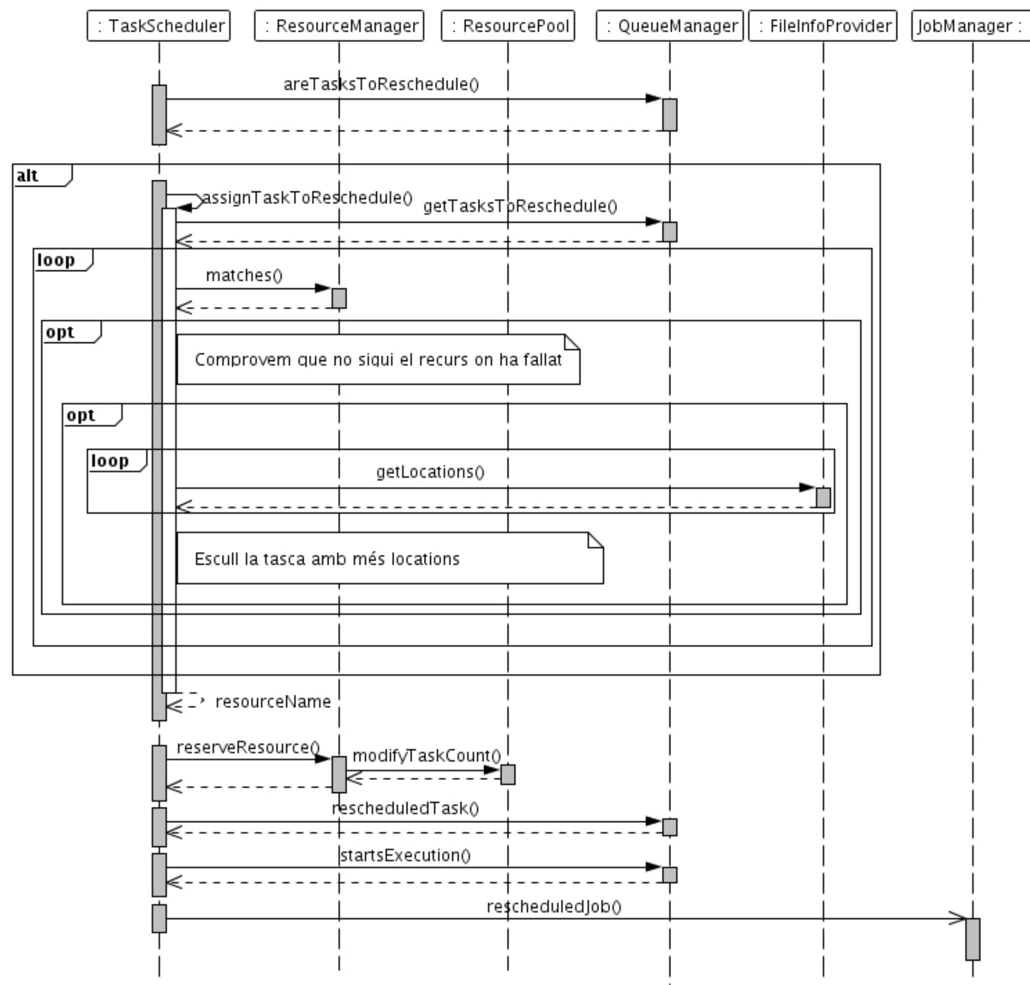

Figura 5.7: Diagrama de seqüència de l'elecció d'una tasca a re-planificar al acabar una tasca

tasca del slot que tingui un temps de major. D'aquesta manera s'aconsegueix un millor balanceig de la càrrega entre slots.

La última opció que hi ha, si tampoc hi ha cap tasca esperant a la màquina, és agafar una tasca que estigui esperant en un altre node. Per tal de reduïr el ventall de recursos d'on es pot agafar la tasca el primer que es fa es escollir únicament aquells recursos que poden executar mètodes comuns, ja que la resta de tasques no les pot executar. Així com fins ara l'objectiu de tota la planificació ha estat balancejar la càrrega el màxim possible entre els diferents nodes i slots. Ja que en aquest cas només es busca una tasca, és preferible escollir la opció que ens doni millor rendiment, això queda traduït en buscar la tasca amb millor localitat de dades. De totes maneres, la millor tasca no és aquella que té més dades en el node, sinó també la que té menys dades en el node on havia estat planificada anteriorment. Totes les tasques dels altres nodes que puguin ser executades en el node seràn avaluades. Per cada fitxer que hagi de ser passat com a paràmetre que es trobi en el recurs, la tasca sumarà 2 punts, mentre que si el fitxer es troba al node origen es restarà 1 punt. En cas de trobar-se tant a l'origen com al recurs, sumarà només 1 punt. La tasca que obtingui una puntuació més alta serà l'escollida per canviar de node i executar-se en slot que ha quedat lliure.

5.4.4 Una tasca falla l'execució

De la mateixa manera que el JobManager rep l'avís conforme la tasca ha acabat correctament, pot rebre'n un que digui que la tasca ha fallat. Com que el JobManager no ha canviat, es farà el mateix procediment que en la versió per Grid de COMPSs, primer s'intenta tornar a enviar la tasca contra el mateix node i, si torna a fallar, s'avisa al TaskScheduler que hi ha hagut un error i que s'ha de reschedular la tasca.

El primer que fa el TaskScheduler és mirar si aquella tasca ja havia fallat un altre cop. Si és així, s'avisa a tot el runtime de que l'execució té un error que no pot evitar-se i s'atura l'execució, tal i com es feia en la versió de COMPSs pel Grid. De la mateixa manera, si és la primera vegada que apareix un error d'aquella tasca s'intenta assignar-la a un altre recurs.

Igual que en el cas de finalitzar la tasca correctament, aquí és on tot el procés canvia degut a que tenim noves estructures de dades. El primer que ha de fer el TaskScheduler és esborrar totes les dades que té referents al Job que ha fallat com si la tasca hagués acabat d'executar-se correctament, però sense avisar al TaskAnalyser que la tasca ja ha acabat. És a dir, que s'allibera el slot del QueueManager que estava ocupant, s'indica al ResourceManager que el recurs pot acceptar una nova tasca, i es busca la millor tasca que pot executar-s'hi tal com s'ha explicat en l'apartat 5.4.2.

Per altra banda, queda veure que es fa amb la tasca que ha fallat i ha de tornar a executar-se. El TaskScheduler demana a ResourceManager quins són els recursos que poden executar-la en aquell moment. És probable que el recurs on ha fallat estigui en aquesta llista, en cas de ser-hi, s'ha de treure per evitar tornar a enviar-la al mateix recurs. En aquest moment, poden donar-se dues situacions diferents: queden altres recursos a la llista o no hi ha cap recurs on es pugui executar.

En el cas de que hi hagi algun recurs disponible, es tracta el cas com si fos una nova tasca: s'assigna el recurs que tingui més fitxers, en qualsevol dels slots lliures. S'aplicaran els mateixos canvis en el ResourceManager i el QueueManager, la diferència serà que a l'hora d'enviar cap al JobManager s'indicar que la tasca ja havia estat planificada anteriorment.

En el cas de que no es disposi de recursos, la tasca no s'executa i es guarda en la cua tasksToReschedule del QueueManager. Tal i com s'ha explicat a l'inici de l'apartat 5.4, les tasques que estàn esperant per ser reschedulades són prioritàries respecte qualsevol altra tasca que pugui ser executada en el node. Així desapareix el cas en que una tasca vella no està aturada molt temps, ja que serà recuperada pel primer recurs que acabi una tasca i pugui executar-la. La figura 5.8 mostra el procés.

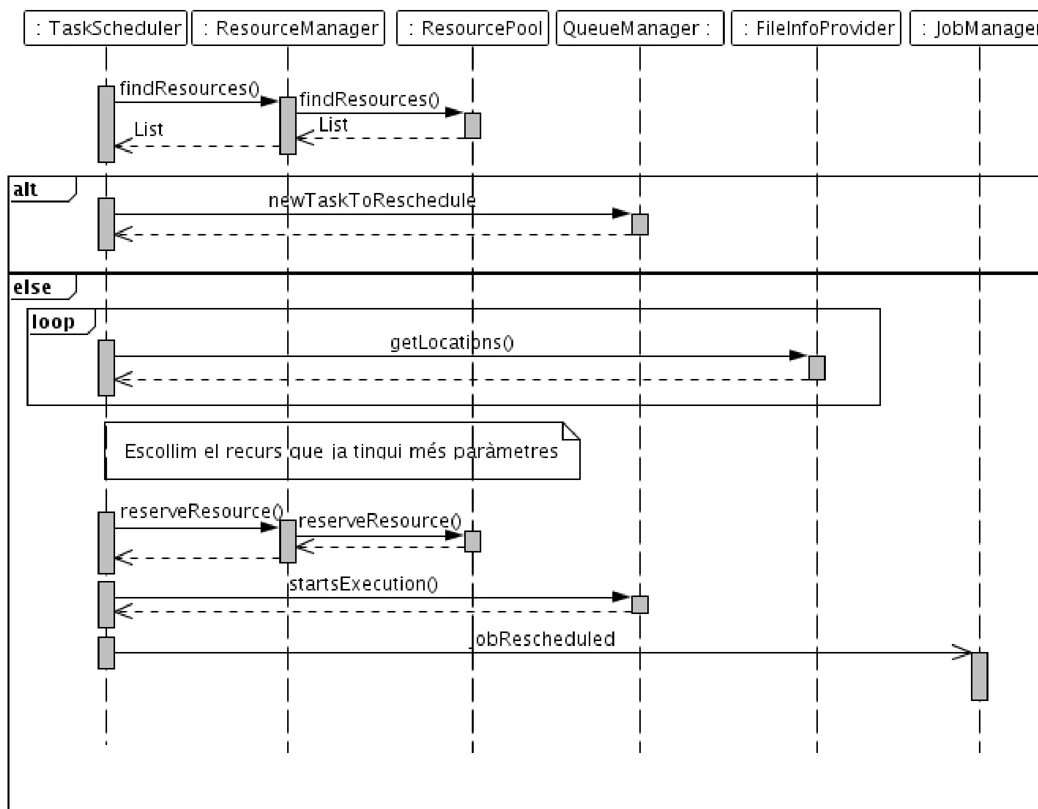

Figura 5.8: Diagrama de seqüència al rebre una notificació de tasca fallida

5.5 Procés de creació i destrucció de màquines

En aquest apartat, es dóna una visió global del procés de creació d'una màquina i del de destrucció. Tot els detalls del procés depenen de la política que dóna l'ordre de crear o destruir una màquina. L'apartat pretén fer veure quins són els problemes que es plantegen a l'hora d'encendre la màquina i apagar-la i explicar la solució per la que s'opta per cada un d'aquests problemes. S'explica en quin ordre s'han de comunicar les diferents estructures i components per tal de que tots tinguin les dades que han de ser consultades en el moment que toca.

Tota ordre de crear i destruir una màquina en el Cloud s'inicia en el TaskScheduler i és resultat de l'aplicació d'alguna de les polítiques; tot comença quan aquest indica al ResourceManager que les apliqui. En els apartats 5.6 i 5.7 s'explica quines són les diferents polítiques, en quins moments poden aplicar-se i quin dels problemes anteriors s'hauran de solucionar.

5.5.1 Procés de creació

Quan el ResourceManager decideix que és el moment de demanar una nova màquina, mira quines tasques pot executar la màquina que està demanant i es guarda que té pendent la creació d'una serië de recursos per poder executar-les. El segon pas és enviar la petició al connector amb les característiques corresponents de la màquina, la imatge que ha de fer servir i un nom qualsevol per identificar la màquina virtual. Com ja s'ha explicat el

connector crea un nou thread que gestionarà la petició i el control torna al TaskScheduler perquè es faci el que tenia que fer.

Quan la màquina ja està llesta per executar tasques, s'avisa al TaskScheduler invocant la funció addCloudNode i s'indica quines seran les característiques que haurà de tenir i quins paràmetres s'hauran de fer servir per connectar-s'hi. Aquesta funció és la que s'encarrega de gestionar tota la creació en les dues estructures. El primer que es fa és comprovar si la màquina és útil (si connector li demana que ho comprovi). Si la màquina no és útil avisa al Connector i aquest la destrueix invocant el mètode refuseCloudWorker-Request del TaskScheduler per indicar-li que aquella màquina no serà creada i que es netegin totes les dades que es tenen fent referència a aquella màquina, tal i com es mostra a la figura 5.9

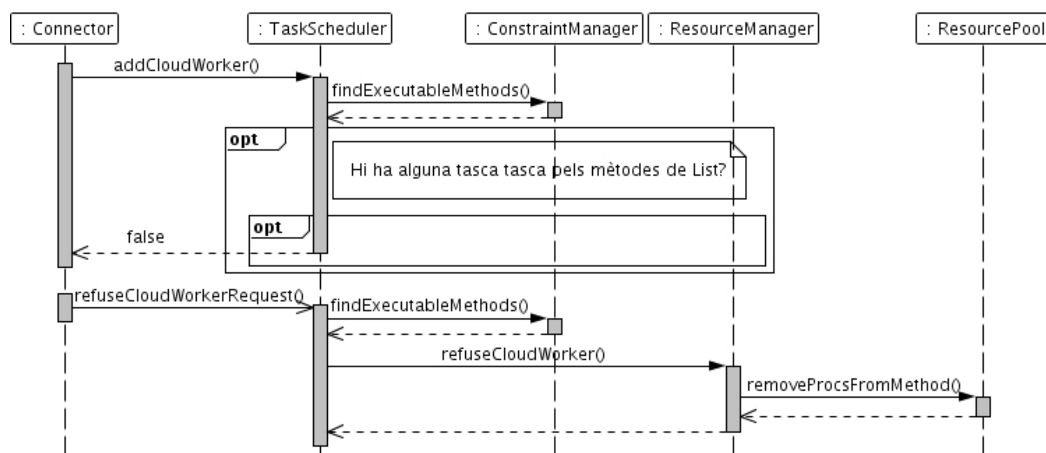

Figura 5.9: Diagrama de seqüència al final de la creació d'una màquina que ja no serà útil

Si la màquina és útil o el connector no creu que s'hagi de fer la comprovació, el procés continuarà i el primer que es s'ha de fer és avisar al ResourceManager que el recurs que esperava ja s'ha creat mitjançant la crida confirmCloudWorker. En aquesta s'indicarà també quines característiques el ResourceManager esperava tenir de la màquina, què li ha retornat el Cloud finalment i quins són els límits de càrrega que pot suportar la màquina. El ResourceManager esborrarà totes les dades referents a la màquina que havia demanat i es guarda en el ResourcePool tota la informació sobre la màquina concedida, incloent-hi les relacions amb els tipus de tasques que pot executar.

La segona cosa que cal fer per fer ús de la màquina serà actualitzar el ProjectManager. S'han d'afegir les característiques del nou worker que ha donat el connector. Al capítol 4 es diu que els components no tenen cap objecte passiu compartit i que el runtime pot estar distribuït en diferents nodes. Una de les conseqüències que té és que el ProjectManager no queda compartit, per tant, quan se'n canvia una instància, s'ha d'avisar a tots els altres components de que facin el mateix canvi. Existeixen tres ProjectManager: el primer és el de TaskScheduler que serveix per saber la càrrega màxima que es vol a cada node. El segon està al JobManager i es fa servir per tenir les dades necessàries per poder enviar una tasca per JavaGAT. El tercer es troba al FileTransferManager, que també servirà els paràmetres que necessita JavaGAT, però aquest cop per enviar els fitxers. Així quan es modifiqui el ProjectManager del TaskScheduler, aquest avisa al JobManager que el canvii a través de la interfície JobCreation que també fa servir per enviar-li les tasques. Quan el JobCreation tracta la petició fa els canvis i afegeix el worker amb les dades que el connector

li ha donat al TaskScheduler.

Finalment, el TaskScheduler envia la mateixa petició cap al FileTransferManager mitjançant la interfície SafeTransfer, quan aquest tracta la petició, actualitza les dades del seu ProjectManager.

S'ha de tenir en compte que ProActive fa que aquestes crides siguin asíncrones, com que no produeixen cap dada mai es queda esperant a que aquestes acabin. Per tant, no hi ha cap risc de deadlock ni pèrdua de rendiment degut a les comunicacions per xarxa ni a la cua de peticions que puguin tenir els components.

La figura 5.10 mostra el diagrama de Seqüència d'aquesta segona part.

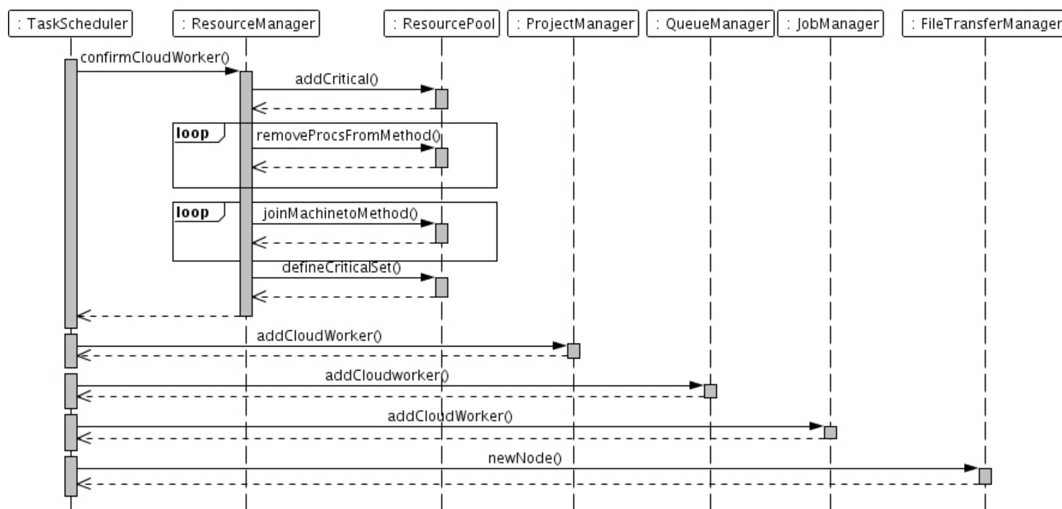

Figura 5.10: Diagrama de seqüència al final de la creació d'una màquina que serà útil

L'últim pas que s'ha de fer és intentar donar-li tantes feines com es puguin i accepti el node. El procés d'assignació de tasques és el mateix que quan un node ha acabat d'executar una tasca. Per fer-ho, actua com si hagués acabat una tasca: primer intenta agafar les tasques per reschedular. Si quan ja se li han assignat totes les tasques a reschedular possibles encara pot executar més tasques de forma simultània, les agafa d'altres nodes de la mateixa manera que s'ha explicat a l'apartat 5.4.3.

Si encara queden més tasques esperant en les altres màquines, però la nova màquina ja no en pot executar més, s'intenta fer un balanceig de la càrrega acumulada en les altres màquines omplint les cues dels slots de la nova amb tasques de les altres. L'objectiu és omplir les cues de la nova màquina; tot i que, sempre vigilant que la nova màquina no passi a ser la més carregada i que en cap cas superi el 50% de la càrrega assumible en el temps de crear una nova màquina. El criteri per decidir quina serà la tasca escollida és simple: s'agafa la tasca que porti més temps a la cua del slot amb més càrrega del node més carregat.

5.5.2 Procés de destrucció

L'altre possibilitat de canviar els workers dels que disposa el runtime és l'alliberament d'un recurs. Així com encendre una màquina és un procés senzill, que només cal afegir la màquina a les estructures quan aquesta ja està llesta per executar tasques, la destrucció

comporta més problemes. Cal tenir en compte que el fet d'alliberar un recurs vol dir perdre l'accés al node i, com a conseqüència de no poder accedir al node, no es pot accedir a la informació que contenia localment.

De la mateixa manera que en el procés de creació d'un nou recurs, existeixen dues fases: demanar el recurs que necessita el runtime i afegir-lo a les estructures; també n'existeixen dues pel procés de destrucció: salvar la informació i apagar-lo. Un cop el ResourceManager decideix que un recurs ha d'eliminar-se ha d'impedir que s'hi enviin noves tasques i donar l'ordre de salvar la informació. Per fer-ho, inicia còpies de tots els fitxers que no es troben en cap altre recurs i encara poden ser utilitzats cap algun altre recurs, per tal que qualsevol fitxer pugui ser llegit sense dependre del node. Quan ja s'han copiat tots els fitxers i estàn accessibles a través d'un altre node, la màquina pot ser apagada.

L'acció torna a començar en el TaskScheduler. Hi ha dues possibilitats d'aplicar una política per apagar les màquines: buscar qualsevol màquina que pugui ser apagada o mirar si una màquina concreta pot ser apagada. Igual que en el cas de la creació de màquines, les polítiques queden descrites més endavant, en l'apartat 5.7. Quan es decideix que una màquina ha de ser apagada, es dóna l'ordre al ResourcePool de que aquella màquina no pot rebre més tasques. Les funcionalitats del ResourcePool són informar al ResourceManager de quines màquines podran executar una tasca i ajudar-lo a prendre decisions de les polítiques. Per tant, en eliminar una màquina del ResourcePool fa que aquesta ja no entri en cap tipus de política, i per tant, no se li pot assignar cap nova tasca.

D'aquesta manera, el ResourceManager ja ha bloquejat l'entrada de noves tasques. Per poder apagar la màquina cal que s'acabin totes les tasques que la màquina té assignades: les que estàn executant-se i les que esperen poder executar en la cua d'un dels seus slots. Existeixen dues possibilitats a l'hora de decidir que fer amb aquestes tasques: poden ser transferides a una altra màquina o es pot esparar a que aquestes acabin abans d'apagar la màquina.

Per les tasques que estan esperant s'opta per migrar-les a un altre node, d'aquesta manera la decisió de la política tarda menys a fer-se efectiva. Per altra banda, per les tasques que ja han començat a executar, l'opció escollida és l'altra: esperar a que acabin les tasques; d'aquesta manera, no es perd el temps que la màquina ha estat calculant aquella tasca que després hauria de tornar a començar a calcular-se en un altre node. El TaskScheduler es guarda que està pendent de que s'acabin tasques de la màquina per apagar-la.

Quan detecti que la màquina no està executant cap tasca, tots els resultats que estava calculant han estat generats; el que s'ha de fer és salvar la informació que només es pugui trobar la màquina. Per salvar la informació se'ns obren varies possibilitats amb els seus avantatges i inconvenients:

- enviar les dades al master: ofereix seguretat. Si el master falla, falla tot el runtime. Per tant, un error en el master és fatal per l'aplicació i la falta de dades no seria el motiu d'error de l'aplicació. Aquest mètode té dos inconvenients. El primer és que el master no soluciona l'aplicació, per tant, els fitxers temporals de l'aplicació no serveixen per res en el master. Aquest els haurà de transferir per que es pugui operar amb ells. El segon motiu és que si tots els fitxers es còpien en el master, aquest passa a ser un possible coll d'ampolla degut a les transferències.
- enviar totes les dades a un sol worker: ofereix un petit avantatge respecte abans, almenys el destinatari podrà executar amb les dades que tingui. El problema que hi

ha és que aquest node segueix sent un coll d'ampolla.

- enviar les dades a workers aleatoris: desapareix el problema del coll d'ampolla, però es manté un problema, que la probabilitat de que les tasques que s'executin en el node no facin servir els fitxers és alta.

- enviar les dades a workers en funció del seus lectors: Mantenim la solució al coll d'ampolla i si la màquina escollida per guardar les dades, és una màquina capaç d'executar les tasques que llegiran el fitxer, és molt més probable que aquest vagi a parar a un node que el faci servir.

La solució escollida és la quarta. De totes maneres, un problema que pot aparèixer escollint qualsevol dels workers: que el worker que rep el fitxer també sigui apagat. En aquest cas, el fitxer ha de tornar a ser copiat en un altre node, havent fet una transferència inútil. Una possible solució a aquest problema és enviar les dades a un worker que estigui en el conjunt de màquines crítiques o físiques.

L'encarregat de transferir els fitxers és el FileTransferManager, per tant, el TaskScheduler farà la petició de salvar les dades a través de la interfície FileTransfer invocant el mètode transferStopFiles i indicant quin node es vol apagar i, per cada mètodes remot, quin worker pertanyent al conjunt crític ha de rebre les dades. A partir d'aquí el TaskScheduler pot seguir tractant peticions ja que la crida és asíncrona, i així planificar altres tasques. Tota aquesta primera part del procés pot seguir-se a la figura 5.11

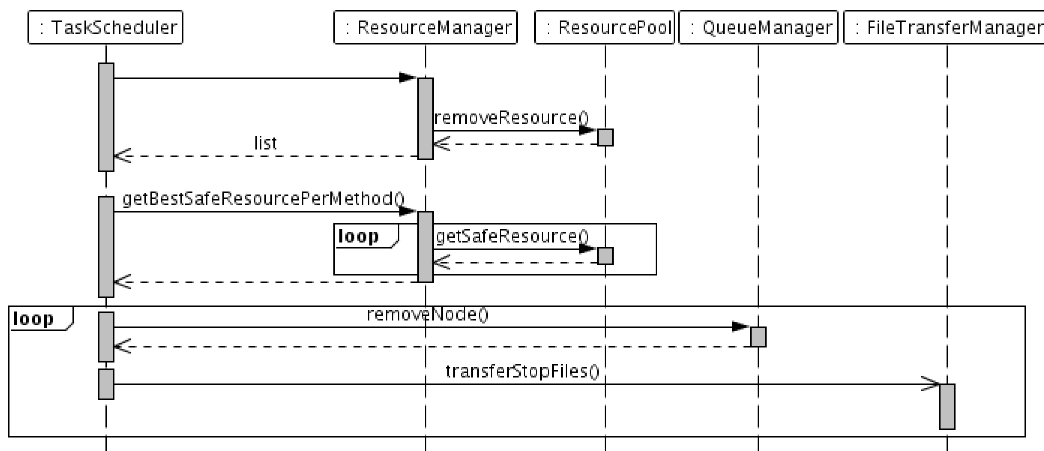

Figura 5.11: Diagrama de seqüència de manar la destrucció d'una màquina (TaskScheduler)

Per la part del FileTransferManager, queda tota la feina per fer. El primer que fa és preguntar al FileInfoProvider quins fitxers conté el node i quins són els seus mètodes lectors. Com que la màquina no pot rebre més tasques, tota la informació referent al node que s'ha d'apagar ja pot ser eliminada, és a dir, les ubicacions de tots els fitxers en la màquina. A partir de la llista de fitxers, FileTransferManager mira quins d'aquests fitxers no es troben únicament al node a eliminar. Si el fitxer ja es troba en un altre node (fins i tot al master), aquest no cal salvar-lo i n'elimina totes les ubicacions referents a la màquina que es vol apagar per evitar que es faci com una font d'aquell fitxer i s'iniciin transferència amb ell com a origen, cosa que no deixaria eliminar la màquina fins que acabés.

Per cada fitxer que hagi de ser salvat s'eliminen totes les ubicacions conegudes, ja que totes són del node, es miren els comptadors dels mètodes lectors i es comença a fer una

còpia del fitxer des de la màquina al recurs que el TaskScheduler ha indicat pel mètode amb més lectors, en cas d'empat el que tingui un identificador més baix. Quan tots fitxers que hagin de ser salvats s'hagin copiat es pot donar l'ordre d'apagar la màquina al TaskScheduler.

És possible que algun dels fitxers que s'han marcat com a necessaris per apagar la màquina ja estiguin a mig ser transferits, abans d'apagar la màquina s'haurà d'esperar a que les transferències que els involucren acabin. També pot donar-se el cas que, tot i haver evitat començar noves transferències, existeixin transferències en procés que poden tenir com a font el node tot i ja tenir rèpliques del fitxer. En aquest cas, la màquina no pot ser aturada ja que la transferència podria ser aturada i haver de tornar a començar o donar-nos un error de transferència.

La manera que hi ha de controlar quines còpies s'han d'acabar abans d'apagar la màquina és tenir un comptador. En l'anterior versió de COMPSs a cada conjunt d'operacions, per exemple: totes les transferències necessàries per començar una tasca s'agrupen en un grup. Aquests grups són un comptador del nombre d'operacions que han d'acabar abans d'indicar que el conjunt de transferències ordenades ja han acabat. Aquesta idea s'extèn a la nova versió, per tal d'avisar que una màquina pot ser apagada. El comptador s'incrementa en 1 per cada còpia en procés que pugui involucrar al node i per cada fitxer que contingui i sigui necessari fer-ne una còpia de seguretat. Cada cop que acabi una operació, s'avisa a tots els grups que estàn pendents d'ella i decrementen en 1 el comptador. Si aquest arriba a ser 0 ja pot indicar-se que ha acabat el conjunt d'operacions necessari i que pot realitzar-se la tasca o apagar-se la màquina.

Abans de poder explicar com canvia el procés de còpia, és necessari que s'entengui com està fet el FileTransferManager de la versió per Grid de COMPSs. Cada cop que un es vol fer alguna operació sobre un fitxer es genera una instància de tipus FileOperation. Un FileOperation pot ser de tipus Copy, si es vol fer una còpia, o Delete, si es vol eliminar un fitxer. Independentment de l'acció que es vol portar a terme, totes aquestes instàncies s'encuen una cua de peticions (RequestQueue), anomenada queue, esperant per ser tractats.

FileTransferManager disposa d'un RequestDispatcher, TransferDispatcher, format per una RequestQueue (la mateixa queue) i un conjunt de Threads (per defecte 5) que operen tots de la mateixa manera: intenten desencuar un FileOperation de queue, processen l'operació (processRequest) i avisen a qui fos necessari de que l'operació ha acabat correctament (checkNotifications) o que ha tingut un error (notifyFailure). A més a més, durant la petició es vigilen coses com que la còpia demanada no estigui ja feta o fent-se en aquell moment. Per no tenir tots els Threads encessos tota l'estona preguntant per la queue, si aquesta estava buida el que es fa és que els threads que no processen peticions estiguin adormits i se'n deperta un cada cop que s'encués una nova petició.

En la versió de COMPSs per Cloud, tot això canvia una mica. Les diferències es poden veure a la figura 5.12. La queue passa a anomenar-se copyQueue i, per tal de suportar aquestes còpies especials per apagar la màquina, hi ha un segon RequestDispatcher, SafeTransferDispatcher. Aquest té el seu propi ThreadPool (per defecte de tamany 1) i una cua anomenada safeQueue, en aquesta s'encuen totes les còpies que s'han de fer per apagar qualsevol màquina.

La decisió de tenir dos RequestDispatchers diferents té dos motius: evitar llargs temps de resposta i que han d'actuar diferent si la còpia és normal que si la còpia és per salvar un fitxer.

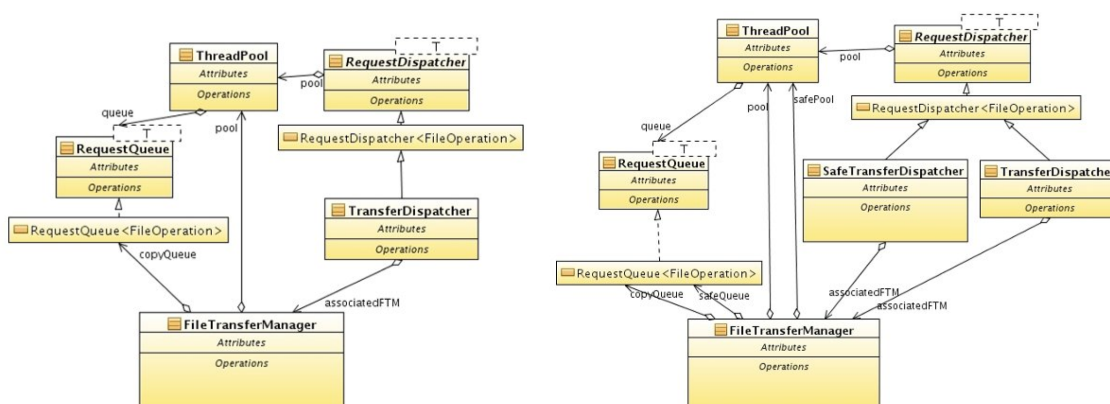

Figura 5.12: Diagrames de classe (parcials) del FileTransferManager de les dues versions de COMPSs

El primer motiu és fàcil d'entendre, en el cas de tenir una màquina que ha realitzat moltes tasques petites i té molts fitxers temporals de manera exclusiva, per exemple a la incialització de l'aplicació. Els altres workers hauran de llegir els fitxers d'aquesta màquina. Si només hi hagués 1 sol RequestDispatcher podria donar-se el cas que en un moment donat tots els fitxers de la màquina estiguessin copiant-se en una altra màquina per tal de poder apagar-la. El fet d'estar copiant aquests fitxers impediria que es fessin transferències per executar tasques en altres màquines i baixaria el rendiment del runtime.

El mateix pot passar per l'altra banda, a l'hora de donar l'ordre hi ha demanades moltes peticions per copiar fitxers per executar tasques i es tarda molt a poder fer aquestes còpies de seguretat. Com que no es fan aquestes còpies es tarda molt a poder apagar una màquina que no s'aprofita, i per tant, es gasten diners. El fet de tenir dos RequestDispatchers permet evitar casos com aquests.

Com s'ha dit, el segon motiu té a veure amb el tractament de les còpies i de les còpies de seguretat. Bàsicament només hi ha una diferència: per les còpies de seguretat es pot assumir que existeix alguna font del fitxer i que és el node que es vol apagar. Per les còpies de lectura no es pot assegurar que hi hagi cap ubicació on es pugui trobar el fitxer. A l'inici del procés, s'han eliminat les ubicacions que permetien copiar des del node que es vol apagar. Això fa que el fitxer pogués no tenir fonts disponibles en aquell moment mentre se n'està fent una còpia de seguretat.

El procés per fer una còpia per lectura d'un fitxer comença comprovant que la còpia que es vol fer del fitxer no estigui en procés. En cas d'estar-ho, s'ignora la petició, però es guarda a la còpia en procés que quan acabi s'avisi també al grup de la còpia descartada com si s'hagués acabat la còpia desitjada. En cas contrari, guarda que la còpia està en progrés.

Com s'ha dit, molts Threads poden estar accedint a la vegada sobre les estructures que guarden les ubicacions dels fitxers (LogicalFile de JavaGAT), afegint i treient noves fonts. Per evitar que algun altre thread elimini les fonts que s'estan fent servir per fer la còpia, enlloc de treballar directament sobre el LogicalFile, s'utilitzarà una còpia amb les ubicacions reals en el moment de començar la còpia. Si han eliminat totes les fonts del fitxer, aquest no pot ser copiat, per tant, haurà d'esperar-se a que aparegui una nova font del fitxer. Es guardarà la còpia esperant trobar alguna nova font a l'estructura fileToCopy

i el thread abandonarà la còpia indicant que està esperant. Si existeixen ubicacions des d'on es pot copiar s'inicia la còpia i el thread la monitoritza.

Quan la còpia acaba, el primer que es fa és reactivar totes aquelles còpies que estiguessin pendents de que aparegués una nova font del fitxer, és a dir, les que estaven esperant a fileToCopy. Totes aquestes còpies són còpies que ja havien començat a fer-se i, per tant, són prioritàries per sobre de les que estan esperant a la cua, s'afegeixen al principi de la cua. També es comprova que no hi hagi cap màquina que estigués esperant aquella còpia per apagar-se.

L'últim que s'ha de fer és afegir la nova ubicació i donar la còpia per acabada correctament.

Aquest és el procés que segueix qualsevol còpia de lectura, però falta explicar que passa amb les còpies que ha manat explícitament el FileTransferManager per poder apagar la màquina.

En el cas de les còpies ordenades explícitament per poder apagar la màquina, el procés és molt més senzill ja que es parteix de tres fets importants:

- el fitxer té una única font: el node que es vol apagar.
- no hi ha cap altra còpia que estigui en procés amb el mateix fitxer origen ni destí.
- les ubicacions no poden canviar (No hi ha cap còpia en progrés i l'ordre d'apagar la màquina només es dóna un cop)

Això fa que no calguin moltes de les anteriors comprovacions, el procés comença directament guardant que la còpia està en procés i directament la comença a fer-la a través de JavaGAT.

Quan la còpia acaba, es mira a l'estructura fileToCopy si existeix alguna còpia que estigués esperant una font pel fitxer que s'ha copiat. Si és així, torna al principi de la copyQueue perque pugui començar el més aviat possible i també es mira que no fos la última còpia del grup per apagar la màquina. Finalment indicarà que la còpia ha acabat correctament.

Independentment de si era una còpia per lectura o era una explícitament creada per poder apagar la màquina, si era la última còpia d'un grup per apagar la màquina, el FTM avisarà al TaskScheduler que ja pot apagar-la. Després d'això, continua el seu procés normalment, avisa a qui hagués d'avisar segons el tipus de còpia que fos, per exemple, si hi havia una tasca que depenia d'aquella transferència, avisa també al JobManager de que la còpia ha acabat.

En el moment en que el TaskScheduler rep que tots els fitxers ja tenen una nova localització i que la màquina ja pot apagar-se, s'avisa al ResourceManager de que la petició ha estat completada i que ja pot eliminar la màquina amb tota la informació que contingués. Aquest retransmet la petició al connector que encen un thread que controla que la màquina s'apagui.

L'últim que queda per actualitzar són les dades de connexió i de treball, per això, el TaskScheduler es posa en contacte amb el ProjectManager i elimina el recurs. Igual que en la creació, quan es destrueix un recurs és necessari que la petició es retransmeti a JobManager i FileTransferManager.

5.6 Demanar nous recursos al Cloud

A l'iniciar l'execució, l'usuari pot determinar un conjunt de recursos físics en els que COMPSs pot delegar l'execució de les tasques. A més a més, d'aquests recursos inicials el runtime pot demanar recursos nous al Cloud tant al moment d'iniciar l'execució com al llarg d'aquesta.

Aquest apartat descriu quines polítiques segueix el runtime. Aquest pot demanar recursos a l'inici i durant l'execució per anar adaptant els recursos disponibles a les necessitats de l'aplicació.

5.6.1 Política de recursos inicials

Quan s'inicia l'aplicació, el runtime pot disposar d'un conjunt de recursos físics on executar les tasques o no tenir-ne cap. L'usuari pot conèixer l'aplicació i pot saber el seu nivell de paral·lelisme. Per tant, pot fer-se una idea de quina quantitat de màquines virtuals és òptima per la fase inicial de l'aplicació. Per això, es dóna l'opció a l'usuari d'indicar quin nombre de màquines virtuals vol que s'encenguin automàticament a l'inici de l'execució, sense esperar a que hi hagi cap petició. Així, si l'aplicació tarda una estona a començar a enviar tasques als workers, durant tot aquest temps pot solapar-se la computació inicial en el master i la creació d'aquestes màquines virtuals. Així doncs, s'afegeix un sisè paràmetre d'execució (ja existien el connector, el nom del servidor, el port i la imatge que es vol fer servir al crear les màquines i el límit en la creació de màquines): la quantitat de màquines que l'usuari vol a l'arrencar l'aplicació.

El principal problema que hi ha és que el runtime sap quantes màquines vol l'usuari, però no sap quines característiques han de tenir. De totes maneres, pot especular quin tipus de màquines necessitarà a partir de les restriccions dels mètodes i pot repartir la quantitat de la forma que sembli més convenient.

Un altre problema amb el que es pot trobar el runtime és que un conjunt de tasques no pugui executar-se en cap dels recursos físics que té. En la versió original de COMPSs, l'aplicació no acabaria i es quedaria esperant a que s'alliberés un recurs que no existeix. L'avantatge que ofereix el Cloud és que pot demanar recursos que compleixin les restriccions per tal de solucionar el problema.

Així doncs, al començar l'aplicació el runtime ha prendre una decisió en funció de tres paràmetres:

- les màquines físiques que té disponibles
- les restriccions dels mètodes
- el nombre de màquines que vol l'usuari.

La prioritat a l'hora de crear recursos inicial és que totes les tasques puguin executar-se en algun recurs. Encara que això signifiqui crear més màquines de les que ha indicat l'usuari, mai menys.

El primer que s'ha de fer és comprovar que les restriccions dels mètodes i que puguin complir-se amb els recursos físics que el runtime té disponibles. Si n'hi ha alguna que no pugui complir-se, ha de demanar una màquina que ho sigui. De totes maneres, això no vol dir que per cada tipus de tasca que no pugui ser executada es creï un recurs. La decisió

intenta respectar al màxim el número de màquines indicat per l'usuari, per tant, intenta crear una màquina que satisfaci les restriccions de més d'un mètode a la vegada, s'ajunten de manera que s'aprofitin al màxim els recursos. Per exemple, si els següents mètodes:

- a: demana 5 CPUs i 1GB de memòria física.
- b: demana 3 CPUs i 1GB de memòria física.
- c: demana 2 CPUs i 2 GB de memòria física.

El mètode b requereix una màquina més petita que l'a. Per tant totes les tasques de tipus b podran executar-se en una màquina que satisfaci a. En canvi, c no pot executar-se en la mateixa màquina que a. Per tant, el runtime hauria de crear dues màquines.

Pot ser que el fet de crear dues màquines faci que se superi el número de màquines que vol l'usuari. En aquest cas el que es demana ajuntar les màquines que siguin més similars entre elles fins obtenir el número de màquines que vol l'usuari o no es pugui ajuntar-ne més. En aquest cas, es demanaria una sola màquina de 5 CPUs i 2 GB de memòria física.

Hi ha dos restriccions que no permeten ajuntar recursos:

- el sistema operatiu.
- l'arquitectura del processador.

Si l'usuari vol més màquines de les que s'han creat, la política a seguir és crear màquines capaces d'executar el màxim de mètodes diferents sense necessitat de crear màquines especials, com la de 5 CPUs i 2 GB de memòria del és el cas anterior.

En el moment de decidir quines màquines es creen, el runtime no té cap tipus d'informació sobre quines seran les necessitats de l'aplicació. Aquesta política intenta distribuir les quantitats de màquines creades d'una forma equitativa tenint en compte el nombre de tasques que aniran a parar en aquella màquina. Això provoca que a l'iniciar l'aplicació qualsevol tasca pugui ser executada en algun dels nodes i possiblement pugui fer-ho en més d'un si les seves restriccions no són molt fortes comparades amb les de la resta de mètodes.

El problema pot sorgir quan a l'inici hi ha molt paral·lelisme i es poden llençar moltes tasques molt restrictives de cop. El més probable és que només es disposi d'una sola màquina capaç d'acceptar aquestes tasques i, per tant, el número de recursos disponibles acabaria fent reduir el rendiment d'aquesta aplicació.

Descòneixer l'aplicació porta al runtime a haver de seguir una política especulativa, que comporta una probabilitat molt alta de que l'elecció de recursos inicial no sigui òptima.

5.6.2 Implementació de la política de recursos inicials

Tal com diu l'apartat anterior, aquesta política només es té en compte en un moment molt concret de l'execució. El moment d'iniciar l'aplicació, abans de començar a rebre tasques. De fet la política s'invoca un únic cop durant la inicialització del component TaskScheduler. En aquesta inicialització s'inicialitzen tots els objectes que necessita el TaskScheduler: el ResourceManager, el ProjectManager, el ConstraintManager (llegeix la interfície anotada que ha programat l'usuari per donar al TaskScheduler totes les restriccions dels mètodes)

i altres estructures internes que mantenen informació sobre l'estat de les tasques que han de ser planificades o ja ho han estat. Un cop totes aquestes estructures han estat inicialitzades amb els valors corresponents, el ProjectManager i ResourceManager (també el ResourcePool) tenen totes les informacions sobre els recursos físics ja que s'han llegit dels respectius XMLs.

Amb tota aquesta informació, ja n'hi ha prou per poder aplicar la política que s'ha descrit en l'apartat anterior. L'aplicació de la política es fa en dues fases: la fase en que es creen els nodes necessaris per poder executar l'aplicació, encara que això signifiqui sobrepassar el límit de l'usuari; l'objectiu és que cada tasca tingui una màquina on executar-se. La segona fase es fa només si encara s'han de crear més màquines per tal d'arribar al número indicat.

El primer que fa el TaskScheduler és demanar al ConstraintManager, les restriccions de tots els mètodes que hi ha. I, directament, delega en el ResourceManager la creació de les màquines necessaries tenint en compte que alguns mètodes ja tenen màquines físiques on executar-se, per això invoca el mètode addBasicNodes. Aquesta funció el primer que fa és de tots el mètodes, en filtra aquells que ja tenen alguna màquina física on executar-se. Així evita crear màquines per mètodes que ja tenen una màquina on executar-se.

Ara que només queden els mètodes que necessiten alguna màquina virtual el següent pas és escollir-ne un nombre suficientment gran com per no haver de demanar màquines amb molts recursos, però intentant agrupar els requeriments per no crear moltes màquines similars i tenir tasques que podrien córrer en varies màquines. Els dos factors que fan que els requisits de dos mètodes siguin impossible d'ajuntar són: arquitectura i Sistema Operatiu. Es crea una llista per cada arquitectura i sistema operatiu, mantenint com una arquitectura i un sistema operatiu apart els que no tenen cap restricció d'aquests tipus.

El segon pas que cal fer és escollir les màquines que permeten executar diferents tasques. En el cas de que el conjunt de mètodes que pot executar una màquina sigui un subconjunt dels d'una altra, la primera no és necessari crear-la. D'aquesta manera es redueix el nombre de màquines a crear.

De totes maneres, l'usuari pot haver especificat un nombre de màquines més baix que el nombre d'arquitectures-sistemes operatius que han quedat definides. En aquest cas el que es fa és assignar un sistema operatiu o una arquitectura arbitrària a les màquines sense cap restricció d'aquest tipus. Les eleccions del sistema es fan en funció del nombre de processadors, memòria física i espai de disc que requereixen, en aquest ordre. La més pròxima és la que s'escollirà.

El fet d'assignar aquestes màquines fa possible que es pugui tornar a ajuntar algunes de les restriccions amb subconjunts menors.

Ara sí que ja no es pot reduir més el nombre de possibles combinacions sistema operatiu-arquitectura. Es reparteix el número de màquines entre les diferents combinacions. Cada combinació ha de tenir almenys una màquina i com a màxim el número de màquines del tipus que hagin sobreviscut als diferents filtres que han anat passant. La repartició es fa en funció de la relació #mètodes que requereixen la combinació / # màquines per la combinació. La que tingui aquesta relació amb un valor més alt, obté el dret de demanar una nova màquina. Es reparteixen totes les màquines que hagi especificat l'usuari. Amb aquesta relació, el número de màquines que pot crear de la combinació és proporcional al nombre de mètodes de la combinació arquitectura - sistema operatiu.

Ara ja només queda passar un últim filtre, han tocat prou màquines virtuals a l'arqui-

tectura per satisfer els tipus de m ‘aquina que han sobreviscut? Si és així es demana una màquina per cada un. En cas contrari, s’han d’ajuntar tipus fins arribar a tenir prou màquines. Ràpidament es pot pensar que un possible criteri per ajuntar aquestes màquines és el cost que tenen. Així, les màquines A (5 processadors i 1 GB), B(2 processadors i 1GB) i C(1 processadors i 3GB) amb uns preus de 5 per processador i 1 per GB de memòria, tenen un cost de 26, 11 i 4. Les màquines més semblants són la B i la C, per tant s’han d’ajuntar per crear una màquina D (2 processadors i 3 GB) amb un cost de 13, en total 39.

El problema que té aquest sistema és que no es té en compte l’origen de la diferència, així si tenim les màquines A (5 processadors, 1 GB), B(3 processadors , 2 GB) i C(2 processador, 15GB), els costos seràn 26, 17 i 25 respectivament. Ajuntant les màquines A i C i es crea una màquina D (5 processadors, 15 GB) amb un cost de 40, 57 en total. Si s’hagués ajuntat A i B, la màquina D tindria un cost de 27, en total 52. A més a més quan estigui executant tasques de tipus A no utilitzarà 14 GB de memòria i quan executi tasques de tipus C no utilitzarà 4 processadors. Les màquines molt grosses no s’aprofiten.

Així doncs, utilitzar el cost com a element d’unió no dóna cap garantia. Per això el que ha de fer la política és ajuntar les màquines per característiques. Es prioritza l’ús dels processadors, de manera que tots els processadors quedin el màxim ocupats possibles (són l’element més car en el comput de preu) fent que s’ajuntin les màquines amb un número similar. D’aquesta manera el resultat que dóna és el segon.

Tots els tipus de màquina que hagin passat tots filtres es demanaran al connector per tal de crear-les tal i com indicat al apartat 5.5.1.

La segona fase serveix per repartir les màquines sobrants. El procés comença amb totes les tasques (ara totes ja tenen almenys una màquina on poden executar-se) . Per tal d’aprofitar el màxim el nombre de màquines, s’ajuntaran utilitzant els conjunts de tasques tal i com ja feia abans. Per acabar es repartiran totes les màquines que queden en funció de la mateixa relació que abans #mètodes que pot executar el tipus / # màquines assignades al tipus.

5.6.3 Política de demanda periòdica

Només tenint en compte el que s’ha explicat fins ara l’usuari pot disposar com a molt de les màquines físiques que l’usuari hagués indicat a l’inici de l’execució i de les màquines virtuals demanades per l’usuari a l’inici. Amb això l’usuari ja és capaç d’executar les tasques paral·lelament i utilitzant els recursos que ofereix el Cloud. De totes maneres, no n’hi ha prou només amb això.

El principal avantatge que ofereix el Cloud és la possibilitat d’adaptar-se a les necessitats de l’usuari en cada moment. Disposar de recursos físics i demanar inicialment un nombre concret de màquines no s’adapta al que necessita l’usuari en aquell moment tot el que es podria. Per aquest motiu és necessari tenir una política que permeti al runtime demanar més màquines al Cloud Provider, i així ajustar la demanda de màquines a les necessitats en cada moment.

Existeixen dos maneres de poder controlar quines són les necessitats de l’usuari:

- Comprovar les necessitats de l’aplicació periòdicament.
- Comprovar que no es necessitin més recursos cada cop que el volum de feina varia.

Cada una de les maneres té els seus avantatges i els seus inconvenients. La consulta periòdica permet fer un anàlisi detallat del que necessita el runtime en moments concrets. De totes maneres s'obvien les necessitats de l'execució en l'interval entre dues mostres. La solució a aquest problema és utilitzar intervals de temps que no siguin massa grans i permetin veure les necessitats de l'aplicació a temps per poder reaccionar.

Per contra, la comprovació per canvi de volum permet conèixer aquestes necessitats tan aviat com apareixen. Però té el problema que si hi ha canvis massa sovint l'overhead que genera és major que el temps dedicat a planificar les tasques. Per evitar aquest inconvenient és necessari aplicar polítiques molt simples.

Un altre aspecte a tenir en compte a l'hora de decidir quin tipus de política implementarà el runtime és la rapidesa del Cloud Provider en proveir al runtime d'un nou recurs. El període des del moment en que la política marca que es necessita un nou recurs fins que realment el runtime pot fer ús d'aquest és llarg. Això vol dir que per molt aviat que el runtime vegi les necessitats de l'aplicació i prengui les decisions de crear noves màquines, aquestes decisions poden tardar en tenir algun efecte sobre el runtime.

Un altre problema que porta aquest temps de resposta és que pot ser variable en funció del nombre de peticions que ja s'hagi fet sobre el mateix Cloud Provider. La solució per la que opta la implementació del runtime és que la política s'apliqui periòdicament, però que l'interval entre diferents mostres sigui variable en funció del temps que es tardaria a donar resposta a una nova petició.

Ara que ja està explicat quan es prenen les decisions, queda per veure en funció de què i com es prendran. L'objectiu de la política és determinar si l'aplicació obtindria un rendiment millor si tingués una màquina més capaç d'executar un conjunt de mètodes, però sempre tenint en compte que la resposta no serà instantània.

La idea és que la política demani una màquina sempre i quan existeixi alguna una tasca de les que els recursos tenen pendents que s'executaria abans si en aquell moment es demanés una nova màquina que si s'esperés a que s'executés en els recursos actuals. És a dir, que demanarà una màquina capaç d'executar un tipus de tasca (o varios) si la càrrega de les màquines que poden executar un mètode és més gran que la càrrega que poden executar durant el temps de creació d'aquesta nova màquina.

Si només tingués un sol recurs amb 2 slots d'execució capaços d'executar els mètodes 0 i 1. La màquina disposarà de 2*TC[1] segons de CPU abans de que el runtime disposi d'una nova màquina. Si el temps que tarden les tasques en execució a acabar més el temps de les tasques que hi ha en cua per cada slot és més gran que 2*TC, s'acabaria l'execució abans si es disposes d'aquesta nova màquina després de passar TC..

En cas de tenir varies màquines, per exemple: A amb 1 slot per executar els mètodes 0 i 1 al 135% del TC i B amb 1 slot per executar 1 i 2 al 25%. Existeix un sol slot pels mètodes 0 al 135%. El mètode 1 té un slot al 135% i un al 25%, és a dir, que realment està al 80% de la seva capacitat ((1.35+0.25)*TC/2*TC). El mètode 2 té un 25% de la càrrega que pot assumir-se amb la configuració de màquines actual. Si en aquest moment es demanés una màquina capaç d'executar mètodes de tipus 0, quan acabés de crear-se encara hi hauria tasques a la màquina A que pogués executar.

El percentatge del temps de creació ocupat per demanar una nova màquina podria ser un paràmetre a configurar per l'usuari. Un percentatge per sota del 100% implicaria la possibilitat de crear més màquines de les que en realitat necessita l'execució en aquell mo-

[1]TC=temps de creació

ment, per tant, s'enfocaria més la política a tenir un excés de recursos, però incrementant així el nombre de tasques que poden executar-se a la vegada i, per tant, reduint el temps d'execució.

Si per contra el percentatge és superior al 100%, l'enfoc que es dóna és minimitzar el nombre de màquines que utilitzem. Per tant, es reduiria el cost total de l'execució.

5.6.4 Implementació de la política de demanda periòdica

Tota aquesta política s'executa el TaskScheduler, tal com es diu en els apartats anteriors. Aquest té un Thread d'execució que s'encarrega d'executar la política dinàmica, el PolicyChecker. Aquest thread pregunta quan es tardaria a crear una nova màquina, aplica la política i un cop acabi d'aplicar-la i fer les peticions corresponents, queda adormit durant un quart del temps de creació. Prendre la decisió cada quart del temps creat és degut a la granularitat. Si no hi ha cap altra petició en marxa, com més petit sigui l'interval, abans es pot detectar que es necessita una màquina i abans el runtime pot disposar d'ella. El problema està quan ja hi ha una petició sent tractada, que per molt aviat que es detecti el resultat es tarda el mateix en donar resposta. Un quart del temps és un bon equilibri entre totes dues versions, sempre es podria treure com un altre paràmetre configurable per l'usuari.

Com diu la descripció de la política, les decisions es prenen en funció dels nivells de cua, per tant, el millor lloc on prendre les decisions és al QueueManager que sap en detall els temps de cua de cada slot i màquina tal i com s'ha explicat a l'apartat de planificació, 5.4. Per aplicar la política el primer que s'ha de fer és conèixer la duració de les tasques. Com a temps de referència per calcular el temps d'espera es fa servir la mitjana dels temps de tasques del mètode ja tractats abans. Si encara no s'ha executat cap tasca del mètode, però n'hi ha alguna en execució se suposa que la tasca primera tasca del mètode en començar a executar-se acaba en aquell moment i el temps transcorregut des del seu inici és el temps mig d'execució per les tasques d'aquell mètode. Si un mètode no té cap tasca executada ni en execució, se suposa que tardarà 100 ms en executar-se.

Si existís un mètode amb un temps mig d'execució major que el temps de creació, el percentatge de temps ocupat es dispararia provocant la creació d'una màquina per executar una tasca que ja està sent executada. Es limita la duració prevista per totes les tasques al temps de creació d'una màquina i evitar així aquest problema.

En el moment en que se sap el temps mig d'execució d'una tasca pot actualitzar-se el percentatge ocupat dels mètodes que pot executar una màquina. S'incrementa el comptador de temps de cada un dels mètodes que pot executar amb el temps de cada un dels slots de la màquina. Al acabar es pot saber el temps total ocupat d'un mètode.

En l'exemple anterior que amb 2 màquines i 80 segons per servir una petició: Si les tasques tarden A 5 segons, B 3 segons i C 2 segons, A està executant una tasca de tipus 0, que encara tardarà 3 segons en acabar i té 19 tasques de tipus 0 en espera. B podria estar executant una tasca de tipus 2 que tardarà 1 segon en acabar i 5 més esperant a a la cua i 3 de tipus 1. A tindrà un temps de cua de 3 segons + 19 * 5= 108 segons B tindrà un temps de cua de 1 segon + 5*2 + 3*3=20 segons El mètode 0 tindrà 118 segons ocupats, el mètode 1, 138; i el mètode 2, 20 segons.

Ara només cal el nombre total de slots per poder tenir els percentatges d'ocupació. El mètode 0 té 1 sol slot, per tant, 108/(1*80) té un percentatge d'ocupació del 135%. El mètode 1 té 2 slots, ocupa 128/(2*80)= 80% i el mètode 2 té 1 slot amb 20 segons ocupats,

que dóna un 25% d'ocupació.

El QueueManager escull totes aquelles tasques que tinguin més d'un 100% d'ocupació i les envia al ResourceManager, perquè determini quina màquina pot crear que solucioni el problema que hi ha per aquells mètodes i la demana al connector. Amb la implementació actual descriu un recurs capaç d'executar el mètode amb l'identificador més baix del conjunt. En el cas de l'exemple, demanaria una màquina capaç d'executar el mètode 0, ja que seria la única que hi ha. La resta del procés per incorporar el recurs ja està explicat en l'apartat 5.5.1.

Un cop s'ha fet la petició, retorna el control al PolicyChecker, que pregunta quan es tardaria a crear una altra màquina, i s'adormirà com s'ha dit a l'inici de l'apartat durant un quart del temps de servir una petició.

5.7 Alliberar recursos del Cloud

Al llarg de l'execució, existeix un esdeveniment clau per determinar que una màquina pot eliminar-se o no: quan el runtime rep la notificació de que ja no entraran noves tasques que han d'incorporar-se al graf. En aquest moment que el runtime ja coneix totes les tasques que hauran de ser executades en els workers, aquest pot saber quins mètodes s'executaran i, per tant, quines màquines no seràn útils a partir d'aquell moment.

Igual que en la creació existeixen diversos moments al llarg de l'execució d'una aplicació que permeten detectar amb seguretat que una màquina ja no pot ser utilitzada i, per tant, pot ser aturada. Al llarg de l'apartat s'explica quin és el procés que el runtime segueix i quins mecanismes utilitza per decidir quines màquines ja no seran útils quan:

- el runtime rep l'avís de que ja no hi entraran noves tasques.
- s'acaba de crear una màquina i abans de que executi res es detecta que no es útil.
- una màquina acaba d'executar una tasca.
- s'acaba l'execució de l'aplicació.

També s'explica en quins casos aquests esdeveniments són insuficients per apagar totes les màquines que han d'estar apagades i de quina manera es soluciona aquesta mancança.

5.7.1 Política de destrucció a l'indicar que no hi haurà noves tasques

En el moment que el programa principal executa un stopIT, el TaskAnalyser rep aquest avís sap les tasques que es troben en el graf són totes les tasques que haurà d'executar i, per tant, es poden prendre una sèrie de mesures que abans no podia. A partir de que es rep l'avís, es poden prendre decisions sobre si una màquina serà útil o no ja que es pot detectar si una màquina pot rebre més feina, o si ja ha fet tota la feina que podia i pot ser alliberada.

Aplicar una política en aquest moment és útil en dos escenaris:

- la creació inicial de les màquines ha creat una màquina únicament capaç d'executar un conjunt de mètodes que no seràn cridats al llarg de l'execució.

- inicialment o dinàmicament s'han creat un conjunt de màquines capaces d'executar un conjunt de tasques. En el moment en que es rep l'avís, totes les tasques del conjunt poden haver-se executat, per tant, les màquines ja no són útils i poden ser apagades.

Cal tenir en compte existien dos tipus de stopIT, els que només adormen el runtime fins al pròxim startIT() i els que eliminen els components perque ja no s'utilitzarà més el runtime. Si ha arribat un stopIT que no apaga els components, s'ha de respectar la norma que diu que per cada tasca almenys hi ha d'haver una màquina que pugui executar-ho. Només es poden apagar les màquines que pertanyin al conjunt de no crítiques. D'aquesta manera quan arribi el startIT no caldrà tornar a crear els workers en el Cloud.

Si arriba un stopIT que obliga apagar totalment el runtime, la norma pot trencar-se. En el moment en que s'acabin les tasques que hi ha al graf, no es tornarà a fer servir el runtime, per tant, totes les màquines virtuals que ja no rebran noves tasques poden ser apagades independentment de si formen part del conjunt de crítiques o de no-crítiques.

El procés que segueix el runtime comença en el programa principal quan executant-lo s'arriba a un stopIT i es fa la crida a l'API indicant quin tipus de stopIT és pel paràmetre terminate de la funció. La implementació de l'API indica al TaskAnalyser que ja ha arribat a un stopIT indicant si ja no entraran noves tasques fins que l'aplicació arribi al pròxim startIT() o si ja no n'entrarà cap més.

El Task Analyser té dues feines a fer:

- notificar al TaskScheduler que ha arribat la petició i, per tant, que és el moment de mirar si es poden apagar màquines
- avisar al programa principal que pot seguir executant quan totes les tasques que pendents d'execució hagin acabat.

L'avís al TaskScheduler té doble funcionalitat, indicar al TaskScheduler que ha d'apagar les màquines i indicar-li quantes tasques de cada mètode queden per poder prendre les decisions. A partir d'aquest moment, el TaskScheduler manté el número de tasques que queden pendents d'executar per cada un dels mètodes i així evitar preguntar cada vegada al TaskAnalyser una informació que pot deduir a partir de les tasques que van acabant.

Un cop el TaskScheduler té aquestes dades, crida al ResourceManager perquè a partir d'aquestes decideixi quines màquines ja no poden executar res i, per tant, que poden ser apagades. El ResourceManager també ha de fer dues feines:

- fer la comprovació de quines màquines existents hauran d'apagar-se per reduir el cost.
- avisar al Connector que el graf té totes les tasques i que a partir d'aquest moment s'haurà de comprovar totes les màquines que es creïn.

Tota la informació important dels worker que fa servir l'execució és troba dintre de l'estructura ResourcePool. El ResourceManager delega el càlcul al ResourcePool. A partir de les dades que ha enviat el TaskAnalyser i de les dades dels recursos, el ResourcePool decideix quines màquines s'eliminen i esborra les dades que tingui d'elles per evitar que entrin noves tasques. Quan ja sap totes les màquines que ha d'apagar (i n'ha bloquejat

l'entrada de tasques) es retorna al ResourceManager quines màquines han de començar el procés d'apagada salvant les dades; tal i com s'ha explicat en l'apartat 5.7

La decisió es pren seguint el seguent algoritme:

Per cada mètode sense noves tasques **fer**
  Per tota màquina eliminable capaç d'executar-lo **fer**
    Si cap dels mètodes que pot executar no té tasques noves **llavors**
      Elimina les dades

5.7.2 Política de destrucció al moment de la creació

El primer que cal fer és preguntar-se si és útil una política que únicament serveixi per destruir una màquina que no ha fet res. Si és crea una màquina i no es fa servir significa que no caldria haver-la creat i que, per tant, les polítiques de creació no són bones. Existeixen dos escenaris que demostren que és un plantejament erroni.

El primer escenari es dóna en aplicar la política de creació de recursos inicials. Pot donar-se el cas que existeixi un conjunt de mètodes que requereixin crear unes màquines amb unes determinades característiques on únicament aquests mètodes puguin ser executats, però que realment no hi hagi cap tasca que es correspongui amb aquell mètode.

En l'espai de temps entre que es demana la màquina i que es disposa d'aquesta per poder-hi executar tasques és possible que es rebi l'avís conforme no es rebran noves tasques. Com que la màquina encara no ha estat creada, no pot destruir-se i, per tant, la política de destrucció que s'aplica al executar un stopIT no l'hagués pogut apagar. És necessari que es comprovi la seva utilitat quan màquina s'acabi de crear .

El segon cas és que es demani una màquina degut a la política de creació periòdica. Si el proveïdor tarda més del temps estimat per servir la petició o s'ajusta el paràmetre de la política per demanar recursos el més aviat possible, pot ser que la màquina acabi de crear-se un cop ja ha acabat l'execució o no quedin tasques dels seus mètodes i pugui ser apagada.

A l'inici de l'apartat 5.5, quan s'explica el procés de creació i destrucció de les màquines, s'ha donat una visió global del procés. En el que queda de secció es detalla quins passos se segueixen per tal de determinar si la nova màquina s'ha d'eliminar. Cal tenir en compte que la màquina acaba de ser creada, per tant, no conté cap fitxer que calgui salvar, ni està a la llista de recursos a utilitzar. Tot el procés de destrucció explicat al principi queda reduït simplement a anul·lar la reserva que tenia el ResourceManager i ordenar al Cloud Proveider la destrucció.

L'objecte que controla la creació de la màquina és el Connector. Aquest pot decidir que una màquina pot apagar-se en dos casos:

- Ja s'ha acabat l'execució de l'aplicació. La màquina segur que no és útil i, per tant, ha de ser apagada. S'envia l'ordre al Cloud Provider de que la màquina ha d'apagar-se i s'avisa al TaskScheduler que la màquina no serà concedida mitjançant la crida refuseCloudWorkerRequest. Aquesta s'encarrega d'avisar al ResourceManager de que la petició ha estat desestimada i que no es pot comptar amb els recursos a l'hora de planificar.
- S'ha demanat crear una màquina abans de rebre l'avís, per tant, s'ha demanat pensant que pot ser utilitzada. Durant la creació d'aquesta s'executa stopIT, i per

tant, ja es pot determinar si la màquina pot ser útil. El connector no pot saber-ho per si mateix, el que fa és indicar al TaskScheduler que s'ha completat la creació, però que abans de confirmar-la al ResourceManager ha de comprovar si la màquina serà útil. La mateixa funció de confirmació, addCloudNode, s'encarrega de fer-ho si se li activa el flag de comprovació. El TaskScheduler farà tota la comprovació, si la màquina és útil s'incorporarà la màquina a la llista de recursos tal i com ja s'ha explicat; sinó, simplement s'indicarà al connector que la màquina no era útil perquè l'apagui i cridi la funció refuseCloudWorkerRequest.

Només queda per veure com el TaskScheduler sap si la màquina recent creada serà útil o no. De fet l'objectiu d'aquesta política no és veure si la màquina s'utilitzarà o no, el que s'ha de comprovar és si la màquina té possibilitats de ser utilitzada de manera que es millori el rendiment. La comprovació que realment es fa és la següent: existeix alguna tasca al graf que pugui executar la màquina?

La comprovació afegeix molt poc cost computacional. Aprofitant que igualment s'ha de preguntar quins mètodes pot executar la nova màquina per afegir-la al ResourceManager, s'utilitza la llista dels mètodes i per cada un es consulta el número de tasques que queden en el graf. Aquesta informació és la que el TaskScheduler manté actualitzada a partir del moment en que el TaskAnalyser propaga que s'ha executat un stopIT, només cal fer una consulta a l'estructura que la manté. Si existeix un sol mètode que pugui ser executat en la màquina aquesta és útil i es completa el procés.

Si s'ha de comprovar **llavors**
  eliminar:= cert
  Mentre eliminar=cert i per cada mètode mId executable a la màquina **fer**
    Si counts.get(mId)>0 **llavors**
      eliminar:= false;

5.7.3 Polítiques de destrucció al final de l'execució

Una execució a COMPSs es dóna per acabada quan l'aplicació arriba a un StopIT amb terminate cert. L'actuació segueix els mateix patró que en cas de que el paràmetre terminate sigui fals, esperar a que:

- s'apaguin les màquines que el runtime ja no es faran servir per haver rebut un stopIT.
- s'acabin d'executar totes les tasques que hi ha esperant al TaskAnalyzer.
- es transfereixin tots els fitxers resultat des dels workers cap al master.
- s'eliminin tots els fitxers temporal.

La diferència es que, al complir-se aquestes quatre condicions, l'API dóna l'ordre d'apagar els components. Si ja no s'ha de fer servir més el Runtime, tampoc s'hi enviaran noves tasques a cap de les màquines virtuals de les que disposa; per tant, totes poden ser apagades. Durant el tractament de l'StopIT, el runtime espera a que tots els fitxers resultat tornin al master, qualsevol fitxer que estigués en aquestes màquines no és necessari. Els fitxers que pot contenir són: o bé, una còpia d'un fitxer d'entrada que es troba al master des de l'inici, o bé, una versió temporal que no importa perdre, o bé, la última versió del fitxer

i que s'ha portat de tornada al master. Tota la informació de qualsevol de les màquines virtuals es pot perdre, no cal mirar quins fitxers salvar.

Quan arriba l'ordre d'apagar el component TaskScheduler, aquest fa que el ResourceManager utilitzi el Connector per apagar totes les màquines que tingués enceses en qualsevol Cloud provider, independentment de si forma part del conjunt de màquines crítiques o de les no-crítiques.

5.7.4 Política de destrucció al completar una tasca

S'han comentat tres moments en que es pot decidir apagar una màquina, però no n'hi ha prou per detectar que una màquina pot aturar-se. Un escenari possible és el següent: totes les màquines estàn executant tasques, l'aplicació principal arriba a l'StopIT, envia les tasques que queden al graf, però com que les màquines estan executant no poden ser apagades. A partir d'aquell moment s'analitza la utilitat de totes les màquines que es creïn, però no de les que ja han estat creades. L'única opció que queda és apagar la màquina a l'acabar l'execució del runtime, la qual cosa implica estar consumint uns recursos que no es faran servir més.

Una solució és afegir un altre moment en que el runtime pugui decidir que una màquina pot ser apagada: quan aquesta acaba d'executar una tasca. Quan el TaskAnalyser disposa de tot el graf de dependències, el runtime pot decidir si la màquina que ha quedat lliure pot ser apagada o no. La decisió és pren seguint el mateix criteri que en els casos anteriors: la màquina s'apaga si no està executant res, no és una màquina física ni del conjunt crític i per cada mètode que sigui capaç d'executar la màquina a apagar existeixen més processadors (sense comptar els de la màquina) que tasques del mètode en el graf.

Aquesta política s'inicia al TaskScheduler, concretament cada cop que rep una notificació del JobManager indicant que ha acabat d'executar una tasca. Aquest intenta enviar una tasca a la màquina tal i com s'ha explicat a l'apartat 5.4. En el cas que no trobi cap tasca que pugui executar-s'hi i s'hagi rebut l'avís conforme el TaskAnalyser no rebrà noves tasques, se'n comprova la utilitat invoca la funció tryToTerminate del ResourceManager indicant el nombre de tasques que queden de cada mètode. Aquest pregunta al ResourcePool si és una màquina que es pot apagar, és a dir, si no és física ni forma part del conjunt crític. Si ho és, continuarà encesa fins que es decideixi apagar-la.

En cas contrari, el ResourceManager delega en el ResourcePool la comprovació de la utilitat de la màquina. Una màquina es declara inútil si no està executant cap altra tasca i si es compleix la relació entre tasques i processadors dels seus mètodes. Per determinar-ho s'aplica el següent algoritme, on $procs_m$ representa el número de slots pel mètode m, $procs_{maquina}$ són els de la màquina i $tasques_m$ són tasques del mètode m.

util:= resource.taskCount!=0
Mentre !util i **per cada** mètode m capaç d'executar la màquina **fer**
  util:= #$procs_m$ - #$procs_{maquina}$ $<$ #$tasques_m$
retorna util

Si el ResourcePool determina que la màquina és inútil, el ResourceManager elimina totes les dades que es tenen d'ella i avisa al TaskScheduler de que s'ha d'iniciar tot el procés per apagar la màquina correctament.

A l'inici de l'apartat es demostra que amb 3 moments no n'hi ha prou per apagar les màquines quan toca. Afegir una comprovació al completar una tasca permet detectar alguns casos que s'escapen. De totes maneres la política que s'ha decidit seguir en aquests casos no soluciona totalment el problema. Per exemple el cas en que al TaskAnalyser hi ha el graf de la figura 5.13, en que les tasques 1 i 2 estan executant-se en les màquines A i B respectivament, sent A un màquina del conjunt crític. Quan B acaba la seva tasca fa la comprovació i veu que encara queden 2 tasques del mateix tipus per acabar, 3 i 4. La decisió que pren és mantenir-se encès. Quan A acabi serà una màquina del conjunt crític i, per tant, no s'apagarà. Quan les tasques 3 i 4 entrin al TaskScheduler, cada una al seu moment poden executar-se sobre la màquina A, que no s'apagarà pel mateix motiu. Com que B no executa cap més tasca, no tornarà a comprovar si ha de ser eliminada, i per tant, la màquina seguirà encesa durant tota l'execució.

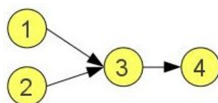

Figura 5.13: Graf de dependències que pot portar a errors a la destrucció de màquines a l'acabar una tasca

Intentar detectar casos com aquest és complex, ja que això implicaria recórrer el graf i pre-assignar les tasques. És un procés molt costós temporalment. Si el runtime es trobés executant una aplicació amb tasques d'una duració molt curta i amb molt paral·lelisme, estaria més temps executant aquesta nova política que no tractant les tasques. Per evitar-ho, s'assumeixen aquests possibles errors que pot donar.

5.7.5 Polítiques de destrucció periòdica

Les polítiques vistes fins aquest apartat actuen en moments determinats, per tant, la seva actuació és immediata a partir del moment en que la possibilitat d'apagar una màquina és detectable.

A part dels casos concrets, totes les decisions que prenen les polítiques explicades s'apliquen a partir de que el runtime sap totes les tasques que han d'executar-se remotament per acabar l'execució. Això significa que mentre el graf no té totes les tasques el número de màquines només pot incrementar. A més a més, en el cas que el número de màquines de les que disposa el runtime està limitat, el conjunt de tasques inicial determina de quines màquines disposa el runtime fins a arribar a executar un stopIT.

En el cas de la creació de màquines existeix una política que s'aplica cada cert temps, per la destrucció de màquines s'utilitza la mateixa solució. El que es fa periòdicament no és comprovar si cal alguna creació o alguna destrucció, sinó que periòdicament s'analitza si cal modificar el conjunt de màquines virtuals. Un dels aspectes que l'apartat 5.1 marca com a criteri de disseny és que només es pot fer un canvi de cop en aquest conjunt de màquines. Això vol dir que, s'ha de decidir quina política periòdica té més prioritat: la de creació o la de destrucció.

Prioritzar la destrucció aporta una reducció del cost de l'execució ja que les màquines s'apagarien tan aviat com fos possible. Tot i aquest avantatge, té l'inconvenient de que redueix el rendiment. El runtime ha d'esperar fins a la següent comprovació per demanar una nova màquina. En el cas que l'aplicació redueïs de cop el número de tasques que poden executar aquelles màquines, el runtime aniria apagant-ne una en cada comprovació i, per tant, no encendria noves màquines.

Per altra banda, prioritzar la creació aporta un augment en el rendiment de l'aplicació ja que amb més màquines es pot aprofitar millor el paral·lelisme. Al revés del cas anterior, ara l'inconvenient és que al decidir encendre una màquina no se'n pot apagar una; per tant, augmenta el cost de l'execució. Es pot donar el cas simètric a l'anterior, és a dir, que de cop l'aplicació obri molt paral·lelisme i requereixi moltes màquines, això faria que durant molta estona s'estiguessin pagant màquines que podrien estar apagades.

La decisió per la que s'opta en el prototip és prioritzar la creació. La decisió es pren per dos motius. El primer és el temps de resposta en la creació és més alt que el de la destrucció i d'aquesta manera es redueix una mica la pèrdua de rendiment que això suposa.

El segon motiu és que existeixen dos límits de creació: el límit de creació indicat al connector i l'augment del temps de creació de la següent màquina. El primer és simple, si s'arriba a aquest límit no es crearan més màquines i, per tant, la destrucció podrà executar-se. El límit en el temps de creació de la següent màquina és degut a la política de creació periòdica explicada a l'apartat 5.6.3. En el cas que el prototip estigui fent servir un cloud en que el número de peticions influeix en el temps de resposta, el fet de demanar varies màquines de manera consecutiva provoca que aquest temps augmenti i, per tant, la política no demana més màquines ja que no arribarien a ser utilitzades. Al no haver de crear més màquines es pot aplicar la política de destrucció.

Cal una política que sigui capaç de decidir en qualsevol moment si el conjunt de màquines de les que disposa el runtime s'excedeix de les necessitats de l'aplicació en aquell moment. Per determinar si el conjunt és excessiu es compara el nivell de càrrega que és capaç d'assumir el runtime per cada mètode (temps de creació d'una nova màquina multiplicat pel nombre de slots del mètode) amb el nivell de càrrega que suposen les tasques que podrien estar executant-se (ja s'han enviat al TaskScheduler) i les tasques que tenen una dependència directa amb aquestes. La figura 5.14 il·lustra les tasques que es tindran en compte, marcant en vermell les que ja es troben en el TaskScheduler i en groc les que hi entraràn quan aquestes acabin.

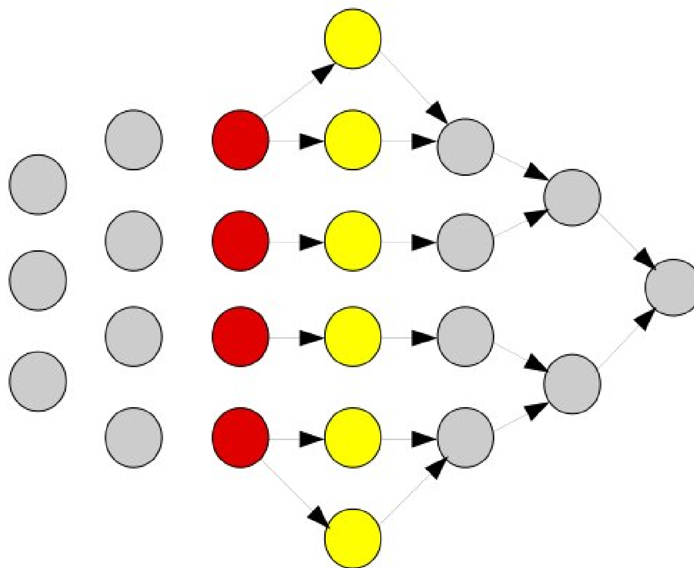

Figura 5.14: Graf de dependències marcant les tasques que es faran servir per la política de destrucció periòdica

Les tasques que ja es troben en el TaskScheduler, les vermelles, representen les ne-

cessitats del runtime en aquell moment. Amb aquestes ja n'hi ha prou per determinar que el runtime s'està excedint de les necessitats de l'aplicació en aquell moment. La lenta reacció a l'hora d'encendre les màquines virtuals obliga a buscar algun sistema per evitar apagar màquines que poden ser necessàries en un plaç curt de temps. La única manera és anticipant-se a les necessitats futures. Per fer-ho és necessari obtenir informació del graf. Recorre'l sencer i analitzar-lo és un procés costós. La solució adoptada en el prototip és avançar-se un únic pas en l'execució i incorporar al càlcul les tasques que podran executar-se quan es completin les que poden executar-se en el aquell moment, les que apareixen en el graf de color groc.

Un cop explicades les dades que es fan servir per prendre la decisió, falta definir quin criteri se segueix. Com que la política de creació periòdica s'executa a la vegada que la de destrucció s'intentarà aprofitar el màxim de dades possibles per evitar afegir més overhead al runtime. La creació dinàmica utilitza:

- el temps mig d'execució d'una tasca per cada mètode
- el temps d'espera en cada un dels slots que té cada recurs
- el temps necessari per executar totes les tasques d'un mètode
- el temps de creació d'una màquina al Cloud
- el temps total de computació assumible per cada mètode en el temps de crear una màquina

A partir d'aquestes dades, s'ha d'intentar encaixar la política de destrucció. Es sumen els temps de computació de les tasques que estan en el graf i que poden entrar en el moment en que acaba una execució i els temps necessaris per cada mètode en el TaskScheduler. Per exemple, en el cas de tenir una aplicació amb quatre mètodes:

0 tarda 5 segons

1 tarda 4 segons

2 tarda 10 segons

3 tarda 30 segons

Si el runtime disposa de les cinc màquines següents:

A amb un slot capaç d'executar els mètodes 0 i 1 amb 15 segons de cua

B amb un slot capaç d'executar els mètodes 1 i 2 amb 40 segons de cua

C amb un slot capaç d'executar els mètodes 0 i 1 amb 10 segons de cua

D amb dos slots capaços d'executar els mètodes 1 i 2 amb 46 i 50 segons de cua

E amb un slot capaç d'executar el mètode 3 amb 75 segons a la cua

Si en el graf estan esperant a que acabi alguna de les tasques planificades vuit tasques de tipus 2. Les càrregues que el runtime ha de ser capaç d'acceptar són les següents:

0 té dos slots amb una càrrega total de 25 segons en el TaskScheduler

1 té cinc slots amb una càrrega total de 151 segons en el TaskScheduler i 32 segons més que poden executar-se al acabar alguna tasca. En total 183 segons.

2 té tres slots amb una càrrega total de 136 segons en el TaskScheduler

3 té un slot amb 75 segons en el TaskScheduler

Si el temps de computació disponible en el temps de crear una màquina per algun mètode és molt gran (per exemple: 5 cops més gran) que el temps que es tardar a executar les tasques del mètode, el mètode passa a ser un candidat a la reducció. En l'exemple, si es tarda 140 segons a encendre una màquina, el mètode 0 és l'únic candidat ja que és 11 cops major.

0 pot arribar a executar fins a 280 segons dels que n'ocupa 25 (11.2x)

1 pot arribar a executar fins a 700 segons dels que n'ocupa 183 (3.8x)

2 pot arribar a executar fins a 420 segons dels que n'ocupa 136 (3.1x)

3 pot arribar a executar fins a 140 segons dels que n'ocupa 75 (1.9x)

En el cas de necessitar reduir el nombre de màquines, una de les màquines capaces d'executar aquests mètodes candidats és la que ha d'apagar-se. Per escollir la màquina que s'apaga primer es passa un filtre buscant candidats comprovant que el temps necessari per executar les tasques del mètode no sobrepassi la meitat del temps de comput que poden assumir les màquines del mètode sense comptar amb la màquina candidata en el temps de crear una nova màquina. De totes les màquines que superen aquesta condició s'escull aquella en que el màxim temps d'espera en un dels seus slots sigui mínim. En el cas de l'exemple el mètode 0 té dues màquines amb un sol slot: A i C amb 15 i 10 segons a la cua respectivament. Totes dues tenen un slot capaç d'executar el mètode 0 i el mètode 1 i, en tots dos casos, eliminar aquest slot no fa que la càrrega actual pel mètode per cap dels dos mètode superi el 50% de la càrrega assumible pel mètode. La màquina escollida per apagar-se és la C ja que el màxim temps d'espera en un dels seus slots és 10 segons, menor que els 15 de la màquina A.

La política és capaç de decidir que s'ha d'apagar una màquina, però aquesta pot estar executant alguna tasca. Quan es decideix que la màquina s'apaga, es bloqueja l'entrada de noves tasques, de totes maneres no es pot encendre tot el procés de salvar la informació i apagar-la ja que encara pot faltar algun fitxer. En aquest cas el runtime ha de fer dues coses: buidar les cues que tinguin els slots re-assignant les tasques a altres màquines i esperar a que acabin d'executar-se les tasques que ja han començat a executar-se en el node.

5.7.6 Implementació de la política de destrucció al llarg de l'execució

Al llarg d'aquest punt cal detallar tres aspectes bàsics que s'han explicat al punt anterior. El primer és com s'ha implementat la política en sí. També que fa falta explicar com està implementat el recompte del nombre de tasques que estan esperant a que acabi una tasca per poder entrar. Finalment, cal explicar quines estructures i procediments són necessaris

per tenir en compte que les màquines han d'esperar-se a acabar l'execució d'algunes tasques pendents quan són seleccionades.

Com ja es diu en la secció anterior, per implementar tota la part de la política s'ha intentat re-utilitzar el procés que seguit per la creació periòdica. És manté el Thread PolicyChecker i tot el procés de creació de la mateixa manera. De totes maneres, la funció encarregada d'aplicar la política de creació, checkPolicies canvia. La funció no només comprova la creació sinó que, a més a més, revisa la política de destrucció. A part dels paràmetres necessaris per la creació, també necessita que el TaskScheduler li doni el nombre de tasques que hi ha al graf esperant a que una tasca acabi per entrar. El retorn de la funció també canvia, retorna una llista amb totes les màquines que han de ser aturades, tot i que com a molt té un element.

La funció comença de la mateixa manera que en el cas de la creació. Fa els mateixos càlculs i mira si pot crear alguna màquina. En el cas que es crei una màquina la funció acabarà i retornarà una llista buida.

Per contra si la política de creació decideix que no cal modificar el conjunt, s'aplica la de destrucció. El primer canvi que es fa és l'actualització dels temps de computació de cada mètode afegint les tasques en espera del TaskAnalyser. Tenint el temps total de computació necessari per cada mètode de les tasques que poden executar-se, el temps mig de les tasques de cada mètode i el número de tasques en el graf és una actualització dels valors molt simple que es pot fer amb la fórmula $T_m = T_{em} + \#tasques_m * TM_m$ on T_m és el temps de computació pel mètode m; T_{em}, el temps de computació necessari per executar les tasques en espera del mètode m; $\#tasques_m$, el nombre de tasques del mètode m al graf que poden entrar i TM_m és temps mig d'execució d'una tasca del mètode m.

A partir d'aquí només cal aplicar el següent algoritme per decidir quina màquina es vol utilitzar:

Per cada mètode m **fer**
    CPUTime:= temps per crear una màquina * slots que poden executar m
    Si T_m/CPU $<$ 0.2 **llavors**
        Per cada recurs R capaç d'executar m **fer**
            Per cada mètode m2 que pot executar R **fer**
                Si compleix $T_{m2}/(T_{creacio}$*(slots per m2 - slots de R))$<$50% **llavors**
                    m2 passa a ser candidat
Minimitzar el temps d'espera entre els candidats

Un cop ja està decidida quina màquina s'ha d'apagar, el que cal fer és distribuir les tasques que tingués a les cues dels seus slots. El procés seguit és similar al que s'utilitzaria per la planificació en cas de tenir varies tasques que entren de cop al TaskScheduler. S'agafen totes les tasques que estiguessin esperant en els slots i una per una es busca quines màquines poden executar-lo i s'assigna a l'slot amb menys temps d'espera.

Un dels aspectes que també cal veure són les estructures necessàries per mantenir la decisió de la política tot i no poder-se aplicar immediatament ja que hi poden haver tasques executant-se. Un cop la política dóna la seva decisió d'apagar la màquina, retorna al TaskScheduler la llista amb una màquina, el primer que fa PolicyChecker, independentment de si la màquina està executant tasques o no, és eliminar-la del ResourceManager d'aquesta manera la màquina ja no apareix a les polítiques de planificació i no s'omplen les cues que acaba de buidar el QueueManager.

Un cop bloquejada l'entrada de noves tasques el PolicyChecker, comprova si la màquina està executant alguna tasca. Si no executa res inicia el procés d'apagar-la tal com s'ha explicat anteriorment. Per contra si la màquina està executant no es pot donar l'ordre ja que encara falta algun fitxer per generar. Es guarda a stoppedResources, dintre del TaskScheduler, que la màquina ha d'apagar-se quan acabin les tasques i el PolicyChecker torna a dormir un quart del temps de creació.

La segona part d'aquest procés es fa a mesura que acaben les tasques. A cada final de tasca, igual que s'explica a la planificació, s'actualitzen totes les estructures i s'avisa al TaskAnalyser que la tasca ha acabat. Abans de mirar quina és la pròxima tasca que pot executar, el TaskScheduler mira a stoppedResources si el recurs està marcat com a màquina a eliminar. Si no està marcat el procés segueix de la mateixa manera, però, si ho està, comprova si encara té alguna altra tasca executant-se. Si no hi ha cap tasca executant-se ja pot iniciar-se el procés d'apagar salvant els fitxers, si n'hi ha alguna segueix esperant a que acabin les tasques però no li se li assigna més tasques.

L'últim procés a explicar és com es fa la comptabilitat de les tasques que poden entrar a executar-se. És un procés que ha de fer-se de forma eficient ja que s'ha d'evitar recórrer el graf cada cop que s'activi la política de destrucció periòdica. Per començar existeixen tres possibilitats perquè la política pugui tenir aquesta informació:

- abans d'aplicar-la es demanen aquestes dades i el TaskAnalyser recorre el graf comptant-les: cal recórrer el graf i això s'ha d'evitar.
- abans d'aplicar-la es demanen aquestes dades i el TaskAnalyser les retorna directament, tota la comptabilitat es fa durant els canvis en el graf: si el TaskAnalyser ha de tractar altres peticions prèvies, la política es queda bloquejada esperant resposta fins que acabi de tractar-les.
- TaskScheduler té aquestes dades i TaskAnalyser actualitza els valors cada cop que es fa algun canvi en el graf: Soluciona tots dos problemes anteriors.

L'opció per la que el prototip es decanta és per la tercera. TaskScheduler té un comptador per cada mètode i es va incrementant o decrementant en funció del que TaskAnalyser li indiqui que té. Existeixen 2 moments que produeixen canvis en el graf i poden modificar els comptadors:

- arriba una nova tasca al TaskAnalyser
- TaskScheduler notifica que s'ha acabat una tasca i TaskAnalyser envia a executar una nova tasca.

Quan es detecta una nova tasca, el TaskAnalyser ha de començar el tractament. Pensant en termes del graf, el que es busca comptabilitzar són les tasques que tenen dependències, però que les tasques de les que depenen no en tenen. En el moment en que s'afegeix una nova tasca al graf, si la tasca no depèn de ningú, l'envia a executar i no cal preocupar-se de comptar res; si la tasca depèn d'una altra tasca, s'afegeixen arestes al graf entre la tasca vella i la nova. En el moment en que s'afegeix aquesta aresta, el TaskAnalyzer pot determinar si la tasca de la que depèn està executant-se o no. Si no està executant-se, no s'ha de comptabilitzar; per contra, si la tasca de la que depèn s'ha enviat al TaskScheduler pot estar executant-se en aquell moment i s'ha de comptabilitzar. TaskAnalyser avisa al TaskScheduler de que ha d'incrementar en 1 el comptador pel mètode de la tasca.

El moment més delicat és quan TaskScheduler avisa a Task Analyser que una tasca ha acabat i s'alliberen dependències i s'esborren arestes del graf. És en aquest moment en que un conjunt de tasques pot passar de ser comptabilitzat a executar i tasques que fins ara no calia tenir en compte s'han de comptabilitzar. Com que el final d'una tasca pot suposar l'inici de varies, el procés queda dividit en 2 fases: la detecció de canvis, que es fa al rebre l'avís al TaskAnalyser, i l'actualització de l'estructura, que es fa al TaskScheduler quan es rep la petició de planificació de tasques.

La primera part del procés és senzilla, Task Analyser rep la notificació de final d'una tasca, s'esborren totes les dependències i es crea el grup de tasques que han de ser planificades de la mateixa manera que es feia quan no es feia aquest càlcul. Generat el grup de tasques que queden lliures de dependències ja només ens queda analitzar els successors d'aquestes tasques amb el següent algoritme:

```
tasquesWaiting:= nou conjunt
Per cada tasca T del conjunt a planificar fer
    Per cada successor S de la tasca T fer
        waiting:= cert
        Per cada antecessor A de la tasca S fer
            waiting:= waiting i A no té antecessors
        Si waiting llavors
            afegir S a tasquesWaiting
```

D'aquesta manera s'obtenen fàcilment tots els successors de les tasques del conjunt que s'envia a executar. Aquests successors depenen únicament de tasques lliures de dependències. Com que les tasques del conjunt fins ara depenien de la tasca que ha acabat, encara no estaven comptabilitzades a l'estructura.

La segona fase és molt més simple, aprofitant el missatge on s'envien totes les noves tasques, s'envien també les tasques que han d'incrementar el comptador. I això és el que es fa al tractar la petició: incrementar el valor per cada tasca del conjunt. De totes maneres, això no és tot el que es fa en aquest moment. A mesura que el TaskScheduler planifica les tasques que li han arribat ha d'anar decrementant el comptador del mètode de la tasca que planifica, ja que a partir d'aquell moment és una tasca que ja pot executar-se en aquell moment i no depèn de cap que estigui executant-se.

5.8 Exemple d'aplicació de les polítiques de creació i alliberament de recursos

En aquest últim punt del capítol, es posa un exemple de com les diferents polítiques per demanar nous recursos o alliberar-los actuen. L'aplicació que es fa servir per l'exemple consta de 8 tasques que poden ser executades en un recurs de les mateixes característiques. El flux de l'aplicació queda descrit en el graf de la figura 5.15.

La figura 5.16 mostra la clau per poder interpretar la traça de la figura 5.17 i així poder saber quina política s'aplica en cada moment de l'execució i quina decisió prenen les polítiques en aquell moment.

Tal com es diu al punt 5.6.1, la primera política que s'executa en qualsevol execució de qualsevol aplicació és la política de creació de recursos inicial. Que demana les màquines

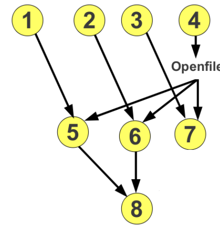

Figura 5.15: Graf de flux de l'aplicació d'exemple d'aplicació de les polítiques de creació i alliberament

- Creació Inicial
- Creació periòdica
- Destrucció al rebre StopIT
- Destrucció al completar una tasca
- Destrucció al final d'una creació
- Destrucció al final d'execució
- Destrucció periòdica
- + Es decideix demanar una màquina
- - Es decideix destruir una màquina

Figura 5.16: Llegenda de l'histograma d'exemple 5.17

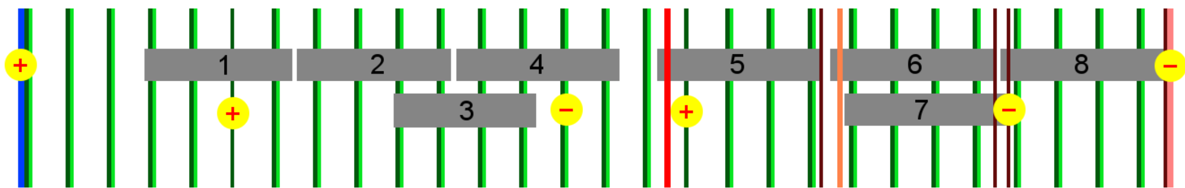

Figura 5.17: Traça d'exemple d'aplicació de les diferents polítiques de creacií alliberament de recursos

necessàries per poder executar l'aplicació o que hagi demanat l'usuari. En aquest cas, totes les tasques poden executar-se sobre un mateix tipus de màquina, per tant, se'n demana una. I es comença l'execució del programa principal que fa entrar les tasques 1, 2, 3 i 4 al graf i es queda esperant a tenir el resultat de la tasca 4 per poder fer un l'OpenFile.

Ja a partir d'aquell moment, comencen a aplicar-se les polítiques periòdiques, tant la de creació com la de destrucció. Com que encara no hi ha cap màquina creada i no s'ha començat l'execució de cap tasca, les polítiques no tenen dades del que tarda a executar-se una tasca i no poden demanar més màquines. Quan al cap d'uns segons apareix la màquina demanada per la política de recursos inicials, comença a executar la primera tasca i dóna dades del que es tarda a executar les tasques. Al cap de poques execucions de les polítiques periòdiques, la política de creació ja demana una nova màquina. Encara no coneix el temps total d'execució d'una tasca, però ja supera el límit de temps necessari perque surti a compte demanar-ne una nova. En l'execució que es demana la nova màquina, la política periòdica de destrucció no s'executa perque ja es fa un canvi.

La primera màquina acaba d'executar la tasca 1 i comença l'execució de la segona màquina. Com que el temps previst per crear una nova màquina és major que el necessari per executar una tasca, es decideix no crear-ne cap més. Al cap de poques execucions de les dues polítiques apareix la màquina que havia demanat la política periòdica i comença a executar la tasca 3.

La tasca 2 acaba d'executar-se en la primera màquina que s'havia demanat i aquesta comença a executar la tasca 4. Quan l'últim recurs demanat acaba d'executar la tasca

3, es troba que no hi ha més tasques que pugui executar i es queda esperant. Quan la política periòdica de destrucció, s'adona que la càrrega del runtime és baixa i que no hi ha cap altra tasca en el graf que pugui executar-se, decideix apagar-la. De manera que l'execució torna a quedar-se amb una sola màquina que està executant.

Quan acaba d'executar la tasca 4, l'OpenFile ja pot executar-se i ja poden fer-se totes les modificacions en el fitxer de forma local en el master. Durant el temps de fer aquestes modificacions al fitxer, s'executen les polítiques periòdiques. Com que no hi ha càrrega en el graf, no es demanen noves màquines. La política de destrucció periòdica hauria de destruir aquesta màquina, però al ser la única màquina, forma part de conjunt crític i no pot ser apagada.

Quan s'acaben de fer les modificacions en el fitxer, el programa principal introdueix la tasca 5 al graf, que passa a executar-se en la màquina que estava aturada. Introdueix també les tasques 6 i 7 totalment lliures de dependències i la tasca 8 depenent de les tasques 5 i 6. En aquest moment, el programa principal acaba i, per tant, executa un stopIT. Això provoca que s'apliqui la política de destrucció que tracta aquest esdeveniment, tot i que, en aquest cas no dóna l'ordre d'apagar cap màquina.

La primera execució de la política de creació temporal, ja coneix la duració de les tasques que hi ha en el graf ja que encara les té de les tasques que ja han acabat d'executar-se, i per tant, en veure que hi ha les tasques 6 i 7 pendents d'executar demana una nova màquina al Cloud Provider i espera a rebre-la.

Com que en aquest punt de l'execució el programa principal ja ha executat un stopIt, al acabar qualsevol tasca s'ha d'aplicar també la política de destrucció que comproba si la màquina que ha executat ha d'apagar-se. En aquest cas encara queden tasques per executar i la màquina forma part del conjunt crític, per tant, es decideix mantenir la màquina i comença a executar la tasca 6.

La política de destrucció en cas d'acabar una tasca no és la única que ha d'aplicar-se a partir del moment en que s'executa un stopIT. Quan el Cloud Provider retorna la màquina demanada, és necessari aplicar la política de destrucció que s'aplica al acabar la creació d'una màquina. Com que encara pot executar tasques, la màquina no es destrueix i comença a executar la tasca 7.

A l'acabar l'execució de la tasca 6, es torna a aplicar la política de destrucció corresponent i es manté la màquina pels mateixos motius que abans i comença a executar la tasca 8 que acaba de quedar lliure de dependències.

Quan la tasca 7 acaba, també s'executa la política de destrucció. Aquesta vegada el resultat varia, ja no queden tasques al graf i al no ser una màquina que forma part del conjunt crític; per tant, la màquina deixa de ser útil i es poden destruir aquests recursos. Així que només queda la màquina del conjunt crític executant la última tasca, la 8.

Quan aquesta màquina acaba d'executar la tasca, es torna a executar la política de destrucció. En aquest cas no queden tasques per executar i la màquina hauria de ser apagada. Al ser una màquina del conjunt crític aquesta política no pot destruir-la i es manté. De totes maneres, al haver-se executat la última tasca del graf, quan tots els fitxer resultat s'hagin transferit al master, l'aplicació es dóna per finalitzada i, per tant, s'aplica la política de destrucció en cas de finalitzar l'aplicació que apaga totes les màquines demanades encara estiguin enceses.

Anàlisi del Prototip

Capítol 6

Anàlisi del Prototip

Entre tots els objectius que es van marcar a l'inici projecte, i que es poden llegir a la secció 1.2, n'apareix un que diu que s'ha d'avaluar el prototip resultant del projecte i el Cloud Computing.

Al llarg d'aquest capítol s'aporten i s'expliquen els resultats d'una sèrie de proves que s'han fet utilitzant el prototip. Cada una de les proves realitzades busca aprofundir en un aspecte concret: el rendiment de les noves polítiques de planificació, la manera en que es gestionen els recursos, la diferència de costos, tant computacionals com econòmics, o com influeix el Cloud Provider sobre el rendiment del runtime.

A partir d'aquestes dades, es treuran les conclusions presentades en el capítol 7.1.

6.1 Entorn de proves

La figura 6.1 mostra el testbed en que s'han realitzat totes les proves que es presenten al llarg del capítol.

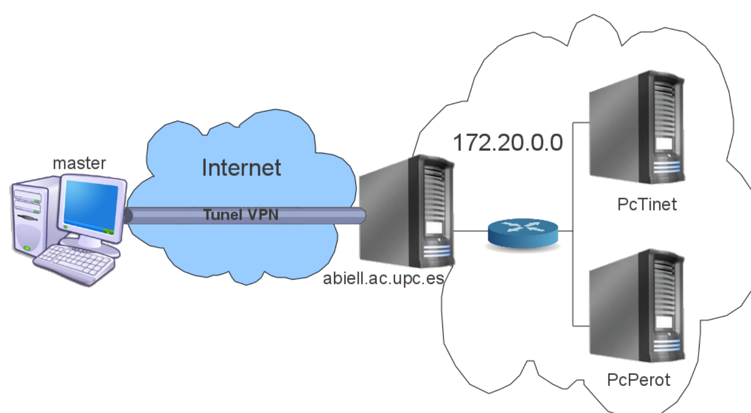

Figura 6.1: Esquema de la xarxa utilitzada com a testbed

En totes elles, la mateixa màquina que executa l'aplicació principal és la que alberga tot el desplegament dels components del runtime. El master és un pc sobretaula equipat amb un processador Intel Core Duo, dos cores a 3GHz i 4 GB de RAM i un disc dur de 160GB.

El Cloud utilitzat durant les proves funciona sobre dos servidors: PcTinet i PcPerot. PcTinet té un processador Intel Xeon, quatre cores a 3GHz amb 16GB de RAM i un disc dur amb 110GB disponibles de 250 a 7200rpm, capaç de donar un ample de banda de 53.26 MB/sec. PcPerot també té un processador Intel Xeon, quatre cores a 2.66GHz, 16GB de RAM i 14GB disponibles de 150 en un disc dur de les mateixes característiques. Com ja s'ha comentat al llarg de tota la memòria, els Cloud està gestionat amb EMOTIVE Cloud. Cada un dels servers té el seu propi VtM i el Scheduler es troba a PcTinet.

La connexió entre el master i el Cloud es fa a través d'Internet, utilitzant una vpn contra un tercer node accessible des d'Internet que es troba dins la mateixa subxarxa que els nodes del cloud i que fa accessible el Cloud. La connexió màxima entre els Cloud i el master és de 10MBps. La xarxa interna del Cloud és capaç d'enviar a 50 MBps.

A l'hora de fer proves amb la versió de COMPSs per Grid, el desplegament del runtime i l'execució del master és igual que en el cas de Cloud. Per evitar tenir un overhead generat pel fet d'utilitzar màquines virtuals enlloc de físiques, els recursos que formen el Grid són màquines virtuals creades per EMOTIVE Cloud. Així els workers executaran les tasques en les mateixes condicions.

6.2 Aplicacions

6.2.1 SparseLU

SparseLU és una aplicació per factoritzar una matriu com el producte d'una matriu triangular inferior i una superior.

La matriu queda dividida en blocs de NxN sobre els que es van aplicant 4 tipus d'operacions que modifiquen un cert bloc: lu0, fwd, bdiv i bmod. En la implementació, aquestes 4 operacions coincideixen amb les tasques remotes. Lu0 és una tasca que només depen d'un bloc i el modifica. Fwd i bdiv són dos mètodes que necessiten 2 blocs de la matriu i en modifiquen un d'ells. L'últim tipus de tasca és el bmod, que utilitza tres blocs d'entrada, per tal de poder modificar-ne el tercer.

La figura 6.2 mostra el graf de depenencies que genera el seu codi principal:

```
Per k:= 0 fins NB fer {
    lu0(A[k][k]);
    Per j:= k+1 fins NB fer {
        fwd(A[k][k], A[k][j]);
    }
    Per i:= k+1 fins NB fer {
        bdiv(A[k][k], A[i][k]);
        Per j:= k+1 fins NB fer {
            bmod(A[i][k], A[k][j], A[i][j]);
} } }
```

Tal com es pot veure en el graf, només una tasca del mètode lu0 pot ser executada a la vegada. Les tasques bmod, fwd i bdiv de varies iteracions poden solapar-se a mesura que les dependències es solucionen. Les execucions es fan en 3 escenaris diferents:

- 1 sol tipus de màquines: totes les tasques poden executar-se en qualsevol recurs.

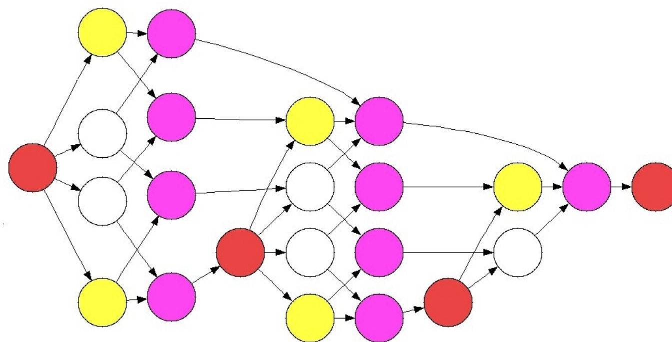

Figura 6.2: Graf de dependències del SparseLU

Per tant, a l'hora d'escalar es depen únicament del temps d'execució de les tasques lliures de dependències.

- 2 tipus de màquines: les tasques lu0 només poden executar-se en màquines d'arquitectura IA32 i les fwd, bdiv i bmod només poden fer-ho en màquines d'arquitectura PPC. Això farà que a l'hora d'escalar dinàmicament l'aplicació no s'hagin de demanar més d'una màquina IA32, però en canvi s'hauran de demanar tantes màquines PPC com siguin necessàries.

- 3 tipus de màquines: igual que en el cas anterior les tasques lu0 s'executen sobre IA32 i les fwd, bdiv i bmod sobre PPC. La diferència amb l'anterior escenari és que les tasques bdiv requereixen 2GB de RAM mentre que per executar tasques bmod i fwd només és necessari 1 GB. Això fa que les tasques bmod i fwd puguin executar-se sobre màquines de 1 GB i de 2 GB, però les bdiv són més restrictives i només s'executen sobre màquines amb 2GB de memòria. A l'hora d'escalar això pot tenir les seves conseqüències. Si en un moment donat hi ha moltes tasques bmod i fwd llestes esperant per executar i poques bdiv, es crearan màquines de 1GB per reduir el cost sense comprometre el rendiment.

Totes les execucions que es fan al llarg de les proves factoritzen matrius 40x40 elements, 10x10 blocs de 4x4 elements. Això dóna un graf de 145 tasques. Donada la forma del graf, és una mida que permet al runtime detectar quines són les seves necessitats i ampliar o reduir el nombre de màquines de cada tipus. El problema que hi ha és que el temps dedicat al càlcul és molt reduït. Això fa que l'aplicació acabi més ràpid del que és necessari per poder fer canvis en el número de recursos de que disposa el runtime.

Per evitar aquest problema una solució pot ser ampliar el tamany del bloc, i fer matrius de més elements amb el mateix nombre de blocs. D'aquesta manera, el temps de computació creix i l'overhead produït pel runtime és menys significatiu. De totes maneres, l'overhead per transferències segueix sent molt alt i s'ocupa molt d'espai de disc virtual. La solució per eliminar aquest overhead és afegir al càlcul de les tasques temps de computació. En aquest cas s'ha afegit un wait a dintre de cada tasca de 60 segons. Així el temps global de l'aplicació és suficientment gran per poder fer peticions al cloud.

Un altre efecte que s'aconsegueix amb aquest padding és que totes les tasques tardin més o menys el mateix. Comptant l'overhead del runtime i de transferir els fitxers, els temps mitjos d'execució són els següents:

- lu0: 61.559s

- bdiv: 61.352s
- fwd: 61.480s
- bmod: 61.515s

Amb SparseLU el benchmark disposa d'una aplicació amb un nombre limitat de tasques de duració similar amb una forta dependència de càlcul. El màxim nivell de paral·lelisme es trobar a la part mitja de l'execució, tot i que la forma d'aprofitar les màquines depen de quin dels 3 escenaris es faci servir.

6.2.2 Hmmer

HMMER és una suite d'aplicacions utilitzada en l'àmbit de la bio-informàtica. La suite es fa servir per analitzar models HMM (Hidden Markov Model), que representen famílies de proteïnes. Una de les aplicacions més important és l'HMMPfam que llegeix un conjunt de seqüències d'aminoàcids i compara cada una de les seqüències contra una base de dades buscant seqüències similars en el model. L'objectiu final de l'aplicació és trobar, per cada seqüència, el conjunt de famílies de la base de dades que més similituds tenen per cada una de les seqüències. És un procés de càlcul molt intens, però molt paral·lelitzable.

Per paral·lelitzar l'aplicació l'algoritme es divideix en 3 fases:

- Segmentació: els fitxers amb les seqüències i la base de dades és divideixen en funció del número de processadors i de memòria disponible.
- Execució: S'executa el binari HMMPfam de la suite per cada parella de fragments de base de dades i seqüència
- Reducció: els resultats d'aquestes execucions s'ajunten per donar un sol resultat final.

Una execució d'aquesta aplicació dóna un graf com el que mostra la figura 6.3. L'execució de la fase de segmentació és fa de manera seqüencial en el master de COMPSs. El nivell més alt de l'arbre correspon a la segona fase. Els nivells inferiors representen com la fase de reducció ajunta els diferents fragments en un únic resultat.

De fet, en la implementació existeixen 2 tipus de tasca de reducció en funció dels fragments que ajunti: si s'ajunten fragments de la mateixa base de dades o si s'ajunten fragments amb les mateixes seqüències. El temps de cada unió varia en funció de quin dels 2 es faci servir i de la quantitat de resultats parcials ajuntats. Així doncs, hi ha 3 tipus de tasca:

- hmmpfam: executa en el worker el binari HMMPfam amb un fragment de la base de dades i un fragment del conjunt de seqüències generant un resultat parcial. Els fragments estàn repartits de forma equitativa, per tant, totes les tasques tarden més o menys el mateix en executar-se.
- mergeSameDB: uneix dos resultats parcials que provinguin del mateix fragment de la base de dades generant un nou resultat parcial.
- mergeSameSeq: ajunta dos resultats parcials que provinguin del mateix conjunt de seqüències generant un nou resultat parcial.

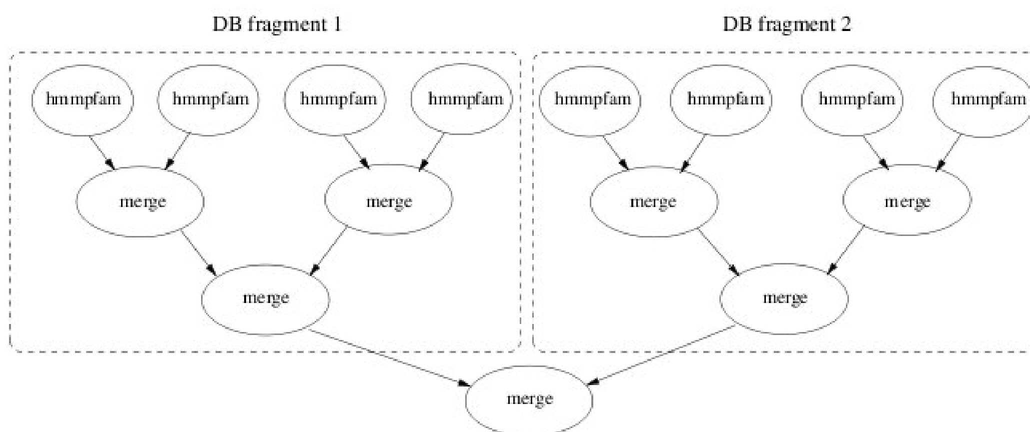

Figura 6.3: Graf de dependències de l'aplicació HMMER

L'última unió de resultats parcials sempre es fa en el master.

Totes les execucions es fan utilitzant la base de dades SMART que conté 725 models amb longituds d'entre 11 i 971 aminoàcids. El que fa variar el tamany de l'entrada i, per tant, l'amplada i alçada del graf és el conjunt de referències. Sempre es fa servir un subconjunt amb un màxim de 4096 seqüències de les que conté el fitxer UniParc. Independentment del tamany d'aquests i del nombre de màquines totes les proves es fan en dos escenaris possibles:

- 1 sol tipus de màquina: totes les tasques poden executar-se en qualsevol màquina que el runtime tingui disponible.
- 2 tipus de màquines diferents: les tasques hmmpfam obliguen a ser executades en una màquina amb arquitectura IA32, mentre que els merge (mergeSameDB i merge-SameSeq) ho han de fer en màquines PPC. D'aquesta manera, a l'inici de l'aplicació es demanen màquines d'arquitectura IA32 i, poc a poc, aquest nombre es va reduint en favor de màquines PPC si el tamany i nombre dels resultats parcials fa que una sola màquina PPC no sigui suficient.

Hmmer aporta al benchmark una aplicació molt dependent del càlcul amb tasques de diferent duració segons el tipus i moment de l'execució. Que té un fort paral·lelisme a l'inici de l'execució que poc a poc va decrementant-se. Segons l'escenari i l'entrada que es fa servir en la prova, el nombre i les característiques de les màquines requerides pot variar.

6.2.3 Stream

Stream és un programa simple i sintètic que forma part del HPC Challange. El programa s'utilitza per mesurar l'ample de banda de memòria sostenible i la corresponent relació de càlcul. L'aplicació executa iterativament 4 funcions: copy, scale, add i triad; que operen sobre la mateixa posició de 3 vectors diferents. Per poder portar l'aplicació a COMPSs el que s'ha fet és dividir aquest vector en N petits fragments i aplicar les funcions sobre aquests. El graf de dependències resultant és el de la figura 6.4. En la versió per OpenMP de l'aplicació cada grup de funcions es feia de manera paral·lela, de manera que primer es fan totes les tasques del tipus copy de la primera iteració, després les scale, ... Un

cop acabada la primera iteració es repetia el procés per les iteracions restants. Al portar l'aplicació a COMPSs s'ha optat per deixar al runtime la planificació de les tasques dintre d'una mateixa iteració, de manera que les tasques puguin executar-se de la manera més àgil possible. Per altra banda, s'ha mantingut l'ordre de les iteracions esperant a que una acabi per començar la següent. Les execucions que es fan utilitzen vectors de 150 blocs de

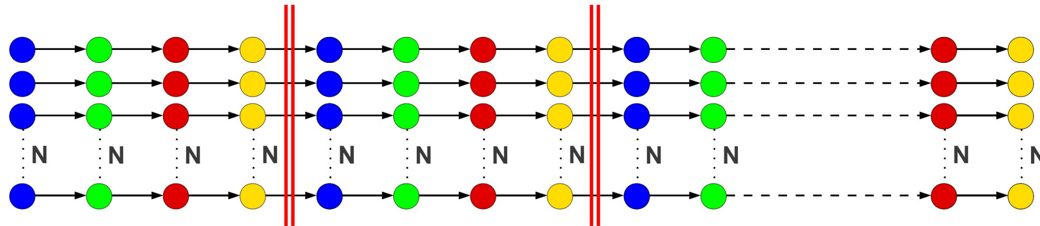

Figura 6.4: Graf de dependències de l'aplicació Stream

3000 elements i es fan 3 iteracions per cada bloc. En total el graf resultant tindrà 1800 tasques, però només podran executar-se'n un màxim de 150 de forma paral·lela, sempre i quan el nombre de recursos ho permeti. Igual que pels altres casos, les proves es fan 2 escenaris concrets:

- 1 tipus de màquina: permet veure que a l'inici de cada execució entren 150 tasques de cop que obliguen al runtime a crear noves màquines. Cap al final de la iteració el volum de tasques baixa degut a que encara no s'han acabat totes les tasques necessàries per superar el Barrier. Això obre la possibilitat a apagar alguna màquina que no sigui aprofitada.

- 2 tipus de màquina: les funcions copy i scale són forçades a executar-se en una màquina amb arquitectura IA32, mentre que les tasques add i triadd utilitzen una arquitectura PPC. Això provoca que a l'inici de l'aplicació totes les tasques que entrin demanen màquines IA32, la qual cosa obliga a augmentar-ne el nombre. Les tasques copy i scale s'executen a les noves màquines . A mesura que acabin les tasques scale, el nombre de tasques add augmenta obligant a crear noves màquines amb arquitectura PPC a mitja iteració i lentament es van apagant les màquines amb arquitectura IA32. Aquestes noves màquines PPC es mantenen fins al final de la iteració que es van apagant les màquines en funció de la política periòdica. Al començar una nova iteració el procés començarà un altre cop.

6.3 Rendiment de la planificació

A l'hora d'avaluar un runtime, un dels factors a tenir més en compte és el rendiment que tenen les aplicació que l'utilitzen. En cas de tenir la mateixa combinació de màquines, el factor que més hi pot influir és la planificació. Les decisions que es prenen en el TaskScheduler poden fer variar el nombre de transferències i els temps d'execució i d'espera de cada tasca. Tot això repercuteix directament sobre el temps total d'execució

Tal com s'explica a l'apartat 5.4, tota la planificació de tasques canvia respecte la versió de Grid. Això pot provocar pèrdues de rendiment en les aplicacions. Per tal de poder analitzar millor tota la part del projecte relacionada amb l'extensió per Cloud i avaluar la tecnologia, s'ha de veure quina és la variació que s'obté degut a aquest canvi.

L'experiment que permet analitzar aquestes diferències consisteix en executar la versió per Grid de COMPSs sobre un conjunt de màquines virtuals com si fossin màquines físiques i agafar els temps d'execució d'algunes aplicacions en diferents escenaris. El mateix s'ha de fer pel prototip del projecte. Cal fer que aquesta segona versió actuï en les mateixes condicions que ho feia la primera i no pugui treure avantatge del fet de poder incrementar el nombre de màquines que utilitza. Per això cal s'indica al runtime que no pot demanar noves màquines (màxim de màquines en Cloud 0).

Per altre banda, el temps de crear les màquines virtuals a l'inici de l'execució, fa que la versió Cloud tingui un factor en contra. Cal eliminar aquest temps i, per això, s'indicara al runtime que es volen 0 recursos en el cloud inicialment i que pot utilitzar com si es tractessin de recursos físics els mateixos que es fan servir per avaluar la versió Grid.

La taula 6.1 mostra els temps obtinguts a l'executar les aplicacions Hmmer amb 512 i 4096 seqüències d'entrada i SparseLU amb diferents configuracions de màquines utilitzables. Com a entorn de proves de tots ells s'ha fet servir un Cloud d'un sol node, PcTinet, per tant el màxim nombre de nodes amb els que es treballa és 4, tot i que, es faran servir diferents restriccions per les tasques tal i com s'explica al capítol 6.2.

Applicació	Escenari	GRID	Cloud
Hmmer-512seq	1 màquina	685.12	683.66
Hmmer-512seq	2 màquines iguals	367.16	375.72
Hmmer-512seq	2 màquines de 2 tipus diferents	652.01	650.55
Hmmer-512seq	4 màquines iguals	223.96	232.92
Hmmer-512seq	4 màquines de 2 tipus diferents	358.12	356.23
Hmmer-4096seq	1 màquina	5577.44	5339.12
Hmmer-4096seq	2 màquines iguals	2947.97	2944.97
Hmmer-4096seq	2 màquines de 2 tipus diferents	5046.64	4994.97
Hmmer-4096seq	4 màquines iguals	1795.12	1842.38
Hmmer-4096seq	4 màquines de 2 tipus diferents	2647.14	2617.93
SparseLU	1 màquina	8839.02	8836.93
SparseLU	2 màquines iguals	4720.24	4724.42
SparseLU	2 màquines de 2 tipus diferents	8420.16	8418.22
SparseLU	4 màquines iguals	2713.14	2720.59
SparseLU	4 màquines de 2 tipus diferents	3274.84	3248.36

Taula 6.1: Temps (segons) d'execució aïllant la planificació

Les execucions amb 1 sola màquina mostren el punt fort d'aquesta planificació: la velocitat de decidir la següent tasca que s'executarà en un node concret. En tots els casos en que la decisió de quin node executarà una tasca està forçada degut al nombre de màquines que poden executar-la el rendiment de la nova planificació és més alt que l'anterior. Mentre que en el moment en que el TaskScheduler té més llibertat de decisió apareixen els temps favorables a l'antiga planificació.

El cas de l'execució del Hmmer amb 4096 seqüències i un sol worker és bo per ressaltar el punt fort d'aquesta planificació. En aquest cas, el temps obtingut és 238 segons menor. Observant la forma del graf i el nombre de tasques de l'execució es pot veure que ja d'inici el planificador disposa de 195 tasques. En la planificació de Grid, al final de l'execució d'una tasca, el TaskScheduler compta per cada una de les tasques quin és el número de fitxers que han de ser utilitzats i que el node té disponibles. De totes elles escull la que en

tingués més. La versió per Cloud, aquest procés es redueix a consultar un sol cop quin és la tasca que ha d'executar-se en aquell slot. Això redueix el temps de consulta i explica la diferència de temps que hi ha entre una versió i l'altre. Es pot dir que l'escalabilitat en quant a nombre de tasques que és capaç d'assumir la planificació és més alta.

Per contra, al donar més llibertat a l'hora d'escollir recurs on executar la tasca, aquest avantatge perd pes. La mateixa aplicació amb la mateixa entrada pot ser l'exemple més clar. Hmmer amb 4096 seqüències executat sobre 4 workers de les mateixes característiques permet al TaskScheduler assignar una mateixa tasca a 4 recursos diferents. En la versió Grid de COMPSs la tasca s'assigna a un recurs quan aquest està lliure i és la tasca que depen de més fitxer que el recurs té. En la versió de Cloud, la planificació ha assignat la tasca a un node en el moment de l'entrada sense preocupar-se del nombre de transferències que és necessari fer per poder executar-la. L'increment del nombre de transferències fetes innecessàriament provoca una pèrdua de rendiment major que la millora obtinguda per la velocitat de la planificació.

6.4 Dependència del Cloud Provider

Un cop analitzada la política de planificació. Un altre paràmetre que es vol analitzar i que influeix fortament sobre el temps d'execució de l'aplicació és el temps que tarda el Cloud Provider en donar resposta al runtime i com es gestionen les màquines virtuals dintre del seu cluster.

L'objectiu d'aquesta prova no és analitzar directament el rendiment del runtime, ni les seves polítiques de planificació, simplement es vol veure la influència del Cloud Provider. Per evitar que tota la resta els canvis en el runtime influeixin en el resultat, s'utilitza el prototip que s'ha fet en la quarta iteració de la metodologia. En aquesta versió encara es manté tota la planificació igual que en la versió per Grid i no es demanen ni alliberen recursos del Cloud fins que s'acabava l'execució que espera a poder apagar tots els recursos que havia demanat.

Com s'ha explicat al capítol 5.6.1, a l'inici del runtime es dóna l'opció d'escollir el nombre de màquines virtuals que es vol demanar al cloud a l'inici de l'execució. Per poder comparar els temps de la versió Grid i de la Cloud, es demanen el mateix nombre de màquines a l'inici com se n'ha utilitzat a la versió Grid. Si tal com es fa la petició, el Cloud retornés la màquina el resultat hauria de ser el mateix. Com que això no és possible, és evident que pel fet de demanar un recurs a l'inici de l'execució es tardarà més a poder acabar una execució ja que hi haurà una estona en que no es tenen tantes màquines com seria necessari i durant aquella estona no es pot treure rendiment del paral·lelisme de dades que té l'aplicació.

La taula 6.2 conté els temps d'execució per la versió Grid. En el cas de treballar amb 1, 2 i 4 workers, els recursos queden mapejats sobre una sola màquina física, mentre que en el cas de treballar amb 8 workers, aquests utilitzen 8 CPU de dos nodes físics. D'aquesta manera s'elimina qualsevol col·lisió deguda a utilitzar més processadors virtuals que físics.

Per comparar-ho amb la versió Cloud, es pot observar els valors de la taula 6.3, que conté els temps d'execució per 1, 2, 4 i 8 nodes tots ells mapejats sobre una sola màquina física de 4 processadors.

Tal i com s'anticipa, el temps de resposta es veu limitat pel temps de creació de la màquina virtual i ho mostren clarament els nombres a l'utilitzar un sol worker, la versió

	1 worker	2 workers	4 workers	8 workers
512 seqüències	685.12	367.16	223.96	139.39
1024 seqüències	1360.73	734.82	452.27	255.98
2048 seqüències	2719.25	1464.88	905.44	531.76
4096 seqüències	5577.44	2947.97	1795.12	1068.81

Taula 6.2: Temps (segons) d'execució per HMMER de 512, 1024, 2048 i 4096 sequències en la versió Grid

	1 worker	2 workers	4 workers	8 workers
512 seqüències	841.47	567.93	661.54	1123.93
1024 seqüències	1487.49	945.99	897.69	1133.28
2048 seqüències	2875.11	1690.54	1347.48	1385.49
4096 seqüències	5737.29	3157.84	2262.51	2321.25

Taula 6.3: Temps (segons) d'execució per HMMER de 512, 1024, 2048 i 4096 sequències en un Cloud d'un node

Grid és al voltant de 143 segons més ràpida ja que pot començar a executar directament. Existeix una desviació estàndard de 7.38, un 5%. Tot i que alguns dels casos descartats tenien fins a 200 segons de retard.

A part d'això, es poden veure també altres límits que imposa el Cloud. El més evident de tots es pot veure en l'execució en un Cloud d'un node del Hmmer amb 512 seqüències d'entrada. L'aplicació escala segons el previst fins als 2 workers a partir d'aquell moment, els temps deixen de baixar i comencen a incrementar-se tant per 4 com per 8 workers. El límit el segueix imposant el temps de creació de les màquines. Quan s'acumulen les peticions, els temps de creació també s'acumulen. Si és llancen una quantitat de peticions que tardin a servir-se més temps que el que dura l'execució, el runtime haurà d'esperar a que s'acabin aquestes creacions per poder apagar les màquines. El cas dels 4 workers (143*4=572) és sospitós, però el dels 8 workers (8*143 =1144) ho confirma, tal i com demostra el diagrama de Gantt de les creacions de la figura 6.5,extret d'una de les execucions del cas.

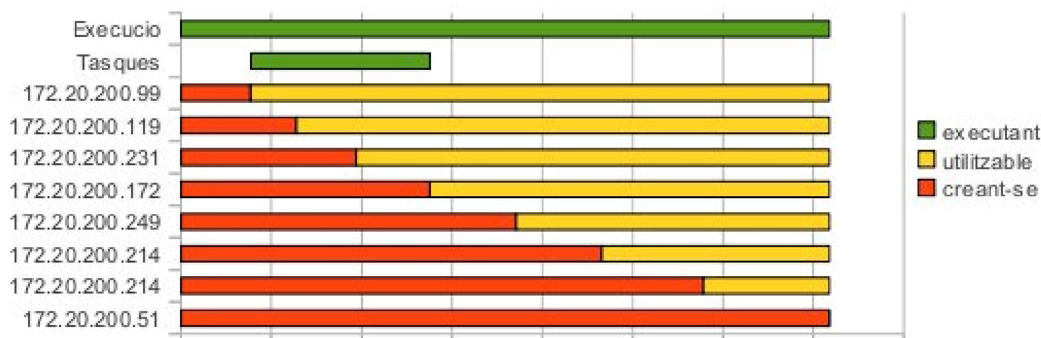

Figura 6.5: Diagrama de Gantt per una execució del Hmmer amb 512 seqüències i 8 workers

Un altre problema detectat és la quantitat de processadors virtuals que es fan servir per màquina física. PcTinet té 4 processadors, quan es mapegen més de 4 màquines d'un processador en el mateix node, hi ha més processadors virtuals que físics. Arribats al punt

en que existeixen tantes màquines virtuals com reals, el rendiment de l'aplicació comença a decaure. El problema que hi ha és que en les tasques de Hmmer el processador està tota l'estona ocupat, per tant el temps no pot millorar, ja que només 4 màquines poden estar treballant a la vegada. De fet, la pèrdua de rendiment entre els 4 i els 8 workers ve donada pel temps ocupat pel planificador del sistema operatiu de la màquina física per fer el canvi de màquines virtuals que estan executant.

En algun cas el fet de tenir més processadors virtuals que físics podria ser-nos útil. Per exemple, en aplicacions amb tasques que tinguin gran part del pes dedicades a entrada i sortida. Mentre les tasques esperen a que les seves peticions siguin tractades, el processador queda lliure i pot ocupar-lo una altra tasca.

La taula 6.4 mostra els temps en cas de fer servir un cloud de 2 nodes. Segueix existint el problema dels 143 segons per encendre una màquina, però ara es poden crear-ne dues de forma paral·lela, així doncs el temps per crear 2 màquines virtuals és la meitat d'abans. El límit de temps per crear 4 workers que existieix al Hmmer amb 512 seqüències amb un sol node desapareix i passa només als 8 workers. En un Cloud de N nodes es podrien encendre N màquines virtuals de cop i, per tant, desapareixeria el límit.

	1 worker	2 workers	4 workers	8 workers
512 seqüències	842.57	508.62	436.53	568.57
1024 seqüències	1487.34	860.45	735.39	607.37
2048 seqüències	2873.05	1630.13	1184.57	883.42
4096 seqüències	5736.98	3090.48	2075.89	1418.79

Taula 6.4: Temps (segons) d'execució per HMMER de 512, 1024, 2048 i 4096 sequències en un Cloud de dos nodes

Al tenir 2 nodes, el gestor disposa de 8 processadors reals, per tant, el problema de tenir més processadors virtuals que reals desapareixeria de la mateixa manera, escalant el cloud amb N nodes i que el gestor pugui oferir M>N processadors virtuals.

Els grafs que mostra la figura 6.6 permeten fer la comparació de temps entre les execucions que utilitzen 1 node i 2 nodes. Mantenint la mateixa planificació de tasques, en incrementar el nombre de nodes del Cloud, la línia del graf tendiria a fer una corba paral·lela a la que dóna la versió Grid de COMPSs, 143 segons per sobre.

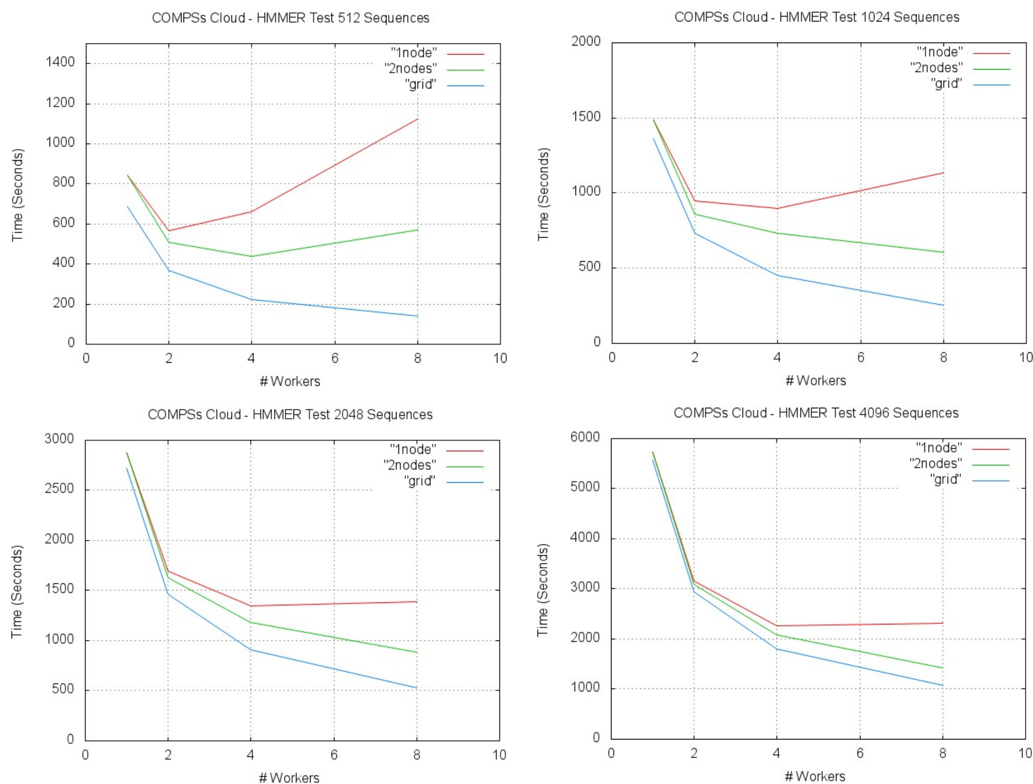

Figura 6.6: Comparació entre número de workers solicitats a l'inici i el temps utilitzats per executar Hmmers de 512, 1024, 2048 i 4096 seqüències amb un cloud d'un i dos nodes i un grid.

6.5 Escalar aplicacions

Un cop comprovades la planificació i l'efecte que té el cloud provider sobre el prototip, queda per veure si les polítiques definides funcionen correctament. L'objectiu d'aquest punt és veure que la creació de màquines es fa d'una manera ordenada i coherent. El nombre de màquines ha de creixer en funció de la càrrega que tenen les màquines capaces de soportar execucions d'un mètode i de la càrrega d'aquell mètode que hi ha en aquell moment. La segona meta que persegueix aquesta prova és comprovar la correcta destrucció de màquines. A mesura que el nivell de càrrega d'un mètode decrementa també ho ha de fer el nombre de màquines capaces de soportar-lo, tal i com s'explica al capítol 5.

Aquesta prova s'ha fet utilitzant un Cloud amb tots dos nodes PcTinet i PcPerot per tal de permetre al runtime demanar tantes màquines com li sigui possible, limitant aquest nombre a 8 màquines virtuals d'un sol processador (ja que només existeixen 8 processadors reals).

Per poder portar a terme aquest experiment s'executaran les tres aplicacions en tots els escenaris descrits a l'inici del capítol. Per tal d'ajudar a veure l'evolució de la quantitat de recursos utilitzats en cada un dels escenaris en que s'ha fet la prova es fan servir unes imatges que contenen dos gràfics. El grafic que es pot veure a l'esquerra de la figura representa l'evolució del nombre de màquines que el runtime ja pot fer servir per executar-hi tasques. Cada línia representa el nombre de màquines capaces d'executar mètodes sota unes restriccions similars.

El graf que queda a la dreta de la figura representa l'evolució de les càrregues que

pot assumir el runtime (Temps de creació d'una nova màquina multiplicat pel nombre de recursos que compleixen certes restriccions) i la càrrega real (nombre de tasques multiplicat pel temps mig d'execució d'una tasca del mètode) que té en aquell moment. La càrrega de les tasques per cada un dels mètodes queden representats en una sola línia, la càrrega assumible pels recursos queda representada per la part ombrejada del graf.

6.5.1 Hmmer

La primera aplicació que es fa servir per veure les proves és el Hmmer, explicat a 6.2.2. Resumint ràpidament l'aplicació, té 3 mètodes: hmmpfam, mergeSameSeq, mergeSameDB; a l'inici de l'aplicació, s'obre molt paral·lelisme amb tasques del mètode hmmpfam i, poc a poc, es generen tasques de reducció. Això provoca una gran una càrrega a l'inici que va decrementant.

Existien 2 tipus d'escenari possible: el que totes les tasques tenen les mateixes restriccions i el que les tasques de merge no poden executar-se en màquines de hmmpfam, ni a la inversa. per cada un d'aquests escenaris s'han fet proves amb diferents tamanys d'entrada per veure la resposta que es dóna en fució de com es veu sobrepassada la capacitat del runtime.

1 tipus de tasca

Tots tres mètodes poden executar-se en totes les màquines existents. Com es pot veure en les 4 proves que s'han fet, tot el sistema creat per tal de poder escalar el sistema funciona correctament en el cas de tenir un nivell molt alt de càrrega comparat amb el que pot assumir. Poc després de començar les execucions, el runtime ja és conscient del que es tarda a solucionar totes les tasques acumulades i demana noves màquines. A mesura que l'execució avança va acumulant més peticions al Cloud Provider.

Les figures 6.8, 6.9, 6.10 i 6.11 mostren el comportament del sistema tal i com s'explica a la descripció de la prova. Per interpretar el gràfic de càrregues es farà servir la llegenda que mostra la figura 6.7. Les barres verticals negres representen el moment en que la política demana una nova màquina, l'efecte d'aquesta petició en la càrrega que pot acceptar el runtime no es veurà fins que el Cloud Provider retorni la màquina.

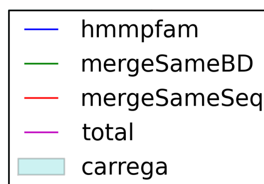

Figura 6.7: Llegenda pels gràfics d'execucions Hmmer amb 1 sol tipus

Amb aquestes 4 execucions es pot veure com es controla l'accés de càrrega inicial. Poc a poc es va incrementant es van demanant màquines capaces de solucionar tasques. En cap cas, s'ha vist que una màquina sigui creada i destruïda sense haver executat cap tasca, ni en cap cas s'ha demanat una màquina i abans de ser proveïda s'ha destruit una altra del mateix tipus.

Un altre fet que es pot observar és que hi ha canvis sobtats en la càrrega assumible en 2 casos:

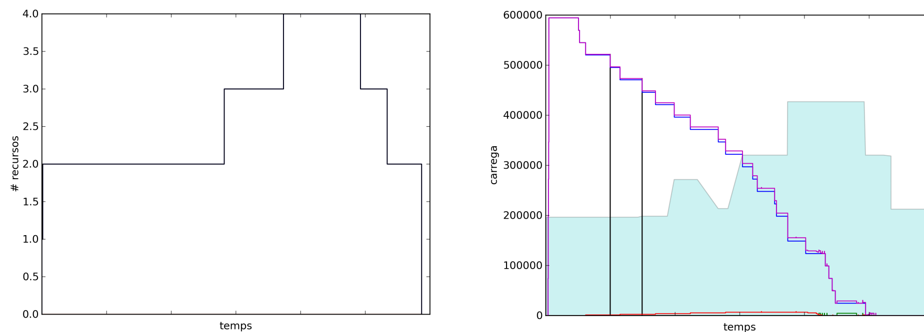

Figura 6.8: Evolució dels recursos en un hmmer de 512 seqs. i 1 tipus de tasca

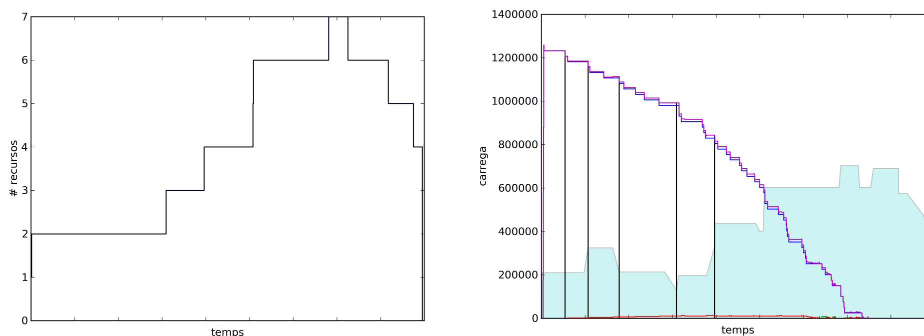

Figura 6.9: Evolució dels recursos en un hmmer de 1024 seqs. i 1 tipus de tasca

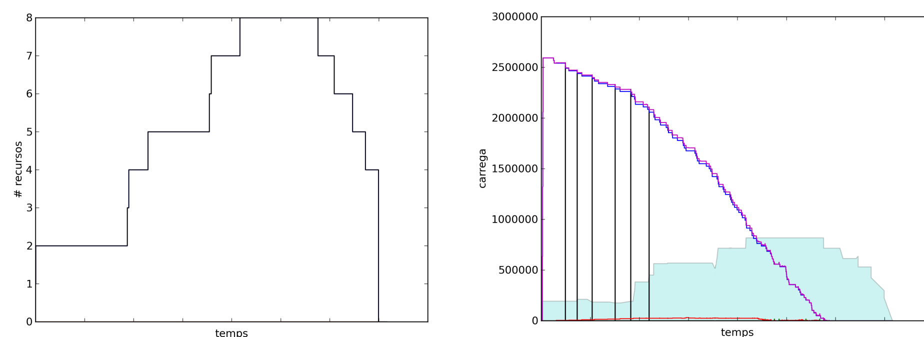

Figura 6.10: Evolució dels recursos en un hmmer de 2048 seqs. i 1 tipus de tasca

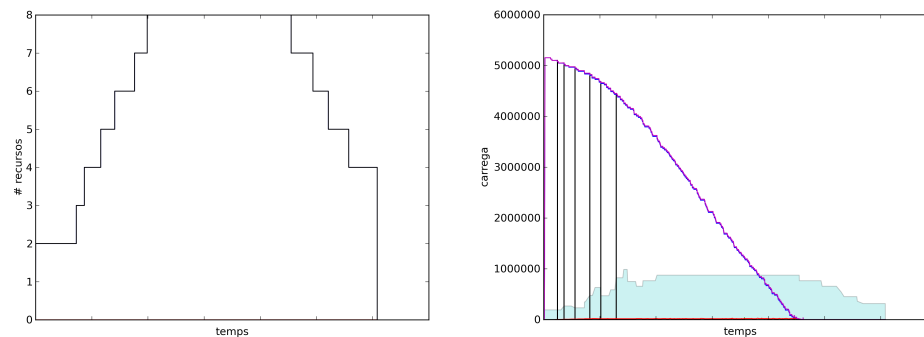

Figura 6.11: Evolució dels recursos en un hmmer de 4096 seqs. i 1 tipus de tasca

- al variar el nombre de màquines.
- al demanar una màquina.

El primer cas és evident, al mantenir el temps de creació d'una nova màquina, la càrrega assumible és directament proporcional al nombre de màquines que tingui el runtime. Si es destrueix una màquina, la càrrega assumible disminueix. Per contra, si s'incorpora una nova màquina virtual als recursos disponibles, la càrrega assumible augmenta.

El segon cas és menys evident. Quan es demana una màquina, el temps de resposta de la següent petició sobre el mateix node és el temps de crear una nova màquina més el temps de donar resposta a totes les peticions anteriors. La figura 6.12 mostra com evoluciona el temps de respondre una petició a l'acumular-se vàries demandes sobre un mateix node. Cada cop que el temps augmenta de cop significa que la creació d'una nova màquina ha estat assignada al node.

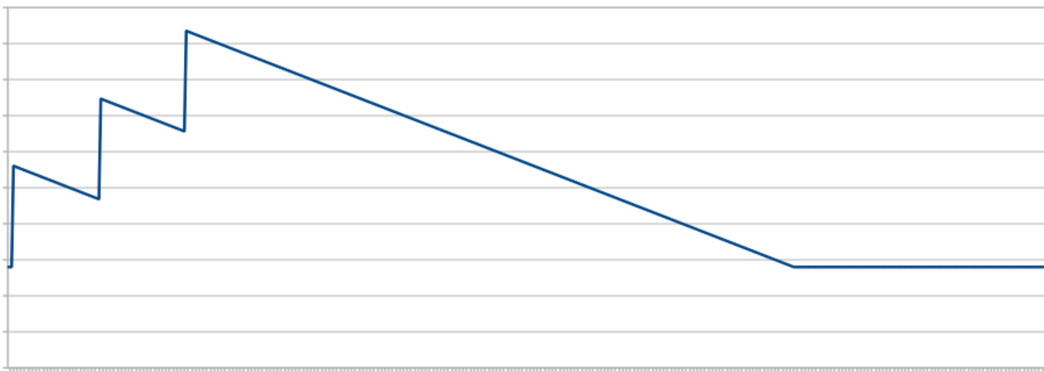

Figura 6.12: Evolució del temps de servir una petició en un node on s'acumulen peticions

Amb el SimpleScheduler d'EMOTIVE Cloud, al fer una petició sempre es canvia el node que tracta la següent petició. Aquest segon node pot ser més ràpid o més lent que l'anterior, per tant, el temps de resposta també canvia. Al fer servir només dos nodes per fer les proves, quan s'acumulen dues peticions sense temps de donar resposta a la primera, el temps de crear una tercera màquina serà el temps que tardarà el node ha crear la tercera màquina més el que falti per servir la primera petició. La figura 6.13 mostra com evoluciona el temps de resposta a una petició en el cas de tenir dos nodes. Les línies del graf representen el temps que tardaria en respondre cada node. L'àrea de color blau és el temps que el Cloud tardaria a respondre la següent petició.

2 tipus de tasca

Les quatre execucions anteriors han permès demostrar que el runtime és sensible a la quantitat de càrrega global del sistema i que era capaç de resoldre el problema demanant noves màquines de forma coherent. Aquest escenari permet comprovar si el runtime és capaç de demanar noves màquines en funció del mètode origen de la càrrega i mantenir el control en cas de tenir una ràfaga de tasques a l'inici. A l'hora de representar-ho gràficament, s'hauran de representar de manera diferent els recursos dedicats al hmmpfam i als dos merges, tant en el graf de recursos com en el del càrregues en el que s'haurà de veure quina càrrega és capaç d'assumir el runtime per cada mètode. La llegenda que conté la figura 6.14 goberna a tots els gràfics resultants d'aquesta prova.

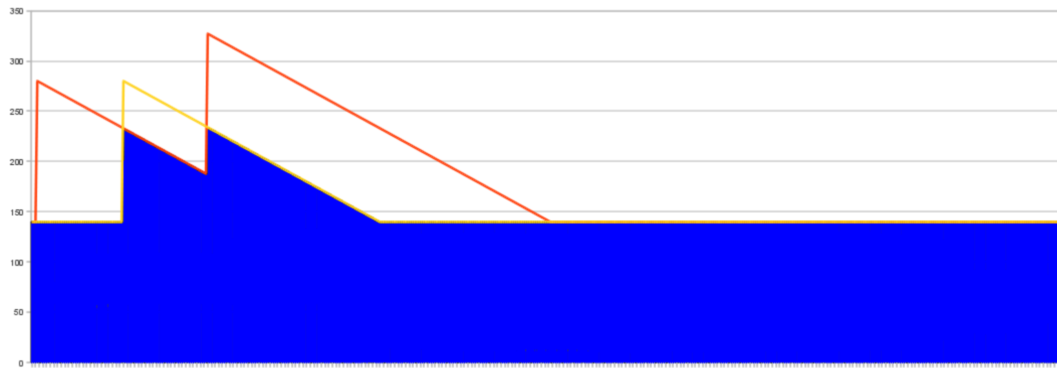

Figura 6.13: Evolució del temps de servir una petició en un Cloud amb 2 nodes

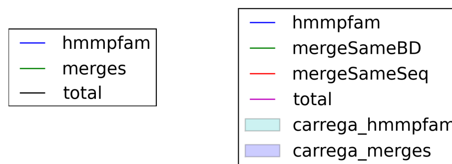

Figura 6.14: Llegendes pels gràfics d'execucions Hmmer amb 2 tipus

El blau mostrat per la carrega dels merges no correspon al que apareix en les imatges. El motiu d'això és que la càrrega de les màquines d'aquest tipus és sempre inferior a les del hmmpfam. Això fa que el color cian de la càrrega del hmmpfam i el blau dels merges apareixi barrejat donant una coloració més clara a tota l'extensió de la càrrega dels merges. El mateix passa en tots els altres grafs que mesclin aquests colors al representar les càrregues.

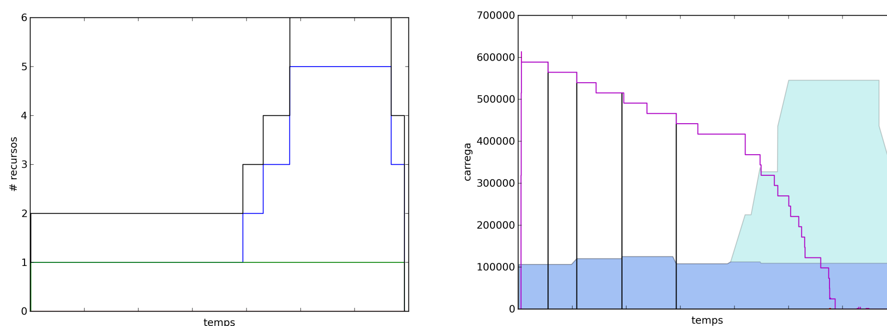

Figura 6.15: Evolució dels recursos en un hmmer de 512 seqs. i dos tipus de tasca

Al tenir una màquina dedicada exclusivament a realitzar els merge (tasques més curtes), difícilment s'arriben a acumular un nombre de tasques que faci necessària la creació d'una nova màquina. I en cap de les execucions s'ha donar el cas en que s'hagi hagut de fer.

Per la banda dels hmmpfam, el nombre de tasques inicial si que ha representat un problema. Amb una sola màquina no es pot donar abast a tota la càrrega d'una manera eficient així que és necessari demanar més màquines capaces de solucionar aquest tipus de tasques. Tal i com mostren tots els grafs de recursos, el nombre de màquines demanades per processar la càrrega d'una forma més ràpida es demanen en funció d'aquest nivell i únicament serveixen per executar tasques hmmpfam.

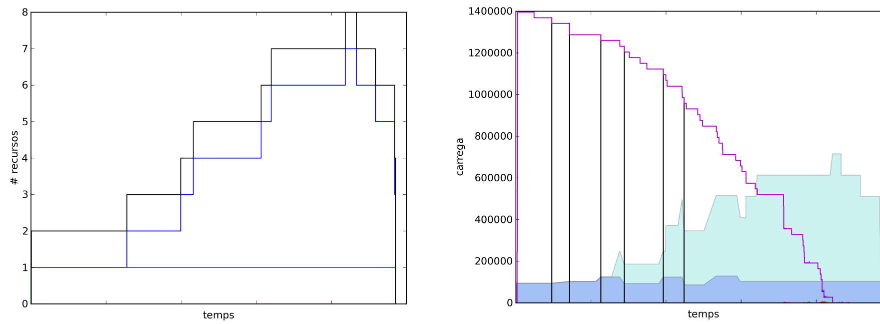

Figura 6.16: Evolució dels recursos en un hmmer de 1024 seqs. i dos tipus de tasca

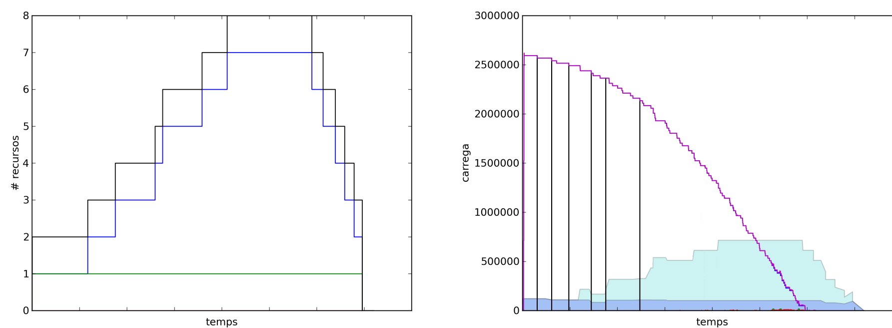

Figura 6.17: Evolució dels recursos en un hmmer de 2048 seqs. i dos tipus de tasca

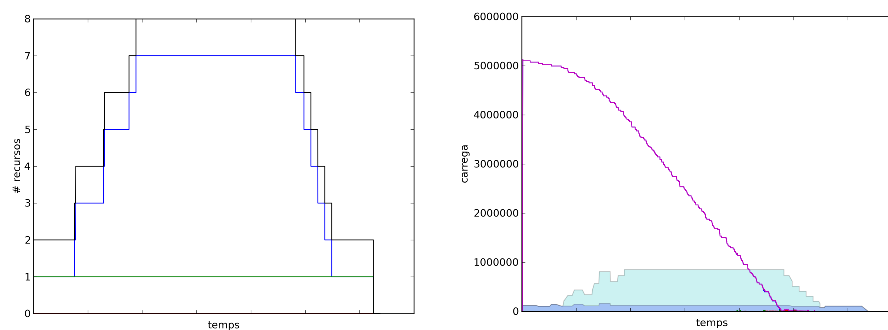

Figura 6.18: Evolució dels recursos en un hmmer de 4096 seqs. i dos tipus de tasca

6.5.2 SparseLU

SparseLU és una aplicació que executa quatre tipus de tasca: lu0, fwd, bdiv i bmod. L'aplicació comença executant una sola tasca de tipus lu0, que dóna lloc a l'execució de varies tasques de tipus fwd i bdiv. Quan aquestes van acabant per cada parell de tasques fwd-bdiv acabades es crea una nova tasca bmod. Aquest esquema es repeteix en cada iteració en que cada tasca de la nova iteració depen d'una de les bmod de l'anterior.

Tal i com es diu a l'apartat 6.2.1, la prova s'executa en tres escenaris diferents. En el primer totes les tasques poden executar-se en totes les màquines. En el segon les tasques lu0 es diferencien de la resta executant-se en una màquina especial. En el tercer escenari lu0 segueix executant-se en una màquina especial i bdiv demana una màquina de característiques similars a fwd i bmod, però demanant més memòria, de manera que bmod i fwd puguin executar-se en una màquina que executi bdiv, però no a l'inrevés.

1 tipus de tasca

Totes les tasques poden executar-se en qualsevol de les màquines. Com en el cas del Hmmer amb 1 sol tipus de màquina, la càrrega global és la que marca el número de màquines que necessita el runtime. La figura 6.20 mostra l'evolució del nombre de recursos que utilitza el runtime i els nivells de càrrega i suportable per cada mètode. La llegenda per interpretar el segon gràfic es troba al la figura 6.19. A diferència del cas del Hmmer, es

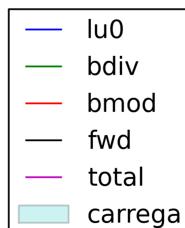

Figura 6.19: Llegenda pels gràfics d'execucions SparseLU amb 1 sol tipus de màquina

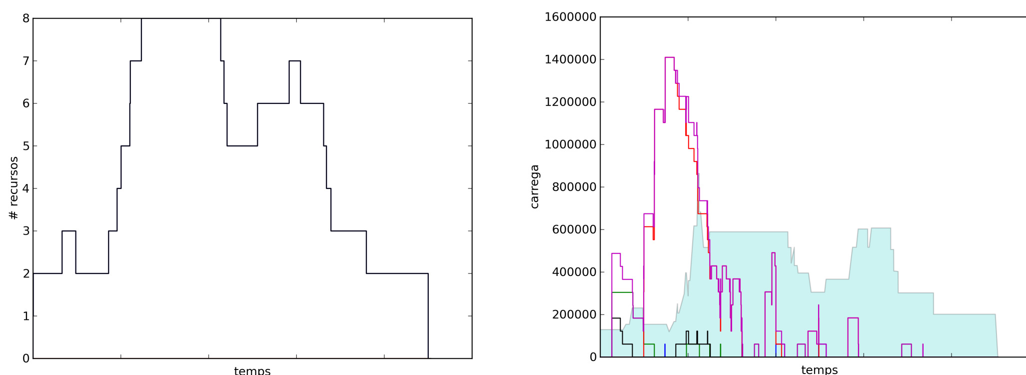

Figura 6.20: Evolució dels recursos en un SparseLU amb 1 sol tipus

pot veure com el runtime és capaç de solicitar més recursos no només quan es veu molt superat per la càrrega inicial de l'aplicació, sinó que a mesura que avança l'aplicació i es va acumulant més càrrega, el runtime també va demanant noves màqunes per poder resoldre la factorització. A mesura que va resolent els pics de càrrega que li arriben, el nombre de màquines decrementa fins que li arribi una nova ràfaga de tasques.

2 tipus de tasca

El segon cas és en el que es fan servir dos tipus de màquina. El primer per executar únicament tasques lu0 i el segon per executar bmod, bdiv i fwd. L'objectiu d'aquest escenari és veure que el runtime respon creant màquines del segon tipus que és l'únic que té carrega de feina. La llegenda de la figura 6.21 és la clau per interpretar el graf d'evolució de 6.22.

En els gràfics es pot veure com no apareix la càrrega deguda al mètode lu0. Això és perquè només se'n pot haver de planificar 1 a la vegada i té una màquina dedicada exclusivament a resoldre-la. Al no poder-se donar càrrega d'aquest tipus no es demanen noves màquines.

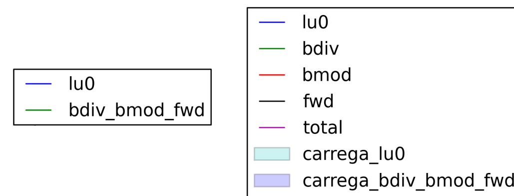

Figura 6.21: Llegendes pels gràfics d'execucions SparseLU amb 2 tipus

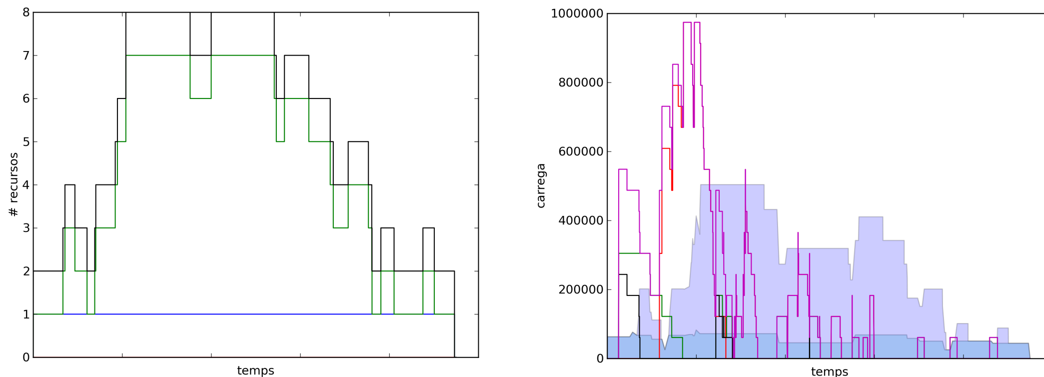

Figura 6.22: Evolució dels recursos en un SparseLU amb 2 tipus

Per contra una sola màquina dedicada a la resta de mètodes no és suficient i el runtime demana més màquina al Cloud Provider per tal d'augmentar els recursos que executen en paral·lel aquest tipus de tasques, de la mateixa manera que ho fa per l'exemple anterior. Amb aquest exemple, es pot veure que el runtime és capaç de demanar recursos tenint en compte les necessitats de l'aplicació en cada moment i no només quan hi ha una demanda molt forta. A més a més, demostra que no només escala bé a l'hora d'incrementar recursos sinó que també és capaç d'alliberar-los en el moment correcte i tornar a demanar-ne més quan torni a fer-ne falta.

3 tipus de tasca

El tercer escenari és més restrictiu. Bdiv s'executa en una màquina que demana més recursos que bmod i fwd. La prova comença amb 2 màquines: una per executar els lu0 i l'altra compleix els requisits de bdiv, i, per tant, també bmod i fwd poden ser-hi executats. L'objectiu d'aquesta prova és comprovar que el runtime demani els recursos necessaris per solucionar el problema i no ho demani màquines que consumeixin més recursos del que realment necessita. Igual que en els casos anteriors la figura 6.24 mostra el que passa al llarg de l'execució segons la llegenda de la figura 6.23.

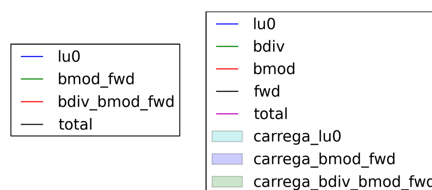

Figura 6.23: Llegendes pels gràfics d'execucions SparseLU amb 3 tipus

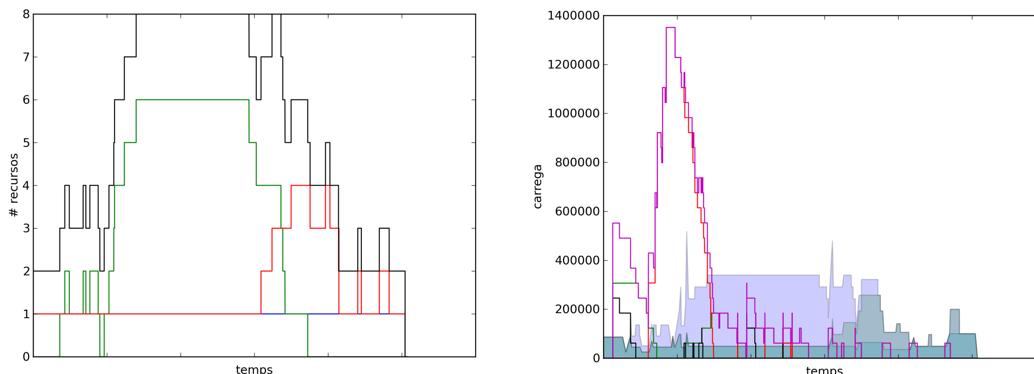

Figura 6.24: Evolució dels recursos en un SparseLU amb 3 tipus

Igual que en el casos anteriors, el nombre de màquines capaces d'executar lu0 es manté constant. El motiu és el mateix, mai arribarà prou càrrega per crear una nova màquina ja que com a molt pot executar-se'n una de cop. La variació de recursos ve donada per les tasques bdiv, bmod i fwd. Com es pot veure, a l'inici de l'aplicació la major part de la càrrega es deguda al mètode bmod, això fa que les màquines que es demanin per resoldre l'aplicació siguin màquines que no puguin executar bdiv. Aquestes noves màquines buiden les cues de la única màquina capaç d'executar bdiv de tasques dels altres mètodes i així aquesta pot executar els bdiv. Aquest comportament es manté fins arribar a la part final de l'execució. En el moment en que s'ha acumulat una càrrega de bdivs que es tarda més a executar-se que la de bmods i fwds (hi ha menys càrrega però menys recursos, per tant, el tractament del conjunt de tasques serà més llarg). El runtime demana màquines capaces d'executar bdiv. Quan aquesta càrrega decrementa la màquina es manté ja que pot executar també altres tipus de tasca

6.5.3 Stream

Stream és una aplicació amb quatre mètodes: copy, scale, add i triad. L'aplicació consisteix en aplicar aquestes quatre funcions de forma iterativa operant un conjunt de 3 vectors. Abans de començar una iteració, l'aplicació espera a que s'hagin completat totes les tasques de l'anterior iteració .

Les execucions que es fan a continuació corresponen als 2 escenaris explicats en el capítol 6.2.3.

1 tipus de tasca

Totes les tasques poden executar-se en qualsevol de les màquines. Com en el cas del Hmmer amb 1 sol tipus de màquina, la càrrega global és la que marca el número de màquines que necessita el runtime. El fet d'incloure un barrier que obliga a esperar fa que no es pugui conèixer el que hi ha més enllà del final de la iteració. Com que no es coneix tot el graf la única política de destrucció de màquines que pot aplicar-se és la destrucció periòdica. L'objectiu d'aquesta prova és veure com influeix aquesta política en el nombre de recursos. La figura 6.20 mostra l'evolució del nombre de recursos que utilitza el runtime, la figura 6.19 mostra la llegenda per poder interpretar els gràfic amb les càrregues acumulada i acceptable en el temps de crear una nova màquina.

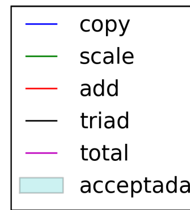

Figura 6.25: Llegenda pels gràfics d'execucions Stream amb 1 sol tipus de màquina

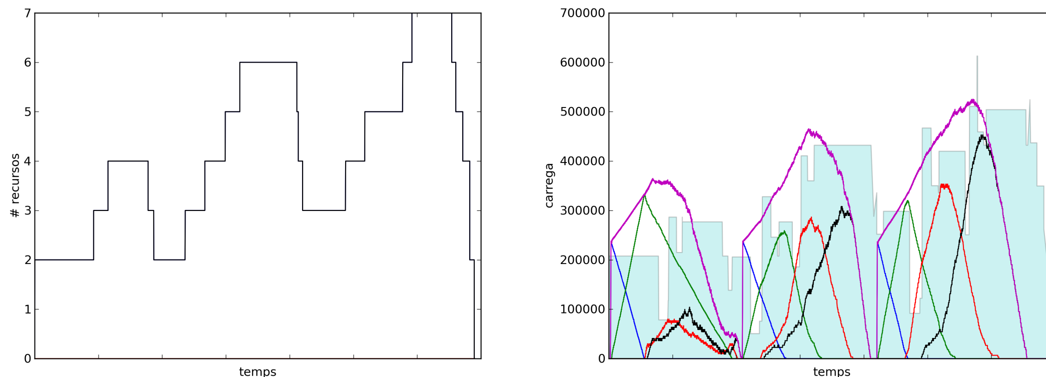

Figura 6.26: Evolució dels recursos en un Stream amb 1 sol tipus

En aquest primer escenari es pot veure com el nombre de màquines evoluciona de forma paral·lela a la càrrega global del runtime. Hi ha tres fets destacables que poden ser observats en aquests gràfics. El primer té relació amb el fer de que stream sigui un benchmark pensat especialment per mesurar l'ample de banda de memòria. És a dir, és una aplicació que requereix moltes transferències. Una bona planificació les hagués reduït i a l'augmentar-ne la localitat de les dades hagués obtingut un millor rendiment. En aquesta planificació, s'executa a la màquina on pot començar a executar-se abans. Al fer-se així, el nombre de transferències augmenta de forma important, i amb això el temps mig de tasca.

El segon aspecte a destacar està relacionat amb l'anterior. En la primera iteració, el runtime preveu que necessita utilitzar quatre workers, en la segona sis i en la tercera set. El comportament de la política és correcte, ja que es creen les màquines que es fan servir. El que no està tenint en compte aquesta política és que el fet d'augmentar el nombre de workers implica fer més transferències.

Al no haver executat un stopIT no s'apaguen les màquines fins que la política temporal ho decideix. Això només es fa periòdicament i només s'apaga una; de manera que el temps entre que aquesta decideix apagar una màquina fins començar la seguent iteració no es suficient per aplicar suficients vegades la política temporal com per apagar totes les màquines.

L'últim punt destacable és l'ordre que segueixen la planificació de tasques. A mesura que avança l'aplicació, les diferents fases queden més marcades. Això és una conseqüència de que s'acumulin els workers de l'anterior iteració. Quan arriba les tasques de la següent iteració, aquestes balancegen la seva càrrega entre més workers. Quan s'afegeix un nou recurs i agafa les tasques d'altres recursos, ho fa de manera més equitativa i ordenada.

La primera iteració és molt més desordenada que les altres degut a que no es coneix el

cost computacional de les tasques fins que ja se n'han planificat moltes. Quan el TaskScheduler coneix el cost que realment tenen, intenta balancejar la càrrega assignant les noves tasques planifica sobre la resta de tasques provocant més desordre.

2 tipus de tasca

Aquest segon escenari, pretén fer veure l'habilitat que té el runtime per adaptar-se segons el tipus de tasca, cosa que ja s'ha fet en escenaris similars d'altres aplicacions. De totes maneres, l'estructura de l'aplicació permet veure com el runtime reacciona a ràfagues d'un tipus que es repeteixen iterativament.

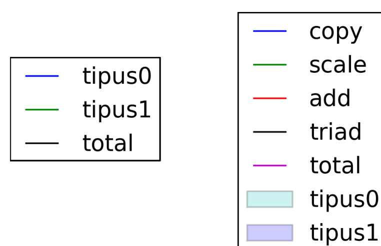

Figura 6.27: Llegendes pels gràfics d'execucions Stream amb 2 tipus de màquina

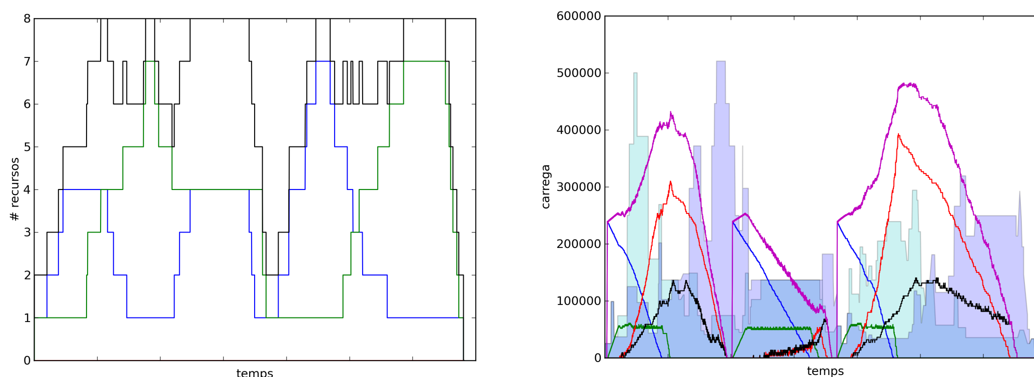

Figura 6.28: Evolució dels recursos en un Stream amb 2 tipus

El fet més destacable és que és capaç de demostrar com el runtime s'adapta al que arriba. El canvi de tipus de màquines dintre d'una iteració és correcte. Es pot veure que a mesura que comença a executar tasques del segon tipus, van desapareixent les màquines del primer permetent crear altres màquines.

De totes maneres no deixa de cridar l'atenció la manera en que s'acaben les dues iteracions. Mentre que la primera acaba amb una màquina pels primers mètodes i quatre pels add i triad, la segona acaba amb un i un. El motiu és el mateix per el que s'acumulaven recursos en el cas anterior: la política periòdica no té temps a apagar els sis recursos extres que s'han demanat i n'apaga tres. Al començar la següent iteració arriba de cop tota la càrrega i es prioritzen les creacions de les noves màquines tres vegades. Durant el temps d'aquestes comprovacions, s'executen algunes tasques del copy, deixant com a executables tasques scale. Quan la política de creació veu que s'ha arribat al límit de sis màquines en el cloud, permet la comprovació de la destrucció periòdica. En aquest moment ja existeixen tasques add que poden entrar a ser planificades un cop s'executin les tasques

copy que estan esperant. Com que la política de destrucció preveu aquestes execucions decideix no destruir les màquines. Igual que en la primera iteració, les màquines del primer tipus decrementen en nombre, permetent crear noves màquines del segon tipus. Ara, però, només se'n demana una ja que el fet de tenir un nombre de màquines i uns temps de tasques propers, fa que sigui difícil acumular tasques del segon tipus, ja que a mesura que acaben les tasques scale ja es van executant els adds en les altres màquines. Al tenir només tres màquines creades en el Cloud, a l'acabar la iteració si que hi ha temps per destruir-les totes, de manera que la tercera iteració comença amb el mateix nombre de màquines que tenia a l'inici.

6.6 Consum de recursos del Master

Aquesta prova vol comprovar de quina manera han influït els canvis realitzats sobre COMPSs en quant a quantitat de recursos consumits, centrant-nos en 4 aspectes: el percentatge de CPU que es fa servir pel runtime, la quantitat de memòria ocupada i l'espai de disc.

Per fer la prova s'utilitzaran 8 workers constants per la versió Grid, mentre que per la versió Cloud l'aplicació començarà executant amb 2 workers i s'estendrà demanant fins a 6 workers més al Cloud, fent un total de 8 workers. L'aplicació escollida per fer la prova és el hmmer amb una entrada de 2048 seqüències i un sol tipus de màquina. S'ha escollit aquest tipus d'aplicació ja que és un exemple que fa treballar durament els dos runtimes.

En el cas del runtime per Grid sempre que es vulgui escollir una tasca per executar haurà de buscar quina de totes les tasques és millor, per tant, haurà de revisar totes les tasques pendents.

Per la versió Cloud, a l'inici ha de planificar moltes tasques, quan té només dues màquines. A mesura que es vagi aplicant la política demanarà noves màquines que miraran de balancejar la càrrega. Degut a haver fet tota la planificació a l'inici, moltes màquines acabaran les seves tasques i hauran de buscar tasques d'altres màquines per executar-se. Quan l'aplicació s'apropi al final, el nivell de càrrega decrementa de tal manera que no seran necessàries la major part de les màquines i les anirà apagant.

El primer aspecte que s'analitza és el percentatge de CPU. La figura 6.29 mostra l'execució entre tots dos runtimes. El gràfic de dalt de tot correspon al percentatge de CPU utilitzada pel runtime de la versió Grid en cada moment i la tercera és el percentatge utilitzat pel prototip. La figura central correspon al nombre de màquines en la versió Cloud que s'està fent servir en cada moment.

Tal com es veu en la figura es poden diferenciar dues fases en les dues execucions. La part inicial on es tracten la majoria de les tasques hmmpfam (més llargues). Que la CPU de tots dos runtimes està més estones de idle. En aquesta primera fase, el runtime de la versió Grid mostra pics consum més alts i llargs de CPU que no pas la versió Cloud, que en mostra pics de poca ocupació, poca estona, però més freqüents. L'explicació és senzilla. Al tenir 8 workers treballant en paral·lel per la versió COMPSs, tasques amb una duració similar i entrar moltes tasques de cop a l'inici, fa que les 8 tasques que entrin a executar en un moment donat acabin gairebé a la vegada, això significa que s'executarà l'algoritme d'escollir una nova tasca 8 vegades, donant pics d'una longitud més llarga. A més a més, l'algoritme d'elecció d'aquesta nova tasca és molt més intensiu en la versió Grid, de manera que donarà pics més alts (aproximadament un 10% més alts). En la versió

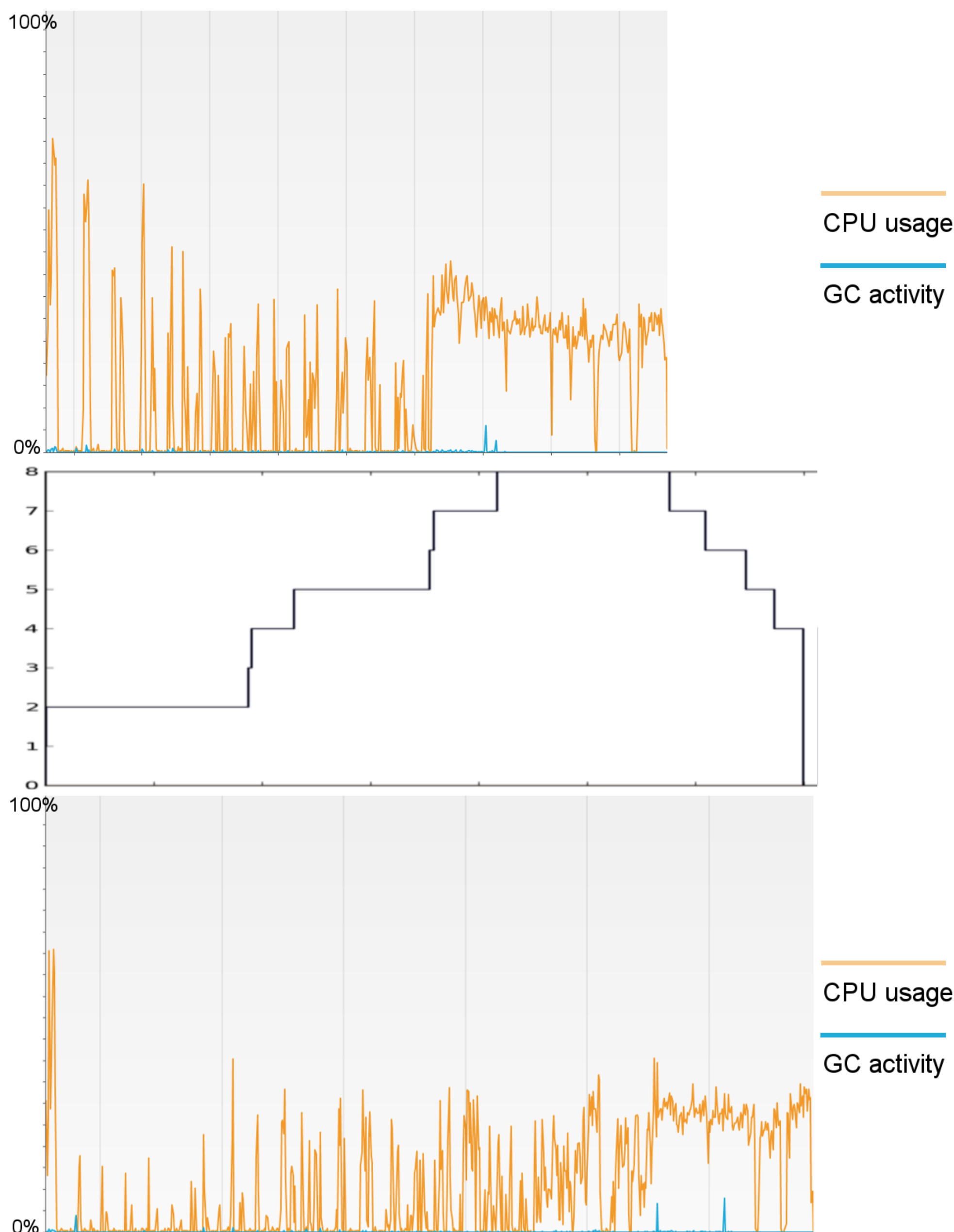

Figura 6.29: Comparació de l'ocupació de CPU entre les versions Grid i Cloud

Cloud, el balanceig de càrrega en el moment en que s'afegeix una nova màquina provoca que es puguin veure pics una mica més alts que en la resta d'estona. L'útlim fet que cal explicar és la freqüència en que es fa ús de la CPU. En la versió Grid es veuen (sobretot a l'inici), que el runtime entra en acció amb intervals de pausa llargs (aproximadament els 20 segons que dura una tasca). En la versió Cloud, es veu com el runtime té pics molt petits en intervals curts de temps. Aquest es corresponen a l'entrada en acció de la política periòdica tant de creació com de destrucció.

En la segona part de l'execució es pot veure un comportament pràcticament calcat. El motiu és que s'han tractat totes les tasques hmmpfam i només queden tasques merge que entren ràpidament en el TaskScheduler i se'ls hi assigna el millor recurs on poden executar-se. En aquest cas, l'algoritme d'assignació és exactament el mateix, ja que els

recursos comencen a tenir espais buits i, per tant, només cal escollir el recurs. El fet de reduir el nombre de workers en la versió Cloud, provoca que l'algoritme sigui una mica menys pesat (5% d'ocupació menys).

El segon aspecte que cal comprobar és la quantitat de memòria ocupada. La figura 6.30 mostra els tamanys de heap i el heap ocupat per totes dues versions en el mateix ordre que en la figura de les CPU.

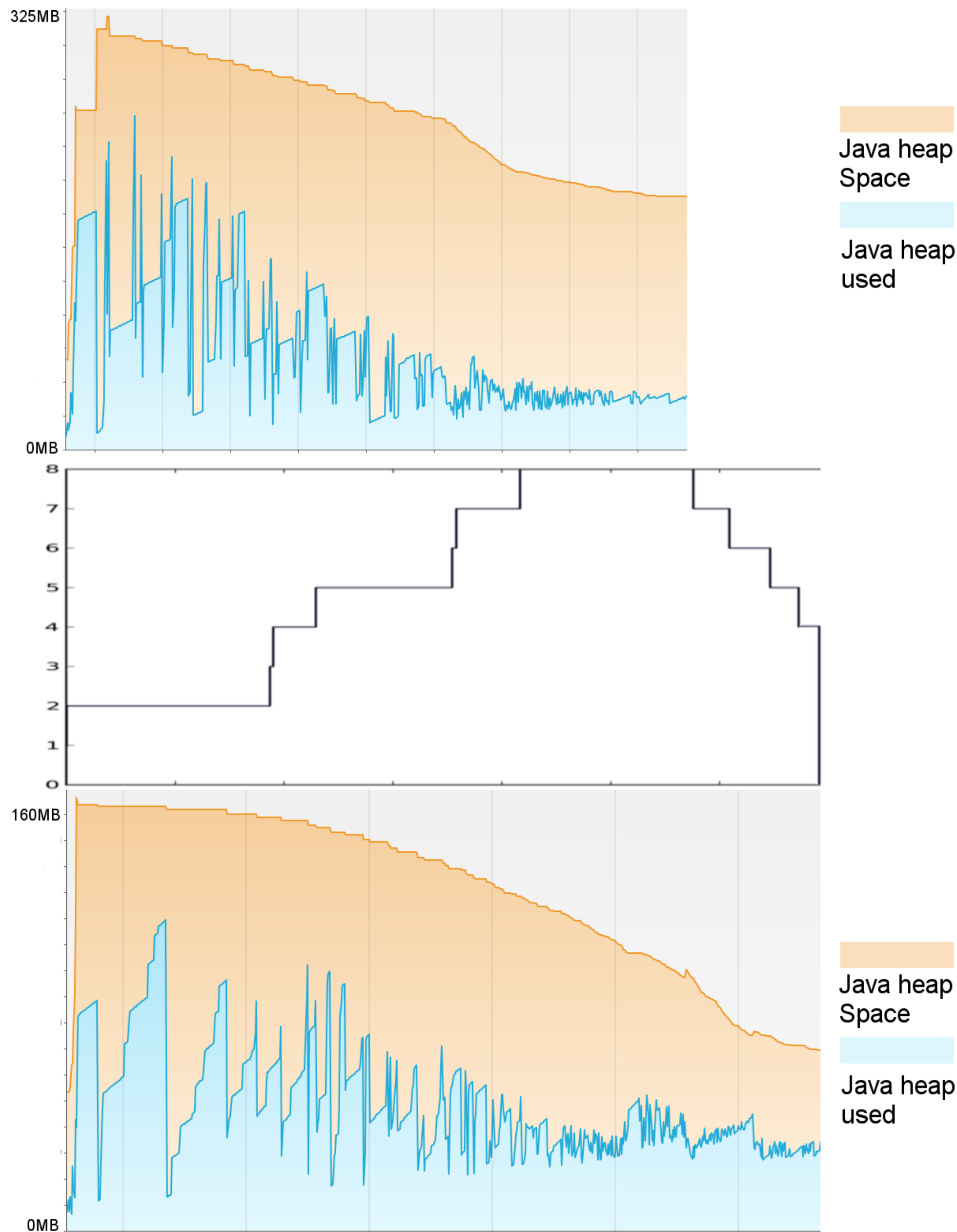

Figura 6.30: Comparació del consum de memòria entre les versions Grid i Cloud

L'ocupació en memòria del runtime és un punt complicat d'avaluar. Existeixen dos problemes: ProActive i l'alliberament d'espai del heap en Java. Com es diu a l'explicació de ProActive en el punt 3.2, per tal de poder enviar una estructura de dades entre dos components diferents és necessari fer una còpia de l'objecte. De manera que com més comunicacions es facin entre els diferents components, més espai de memòria s'ocupa.

El fet de ser un runtime en Java, tampoc ajuda en reduir aquest espai. L'encarregat d'alliberar dades del heap és el Garbage Collector, per netejar totes aquestes estructures que s'han hagut de copiar per poder comunicar els components s'ha d'esperar a que aquest actuï.

Igual que en el cas anterior, existeixen dues fases en l'execució. En la primera el comportament dels dos runtimes és totalment diferent. Mentre que a l'arribar al moment en que tots dos runtimes es comporten igual el consum de memòria és totalment diferent.

Un altre cop la planificació de tasques resulta clau. En la versió per Grid, cada vegada que es vol planificar una tasca per un node qualsevol, el Task Scheduler fa una petició al File Information Provider d'on es pot trobar cada un dels paràmetres de cada tasca. Això implica tenir en memòria moltes còpies amb els mateixos valors, ja que dues tasques poden compartir fitxers d'entrada. Tant es així, que el consum de memòria per aquesta versió arriba a 250MB en algun moment. En canvi, en la versió per Cloud, en la majoria dels casos la planificació no requereix cap transferència de dades entre Task Scheduler i FileInfo Provider, de manera que el consum de memòria augmenta fins a 125MB.

Com es veu en els gràfics, el consum de memòria degut a les noves estructures que s'han creat al llarg del disseny resulta negligible comparat amb el consum degut a comunicacions del runtime.

L'espai de disc que requereix la nova versió de COMPSs ha augmentat respecte la versió de Grid. Mentre que la versió anterior de COMPSs ocupava un total de 165.8MB tenint en compte JavaGAT i ProActive, la nova versió n'ocupa 171.9MB. Aquesta diferència de 6.1MB es deu a dos aspectes. El primer és que s'ha afegit més codi al runtime. Per això el jar que el conté ha passat d'ocupar 184.4KB a 259.3KB. Els 6 MB restants corresponen a l'aparició d'una nova dependència: el client necessari per comunicar-se amb l'scheduler d'EMOTIVE Cloud. Aquest client conté no només l'API, sinó que també conté totes les llibreries que necessita per executar-se.

6.7 Comparació de costos entre Grid i Cloud

L'última prova que s'ha volgut fer, consisteix en una comparació entre el cost que tindria una execució en un Grid i una execució utilitzant recursos únicament Cloud oferts públicament. El càlcul s'ha fet utilitzant els preus dels recursos oferts per Amazon com a preu de referència.

Per calcular el cost d'una execució en el Grid, s'ha suposat que es disposa d'un cluster format per 2 servidors amb unes característiques similars als que s'han fet servir durant l'execució de les altres proves i queden descrits a l'apartat 6.1. A l'hora d'avaluar els costos que suposa tenir un cluster només s'han tingut en compte dos factors: l'amortització de les màquines amb un cost de 2.000€ per cada servidor i el consum energètic que suposa tenir-les enceses amb un preu de és de 0'1497€ el KWh. El consum elèctric varia en funció de la càrrega que les tasques generin en aquestes màquines. La potència consumida varia entre 240W, si la màquina el servidor està en idle, i 320, si les 4 CPUs s'estàn fent servir amb un 100% d'ocupació.

Donats aquests preus, ja es pot fer una primera comprovació sobre la diferència de costos entre executar en el Cloud i executar en el Grid. La primera pregunta interessant a fer-se és: a partir de quantes hores de computació surt més a compte tenir un Grid que fer servir un 8 workers permanentment en el Cloud? Tal i com mostra el gràfic de la

figura 6.31, el mínim d'hores variarà en funció de la càrrega que hagin de suportar aquestes màquines. Si les CPUs de les màquines estan pràcticament al 0% d'ocupació, el mínim són 8.069 hores, mentre que si les CPUs de les màquines estan calculant el 100% del temps, surt a compte adquirir un Grid a partir de les 8.425 hores d'execució.

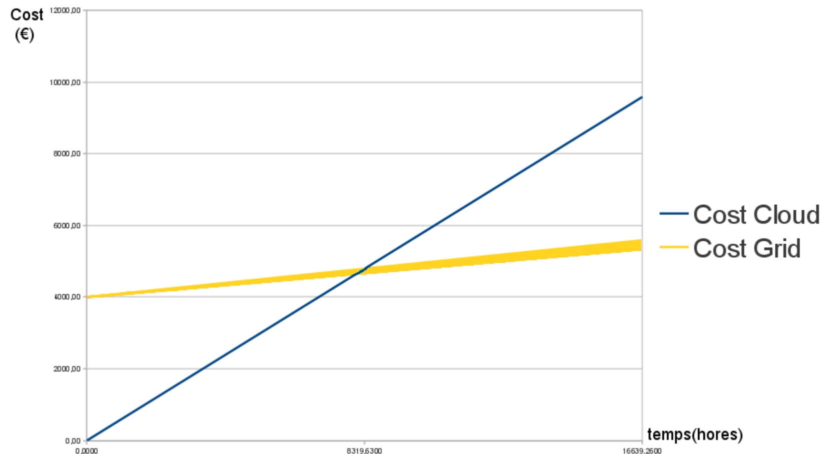

Figura 6.31: Relació entre els costos executant 8 recursos en el Cloud i fent executant un cluster de 2 nodes

D'altra banda, l'objectiu del projecte no és mantenir aquest nombre de workers permanent en el Cloud, sinó poder augmentar-lo i disminuir-lo en funció de les necessitats de l'aplicació. Per això, fa falta comprovar quin és el límit d'hores de computació que seràn necessàries perquè la compra dels servidor sigui més barata que utilitzar nodes Cloud en funció del temps que aquest recursos estiguin sent utilitzats. En cada fila de la taula 6.5 es poden trobar els diferents nombre d'hores necessàries perquè el cost de compra sigui inferior al del Cloud en funció del percentatge de temps d'us que es fa dels 8 recursos en el Cloud al llarg de l'execució per un ús mig de les CPUs.

Ús de les CPU \ temps d'ús dels recursos	20%	40%	50%	60%	80%	100%
20%	130.010	27.631	19.825	15.458	10.731	8.217
40%	144.026	28.214	20.123	15.639	10.817	8.268
50%	152.231	28.515	20.276	15.731	10.861	8.294
60%	161.428	28.823	20.431	15.824	10.906	8.320
80%	183.614	29.458	20.749	16.014	10.996	8.372
100%	212.870	30.123	21.076	16.208	11.087	8.425

Taula 6.5: Quantitat d'hores de computació necessàries perquè recòrrer al Cloud deixi de ser útil.

Com mostra la taula, depenent de la forma del graf de l'aplicació i de la intensitat del càlcul que tinguin les seves tasques les 8.425 hores poden augmentar fins a 212.870. Depenent de l'ús de les màquines que en fa l'aplicació, en alguns casos, el nombre d'hores que seria necessari que el cluster estigués executant perqué surti a compte adquirir-lo és major que no les 25.632 hores que tenen els 3 anys que és el temps que normalment s'utilitza al calcular l'amortització d'una màquina.

Per fer aquests càlculs s'ha partit de la hipòtesi de que sigui quina sigui l'aplicació, s'executa de la mateixa manera en el Grid que en el Cloud i, per tant, tarda el mateix a executar-se. Tal i com es demostra en els tests anteriors, aquesta hipòtesi és errònia, i,

per tant, el nombre d'execucions que pot fer-se en cada tipus no té una relació directa en funció del que es tardi en executar l'aplicació. Per tant, el nombre d'execucions necessàries perquè el Cloud deixi de ser la solució més barata dependrà de la cada aplicació concreta.

Per tal d'avaluar aquest aspecte, s'ha executat un SparseLU i un Hmmer de 4096 seqüències i se n'ha calculat el cost que té una execució en el Cloud i el consum energètic que té executar-la en el cluster. A partir d'aquests dos costos se'n pot deduir el nombre total d'execucions que pot fer-se'n.

En el cas del SparseLU, l'execució en el cluster ha tardat 1.968 segons, ocupant aproximadament menys d'un 1% de les CPUs (cal recordar que l'aplicació porta un sleep a dintre de les seves tasques). El cost de l'energia consumida per poder fer una execució és de 0'044€.Mentre que una execució en el Cloud dura de mitjana 2.404 segons amb un cost total de 0'204. La gestió dels recursos fa que només es cobri el 54% del cost del màxim de 8 recursos que igualarien el Grid. El nombre mínim d'execucions d'un sparseLU necessari per justificar la compra d'un Grid és de 24.915.

En el cas del Hmmer amb 4096 seqüències, l'aplicació ha tardat 1.068 segons ocupant una mitja del 85% de CPU. El cost energètic en aquest cas és de 0'02758€. Mentre que l'execució ha tardat 1.525 segons amb un cost total de 0,160€. La gestió dels recursos redueix l'ús dels recursos a un 66%. El nombre mínim d'execucions en aquest cas és de 32.406 execucions.

Conclusions del Projecte

Capítol 7

Conclusions del Projecte

7.1 Conclusions

En aquesta última part del document es busca analitzar tot el que ha estat el projecte i extreure'n unes conclusions. El primer que s'analitza és el prototip resultant del projecte per veure si s'han complert els objectius marcats a l'inici del projecte i que poden llegir-se a l'apartat 1.2. El capítol anterior mostra les dades de rendiment i l'evolució del nombre de màquines en diferents casos d'ús. De totes maneres encara no s'ha analitzat en cap moment el nou runtime de forma global.

Un altre aspecte que el runtime ha permès estudiar és la tecnologia Cloud aplicada al càlcul distribuït. El capítol 2, explica el costat més teòric i comercial, queda per veure l'experiència com a usuari i realment quins beneficis ha aportat tant en aquest camp específic com pot fer-ho en altres camps.

Hi ha dos temes més als que qualsevol projecte està vinculat: el cost econòmic i la planificació. Cal fer un anàlisi de tots els costs que pot comportar realitzar aquest projecte en un àmbit comercial per veure'n la viabilitat. També cal veure com la planificació inicial ha canviat al llarg del projecte i justificar aquests canvis.

Per acabar, en un últim apartat, s'analitzen tots aquells camps en que queda feina per fer i millorar l'ús del Cloud en el camp del càlcul distribuït.

7.1.1 Conclusions del Prototip

El primer que cal fer és recordar quins objectius s'havíen marcat a l'inici del projecte, a l'apartat 1.2:

- Estendre el model de programació per tal de que es pogués fer ús de les màquines que pertanyen a un Cloud per resoldre tasques.
- Ajustar dinàmicament el nombre de màquines que utilitza el runtime.
- Dissenyar una arquitectura que faci el runtime independent del Cloud Provider que es vulgui utilitzar.
- Definir polítiques amb una actuació amb sentit, sense la necessitat de que minimitzin el cost o el augmentin rendiment.

- Avaluar el rendiment del prototip generat pel projecte.
- Avaluar la tecnologia que suporta el Cloud.

Al llarg de la tercera part de la memòria, Desenvolupament del projecte, es veu com es compleixen els 3 primers objectius. El nou runtime és capaç d'utilitzar workers en el Cloud. Les polítiques creades mantenen un nombre de màquines virtuals útil, i en demanen més o en destrueixen en funció de la càrrega del runtime. Per aconseguir la independència del Cloud Provider s'ha creat una capa abstracta: el Connector. La seva implementació és la que marca les comunicacions amb el Cloud Provider.

És pot dir que complir aquest tres primers objectius és el punt més fort del prototip. Per si sols no milloren el rendiment, ni suposen cap reducció dels recursos que necessita el runtime. Tot i així, és el que obre les portes a utilitzar els recursos Cloud per paral·lelitzar el tractament de dades o càlculs complexos i, com a conseqüència, a tots els avantatges que ofereix la tecnologia Cloud Computing i cada un dels proveïdors comercials d'Utility Computing.

El canvi en l'arquitectura del runtime no és l'únic canvi que s'ha fet. També s'han realitzats canvis importants en la gestió de workers i en la planificació de tasques.

Els canvis per poder gestionar els workers és on s'han incorporat nous aspectes al runtime. El benchmark que s'ha fet servir per testar el prototip i explicat a la quarta part d'aquest document, Anàlisi del prototip, mostra que el conjunt de les polítiques actua d'una manera lògica. Tot i que, la definició d'unes polítiques amb sentit era part dels objectius, aquests també marcaven que el rendiment d'aquestes no era part del projecte i, per tant, no han de ser més analitzades del que ja s'ha fet al llarg dels experiments realitzats en el capítol 6.5.

Com ja s'ha vist en el test presentat en el punt 6.6, la quantitat de recursos que el runtime requereix per fer tota aquesta gestió és negligible a nivell de memòria i redueix l'ús que es fa de la CPU.

Un efecte col·lateral d'incloure les polítiques ha estat modificar la planificació de tasques. S'ha incorporat una planificació que redueix l'overhead respecte l'anterior planificació. De manera que per si sola fa el runtime més ràpid. El runtime funciona millor que el seu predecessor en totes aquelles aplicacions intensives en la part de càlcul, amb poca transferència de dades, amb un conjunt de workers reduït o que les transferències vinguin obligades pels tipus de màquines que requereix cada tasca. A més a més, minva l'impacte del problema que tenia la planificació anterior quan tenia moltes tasques pendents d'execució i havia de comprobar-les totes per escollir-ne una.

El problema que es pot detectar és que no té en compte les transferències que cal fer per executar la tasca provoca que el rendiment de l'aplicació es redueixi dràsticament, com en el cas del stream que al tenir tasques tan simples, el límit de transferències realitzant-se de forma paral·lela passa a ser un coll d'ampolla. Al no tenir en compte la localitat de les dades es perd rendiment en aquelles aplicacions que tinguin poc càlcul i puguin ser realitzades amb molts workers.

Comparat amb Google MapReduce, COMPSs és capaç d'albergar un ventall d'aplicacions molt més ample ja que permet definir qualsevol flux d'execució, igual que Microsoft Dryad. La diferència amb el segon és que la simplicitat per indicar-lo. Mentre que a Microsoft Dryad cal definir un graf que representi el flux de l'aplicació utilitzant una sèrie d'operacions de la seva API, COMPSs és capaç de crear aquest graf a partir d'un co-

di seqüèncial. Això dóna a COMPSs, i a tots els models de programació de la família SuperScalar, molta elegància i facilitat a l'hora de programar.

Mantenir aquesta facilitat i elegància era un dels punts que es marcava en les directrius del disseny, explicades a l'apartat 5.1. Un cop acabat el disseny i la implementació es pot dir que també s'ha complert aquest objectiu. Les mateixes aplicacions que poden executar-se en COMPSs per fer servir el Grid són capaces d'utilitzar de manera totalment transparent a l'usuari màquines de qualsevol Cloud Provider. Per tant, aquest prototip hereta aquest punt fort de tota la família.

Per acabar, aquest prototip no deixa de ser una modificació superficial COMPSs. El nucli de COMPSs segueix estant format per 5 components GCM i segueixen comunicats de la mateixa manera que abans mitjançant ProActive. Manté l'avantatge de tenir una gran modularitat i que cada component pot ser desplegat en una màquina diferent per repartir la càrrega. Per contra, segueix apareixent el problema de les comunicacions entre els components, que consumeixen molta memòria i es poc eficient si està tot en una sola màquina.

L'execució de tasques i còpia de fitxers se segueix fent mitjançant JavaGAT amb l'adaptador per ssh. Igual que amb ProActive, utilitzar un middleware aporta els seus avantatges i inconvenients. Per una banda, el runtime pot ser executat sobre qualsevol conjunt de màquines, només cal tenir l'adaptador necessari per fer servir el software necessari per les comunicacions. Per contra, el runtime no pot tenir control sobre alguns aspectes de les comunicacions i pot perdre una mica de rendiment.

7.1.2 Conclusions del Cloud Computing

De la llista d'objctius feta a l'apartat anterior, encara en falta un per complir: avaluar la tecnologia que dóna suport al Cloud. Al llarg del capítol 2, a part de descriure què era el Cloud i de quines maneres es podia accedir a aquest servei, també s'ha vist els avantatges i els inconvenients teòrics i tècnics que hi ha avui en dia. A la pràctica encara es poden veure com encara queda camí per desenvolupar la tecnologia que dóna suport a aquest model comercial. Aquest apartat pretèn fer una crítica al que s'ha fet servir al llarg del projecte: els serveis Cloud IaaS i el middleware EMOTIVE Cloud.

És evident l'avantatge que el Cloud dóna al programador al tenir un entorn virtualitzat on s'executa el seu software. El programador pot saber exactament amb quins problemes es pot trobar. El fet de tenir un SLA que defineix la manera en que el Cloud Provider ofereix els recursos del Cloud permet al client definir d'una manera més simple de quina manera pot oferir ell el seu producte. A més a més, permet al client oblidar-se de tota la gestió i manteniments de la màquina i, tal com demostra l'últim test fet, reduïr el cost de realitzar certs càlculs.

Un segon aspecte que és molt positiu és la manera en que s'està desenvolupant aquests serveis IaaS: estandaritzant l'accés, però mantenint independent la implementació que té cada una de les solucions. OCCI (Open Cloud Computer Interface) és una interfície que encara s'està definint per un grup de treball de l'OGF [1] que pretén ser la interfície estàndard que tots els proveïdors i gestors de Cloud privats implementin per poder permetre als programadors/clients utilitzar un proveïdor o un altre sense necessitat de canviar els software que utilitza.

[1] Open Grid Forum - http://www.gridforum.org/

L'únic problema que s'ha detectat en aquest aspecte ha estat el que mostra el test que comprova la dependència del Cloud Provider (capítol 6.4). Aquest mostra que el punt on el runtime perd més rendiment és el Cloud Provider. En el cas del prototip, es fa servir un Cloud gestionat amb EMOTIVE Cloud i el mateix test es fa sobre un Cloud de 1 node i un de dos. Els temps obtinguts en l'experiment mostren els dos problemes que dóna el Cloud Provider. El primer és el temps de resposta en donar una nova màquina: 140 segons en cas de ser una màquina de 1.5GB de disc. En experiments posteriors es veu com aquest temps es redueix segons l'espai de disc demanat. Per tant, el rendiment del runtime va subjecte a tota la infraestructura d'emmagatzematge que té el cluster que servirà al Cloud. Si el Cloud provider disposes d'un emmagatzmatge en RAID amb discs SSD capaços de reservar tot aquest espai d'una manera més ràpida aquest temps es disminuiria i el runtime seria capaç d'aprofitar millor el paral·lelisme de l'aplicació.

El segon problema que dóna el Cloud provider i que mostra la comparació entre el Cloud d'un node i el de dos és la manera en que s'acumulen les peticions. Si es disposes d'un Cloud de 8 nodes, el sistema seria capaç de servir-nos les 8 màquines que té com a límit l'execució amb un temps constant i que reduiria l'error de gestió que pot tenir el runtime en aquests casos. Permetent donar resposta d'una forma més ràpida als pics de demanda.

Així doncs, el problema del Cloud Computing és simplement d'infraestructura. Cal una inversió molt més gran per donar un bon servei de Cloud que per tenir un cluster, i molt més que en el cas d'un Grid. Per aquest motiu i pel poc temps que tenen les tecnologies en que es suporta el Cloud Computing, molt poques empreses s'han embarcat a oferir aquest tipus de servei.

Al llarg del capítol 2, es fa una comparació a nivell teòric dels serveis que ofereixen els diferents Cloud Providers i els Clouds privats. Al llarg de tot l'anàlisi del prototip, només s'ha tingut en compte un Cloud Privat gestionat amb EMOTIVE Cloud i amb recursos molt limitats. Caldria veure com es comporta el runtime en el cas d'estar executant fent servir Amazon EC2 o el servei de qualsevol altre Cloud Provider. No fer aquesta comparació fa que no es pugui arribar a veure l'avantatge d'utilitzar un Cloud comercial o un Cloud Privat ni veure quins són els desventatges que EMOTIVE té comparat amb altres middlewares semblants.

7.2 Planificació

Al llarg de la realització del projecte han sorgit alguns inconvenients que han fet canviar la planificació tal i com es presenta a l'apartat 1.3. Durant els primers mesos del projecte, totes les tasques van anar sortint tal i com s'havia previst en la planificació inicial, fins i tot, la segona iteració de la implementació s'ha realitzat amb menys temps del que estava previst, permetent avançar tota la planificació una setmana.

Tot i portar una setmana d'avantatge respecte la planificació inicial, també van aparèixer alguns inconvenients per diferents motius que van provocar alteracions en la planificació. El primer contratemps que va apareixer cronològicament va ser en el desenvolupament de la cinquena iteració, afegir dinamisme al runtime. Concretament, en la part de destrucció. Ja s'havia previst que afegir dinamisme per destruir una màquina seria més complicat que no pas crear-la. De totes maneres, no es va tenir en compte el fet de que eliminar una màquina implicava deixar de tenir accés als fitxer que hi havia emmagatzemats en aquesta. Es va partir de la idea de que EMOTIVE podia emmagatzemar els fitxers d'al-

guna manera que sobrevisquessin a la màquina. Això no és així, per tant, es va haver de dissenyar una de les parts més complexes del projecte: salvar els fitxers. Realitzar aquest disseny va suposar allargar més del previst aquesta tasca. Per evitar quedar encallats en aquest punt mentre no sorgien dissenys eficients per solventar el problema, es va decidir realitzar abans la creació de la política de creació mentre es discutia la solució.

Un cop creada i implementada la política de creació es podia continuar el procés amb les noves idees. La celebració del SIENA[2] networking session at ICT2010 entre el 27 i el 29 de Setembre va provocar també un petit canvi. En aquesta reunió es volia presentar els primers resultats obtinguts per COMPSs utilitzant la infraestructura EMOTIVE. Es va dedicar la segona setmana del mes de Setembre a realitzar una sèrie de proves per obtenir dades. Aquestes proves corresponen a l'experiment presentat en aquesta memòria en l'apartat 6.4. Per tant, una part de l'avaluació del rendiment va ser avançada uns mesos abans de tenir tot el prototip. A diferència del cas anterior, en aquest cas no es va haver de canviar el nombre d'hores dedicades a cap dels aspectes.

L'últim problema en apareixer va obligar a alterar la planificació per tercera vegada. Aquest cas tampoc va suposar un increment de les hores de feina sinó un desplaçament d'aquestes. Un cop realitzat tot el prototip, a l'hora d'avaluar el runtime es va voler utilitzar els dos nodes del Cloud: pctinet i pcperot. Va coincidir l'inici d'aquesta etapa amb un procés d'actualització d'EMOTIVE Cloud. Que va impedir l'ús d'un segon node durant 2 setmanes. Durant aquestes 2 setmanes no es va poder experimentar i, per tant, el procés d'avaluació del runtime va quedar congelat fins a solucionar el problema. Per no quedar aturat es va intensificar la redacció d'aquest document fins a solucionar el problema.

El problema més gran que ha aparegut en quant a hores de feina ha estat la redacció de la memòria. Ha suposat un increment de 100 hores de feina. L'increment de feina no ha estat tant en la seva redacció, sinó en les posteriors correccions. De totes maneres, l'extensió de la segona part també ha estat una de les causes.

La taula 7.1 mostra els canvis en la quantitat d'hores ressaltant en vermell, aquelles etapes en les que s'ha tardat més temps de l'esperat i en verd les que s'ha tardat menys de l'esperat.

De totes maneres això no reflexa totalment els canvis que finalment s'han produït. El diagrama de Gantt inclòs en la figura 7.1 mostra el temps dedicat i l'ordre en que s'ha realitzat cada una de les tasques. Es marca en vermell si s'ha tardat més temps en realitzar-la, verd si es tarda menys del previst i blau si es canvien l'ordre però no s'amplia el temps.

[2]http://www.sienainitiative.eu

Etapa	Planificació inicial	Planificació real
Familiarització	**75**	**75**
Preparació de l'entorn	8	8
COMPSs	57	57
EMOTIVE	10	10
Implementació	**375**	**400**
Creació Manual	25	25
Creació Inicial	75	75
Destrucció Final	50	37
Creació en paral·lel	50	38
Dinamisme-Creació	40	40
Dinamisme-Destrucció	60	110
Creació de Polítiques	75	75
Avaluació	**75**	**75**
Memòria	**100**	**200**
Total	**625**	**750**

Taula 7.1: Canvis en el nombre d'hores dedicades a cada etapa

Figura 7.1: Diagrama de Gantt de la planificació real

7.3 Anàlisi econòmic

L'altre aspecte important a l'hora de desenvolupar qualsevol projecte, és el cost econòmic que comporta. Al llarg d'aquest capítol es detallen els costos econòmics que haguéssin suposat desenvolupar aquest projecte en un entorn empresarial tot i la dificultat de quantificar el cost de l'ús de les màquines i dels serveis del BSC-CNS que s'han fet servir durant el projecte. Es revisen els costos per: recursos humans, el hardware utilitzat, el software que fa servir el projecte i el software fet servir durant el desenvolupament.

Recursos Humans

Per calcular els costos deguts a la contractació de persones, s'ha classificat en tres possibles perfils les persones que desenvoluparien el projecte.

- Cap de Projecte: pren les decisions executives a través dels estudis fets per l'analista.
- Analista: pren les decision sobre la tecnologia utilitzada. S'encarrega de realitzar el disseny i proporcionar al programador la informació necessària per que pugui desenvolupar la seva tasca.
- Programador: implementa el disseny de l'analista en un llenguatge de programació concret.
- Tècnic en sistemes: s'encarrega de realitzar les instal·lacions configuració i el manteniment dels entorns necessaris perquè la resta d'usuaris puguin desenvolupar la seva feina.

La taula 7.2 mostra el repartiment que s'ha fet de les hores que s'han dedicat al projecte. La memòria és una feina que es fa a tots els nivells, però gran part recau en l'analista que és qui ha de donar explicacions del disseny i coneixer les tecnologies.

Perfil	Cost (€/hora)	Hores dedicades	Total
Cap de projecte	70	25	1.750
Analista	60	370	22.200
Programador	30	295	8.850
Tècnic en sistemes	40	32	1.280
Subtotal			**34.080 €**

Taula 7.2: Quadre resum dels costos en recursos humans

En l'apartat anterior es veu que s'han necessitat 750 hores per desenvolupar el projecte. I en canvi sumant les hores dedicades en aquest aspecte en surten 775. Les 25 hores restants són hores en que s'ha necessitat l'ajuda d'algun tècnic en sistemes del BSC-CNS.

Hardware utilitzat

Tant pel desenvolupament del prototip com per les seves proves s'han fet servir únicament les màquines descrites en el capítol d'avaluació del prototip. La mateixa màquina on s'executava el master és la que s'utilitza per desenvolupar. Així doncs, tal i com mostra la taula 7.3, el cost en hardware seria de 1.150 €.

Perfil	Cost (€)	Amortització	Total
Ús de les màquines del BSC-CNS			1.000 €
Ordinador Personal	600	0.25	150
Subtotal			**1.150 €**

Taula 7.3: Quadre resum dels costos en recursos hardware

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 els seus costos. En alguns casos que s'hagi d'utilitzar software complex i molt específic pot arribar a ser una part important del cost total del projecte, cosa que no ha estat en aquest cas. El cost de les diferents llicències de software necessàries per desenvolupar el projecte es pot veure a la taula 7.4.

Software	Cost (€)
OpenSuSE 11.2 (Sistema Operatiu)	gratuït
Java J2SE Software Development Kit	gratuït
NetBeans IDE 6.8	gratuït
apache-ant	gratuït
EMOTIVE Cloud	gratuït
OpenVPN	gratuït
Subtotal	**0 €**

Taula 7.4: Quadre resum dels costos en software per desenvolupar

Software utilitzat per escriure la memòria i fer la defensa

Realitzar el prototip no és l'únic que requereix l'ús de software, l'escriptura d'aquest document també és part del projecte i l'ús d'alguns programes també pot necessitar llicències. La taula 7.5 descriu els diferents programes utilitzats per fer la memòria i el cost d'aquestes llicències.

Software	Cost (€)
gedit	gratuït
package texlive (generació pdf amb latex)	gratuït
Microsoft Visio 2003 (esquemes)	289
GIMP	gratuït
OpenOffice (Writter, Calc i Draw)	gratuït
MathPlotLib (gràfics)	gratuït
Planner (gestió de Projectes)	gratuït
Microsoft PowerPoint (Presentació)	189
Subtotal	**478 €**

Taula 7.5: Quadre resum dels costos en software per escriure la memòria

Anàlisi del cost total

Concepte	Cost (€)
Recursos Humans	34.080
Hardware	1.150
Software utilitzat pel desenvolupament	gratuït
Software per redactar la memòria	478
Total	**35.708 €**

Taula 7.6: Quadre amb els costos totals del projecte agrupats per concepte

Tal com mostra la taula 7.6, el cost total del projecte puja a 35.708 €. A l'esquerra de la figura 7.2, es pot veure un gràfic amb el percentatge que representa cada un d'aquests apartats. Es pot veure que el 95% del cost ve donat per Recursos Humans. El gràfic de la dreta entra en detall sobre el repartiment d'aquest apartat. En ell es veu com l'analista s'emporta un 65% de la inversió. De les 370 hores de feina que se li han assignat a l'analista, 150 eren en concepte de l'escriptura de la memòria. D'aquestes bona part han estat dedicades a l'escriptura de la segona part d'aquest document, l'anàlisi de l'estat de l'art. Al ser un projecte final de carrera, bona part d'aquestes hores s'han dedicat a recolectar, llegir i sintetitzar informació per poder escriure-la. Si s'hagués contractat un analista professional que ja tingués coneixement de la matèria, aquesta quantitat es podria haver reduit en 50 hores, el cost de l'analista a 16.200 € i el cost del projecte a 29.708€.

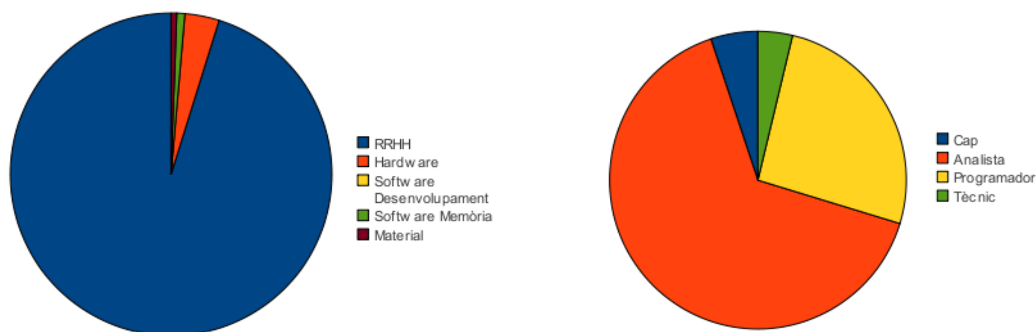

Figura 7.2: Percentatges del cost total i RRHH

7.4 Treball futur

El prototip desenvolupat en aquest projecte simplement demostra que es pot utilitzar la tecnologia Cloud per fer processos de càlcul i tractament de dades. Tot i haver complert amb els objectius del projecte, el prototip no és més que una primera versió. Queda molt per fer per millorar-ne el rendiment i fer-lo més competitiu.

Una de les vies de millora que queden pendents és la planificació de tasques. La planificació del prototip intenta preveure on es podrà executar la tasca abans. Existeixen molts altres paràmetres a tenir en compte que s'han obviat en aquesta primera versió: la quantitat de dades que s'han de transferir, la velocitat en que aquestes es transfereixen, el cost de transferir les dades, el rendiment que obtindrà la tasca a l'executar-se en un node,

les tasques que poden fer servir els seus resultats, ... Tot i haver fet una planificació més ràpida de calcular que l'anterior, en alguns casos el rendiment que obté és inferior degut a aspectes com aquests.

També poden treballar-se molt més a fons les polítiques de gestió dels recursos: en quin moment el runtime decideix prescindir d'una màquina o detectar en quin moment es veu ofegat per la càrrega acumulada. Depenent de l'interès de l'usuari final existeixen moltes polítiques dissenyables. En aquest cas s'ha utilitzat una política que equilibrés rendiment i cost. Es podria fer una política orientada al rendiment, per exemple: intentar preveure la necessitat d'una màquina observant el graf i crear-la abans d'utilitzar-la. O una política orientada a reduïr el cost, per exemple: apagar una màquina en el moment en que no s'utilitzi una màquina encara que s'hagi d'encendre al cap de pocs segons.

Alguns Cloud Providers, entre ells EMOTIVE Cloud, permeten pausar una màquina virtual. Durant l'estona que la màquina està pausada, no s'ocupa ni CPU ni memòria del Cluster, només l'espai de disc. El cost de mantenir una màquina en aquest estat és menor, per tant, una opció a valorar seria afegir algun mecanisme que enlloc d'apagar les màquines directament, les faci passar abans per un estat de pausa i si no es torna a necessitar apagar-la. A més a més, el temps de encendre aquesta màquina pausada és menor al d'encendre'n una nova. Pot ser interessant la idea de tenir-ne una reserva per millorar el temps de resposta del Cloud.

Un dels motius als que no s'ha volgut entrar molt en detall durant el projecte és l'útilització de l'espai d'emmagatzematge virtual de tot el sistema. S'ha treballat únicament amb el disc assignat a cada màquina virtual degut a que els espais compartits no estaven habilitats a EMOTIVE Cloud durant la implementació del projecte. No poder fer-ne ús obliga a fer una sèrie de transferències entre màquines virtuals que poden trobar-se dintre d'una mateixa màquina física amb el disc compartit, o bé que el sistema de fitxers sigui compartit per tots els nodes. En qualsevol dels dos casos és inútil fer-la ja que podria llegir-se directament i amb el mateix retard.

Un dels problemes que el prototip hereta de COMPSs es la quantitat d'espai ocupat amb les diferents versions de tots els fitxers per evitar les dependències WaR i WaW. Les màquines virtuals tenen un disc limitat que es va omplint amb aquests fitxers i, en el cas de tenir una execució amb molts fitxers o fitxers grans, aquest espai pot arribar a omplir-se totalment i causar un error en l'execució. En realitat de tots aquests fitxer només se'n necessita un petit conjunt format per la última versió de cada fitxer i el corresponent a totes aquelles versions que encara tinguin algun lector pendent. Tota la resta de fitxers temporals poden ser eliminats (i de fet es fa indirectament al no salvar-los al apagar una màquina). Cal crear algun mecanisme que permeti eliminar aquests fitxers inútils que ocupen espai.

Possiblement, el tema més interessant en el que es pot treballar és el Connector entre EMOTIVE Cloud i el runtime. En aquest cas, només s'està fent servir un sol Cloud Provider, tot i que se'n podrien fer servir un nombre indeterminat. El connector podria decidir en funció del cost i el temps de resposta de cada provider a quin d'ells demana la màquina. També es podria comunicar el connector amb algun tipus de gestor de recursos que s'encarregués de coordinar peticions per varies execucions del runtime i fer una gestió conjunta dels recursos optimitzant el cost global de les execucions i l'ús dels recursos.

7.5 Valoració personal

Personalment, valoro el treball realitzat molt positivament. El resultat d'aquest projecte és una cosa que al començar la carrera fa cinc anys no m'havia imaginat que pogués fer. En part perquè desconeixia el món del càlcul distribuit, en part perquè el Cloud Computing encara no s'havia desenvolupat.

Al cap d'uns anys de carrera, presentant una pràctica al despatx d'un professor, vaig descobrir el Còmput Voluntari i BOINC. Des d'aquell moment, una de les curiositats que em va perseguir al llarg de tota la carrera era com es podia desenvolupar un programa que fes servir un ordinador qualsevol d'Internet i utilitzar el resultat d'aquest per poder executar tot un programa molt més gran? COMPSs va ser la resposta a aquesta pregunta.

Una altra cosa que he d'agraïr al projecte és la visió que m'ha aportat del Cloud Computing. Quan van començar a publicar-se els primers articles en revistes sobre el tema, el primer que vaig pensar era que simplement era un producte comercial que es llençava per aprofitar els recursos que les empreses amb grans datacenters tenien; però que tenia un ús molt limitat de cara al gran públic i que no tenia gaires perspectives de futur. A mesura que vaig anar llegint publicacions més seriososes sobre el tema i avançant en el projecte, vaig anar veient que els autors d'aquells artícles donaven una visió equivocada que no havia de plantejar-se de cara a la gent sinó de cara a les empreses.

Al definir el projecte, els objectius suposaven un repte per mi. Significaven conèixer una part de la informàtica que encara no havia vist mai, el Cloud Computing. A més a més, havia d'aprendre un nou model de programació, entendre el funcionament del seu runtime i finalment modificar-lo perquè fos capaç d'utilitzar els recursos del Cloud i de decidir com gestionar aquests recursos. De totes maneres no era la gestió de recursos típica que havia vist en assignatures de Sistemes Operatius en que el que es gestionava un sol recurs, el processador, i es repartia entre un conjunt de processos. En aquest cas tenia un problema que fins ara no havia vist: gestionar la quantitat de recursos que pot fer servir aquest conjunt.

Pel que fa al prototip, estic bastant content del resultat ja que compleix tots els objectius marcats a l'inici del projecte. Per una part, crec que els mecanismes desenvolupats per poder crear, destruir i utilitzar les màquines virtuals funcionen bé, i no afegeixen gaire càrrega computacional al runtime i la pèrdua de rendiment que suposen és negligible. I crec que aquesta part del disseny i implementació està molt ben feta. Ja que permet a l'usuari definir el connector de la manera que vulgui sense necessitat de retocar el codi de COMPSs, només ha de crear una classe que implementi les funcions de les APIs del connector. I així s'aconsegueix la independència del proveïdor de Cloud que es marcava com a objectiu.

Estic content del rendiment que tenen les polítiques implementades. De totes maneres, no m'he quedat totalment satisfet amb el disseny de les polítiques de planificació de tasques. Tot i tenir un rendiment, en alguns casos, millor que la versió de COMPSs inicial, també tenen un rendiment més baix en algunes aplicacions per no tenir en compte certs paràmetre a l'hora de decidir on és millor executar una tasca. No són prou completes. Han quedat moltes idees pendents d'implementar i provar per falta de temps i que realment han estat descartades perquè no eren part dels objectius.

Estic content no només pel projecte realitzat sinó també per escollir aquesta carrera i centre. Més enllà dels coneixements tècnics que s'aprenen al llarg dels 5 anys, també s'inculca una forma de treballar i tota una filosofia a l'hora d'afrontar qualsevol problema:

reduir-lo a molts de petits i aprofitar les solucions dels altres. Considero aquest projecte un bon punt i final als meus estudis d'Enginyeria Informàtica i espero que sigui un bon punt d'inici per estudis i treballs futurs.

Apèndixs

Apèndix A

Organització de l'entorn de treball de COMPSs

En aquest annex es comenta l'estructura de l'entorn de treball del model de programació COMP Superscalar, coincident amb el contingut de la carpeta COMPSs del fitxer adjunt a aquesta memòria.

En el directori arrel d'aquesta carpeta s'hi poden trobar les següents carpetes:

gridunawareapps conté les aplicacions preparades per executar-se utilitzant COMPSs. Dintre de la carpeta hi podem trobar:

- **lib** conté tots el jars necessaris per executar les aplicacions en mode seqüencial, inclòs el jar guapp.jar que agrupa totes les aplicacions.
- **src** on es troba el codi font de les aplicacions dividit en 3 carpetes:
  - **api** on hi ha el codi principal d'aplicacions fent servir l'API de COMPSs.
  - **sequential** on hi ha el codi principal d'aplicacions sense fer servir l'API de COMPSs.
  - **worker** amb el codi de les tasques que s'executaran remotament.

integratedtoolkit on es troba el runtime, repartit en 2 carpetes:

- **lib** conté tots el jars necessaris per executar el runtime, inclòs el jar IT.jar que és el runtime en sí.
- **src** on es troba el codi font del runtime:
  - **integratedtoolkit** és el package arrel del runtime. Dins seu hi ha tot el runtime dividit en els subpackages:
    - **api** inclou les interfícies necessàries per invocar el runtime des de les aplicacions i la seva implementació.
    - **components** inclou les interfícies necessàries per comunicar els components del runtime i la seva implementació.
    - **connector** inclou les interfícies necessàries perquè el ResourceManager pugui comunicar-se amb el Cloud Provider. També, inclou el codi del connector específic per EMOTIVE Cloud i un conjunt de classes útils per simplificar la programació de nous connectors.

- **control** conté les classes que defineixen el comportament de ProActive a l'hora d'encendre i apagar els components
- **interfaces** conté la definició de les interfícies necessàries perquè la implementació de l'API pugui comunicar-se amb els components del runtime
- **loader** conté el codi dels diferents loaders amb els que es pot carregar les aplicacions i modificar el codi fent servir javassist.
- **log** únicament conté una classe que permet identificar tots els Apache Loggers que es faran servir dintre del runtime
- **types** conté la definició dels tipus amb els que opera el runtime, per exemple: Task o Method.
- **util** conté les diferents classes que faran servir els components per gestionar la informació, per exemple: el ResourceManager o el QueueManager.
- **worker** conté el codi de l'aplicació que han de tenir els workers i que es necessita per executar una tasca qualsevol en un node remot.

log conté un fitxer de configuració per descriure quins Apache Loggers fan servir les execucions de COMPSs i la seva configuració.

worker conté els scripts i classes necessaris per poder fer el desplegament de les aplicacions als workers físics.

xml on trobem els xml amb la configuració dels diferents elements que necessita el runtime:

- **adl** fitxers necessaris per descriure l'estructura de components que forma COMPSs per ProActive.
- **deployment** els seus fitxers descriuen de quina manera ProActive desplegarà els components que formen COMPSs.
- **control** els seus fitxers descriuen de quina manera es comportarà ProActive per tal d'encendre i apagar els components
- **projects** cada un dels seus fitxers descriu com treballar en els recursos remots en un escenari concret.
- **resources** cada un dels seus fitxers descriu els recursos remots que es poden fer servir en una execució.

resource_list fitxer que conté tots els recursos on s'ha de fer el desplegament del worker de COMPSs de manera automàtica.

Apèndix B

Contingut del fitxer adjunt a la memòria

A continuació es comenta el contingut del fitxer adjunt a aquesta memòria. A l'arrel d'aquest fitxer es pot trobar el fitxer **index.html** que mostra el mateix que aquest apèndix, que facilita l'accés a la informació en el cas de que no es disposi de la memòria. A més a més d'aquest document la informació està organitzada en les següents carpetes:

COMPSs Que conté l'entorn de treball del model de programació tal i com queda descrit a l'apèndix A.

docs On s'hi poden trobar els següents documents relacionats amb el projecte:

Manual d'instal·lació de COMPSs i execució corresponent a l'apèndix C

Manual d'usuari de COMPSs

software conté un document amb les URLs des d'on es pot baixar tot el software necessari per executar el prototip. A més a més, també hi ha la imatge virtual necessària per encendre els workers virtuals i el Scheduler d'EMOTIVE Cloud fet servir.

Apèndix C

Instal·lació de COMPSs i execució d'una aplicació

A continuació s'exposen les instruccions per tal de poder executar les aplicacions d'exemple que porta COMPSs.

Es parteix de la base que ja es disposa d'un Cloud gestionat amb el middleware EMOTIVE Cloud al que es pot accedir directament des de la màquina on s'executarà el master de COMPSs. Primer s'explica de quina manera cal preparar l'entorn de treball del master i, després, quins canvis caldrà fer sobre el Cloud per tal de que sigui capaç de donar suport a COMPSs.

C.1 Preparació del master

El primer que cal tenir en compte a l'hora de preparar l'entorn de treball és que el node que actuarà com a master necessita poder identificar-se per accedir als workers. Per aquest motiu és necessari que el node tingui una parell de claus ssh i que aquest pugui accedir mitjançant ssh a qualsevol dels workers que es vulguin utilitzar sense password. En cas de no tenir-ho cal generar una clau privada sense password amb la comanda.

```
ssh-keygen -t dsa -f /home/user/.ssh/id_dsa -P ‘’
```

I afegir el contingut del fitxer $HOME/.ssh/id_dsa als fitxers $HOME/.ssh/authorized_keys de totes les màquines remotes que es vulguin utilitzar com a workers.

El segon aspecte que s'ha de tenir en compte per poder executar una aplicació de COMPSs són les dependències de software que necessita resoldre el runtime.

C.1.1 Java

La dependència més bàsica de totes és un entorn per desenvolupar en Java. Per tant, s'haurà d'instal·lar un SDK de Java . Proposem el SDK oficial de Sun que es pot descarregar de la pàgina *http://www.oracle.com/technetwork/java/javase/downloads/index.html*. Els passos necessaris per la instal·lació dependràn del SDK que es faci servir, el que sí que és important és que a l'acabar el procés es defineixi la variable d'entorn JAVA_HOME amb el path cap al directori on s'ha instal·lat Java. Per exemple:

export JAVA_HOME=/home/user/jvms/jdk1.6.0_18

En el cas de treballar en un entorn UNIX, pot ser convenient afegir aquesta variable i totes les que caldrà definir més endavant al .bashrc o equivalent segons la distribució que es faci servir

C.1.2 Apache Ant

Apache Ant és una eina utilitzada per la realització de tasques mecàniques i repetitives, en el cas de COMPSs s'utilitza en el desplegament dels workers físics i en la compilació i construcció, tant de les aplicacions com del runtime.

L'eina pot descarregar-se de la pàgina *http://ant.apache.org/*. Per intal·lar-la, només cal descomprimir el fitxer descarregat en una carpeta i definir la variable d'entorn ANT_HOME amb el directori on s'ha descomprimit l'aplicació. Per exemple:

export ANT_HOME=/home/user/ant/

C.1.3 ProActive

Un altre software necessari per poder executar el runtime és ProActive. Com ja hem explicat al capítol 4, ProActive és l'eina que COMPSs fa servir per comunicar els seus components.

La instal·lació d'aquest software torna a ser molt senzilla, només caldrà anar a la pàgina *http://www.activeeon.com/community-downloads* i descarregar l'última versió de ProActive Java API i descomprimir la carpeta en el directori desitjat. Igual que en els altres casos cal definir una variable d'entorn, PROACTIVE_HOME per saber on trobar aquesta aplicació. Per exemple pot definir-se de la següent manera:

export PROACTIVE_HOME=/home/user/proactive/

C.1.4 JavaGAT

Per acabar, l'últim software que COMPSs necessita per poder-se executar és JavaGAT. Com també es diu al capítol 4, JavaGAT és fa servir per les comunicacions entre master i workers i donar a COMPSs independència del middleware o protocol que comuniqui els diferent nodes del Grid.

Igual que en els casos anteriors la instal·lació de JavaGAT consta de 3 passos molt simples. El primer descarregar-lo de la pàgina web *http://gforge.cs.vu.nl/gf/project/javagat/frs/*, el segon descomprimir el fitxer en una carpeta i el tercer definir una variable d'entorn que ens permeti trobar-la, GAT_LOCATION.

export GAT_LOCATION=/home/user/JavaGAT/

C.1.5 COMPSs

Arribats a aquest punt, ja totes les dependències de software s'han resolt, només falta tenir el runtime i les aplicacions. Tots els fitxers que es necessiten per instalar COMPSs

es troben en el fitxer adjunt a la memòria. Per instalar COMPSs s'ha de copiar la carpeta COMPSs del fitxer en el master i definir la variable d'entorn IT_HOME. Suposant que es copia la carpeta a /home/user:

```
export IT_HOME=/home/user/COMPSs
```

Un cop estigui la carpeta copiada i la variable definida, cal afegir una llibreria d'ANT perquè aquest pugui fer servir el protocol ssh a l'hora de fer el desplegament dels workers físics. Per això és necessari executar:

```
cd $IT_HOME
ant install
```

En aquest moment l'únic que falta per tenir el runtime de COMPSs és compilar el runtime. Per això només cal anar a carpeta integratedtoolkit i executar la comanda ant lib que s'encarrega de tota la compilació del runtime.

```
cd $IT_HOME/integratedtoolkit
ant lib
```

C.2 Preparació d'EMOTIVE Cloud

Abans de poder començar a executar les aplicacions cal fer un parell de canvis a EMOTIVE Cloud. Cal modificar el Simple Scheduler perquè; doni el temps que preveu que tardarà a servir una petició. També, cal donar a EMOTIVE Cloud la imatge que hauran de carregar les màquines que es creïn com a workers de COMPSs.

C.2.1 Extensió del Scheduler

Per canviar l'scheduler d'EMOTIVE s'ha de canviar la carpeta $EMOTIVE_HOME/Scheduler/SimpleSchedulerREST/src/main/java/net per la carpeta net que es troba dintre la carpeta software/SimpleScheduler del fitxer adjunt.

Després de substituir tot el package, cal compilar i instal·lar els Schedulers d'EMOTIVE Cloud utilitzant l'script install.sh

```
install.sh scheduler
```

Per acabar només falta reiniciar el servidor Apache amb el Scheduler d'EMOTIVE Cloud, per això cal anar a la carpeta on hi ha l'Apache-Tomcat de la màquina i reinciar-lo amb el següent codi:

```
cd bin
./catalina.sh stop
./catalina.sh start
```

C.2.2 Afegir la imatge dels workers

Descomprimir el fitxer COMPSsworker.img.zip de la carpeta Software del fitxer adjunt i copiar el fitxer que conté, COMPSsworker.img, a la carpeta images de tots els nodes que hi ha al pool del Cloud. En el cas presentat al llarg de tota la memòria amb Pctinet i PcPerot, el que s'hauria d'executar el següent codi:

```
unzip COMPSsworker.img.zip
cp COMPSsworker.img /aplic/brein/pool/pctinet/images/
cp COMPSsworker.img /aplic/brein/pool/pcperot/images/
```

C.3 Compilació i execució d'aplicacions

Un cop preparat tot el que es necessita per poder executar el runtime en el Cloud i crear màquines virtuals capaces d'executar les tasques només falten les aplicacions. El primer que s'ha de fer és compilar les aplicacions. Per fer-ho s'ha d'executar el següent codi:

```
cd $IT_HOME
ant guapp
```

El segon pas per executar qualsevol aplicació consisteix en preparar els workers perquè puguin executar les tasques i indicar al runtime quins workers físics es volen fer servir. Es Modificar el fitxer $IT_HOME/resources_list perquè contingui tots els recursos que es volen fer indicant node i usuari d'accés seguint el següent esquema:

user@node:

Per tal de copiar totes les classes que han de poder executar els nodes remots. Per fer-ho n'hi ha prou executant la comanda:

```
ant worker
```

Per acabar només falta indicar al runtime quins nodes pot utilitzar per executar les tasques. Per això cal fer crear els fitxers xml de recursos i projecte a $IT_HOME/xml/resources/ i $IT_HOME/xml/projects respectivament tal i com es mostra al punt 2.1 del manual d'usuari de COMPSs.

Un cop definits aquest fitxers ja es poden executar les aplicacions. Per fer-ho només cal entrar a la carpeta $IT_HOME/gridunawareapps i executar el script guapp.sh indicant els següents paràmetres:

- Mode: pot ser IT, si es vol fer servir el COMPSs, o sequential, si es vol executar en mode seqüencial.
- Project: ruta del fitxer xml de projecte.
- Resource: ruta del fitxer xml de recursos.
- Connector: Class package i nom de la classe que es farà servir com a Connector.
- Cloud Server: nom del Servidor de Cloud al que el runtime s'ha de connectar.

- Cloud Port: port del Servidor de Cloud al que el runtime s'ha de connectar.
- Cloud image: imatge amb la que s'han d'encendre les màquines virtuals del Cloud.
- Initial CloudVM Count: número de màquines que es volen a l'iniciar l'aplicació.
- Maximum CloudVM Count: número màxim de màquines que es volen de cop al llarg de l'aplicació.
- Loader: indica si es vol utilitzar l'API de COMPSs, partial, o si es vol que el runtime s'encengui a l'iniciar l'aplicació i es mantingui encès fins al final d'aquesta, total.
- Application Main Class: package i nom de la classe que conté el mètode principal de l'aplicació.
- Application Arguments: paràmetres de l'aplicació.

En el cas de voler executar un SparseLU de mida 10 connectant amb PcTinet pel port 8080 com a Servidor del Cloud fent servir el connector per SimpleScheduler amb 0 màquines inicials i un màxim de 6 workers en Cloud i utilitzant les màquines físiques descrites als fitxers $IT_HOME/xml/projects/compss.xml i $IT_HOME/xml/resources/compss.xml, la comanda que s'hauria d'executar seria la següent:

```
./guapp.sh IT $IT_HOME/xml/projects/compss.xml $IT_HOME/xml/resources/
compss.xml emotivecloud.SimpleScheduler pctinet port 8080 compssworker 0 6
total sequential.sparselu.SparseLU 10
```

Bibliografia

[1] M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. Katz, A. Konwinski, G. Lee, D. Patterson, A. Rabkin, I. Stoica, and M. Zaharia. Above the Clouds: A Berkeley View of Cloud Computing. Technical Report 2009-28, UC Berkeley, Febrer 2009.

[2] L. M. Vaquero, L. Rodero-Merino, J. Caceres i M. Lindner. A Break in the Clouds: Towards a Cloud Definition. ACM SIGCOMM Computer Communication Review, Volume 39, Number 1, Gener 2009

[3] 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)

[4] 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.

[5] J. Geelan. Twenty-one experts define cloud computing, 2009. (Disponible a http://cloudcomputing.sys-con.com/node/612375/)

[6] M. Klems. Classification of Cloud Computing Stakeholders, Juliol 2008. (Disponible a http://markusklems.wordpress.com/2008/07/10/classification-cloud-computing/)

[7] J.C. Moreno. FAQ Cloud Computing, Març 2009. (Disponible a http://www.saasmania.com/faq-sobre-cloud-computing/)

[8] Platform Computing. Enterprise Cloud Computing: Transforming IT, Juliol 2009. (Disponible a http://www.hpcwire.com/specialfeatures/cloud_computing/casestudies/Enterprise-Cloud-Computing-Transforming-IT-81574347.html)

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

[10] F. Baude, D. Caromel, C. Dalmassp, M. Danelutto, V. Getov, L. Henrio i C. Pérez. GCM: A Grid Extension to Fractal for Autonomous Distributed Components, Setembre 2008. (Disponible a http://hal-unice.archives-ouvertes.fr/docs/00/32/39/19/PDF/GCM.pdf)

[11] E.Bruneton, T.Coupaye, i J.B. Stefani. The Fractal Component Model. Technical Report, ObjectWeb Consortium, February 2004. (Disponible a http://fractal.objectweb.org/specification/index.html)

[12] J.Czyzyk, M.Mesmer, i J.J. Moré. The Neos Server. IEEE Journal on Computational Science and Engineering, Volum 5, Setembre 1998. (Disponible a http://ieeexplore.ieee.org/search/freesrchabstract.jsp?arnumber=714603)

[13] D.Chappell. Introducing Windows Azure, Desembre 2009. (Disponible a http://go.microsoft.com/?linkid=9682907)

[14] D.Chappell. Windows Azure and ISVs, Juliol 2009 (Disponible a http://go.microsoft.com/fwlink/?LinkID=157857)

[15] 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)

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

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

[18] Web d'Amazon (http://aws.amazon.com/ec2/)

[19] Web de Microsoft Azure (http://www.microsoft.com/windowsazure/)

[20] Web de Google AppEngine (http://code.google.com/intl/es-ES/appengine/docs/)

[21] Web de Rackspce (http://www.rackspacecloud.com/)

[22] Web GoGRID (http://www.gogrid.com/cloud-hosting/)

[23] http://wiki.appnexus.com/

[24] E. Tejedor, R. Badia, R.Royo i J.Gelpi.Ennabling HMMER for the Grid with COMP Superscalar. Procedia Computer Science, 2010 (Disponible a http://www.sciencedirect.com/science)

[25] A.Quinn i S. Hunter. HMMer Benchmarking for ELIXIR. Març 2008.

[26] P. Wayner. Cloud versus cloud: A guided tour of Amazon, Google, AppNexus, and GoGrid. Juliol 2008. (Disponible a http://www.infoworld.com/print/37122)

[27] http://www-05.ibm.com/services/es/igs/cloud-development/

[28] IBM Global Technology Services. IBM Smart Business Development and Test on the IBM Cloud. Agost 2010

[29] http://www.eucalyptus.com/

[30] Eucalyptus Systems, Inc. Eucalyptus Open-Source Cloud Computing Infrastructure - An Overview. Agost 2009 (Disponible a http://www.eucalyptus.com/pdf/whitepapers/Eucalyptus_Overview.pdf)

Glossari

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

CDN, Content Distribution Network sistema de computadors que contenen diferents còpies de les dades situades en diferents punts de la xarxa. Els clients accedeixen a les dades que es troben més pròperes, maximitzant l'ample de banda.

Depèndencia Relació entre dos blocs d'instruccions en que un fa referència a algun dels operands o resultats d'un anterior que encara no ha acabat. Alterar l'ordre d'execució pot provocar un comportament erroni de l'aplicació.

Dependència RaW Lectura després d'una escriptura. El segon bloc ha de llegir un valor que genera el primer. Al llegir abans de que el bloc més vell escrigui pot provocar una lectura incorrecta.

Dependència WaR Escriptura després de lectura. Escriure el valor abans de que el primer el llegeixi provoca que el bloc més vell operi amb el resultat de la més jove, podent donar una lectura incorrecta.

Dependència WaW Escriptura després d'una escriptura. Si la segona escriptura del valor s'avança a la primera, totes les lectures posteriors d'aquell valor poden llegir el valor que hi escrigui el bloc més vell.

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[3].

Latència suma de retards que es produeixen en realitzar una acció.

Màquina virtual software que emula un computador i permet executar programes com si fos un computador real, independentment de les equivalències del hardware.

NP-Complet dit d'aquells problemes que no poden ser results en un temps polinòmic en funció de la mida de la seva entrada.

Overhead temps de computació, memòria, ample de banda o altres recursos que es requereixen per atendre un objectiu concret.

Padding augment del temps de computació requerit per resoldre una tasca.

SLA, Service Level Agreement contracte escrit entre un proveïdor de servei i el seu client en què es documenta el nivell acordat per a la qualitat del servei.

Testbed Conjunt de recursos organitzats per tal de fer les proves.

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

TimeStamp seqüència de caràcters que indiquen la data i hora en que es produeix un event.

URI cadena curta de caràcters que identifica inequivocament un nom o un recurs a Internet.

Virtualització creació d'una capa d'abstracció entre el hardware de la màquina física (host) i el sistema operatiu de la màquina virtual (guest).

VPN Tecnologia de xarxa que permet estendre una xarxa local sobre una xarxa pública no controlada, com per exemple Internet.