UNIVERSIDAD DE CHILE
FACULTAD DE CIENCIAS FISICAS Y MATEMATICAS
DEPARTAMENTO DE CIENCIAS DE LA COMPUTACION

BALANCE DE CARGA DINAMICO PARA OBJETOS ACTIVOS MOVILES EN GRILLAS DE COMPUTADORES

TESIS PARA OPTAR AL GRADO DE DOCTOR EN
CIENCIAS, MENCION COMPUTACION

JAVIER ALEJANDRO BUSTOS JIMENEZ

PROFESOR GUIA:
JOSE MIGUEL PIQUER GARDNER

MIEMBROS DE LA COMISION:
GONZALO NAVARRO BADINO
MAURICIO MARIN CAIHUAN
PIERRE COINTE

SANTIAGO DE CHILE
DICIEMBRE 2006

UNIVERSIDAD DE CHILE
FACULTAD DE CIENCIAS FISICAS Y MATEMATICAS
DEPARTAMENTO DE CIENCIAS DE LA COMPUTACION

BALANCE DE CARGA DINAMICO PARA OBJETOS ACTIVOS MOVILES EN GRILLAS DE COMPUTADORES

TESIS PARA OPTAR AL GRADO DE DOCTOR EN CIENCIAS, MENCION COMPUTACION

JAVIER ALEJANDRO BUSTOS JIMENEZ

PROFESOR GUIA:
JOSE MIGUEL PIQUER GARDNER

MIEMBROS DE LA COMISION:
GONZALO NAVARRO BADINO
MAURICIO MARIN CAIHUAN
PIERRE COINTE

SANTIAGO DE CHILE
DICIEMBRE 2006

to Cristina
... and Amelia

Resumen

Esta tesis apunta a entregar las bases para el desarrollo de los algoritmos de balance de carga para el modelo de objetos activos definido por *ProActive* en el contexto de las redes a gran escala (grillas).

ProActive es un middleware implementado en lenguaje Java, de código abierto, para la programación concurrente, en paralelo, distribuido, y emóvil, poniendo el paradigma de objeto-activo en ejecución. En ProActive, cada objeto activo tiene su propio hilo de control y puede decidir independientemente en qué orden servir los métodos entrantes, las cuales se almacenan automáticamente en una cola de peticiones pendientes. Para agregar eficacia al paradigma de objetos activos, ProActive proporciona un mecanismo del *migración*, obteniendo localización automática y transparencia mediante el uso de *forwarders*. La migración viene con una costo de comunicación: un objeto activo debe emigrar con su estado completo, consistiendo en sus peticiones pendientes (llamadas de método), futuros, y sus objetos pasivos. Por lo tanto, las aplicaciones implementadas con ProActive son sensibles a la latencia.

Cuando varios objetos activos con funcionalidad idéntica se despliegan, un algoritmo de balance de carga puede ser utilizado para mejorar el funcionamiento de la aplicación utilizando esa funcionalidad. La carga de trabajo puede ser equilibrada a través de varios objetos activos *enviando* objetos activos de un procesador altamente cargado a uno menos cargado, o bien *robando* objetos activos de un procesador altamente cargado por uno menos cargado. El ambiente donde funcionan las aplicaciones implementadas usando el modelo de objetos activos se compone generalmente de grupos múltiples de recursos, por ejemplo, un sistema de máquinas interconectadas por una red local de alta velocidad.

Por lo tanto, un algoritmo de balancede carga para objetos activos que pertenecen a una aplicación paralela fue desarrollado y estudiado, fijando las bases para el desarrollo de los algoritmos de balance de carga para el middleware ProActive. Este primer acercamiento se llama el algoritmo *Robin-Hood + Nottingham Sheriff*. Este algoritmo fue validado en el contexto de una red de alta escala (sobre 1.000 nodos) mediante simulaciones, utilizando nuestros modelo de las grillas basados en la observación y la medición de lo que consideramos las características dominantes para el balance de objetos activos: capacidad de procesamiento y latencia entre recursos.

Finalmente, presentamos los contratos de acoplamiento para el despliegue de aplicaciones paralelas, su forma de utilización en el contexto de balance de carga, por ejemplo, elegir entre un planificador local y el balanceador de carga de ProActive

Abstract

This thesis aims to set the foundations for the development of load-balancing algorithms for the active objects model defined by *ProActive* in the context of large-scale networks (Grids).

ProActive is an open-source Java middleware which achieves seamless programming for concurrent, parallel, distributed, and mobile computing, implementing the active-object paradigm. In ProActive, each active object has its own control thread and can independently decide in which order to serve incoming method calls. Incoming method calls are automatically stored in a queue of pending requests (called a *service queue*). To add efficiency to the active objects paradigm, ProActive provides a *migration* mechanism, having automatic location and transparency through the use of forwarders. The migration operation comes with a communication penalty: an active object must migrate with its complete state, consisting of its pending requests (method calls), futures, and passive (mandatory non-shared) objects. Therefore, ProActive applications are sensitive to latency.

When several active objects with identical functionality are deployed, a load-balancing algorithm can be used to improve the performance of an application using that functionality. We call the total work units required by an application to finish by *workload*. To use some active object's functionality, an application puts a work unit into the service queue of that active object. The workload can be balanced across several active objects either by *sending* active objects from a highly loaded processor to a less loaded one, or by *stealing* active objects from a highly loaded processor by a less loaded one. The environment where the active objects run is usually composed of multiple clusters of resources, such as sets of machines interconnected by a high-speed local network.

Therefore, a load-balancing algorithm for active objects which belong to a parallel application was developed and studied, setting the foundations for development of load-balancing algorithms for the middleware ProActive. This first approach is called the *Robin-Hood + Nottingham Sheriff* load-balancing algorithm. This algorithm was validated in the context of a large-scale network (over $1{,}000$ nodes) through simulations. Therefore, we presented our model of Grids based on observation and measurement of what we consider key characteristics for active-objects load-balancing: *processing capacity* and *inter-resource communication latency*.

Finally, we present coupling contracts for deployment of parallel applications, showing how to use them in the context of load balancing by now choosing between a local scheduler (for clusters) and ProActive's load balancer.

Contents

List of Figures **10**

1 Introduction **2**

2 Active Objects **6**

2.1 Active Objects 7

2.2 Reflection 8

2.2.1 Reflective Architecture 9

2.3 ProActive 10

2.3.1 Distribution model 11

2.3.2 Active Objects implementation for ProActive 12

2.3.3 Message Passing for Actives Objects in ProActive 13

2.3.4 Synchronisation: Wait-by-necessity 15

2.3.5 ProActive: Environment and implementation 16

2.3.6 ProActive Meta-Object Protocol 19

3 Networks for parallelism 24

3.1 History of parallel computing 25

3.1.1 Cluster of computers 25

3.1.2 Computer Grids 27

3.1.3 A model overview for Project Grids 28

3.2 Peer-to-Peer Infrastructure of ProActive 29

3.2.1 Bootstrapping: First Contact 31

3.2.2 Discovering and Self-Organising 31

3.3 Theory of Networks 33

3.3.1 Generating random graphs 34

3.3.2 Natural Networks 35

4 State of the Art on Load-Balancing 39

4.1 Static Load-Balancing 40

4.2 Dynamic Load-Balancing 41

4.3 Components of a Load-Balancing Algorithm 44

4.3.1 Load Index 45

4.3.2 Information-Sharing Policy 46

4.3.3 Transfer Policy 47

4.3.4 Location Policy 49

4.4 Related Work 49

4.4.1 Condor 49

4.4.2 Legion 52

4.4.3 Cilk 55

4.4.4 Satin 58

5 Setting foundations for Load-Balancing of Active-Objects 61

5.1 Active-Objects and Processing Idleness 62

5.2 Location policy for load-balancing of active-objects 64

5.3 Information and transfer policies for load-balancing of active-objects 65

5.3.1 Modelling ProActive behaviour to test algorithm policies 65

5.3.2 Implementing the Information-Sharing Policies 66

5.3.3 Hardware and Software 69

5.3.4 Results Analysis 70

5.3.5 Testing the impact of Information-Sharing Policies 74

5.4 Exploiting the Peer-to-Peer infrastructure: Information on-demand 75

5.4.1 Robin-Hood Load-Balancing Algorithm 76

5.4.2 Robin-Hood over ProActive's Peer-to-Peer Infrastructure 77

5.5 Robin-Hood and the Nottingham Sheriff 79

5.6 Testing algorithms in a real environment 80

6 Models, Simulations and Deployment on Large-Scale Networks 83

6.1 Simulating Desktop Grids 84

6.1.1 Characterising nodes of Desktop Grids 84

6.1.2 Modelling Desktop Grids 85

6.1.3 Finding the best processor 87

6.1.4 Scaling towards the "infinite network" 95

6.2 Simulating Project Grids 103

6.2.1 Characterising a Project Grid . . . 106

6.2.2 Modelling a Project Grid . . . 108

6.2.3 Environment-aware Algorithms . . . 110

6.2.4 Experimental Setup . . . 111

6.2.5 Simulation Results . . . 112

6.2.6 Results Confidence . . . 114

6.3 Where to run parallel applications? . . . 118

6.3.1 Problematic of Applications and Descriptors . . . 118

6.3.2 Clauses in ProActive Descriptors . . . 119

6.3.3 Clauses in ProActive Applications . . . 121

6.3.4 Constraints . . . 121

6.4 The real world . . . 124

7 Conclusions and Future Work **129**

A Matrices for Robin-Hood algorithm working alone **133**

B Matrices for Robin-Hood + Nottingham-Sheriff algorithm **138**

C Expected values for Kolmogorov-Smirnov test statistics **143**

Bibliography **146**

List of Figures

2.1 The reflection process, featuring levels of data, reification and reflection. 9

2.2 Parallelisation and distribution with active objects 11

2.3 Execution of an asynchronous and remote method call 14

2.4 Base-level and meta-level of an active object 19

2.5 Migration and tensioning 22

3.1 Grids divided by objective 29

3.2 (a) step two of Watts and Strogatz model with $n = 12$ and $k = 2$; (b) step three with small p_e 36

4.1 A supermarket 40

4.2 Examples of information-sharing policies 47

4.3 Matchmaking process of Condor 50

4.4 Parallel problems solved by *Condor* 51

4.5 Main classes of Legion infrastructure 53

4.6 Legion Resource Management Infrastructure 55

4.7 *Cilk* model: each thread is a circle, grouped in procedures. Each downward arrow is a spawned child, and each horizontal arrow is a spawned successor. Dashed arrows represent data dependency (synchronisations). Also, spawn-levels from the original thread are presented. 56

5.1 Different behaviours for active-objects request (Q) and reply (P): (a) B starts in wait-for-request (WfR) and A made a wait-by-necessity (WfN). (b) Bad utilisation of the active-object pattern: asynchronous calls become almost synchronous. (c) C has a long waiting time because B delayed the answer. 62

5.2 The supermarket abstraction for load-balancing of enqueued tasks. 63

5.3 The supermarket abstraction for load-balancing of Active Objects. 63

5.4 Migration time from the point of view of latency and object' size 64

5.5 Mean response time for all policies 71

5.6 Bandwidth usage of coordination policies during the information-sharing phase . . 72

5.7 Bandwidth usage of coordination policies during all the load-balancing 73

5.8 Impact of load-balancing algorithms over Jacobi calculus 82

6.1 Frequency distribution of Mflops for $200,000$ processors registered at Seti@home and the normal function which models it. 86

6.2 Final distribution for the *Robin-Hood* algorithm only, for $RB = 0.5$ and $T = 0.5$ . . 89

6.3 Final distribution for the *Robin-Hood* + *Nottingham Sheriff* 91

6.4 Tuning for RS considering: a) number of active-objects in $(9,9)$ per total of active-objects; and b) Number of total migrations reaching a stable state. 92

6.5 Tuning for RS considering: a) number of active-objects in $(9,9)$ per total of active-objects; and b) Number of total migrations reaching a stable state. Because the results using 3 to 6 acquaintances were similar, only those for 3 are shown. 94

6.6 Tuning for RS considering: a) mean number of total migrations until each time-step; and b) mean number of overloaded nodes in each time-step. Using $RB = 0.7$, acquaintances subset size = 3, $|x - y| \leq 3$, $\lambda = 0.1, 0.2, 0.3$ and $T = 0.7$ 96

6.7 Tuning the value of RS considering: a) mean number of active objects on a node with $\mu \geq 1$ per total number of active objects; and b) mean number of active objects on a node with $\mu > 1 + \frac{1}{3}$ per total number of active objects. Using $RB = 0.7$, acquaintances subset size = 3, $|x - y| \leq 3$, $\lambda = 0.1, 0.2, 0.3$ and $T = 0.7$ 98

6.8 Scalability for a network using $RS = 0.9, 1.0, 1.1$, $RB = 0.7$ 100

6.9 Scalability in terms of number of processors used, having $RS = 1.0$ 102

6.10 Scalability in terms of number of migrations, having $RS = 1.0$. The plot presents, for an active object, the (mean) number of accumulated migrations performed until a time-step $t \in [0; 1,000]$. 103

6.11 Scalability, having the number of active objects proportional to the number of nodes 104

6.12 Latency between nodes from the PlugTest project grid. 108

6.13 Total number of pending requests in all active-objects using message-size $C = 0.1$ and object size $M = 1$, without synchronisation. 114

6.14 Total number of pending requests in all active-objects using message-size $C = 1$ and object size $M = 10$, without synchronisation. 115

6.15 Total number of pending requests in all active-objects using message-size $C = 0.1$ *services*, object size $M = 1$ *services* and synchronisation each 10 *time-steps*. 116

6.16 % of confidence of load-balancing algorithms, increasing object size (M) 117

6.17 Example of clauses in descriptor. 120

6.18 Example of clauses in application. 122

6.19 Integer Constraint Schema Grammar. 123

6.20 Institutional clusters on Grid5000: Bordeaux, Grenoble, Lille, Lyon, Nancy, Orsay, Rennes, Sophia-Antipolis and Toulouse. 124

6.21 Speed of Jacobi parallel application in iterations per milliseconds. 126

6.22 Mean number of cumulated migrations that an active object performs during the experience. 127

A.1 Final distribution for the *Robin-Hood* algorithm only, for $RB = 0.5$ and $q = 3$ 134

A.2 Final distribution for the *Robin-Hood* algorithm only, for $RB = 0.5$ and $q = 4$ 135

A.3 Final distribution for the *Robin-Hood* algorithm only, for $RB = 0.5$ and $q = 5$ 136

A.4 Final distribution for the *Robin-Hood* algorithm only, for $RB = 0.7$ and $q = 4$ 137

B.1 Final distribution for the *Robin-Hood + Nottingham Sheriff* algorithm, for $RB = 0.5$, $RS = 0.5$ and $q = 3$ 139

B.2 Final distribution for the *Robin-Hood* + *Nottingham Sheriff* algorithm, for $RB = 0.5$, $RS = 0.5$ and $q = 5$ 140

B.3 Final distribution for the *Robin-Hood* + *Nottingham Sheriff* algorithm, for $RB = 0.7$, $RS = 0.7$ and $q = 3$ 141

B.4 Final distribution for the *Robin-Hood* + *Nottingham Sheriff* algorithm, for $RB = 0.9$, $RS = 0.9$ and $q = 3$ 142

Acknowledgements

I would like to thank both the French Embassy at Chile and the Chilean Commission in Research and Technology (Conicyt) that granted me a scholarship that allowed me to pursue further education in France and Chile.

I am specially thankful to my adviser José Piquer, who demonstrated an amazing dedication guiding me through my studies and who encouraged me to go to France. His support has been incommensurable.

I am most grateful to INRIA, the Oasis team, and its former members. Thanks to Denis Caromel, my adviser in France, who accepted to support my thesis. Special thanks to Tomás Barros, Alfredo Illanes, Mauricio Araya, Gonzalo Robledo and Christian Delbé, who helped me with all the paperwork at the beginning of my French life, without their help I probable would had been deported. I would also like to thank all the people who shared their knowledge in useful discussions about my thesis work: Eric Tanter (Objects), Alexandre di Costanzo (ProActive's P2P infrastructure), Nelson Morales (Network Modelling), Angela Ganz (Poisson Processes), Satu Elisa Schaeffer (Natural Networks) and Luis Mateu (Synchronisation).

I would like to thank to all people who helped me in "non-academic" ways during this PhD. First to my Chilean friends whom gave me their support asking me from time to time "where are you now?" (Geddy, Lemus, Benja, Iván, Fernando, Pato, Pancho, Humberto, Gastón, Teresa, Valeria and Vicky). Second to my French friends whom gave me their support asking me from time to time "when you will back?" (Arthur, Nico and all the Garibaldi F.C. team). Third to the tennis players (Fernando, Humberto, Luis, Tomás, Tamara, Ángela and Mario). Fourth to the beach-volley players (the "Argentine team", specially Tamara and Jimena; Igor, Carlos, Marcela, etc.). Fifth to "la vida del estudiante" group (Marcelo, Ángela, Diego, Patricio and Elena). Sixth to the co-author by default (Mario). Seventh to my favourite proof-reader (Elisa). Finally, to three special places for me: *Stade du Ray*, *La foyer (Valrose)* and *Pub van Gogh*. It is also my privilege to thank all those who shared their friendship along these last four years.

Chapter 1

Introduction

> *"A herd of buffalo can only move as fast as the slowest buffalo. And when a herd is hunted, its the slowest and weakest ones in the back, that are killed first...". (Cliff Clavin from "Cheers")*

This thesis aims to set the foundations for the development of load-balancing algorithms for the active objects model defined by *ProActive* (94) in the context of large-scale networks (Grids).

ProActive is an open-source Java middleware which achieves seamless programming for concurrent, parallel, distributed, and mobile computing, implementing the active-object paradigm (128). In ProActive, each active object has its own control thread and can independently decide in which order to serve incoming method calls. Incoming method calls are automatically stored in a queue of pending requests (called a *service queue*). When the queue is empty, active objects wait for the arrival of a new request; this state is known as `wait-for-request`. Active objects are accessible remotely via method invocation. Method calls with active objects are asynchronous with automatic synchronisation using *future objects*, and the synchronisation is provided automatically handled by a `wait-by-necessity` mechanism (31).

To add efficiency to the active objects paradigm, ProActive provides a *migration* mechanism, that is, a way to move any active object from any Java Virtual Machine (JVM) to another JVM (11). The remote references towards the active objects that have been migrated must remain valid after the migration; in ProActive, forwarding of requests and replies provide automatic location and transparency. The migration operation comes with a communication penalty: an active object must migrate with its complete state, that means, its pending requests (method calls), futures, and passive (mandatory non-shared) objects. Therefore, ProActive applications are very sensitive to latency.

When several active objects with identical functionality are deployed, a load balancing algorithm can be used to improve the performance of an application using that functionality (44; 46; 86; 101; 106) . Such an application must make use of the needed functionality several times until it finishes. We denote each functionality use by *work unit*. We denote the total work units required by an application to finish by *workload*. To use some active object's functionality, an application puts a work unit into the active object's service queue. The workload can be balanced across several active objects either by *sending* active-objects from a highly loaded processor to a less loaded one (push model), or by *stealing* active-objects from a highly loaded processor by a less loaded one (pull model). For the Grid case, the environment where the active objects run is usually composed of multiple clusters of resources, e.g., a set of monitor-less machines inter-connected by a high-speed local network. In ProActive, the active objects form a P2P network (24); the load-balancing algorithm should also take into consideration the topology of this network. Note that for ProActive applications latency is a key performance estimator.

Given the impossibility of having access to a large-scale network (over 1,000 nodes) to perform all the tests needed, most of the time we perform network simulation to adjust algorithm parameters. In that case, we present our model of Grids based in observation and measurement of what we consider the key characteristics for active-objects load-balancing: *processing capacity* and *inter-resource communication latency*. The grid computing research community has started to realise the importance of validated models for simulation work. Therefore, there have been several approaches in the last 2-3 years (81; 69; 73; 84; 67). However, to the best of our knowledge, ours is the first approach to research the characteristics of these components of the grid infrastructure.

This thesis is organised as follows. First we explain the concept of *object* in Chapter 2, followed by the *Active-Objects Model* and its implementations. Chapter 3 introduces the concept of *network* and *Grids* in the context of parallel computations. Chapter 4 presents the state of the art in *Load-Balancing models and algorithms*, explaining in Chapter 5 *why* performing load-balancing in a parallel application developed within ProActive will speed up these applications and setting the foundations for Load-Balancing of Active-Objects. In Chapter 6 we present and discuss our Grid and active-objects *models* used during fine-tuning and testing of our load-balancing algorithm for active-objects. Finally, conclusions of this thesis and discussions of future work are presented in Chapter 7.

The time-line of this thesis is:

- First, we tested if the communication architecture of ProActive's active objects fits in the information-collection phase of well known schemes of load-balancing. This work lead to a publication presented in Sixth IEEE International Symposium and School on Advanced Distributed Systems (ISSADS 2006) (27).

- Then, we focused in minimising the number of messages used by the load-balancing algorithm. Using the previous work and well-known facts of *Peer-to-Peer* networks we proposed to perform the information phase on-demand, exploiting the probability of having a response of another processor due the low probability of having overloaded processors and the high number of processors connected to a *Peer-to-Peer* network. We demonstrated that load-balancing of active-obejcts parallel applications can be performed using a minimal set of acquaintances in Section 5.4.2, having better performance than a server-oriented scheme. This work generated a publication presented in 25th International Conference of the Chilean Computer Science Society (SCCC 2005) (24).

- Our load-balancing algorithm performed balancing until a stable-state (without overloaded processors) is reached, but we experimentally determined that commonly this algorithm did not reach optimal configurations, even using best qualified processors, because its objective was to perform fast reactions against overloadings. Therefore, we added a *work-stealing* (13; 18; 36) step to our algorithm, aiming to reach optimal configurations. This work is presented

in Chapter 6 and generated a publication presented on 12th Workshop in Job Scheduling Strategies for Parallel Processing (28).

- We noted that sometimes our load-balancing algorithm did not speed up a given parallel application in some Grids which we defined as *Project Grids* (Section 3.1.3), even though active-objects were grouped on the best qualified processors. Therefore, we improved our model to consider object communication and synchronisation, discovering the usefulness of environment-awareness in load-balancing algorithms. This work is presented in Section 6.2 and generated a publication accepted in CoreGRID Integration Workshop 2006 (25).

- Finally, we noted that for some configurations the first deployment of a given parallel application influenced both application and load-balancer performance (25), therefore we recommend to use of *contracts for coupling* to improve the first deployment of a ProActive parallel application. This work generated a publication presented in CoreGRID Workshop on Grid Middleware (in conjunction with EuroPar) 2006.

Contributions of collaborators that are discussed in this thesis are the following:

- in Section 3.1.3, the definition of *Project Grids* is a joint work of Alexandru Iosup and the author.

- In Section 3.2, the original infrastructure of a *Peer-to-Peer* network developed within active-objects is by Alexandre di Costanzo, the model presented in this thesis is an optimization by the author presented in (24).

- In Chapter 6, simulations of algorithms are inspired on the implementation of "small-world" networks by Kleinberg in (71).

- In Section 6.3, coupling contracts is an idea generated by Mario Leyton, arithmetic and contract use for discovering resources are by the author.

Chapter 2

Active Objects

> *"It has therefore recently suggested that one should combine a shared variable and the possible operations on it in a single, syntactic construct called monitor. It is, however, too early to speculate about what this approach may lead to". (Per Brinch-Hansen, 1973)*

In 1994, Grady Booch (21) documented the model of *objects*, describing which are the characteristics that a *standard object* must provide:

1. **Data Encapsulation**: the techniques of data encapsulation (4) restrict the data access to a set of functions associated to that data. In the Object Oriented Paradigm, the unit of data encapsulation is *the object*. Each object encapsulates a set of variables (a *state*) and a set of used methods to access and to modify the variables (an *interface*). The only way to use the data is by the invocation (*call*) of some of the methods that compose the interface of the object. Therefore, the state of the object between method invocations is preserved.
2. **Inheritance**: In the Object Oriented Paradigm, it appears in a natural way the concept of *class* to express the common characteristics between objects with identical behaviour that

are different only by their state. Each class defines the interface and encapsulates the state of that class. Inheritance (15) is a technique that allows a set of classes to share parts of a common interface and behaviour (methods and variables).

3. **Polymorphism**: polymorphism allows that different objects respond to the same message and will be the system, at runtime, which decides the suitable interpretation of the message based on the concrete instance of an object. It allows to write different behaviours for a same interface, and the decision from which one to use could be taken based on the parameters received during the call. Polymorphism through inheritance consists on the redefinition of a method in such a way that, when a method in an object is invoked, the decision of which method will be executed to answer the messages, is taken at execution time.

More recently, a similar definition of objects is provided by Wegner in (126):

> "*Objects are collections of operations that share a state. The operations determine the **messages** (calls) to which the object can respond, while the shared state is hidden from the outside world and is accessible only to the object's operations. Variables representing the internal state of an object are called **instance variables** and its operations are called **methods**. Its collection of methods determines its interface and its behaviour.*".

In this chapter, we first define an *active-object*, followed by the concept of reflection in object oriented programming, and finishing with the description of an implementation of active-objects using reflection: ProActive.

2.1 Active Objects

Due to the great popularity and acceptance of the Object Oriented (OO) Paradigm, several concurrent OO programming languages have been designed and implemented, based on the model of

concurrent objects where *each object is an active organisation* (128) or *all objects are not active but the active entities are objects* (30; 29). Nevertheless, from the point of view of the operating system, each object was a process with an only thread of control. Therefore, it was imperative to write great amount of additional code to support the abstractions by the objects.

As a consequence of the abstraction overcost, the *object/thread* model (92) was introduced in 1995 in the context of an Operative System called *Clouds* (41). In this model, the *objects* are address spaces with names that provide storage of data and methods for the manipulation. They are passive entities which provide functions to share data and synchronisation. On the other hand, the *threads* represent the control flow in the system by the invocation and later execution of methods.

One advantages of this model is its good performance, because multiple threads can run at the same time in mono-processors with low cost. However, a main disadvantage is the *mutual exclusion*, because threads are running independently. Also, the introduction of external code to the object to perform synchronisation adds complexity to the programming, specially if it is used combined with the inheritance.

2.2 Reflection

In human terms, reflection is the act of thinking about the own ideas, actions and experiences. In the field of the computer systems, the reflectivity appeared first in the field of the Artificial Intelligence and then it was quickly propagated to other fields like programming languages (108), and in the object-oriented technologies, where it was introduced by Pattie Maes (82).

Several definitions of reflectivity exist, and the most extended, with some modifications, it was given by Bobrow, Gabriel and White (19):

> ”*Reflection* is the ability of a program to manipulate as data something representing the state of the program during its own execution”.

In this manipulation two fundamental aspects exist: introspection and intercession. *Introspection* is the ability of a program to observe and to reason about its own state. *Intercession* is the ability of a program to modify its own state of execution or to alter its own interpretation (19; 83). Both aspects require a mechanism that codifies the state of execution like data, *Reification* provides such codification.

2.2.1 Reflective Architecture

A reflective architecture provides a mean to introduce the reflecting computation in a modular way, which makes the system more comprehensible and easy to modify. It is, then, common to think about a structured reflecting system or compound, from a logical point of view, by two or more levels, which build a *reflective tower* (122) (Figure 2.1). Each level serves as a **base level** for the upper level and reflects to the lower level. The base solves the external problem while the reflecting part (**meta-level**) maintains information and determines the behaviour of the bases.

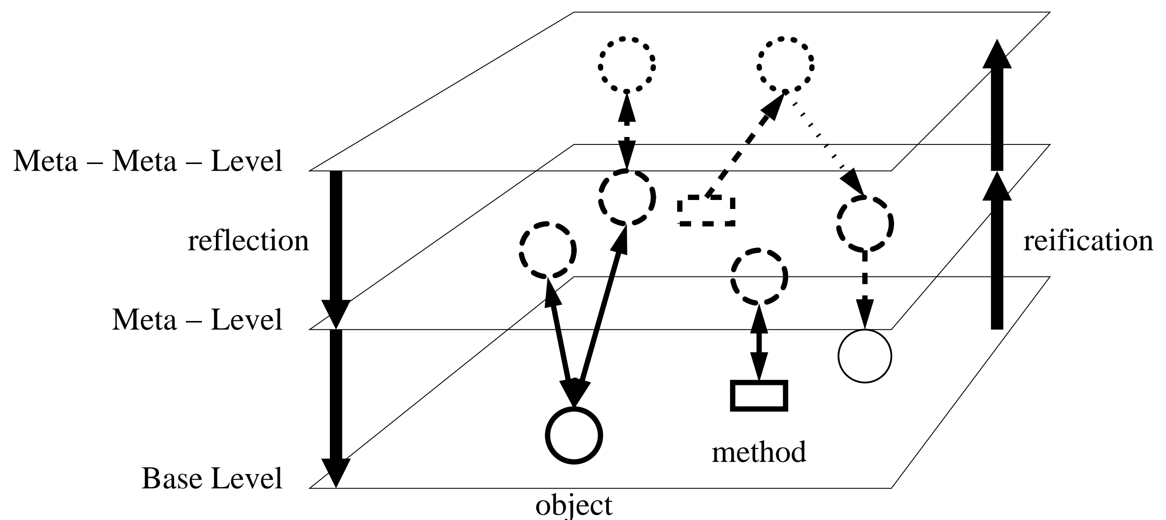


Figure 2.1: The reflection process, featuring levels of data, reification and reflection.

Moreover, the work of Jaques Ferber "*Computational Reflection in Class based Object oriented*

Languages" (51) presents the key features that all reflective architectures must perform:

- A reflective architecture has to determine which aspects are wanted to be reflected, that is to say, what organisations and/or characteristics must be exposed.
- A reflective architecture has to determine the representation of the system within the system. There are, at least, two approaches to build self-representation of a computer system: To assume the existence of a data set that represents the system and to introduce the self-representation of each organisation as an individual form in the system (123).
- A reflective architecture has to maintain the cause-effect relation between the model of the system and the system itself (between base and meta-levels)
- A reflective architecture has to determine how to activate the meta-computation and when the control returns to the base level.

In next section, we introduce *ProActive* as a *reflective* implementation of *Active Objects*.

2.3 ProActive

ProActive is an open source[1] Java library for parallel, distributed, and concurrent computing, also featuring mobility and security in a uniform framework. With a reduced set of simple primitives, *ProActive* provides a comprehensive API allowing to simplify the programming of applications that are distributed on Local Area Networks (LAN), on clusters of workstations, or on Internet Grids. *ProActive* uses only standard Java classes, and requires no changes to the Java Virtual Machine, no pre-processing or compiler modification; programmers write standard Java code. Based on a simple Meta-Objects Protocol, the library is itself extensible, making the system open for adaptations and optimisations. *ProActive* currently uses the RMI Java (Sun Microsystems) standard library as the default portable transport layer.

[1] Source code under LGPL license

2.3.1 Distribution model

The *ProActive* library was designed and implemented with the aim of importing reusability into parallel, distributed, and concurrent programming in the framework of a MIMD[2] model. Reusability has been one the major contributions of object-oriented programming, and *ProActive* brings it into the distributed world. Most of the time, activities and distribution are not known at the beginning, and they change over time. Seamlessness implies reuse, smooth and incremental transitions.

The model of distribution and activity of *ProActive* is part of a larger effort to improve simplicity and reuse in the programming of distributed and concurrent object systems (31; 32), including precise semantics (5). It contributes to the design of a concurrent object calculus named ASP (Asynchronous Sequential Processes) (33; 34). As shown in Figure 2.2, *ProActive* seamlessly transforms a standard centralised mono-threaded Java program into a distributed and multithreaded program.

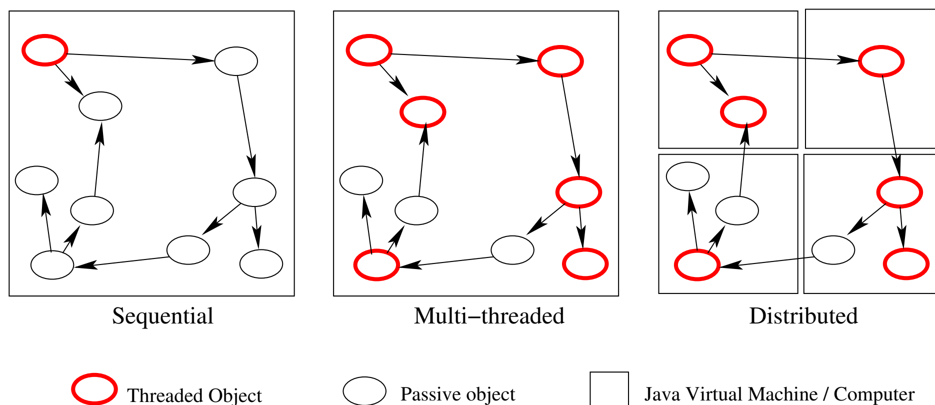


Figure 2.2: Parallelisation and distribution with active objects

[2]MIMD stands for Multiple Instruction Multiple Data

2.3.2 Active Objects implementation for ProActive

A distributed or concurrent application built using *ProActive* is composed of a number of medium-grained entities called *active objects*. Each active object has one distinguished element, the *root*, which is the only entry point to the active object. Each active object has its own thread of control and is granted the ability to decide in which order to serve the incoming method calls that are automatically stored in a queue of pending requests. Objects that are not active are designated as *passive*.

There are three ways to transform a standard object into an active one:

1. The *Class-based* approach is the more static one. A new class must be created extending an existing class, and must implement the `Active` interface. The `Active` interface is a tag interface that does not specify any method. This approach allows adding specific methods useful in distributed environment and possibly to define a new service policy in place of the default *First In First Out* (FIFO) service (see ProActive's documentation at (94) for further details about service policy).

   ```
   public class pA extends A implements Active { }
   Object[] params = new Object[] {"s", new Integer (28)};
   A a = (A) ProActive.newActive("pA", params, node);
   ```

   The array of objects `params` represents the parameters to use for the remote creation of the object of type `A`. The parameter `node` is an abstraction of the physical location of an active object (cf. Section 2.3.5).

2. With the instantiation-based approach, a Java class that does not implement the `Active` interface is directly instantiated without any modification to create an active object. The parameters `params` and `node` play the same role as above.

   ```
   Object[] params = new Object[] {"s", new Integer (28)};
   A a = (A) ProActive.newActive("A", params, node);
   ```

3. Finally, the object-based approach allows transforming an already existing Java object into an active object, possibly remote. It is possible to turn both, active and remote objects, for which the source code is not available, a necessary feature in the context of code mobility. If the `node` parameter is `null` or designates the local JVM, new elements are created to transform the object into active object (those elements are meta-objects presented in Section 2.3.6). Otherwise, if `node` refers to a remote JVM a copy of the object is sent on the remote JVM and transformed into an active object. The original passive object remains on the local JVM.

```
A a = new A ("s", 28);
a = (A) ProActive.turnActive(a, node);
```

2.3.3 Message Passing for Actives Objects in ProActive

The active object creation primitives of *ProActive* locally return an object compatible with the original type because of polymorphism. For instance, at the `A` class:

```
public class A {
   public void methodVoid () {...}
   public V getaV () {...}
   public V getanotherV () {...} throws AnException {...}
}
```

The methods provided by `class A` could be remotely invoked but the communication semantics would differ:

- The method named `methodVoid` does not return any result, so it will perform only a communication from the caller to the callee. This is a *one-way* method call.
- The `getaV` method requires a bidirectional communication. Firstly, from the caller to the callee, then from the callee to the caller in order to return the result. With *ProActive* this

communication is separated into two steps detailed below. Between the steps, activity of the caller does not stop because this is an *asynchronous* method call.

- The `getanotherV` method is quite similar to the previous method except that it can raise an exception. Therefore, the call to `getanotherV` is managed as a *synchronous* method call. Methods returning a primitive type or a final class are also invoked in a synchronous way.

Objects given as parameters are copied on the caller side to be transmitted to the callee side.

Second and third previous cases are explained in Figure 2.3, which exposes an asynchronous call sent to an active object and introduces transparent *future objects* and synchronisation handled by a mechanism known as *wait-by-necessity* (31). There is a short rendezvous at the beginning of each remote call, which blocks the caller until the call has reached the context of the callee. In Figure 2.3, step 1 blocks until step 2 has completed. At the same time a *future* object is created (step 3). A *future* is a promised result that will be updated later, when the reply of the remote method call returns to the caller (step 5). The next section presents synchronisation and control of such *futures*.

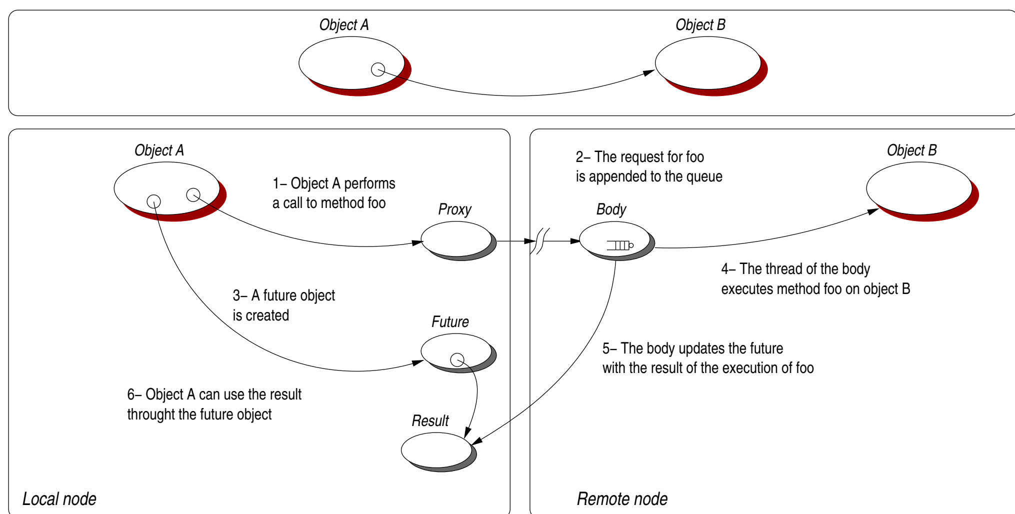


Figure 2.3: Execution of an asynchronous and remote method call

A synchronous method call proceeds in similar steps, with two main differences. Firstly, the future is not created (no step 3). This is due to the incapacity of the Meta-Objects Protocol to create a future in the case the return type does not belong to a class. Secondly, the activity of the caller stops until step 5 has completed (instead of steps 2/3 for an asynchronous call).

ProActive features several optimisations improving performance. For instance, whenever two active objects are located within the same virtual machine, a **direct communication** is always achieved, without going through the network stack. This optimisation is ensured even when the co-location occurs after a migration of one or both of the active objects.

2.3.4 Synchronisation: Wait-by-necessity

Let our active object `a`:

```
A a = (A) ProActive.newActive("A", params, node);
```

and the asynchronous method call:

```
V v = a.method();
```

As previously seen, `v` is a future. *ProActive* automatically deals with future objects with a *wait-by-necessity* mechanism. Consider the new instruction:

```
v.glop();
```

There is no guarantee that the future `v` was updated before the method `glop` is invoked.If the result has arrived and hence the future has been updated when the call to `glop` is executed, activity never stops. However, if the future has not yet arrived, the wait-by-necessity mechanism stops the current activity until the future object is returned, after which the activity is resumed and the method is executed. The *wait-by-necessity* mechanism ensures a maximum efficiency of the asynchronism.

2.3.5 ProActive: Environment and implementation

ProActive is only made of standard Java classes, and requires no change to the *Java Virtual Machine* (JVM), no pre-processing or compiler modification; programmers write standard Java code. Using an unmodified Java development and execution kit, and the standard Java classes ensures portability and allows running applications with all the JVM implementations. For debugging, which is especially critical in a distributed environment, avoiding source code modification is more efficient. *ProActive* uses reflection techniques in order to manipulate runtime events such as a method call. Supplementary code is dynamically generated in the same fashion used by *generative* or *active* libraries (39; 119). Based on a simple Meta-Object Protocol, the library is itself extensible, making the system open for adaptations and optimisations. *ProActive* currently uses the RMI Java standard library as a portable communication layer.

Mapping active objects to JVMs: Nodes

A *Node* is an object defined in *ProActive* whose aim is to gather several active objects in a logical entity. It provides an abstraction for the physical location of a set of active objects. At any time, a JVM hosts one or several nodes. The traditional way to name and handle nodes in a simple manner is to associate them with a symbolic name, which is a URL giving their location, for instance `rmi://sea/node1`.

Consider a standard Java class `A`. The following instruction creates a new active object of type `A` on the JVM identified with `node1`.

```
A a1 = (A) ProActive.newActive("A", params, "rmi://sea/node1");
```

Assigning no third parameter or passing a `null` value will cause the active object to be created on the local JVM (i.e. the JVM in which the `newActive` primitive is called). Also, passing an active object as parameter triggers the co-allocation mechanism. For instance, the active object `a4` will be created in the JVM containing the active object `a1`:

```
A a4 = (A) ProActive.newActive("A", params, a1);
```

Note that an active object can also be bound dynamically to a node as the result of a migration.

Node deployment

Active objects will eventually be deployed on very heterogeneous environments where security policies may differ from place to place, where computing and communication performances may vary from one host to the other, etc. As such, the effective locations of active objects must not be tied to the source code.

A first principle is to eliminate from the source code the computer names, the creation protocols and the registry and lookup protocols. The goal is to deploy any application anywhere without changing the source code. For instance, we use various protocols (`rsh`, `ssh`, Globus GRAM, LSF, etc.) for the creation of the JVMs needed by the application. In the same manner, the discovery of existing resources or the registration of the ones created by the application can be done with various protocols such as RMIregistry, Jini, Globus MDS, LDAP, UDDI, etc. Therefore, the creation, registration, and discovery of resources have to be done externally to the application.

To reach that goal, the programming model relies on the specific notion of *Virtual Nodes* (VNs):

1. a VN is identified by a name (a simple string),
2. a VN is used in a program source,
3. a VN is defined and configured in a deployment descriptor, and,
4. a VN, after activation, is mapped to one or more nodes.

The concept of virtual nodes as entities for mapping active objects has been introduced in (10). Those virtual nodes are described externally through XML-based descriptors which are then read

at runtime when necessary. They help in the deployment phase of *ProActive* active objects (and components).

Active objects are created on Nodes, not on Virtual Nodes. Both concepts, Nodes and Virtual nodes, are justified and necessary. Virtual Nodes are a much richer abstraction, as they provide mechanisms such as cyclic mapping, for instance. Moreover, a Virtual Node is a concept of a distributed program or component, while a Node is actually a deployment concept: it is an object that lives in a JVM, hosting active objects. There is of course a correspondence between Virtual Nodes and Nodes: the function created by the deployment, the mapping. This mapping can be specified in an XML descriptor. By definition, the following operations can be configured in such a deployment descriptor:

1. the mapping of VNs to Nodes and to JVMs,
2. the way to create or to acquire JVMs, and,
3. the way to register or to lookup VNs.

Now, within the source code, the programmer can manage the creation of active objects without relying on machine names and protocols. For instance, the following piece of code allows creating an active object onto the Virtual Node `Dispatcher`. The Nodes (JVMs) associated in a descriptor file with a given VN are started (or acquired) only upon activation of a VN mapping (`virtualNode.activateMapping()` in the code below).

```
Descriptor pad = ProActive.getDescriptor("file://des.xml");

VirtualNode myVirtualNode = pad.getVirtualNode("vnode");
myVirtualNode.activateMapping();
Node node = virtualNode.getNode();
A a = ProActive.newActive("A", params, node);
```

2.3.6 ProActive Meta-Object Protocol

ProActive is built on top of a *Meta-Object Protocol* (MOP) (70) that permits reification of method invocations and constructor calls. As the MOP is not limited to the implementation of the transparent remote objects library, it also provides an open framework for implementing powerful libraries for the Java language. As all other elements of *ProActive*, the MOP is entirely written in Java and does not require any modification or extension to the Java Virtual Machine, unlike other Meta-object protocols for Java (72; 118). ProActive makes extensive use of the Java Reflection API.

An active object provides a set of services, in particular asynchronous communication, but it is important to separate concerns to ensure extensibility and maintenance. A meta-object was introduced for each service provided by an active object. Figure 2.4 shows the final decomposition.

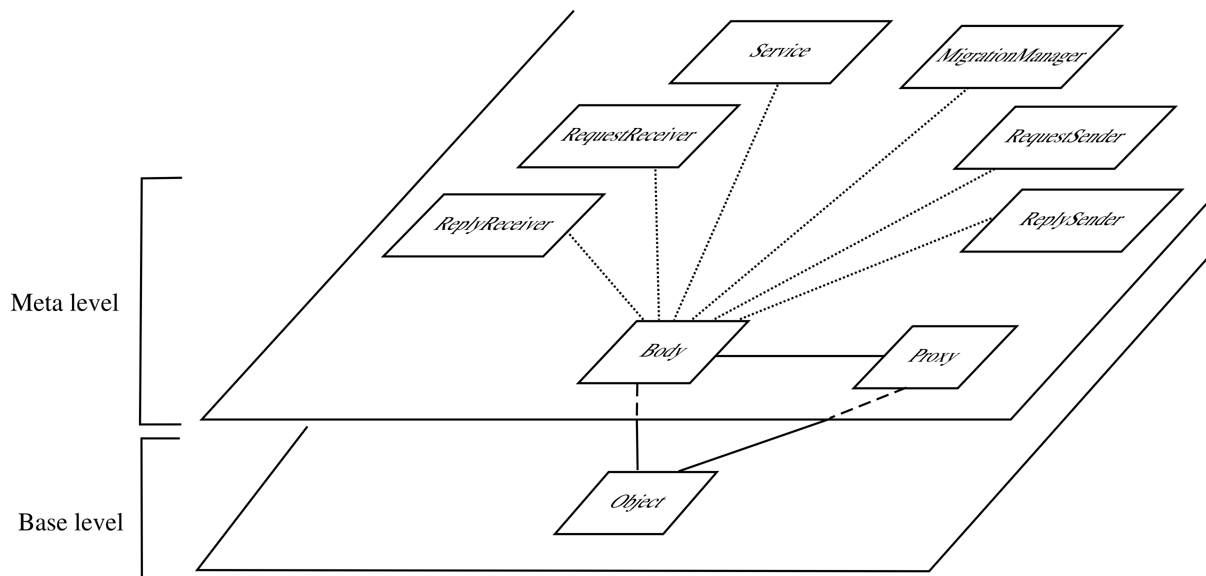


Figure 2.4: Base-level and meta-level of an active object

The MOP creates the *stub/proxy* pair and the *body* with its *meta-objects*. The stub is an entry point for the meta-level and it inherits the type of the object. Being a 100% Java library, the MOP has a few limitations: primitive types cannot be reified because they are not instances of a standard

class nor final classes (including all arrays) because they cannot be sub-classed. So primitive types and final classes are said to be *not reifiable*. The stub overloads the public methods of the class. A method invocation creates a `MethodCall` object that represents the executed method call. This object contains the invoked `Method`, information about the return type, and a copy of each parameter.

The proxy maintains a reference to the active object. It is responsible for the communication semantics:

1. it hides the concept of remote or local reference, and
2. it transmits the `MethodCall` object (embedded into a `Request` object) to the body of the active object.

The body is the entry point for all communications addressed to the active object. It is the only remotely accessible part of the active object. The body is in charge of the meta-objects attached to it. A request queue is attached to the body and it stores messages sent to the body from local objects or other active objects. Requests are served with a FIFO service policy by default, and this can be customised by the programmer.

Migration

Mobility is the ability to relocate at runtime the components of a distributed application. The *ProActive* library provides a way to migrate an active object from any JVM to any other one (11). *ProActive* migrations are *weak*, which means that the code moves but not the execution state (as opposed to *strong mobility*). Activity restarts from a *stable state*.

Any active object has the possibility to migrate. If it references some passive objects, they will also migrate to the new location. Since we use serialisation to send the object on the network, an active object has to implement the `Serializable` interface to be able to migrate. The migration

of an active object is triggered by the active object itself, or by an external agent. In both cases a single primitive will eventually get called to perform the migration. The principle is to have a very simple and efficient primitive to perform migration, and then to build various abstractions on top of it. The name of the primitive is `migrateTo`. For ease of use of the migration, the `ProActive` class provides two sets of static methods.

The first set of methods handles migration triggered by the active object wishing to migrate. These methods rely on the calling thread being the active thread of the active object:

- `migrateTo(Object o)`: migrate to the same location as an existing active object,
- `migrateTo(String nodeURL)`: migrate to the location given by the URL of the node,
- `migrateTo(Node node)`: migrate to the location of the given node.

The second set of methods is intended for migration triggered by some other agent than the active object being migrated. For instance, in this thesis the migration is triggered by *Load Balancing Active Objects*. In this case the external agent must have a reference to the Body of the active object it wants to migrate.

- `migrateTo(Body body, Object o, boolean priority)`: migrate to the same location as an existing active object
- `migrateTo(Body body, String nodeURL, boolean priority)`: migrate to the location given by the URL of the node
- `migrateTo(Body body, Node node, boolean priority)`: migrate to the location of the given node

The `priority` parameter represents two possible strategies:

1. The request is high priority and is processed before all existing requests the body may have received (*priority* = *true*);

2. The request is normal priority and is processed after all existing requests the body may have received (*priority* = *false*).

To answer the location problem (find a migrated object, maintain connectivity), two mechanism were proposed: *forwarders* and *location servers* (63). A forwarder is a reference left by the active object when it leaves a host: this reference points the new location of the object. Multiple migrations create a chain of forwarders; some elements of chains may become temporarily or permanently unreachable because of a network partition or a single machine in the chain failure. Longer chains produce worse performance because of multiple "hops" of the message. Therefore, ProActive uses *tensioning* to shortcut the chain of forwarders: after a migration, the first method call updates the location of the migrated object to the caller and creates a direct link. This mechanism is presented in Figure 2.5.

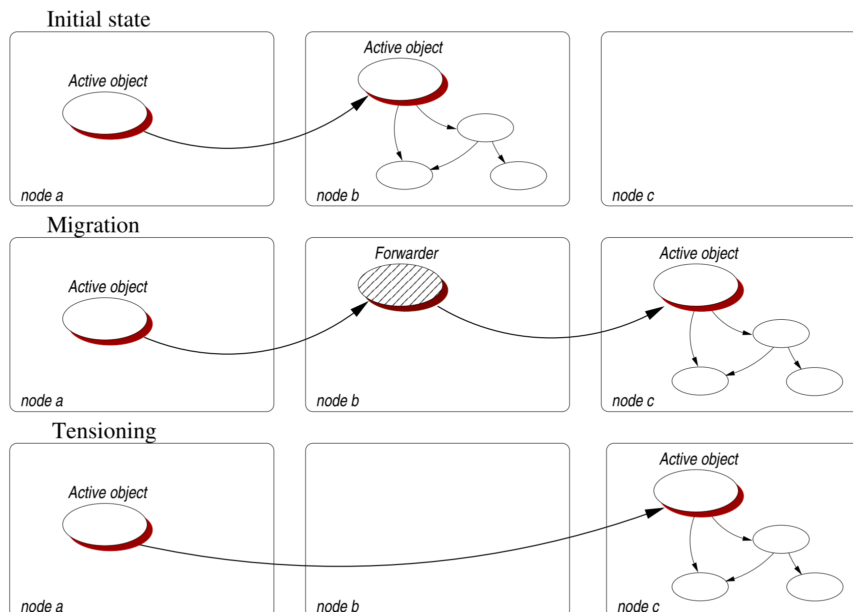


Figure 2.5: Migration and tensioning

With the second solution, the location server tracks the location of each active object. Every time an object migrates, it sends its new location to the location server. After a migration, all the references pointing to the previous location become invalid. When an object attempts to communicate with a migrated active object (through an invalidated reference), the call fails, triggering a lazy mechanism that transparently performs the following steps:

1. queries the location server for the new location of the active object,
2. updates the reference regarding to the server's response, and
3. re-performs the call on the object at its new location.

Contrary to the forwarder approach, the location server approach produces additional messages: first, messages from the migrated object to the server, and second, due to the failed communication attempt. Further discussion about the two approaches in the context of ProActive can be found on the PhD thesis of Fabrice Huet (63).

Chapter 3

Networks for parallelism

"God creates men, but they choose each other." (Niccolo Machiavelli)

Michael Flinn proposed in 1972 a classification of computer architecture based on kind of processing and data (53):

- **SISD** (Single Instruction, Single Data): Sequential processing of instructions and data. Not parallelism at all.
- **SIMD** (Single Instruction, Multiple Data): Exploiting data parallelism to solve parallel problems; for instance, to apply non-deterministic algorithms for NP-hard problems in parallel.
- **MISD** (Multiple Instruction, Single Data): Exploiting instruction parallelism using data redundancy, for instance in real-time architectures.
- **MIMD** (Multiple instruction, Multiple Data): Exploiting full parallelism, having a set of instructions processing different (sections of shared) data.

MIMD derived in Single Program, Multiple Data (**SPMD**) (7), multiples processors executing the same set of instructions (a *program*) on different data.

In this chapter we review the state of the art of networks oriented to parallel computing from a historical point of view of implemented networks and from a point of view of theory of large-scale networks.

3.1 History of parallel computing

Since the birth of *ENIAC* (Electronic Numerical Integrator and Computer), known as the first *computer* capable to solve a set of *computing problems* (107), the scientific world has been searching for ways to use the computer potential in solving hard problems (like *NP-problems*) in a parallel way. Initially, the main problem in this quest was the price of computers: processors where so expensive that most organisations had the means to build only one, or a few but working separately.

Around 1985, the development of *microprocessors* produced computing power at low-cost (compared with previous mainframes), and the scientific world studied again the way to solve their problems using sets of microprocessors. The first attempt was to use microprocessors connected by a data bus and sharing memory and devices using a SIMD scheme (also known as a *multiprocessor* computer), but then the invention of high-speed computer networks allowed the connection of hundred of machines (processor + memory + devices) in a *cluster*.

3.1.1 Cluster of computers

The history of clusters of computers is directly related with the history of computer networks, as one of the primary motivation for the development of a network was to link computing resources, creating computer clusters. Packet switching networks were firstly studied by the RAND corpora-

tion[1] in 1962. Exploiting the concept of a packet switched network, the Research Projects of U.S. Department of Defense (*ARPANET*) built the foundations of what we know today as *the Internet*. Internet is the strong interconnection of computer resources using packet switching and the Internet paradigm is the basis of cluster communication.

The development of clusters started in early 1970s supported by the development of networks (TCP/IP protocol) and the Unix operating system. However, the protocols and tools for easily doing remote job distribution and file sharing were defined around 1983 in the context of BSD Unix (as implemented by Sun Microsystems). In 1984 DEC released their *VAXcluster* (75) product for the VAX/VMS operating system.

The academia presented one of their first infrastructures to interconnect a pool of processors providing a Distributed Operating System (mean to coordinate processors) in 1986 with the *Amoeba* project (111), developed by Andrew Tanenbaum et al. from 1986 to 1995. This project reported on its web-page[2] that:

> *Amoeba is a powerful microkernel-based system that turns a collection of workstations or single-board computers into a transparent distributed system. It has been in use in academia, industry, and government for about 5 years. It runs on the SPARC (Sun4c and Sun4m), the 386/486, 68030, and Sun 3/50 and Sun 3/60. At the Vrije Universiteit, Amoeba runs on a collection of 80 single-board SPARC computers connected by an Ethernet, forming a powerful processor pool.*

A key point in the development of cluster computing was the birth of *Parallel Virtual Machine* (PVM) systems in 1990 (110), which allows the creation of a *virtual supercomputer* made of TCP/IP connected (and low-cost) normal computers. In 1995 the invention of a computer cluster built for the specific purpose of "*being a supercomputer*" using an Internet-like network (called a *Beowulf cluster* (109)), and the development of long-distance high-speed networks allowed that most of the clusters be inter-connected, building kinds of *cluster of clusters* or *computer grids*.

[1] Research team of U.S. Army. http://www.rand.org

[2] http://www.cs.vu.nl/pub/amoeba

3.1.2 Computer Grids

The Grid is the next level of abstraction in computer networks, exploiting the high-speed interconnection of a set of distributed computers and clusters in order to solve large-scale parallel problems as a unique virtual computer architecture. Therefore, Grid computer has to handle interconnection and resource sharing (as a normal cluster); and new services such as resource allocation or resource management.

The name "*Grid*" first appeared on the work of Ian Foster and Carl Kesselman called *Computational grids* (from the book "*The grid: blueprint for a new computing infrastructure*", 1999) (55). Foster is the team leader of Globus Alliance[3], which develops the "Globus Toolkit". Globus Toolkit is a middleware to perform grid management, providing services of CPU and storage management, security provisioning, data movement, monitoring and which also provides a toolkit for developing additional services based on the same infrastructure. The importance of Globus in Grid computing built a direct association of the name of *Ian Foster* with the *Grid* concept.

Ian Foster defined "*a Grid*" in his article "*What is the Grid? A Three Point Checklist*" (54) as a system that:

- coordinates resources that are not subject to centralised control
- ... using standard, open, general-purpose protocols and interfaces
- ... to deliver nontrivial qualities of service.

From the previous definition, we note the main differences with cluster computing: decentralisation and the concept of *quality of service*.

In the literature, Grids are most of the time subdivided by their objective:

- Enterprise Grids (Figure 3.1(a)) have the objective of transparently provide a *business* ser-

[3]http://www.globus.org

vices as a *supercomputer* connected to Internet (e.g: Google), processing distribution is used to increase business quality of service.

- Internet Grids (Figure 3.1(b)) have the objective of exploiting the potential processing capacity of all computers connected to Internet to solve a parallel problem using the Master-Worker paradigm (62) (e.g.: BOINC infrastructure (3) to solve Seti@home problem (95)).

- Scientific Grids (Figure 3.1(c)), also known as Institutional Grids, have the objective of joining clusters, multiprocessors, large equipment (telescopes, particle accelerators) and laboratory computers of several institutions to increase the potential parallel computation of all of them, managing the shared time of parallel architectures (e.g. Condor (86) using Globus Toolkit as Condor-g (57)).

- Desktop Grids (Figure 3.1(d)) have the objective to communicate personal desktop computers using Internet in order to share resources as CPU or storage. Decentralised *Peer-to-Peer networks* as Gnutella[4], developed using open-infrastructures and which fulfil with the minimal requirements of *quality of service* are Desktop Grids.

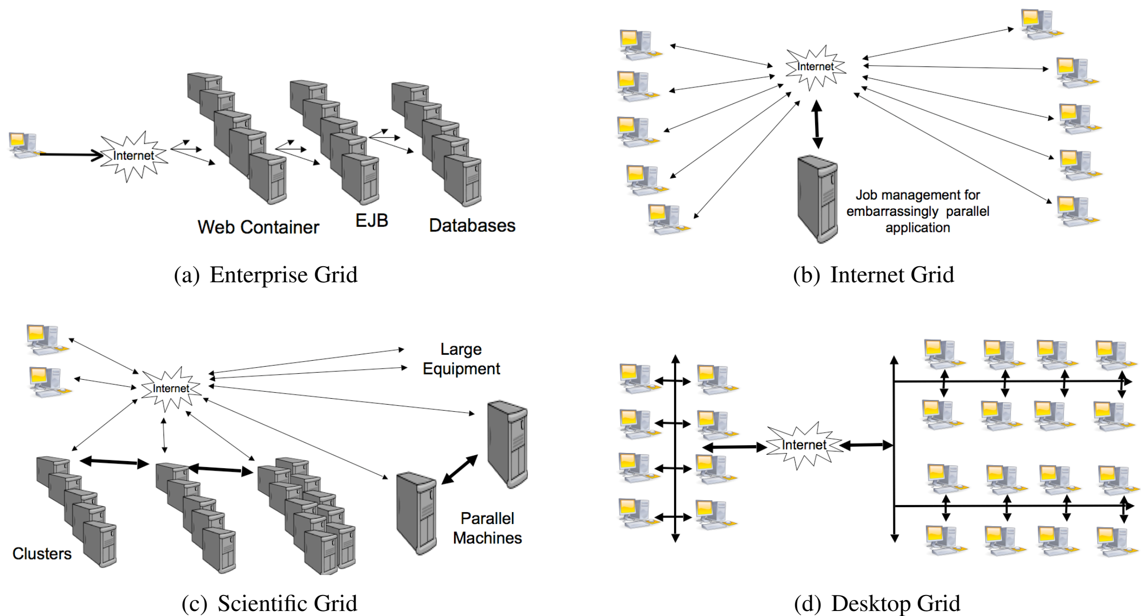
We noted that previous definitions of Grids were too static to fit on studied infrastructures. Therefore, we placed ourselves on the next level of abstraction, *virtual infrastructures*, defining the concept of *Project Grids*.

3.1.3 A model overview for Project Grids

We define as *project grid* a multi-institutional project's virtual environment, created from resources coming from a deployed grid infrastructure. A Project Grid is the infrastructure where a virtual organisation is deployed.

Note that the physical topology of a project grid may be very different from the topology of the physical infrastructure from where its resources originate. First, while the original infrastructure may comprise hundreds of clusters, each with hundreds of resources (possibly the number of

[4]http://www.gnutella.com



(a) Enterprise Grid (b) Internet Grid

(c) Scientific Grid (d) Desktop Grid

Figure 3.1: Grids divided by objective

resources is a power-of-two (69; 81)), the project grid contains only as many resources as were allocated for the project, either from the beginning or dynamically. An institution, assuming the role of *project leader*, provides all of its resources, which will possibly become a large part of the project grid's pool of resources. *Contributing institutions* provide only a part of their available infrastructure. All the applications that run in a project grid are specific to the project, and may come from a very restricted set, with very similar characteristics. This model of operation is being used by more and more projects, including CERN's LCG (112), and ProActive's PlugTests (ETSI and INRIA).

3.2 Peer-to-Peer Infrastructure of ProActive

The *Peer-to-Peer* (P2P) Infrastructure for ProActive middleware began as the Master thesis of Alexandre Di Costanzo (42) and some improvements for its use in load balancing were added by the author in a work called "*Balancing Active Objects on a Peer to Peer Infrastructure*" (24). In

this section we explain the basis of the P2P Infrastructure and the improvements for load balancing.

The goal of the work of Di Costanzo was to use sparse desktop computer processors (called CPU) cycles from institutions; personal desktop computers, grids, and clusters to deploy Java Virtual Machines (JVMs), building an Infrastructure where ProActive active objects might run safety. As he noted, the management of several kinds of resources (grids, clusters, desktop computers) as a single, highly unstable network of resources, needs a fully decentralised and dynamic approach. Therefore, mimicking data Peer-to-Peer networks is a good solution for sharing a dynamic JVM network, where JVMs are the shared resources.

The work of Di Costanzo aimed to comply with the definition of *Pure P2P* given by Rudiger Schollmeier: (104)

> "A distributed network architecture may be called a Peer-to-Peer (P-to-P, P2P) network, if the participants share a part of their own hardware resources (processing power, storage capacity, network link capacity, printers). These shared resources are necessary to provide the Service and content offered by the network (e.g. file sharing or shared workspaces for collaboration). They are accessible by other peers directly, without passing intermediary entities. The participants of such a network are thus resource (Service and content) providers as well as resource (Service and content) requesters (Servent-concept).
>
> A distributed network architecture has to be classified as a *Pure* Peer-to-Peer network, if it is firstly a Peer-to-Peer network according to previous definition and secondly if any single, arbitrary chosen Terminal Entity can be removed from the network without having the network suffering any loss of network service."

The previous definition gives the notion of a P2P network with *quality of service*, and given that ProActive is an open source middleware, the ProActive P2P Infrastructure can be catalogued as a *Desktop Grid*. Therefore, this thesis will use ProActive P2P Infrastructure algorithm to model Desktop Grids.

3.2.1 Bootstrapping: First Contact

A fresh (or new) peer which would like join the P2P network, will encounter a serious bootstrapping problem or first contact problem: *How can it connect to the P2P network?*.

There are different solutions to join a P2P network, such as using specific discovering protocols like JINI (121). The ProActive P2P Infrastructure solution is inspired from Data P2P Networks. The ProActive P2P bootstrapping protocol works as follows:

- A fresh peer has a list of "*server*" addresses. These are peers, which have a high potential to be available and to be in the P2P network, they are in a certain way the P2P network core.
- Using this list, the fresh peer tries to contact each server. When a server is reachable, the fresh peer adds it in its list of known peers (acquaintances).

Using the previous algorithm a fresh peer may be connected to a very distant network, and we will see in Section 5.4.2 that we would like to have peers nearly interconnected. Therefore, we added the requirement that *a fresh peer has to connect only to its nearest server, and servers will maintain a list of other servers.*

3.2.2 Discovering and Self-Organising

ProActive P2P infrastructure aims to maintain a created P2P network alive while there are available peers in the network, this is called *self-organising* of the P2P network. Under condition that P2P does not have exterior entities, such as centralised servers, to maintain peer databases, the P2P network has to be self-organised. That means all peers should be enabled to stay in the P2P network by their own means. There is solution which is widely used in data P2P networks: for each peer to maintain a list of their neighbours, a peer's neighbours is typically a peer close to it (IP address or geographically).

This same solution was selected to keep the ProActive P2P infrastructure up. All peers have to maintain a list of *acquaintances* (most of them geographically close). At the beginning, when a fresh peer has just joined the P2P infrastructure, it knows only peers from its bootstrapping step. Knowing a very small number of acquaintances is a real problem in a dynamic P2P network, as all servers will be unavailable the fresh peer will be disconnected from the P2P infrastructure. Therefore, ProActive P2P infrastructure uses a specific parameter called: *Number Of Acquaintances* (NOA). This is a minimum size of the *list of acquaintances* of all peers. Thereby, a peer must to discover new acquaintances through the P2P infrastructure by sending exploration messages to its acquaintances which forward messages to their own acquaintance until a time-to-live in the messages expire. A fresh peer will not be part of the P2P Infrastructure until the size of its acquaintance list be equal or greater than *NOA*. In order to do not have isolated peers in the infrastructure, we defined that all peer registrations are symmetric.

We discovered that previous solution generated larger and hard to manage networks, therefore we gave to each peer the capacity of decide when to stop message forwarding: when a peer receive the discovering message, it has to decide to respond to be an acquaintance with a given probability (experimentally defined between 0.66 and 0.75).

As the P2P infrastructure is a dynamic environment, the list of acquaintances must be also dynamic. Therefore, all peers keep frequently their lists up-to-date, introducing a new parameter: *Time To Update* (TTU). This is the frequency, which the peer must check its own acquaintances' lists to removed unavailable peers and in case of need discovering new peers. To verify the acquaintances availability, the peer sends a *Heart Beat* to all of its acquaintances. The heartbeat is sent every TTU.

The previous resource query mechanism is similar to the communication system of Gnutella, called *Breadth-First Search algorithm* (BFS). The Gnutella BFS algorithm got a lot of justified critics (100) for scaling, bandwidth using, etc. However, in our experiments, the network size of 250 desktop computers with 100 Mb/s Ethernet connections the message traffic has not posed a significant problem. We made a permanent infrastructure with INRIA laboratory desktop computers and we have been experiment about massive parallel applications for 2 years.

3.3 Theory of Networks

Networks is a well studied field in mathematics, by the name of *Graph Theory* and originated from the works of Leonard Euler in the 18th century. A good introduction in this field is a textbook by Reinhard Diestel (43). In the field of *distributed systems*, the study of *random graphs* became a powerful tool for understanding the main behaviour of distributed algorithms and processes.

A network is represented by a graph, and its nodes are called *vertices*. A set of nodes is denoted by V and the symbols u,v,w are commonly used to refer to specific nodes. The number of nodes $n = |V|$ is known as the *order* of a given graph.

A link between two given nodes u, v is represented by an *edge*. An edge representing an undirected link is denoted by the set $\{u, v\}$. The number of links of a given node is known as its *degree* (Deg). An edge representing a directed link is denoted by $\langle u, v\rangle$, which means that the link goes from u to v. In *weighted graphs*, a weight function is defined which assigns a weight to each node. In this work, the most used weight function is the *latency* $l(u, v)$ which is the time it takes from a message sent from a node u to be received by node v.

A set of edges in a graph is denoted by E, having an edge count of $|E|$. A graph is characterised by the two sets E and V and in the literature it is denoted by $G = (V, E)$.

In graph theory, it is said that two vertices u and v are neighbours if they are connected by an edge, that is, $\{u, v\} \in E$. The set of neighbours for a given node is called its *neighbourhood*. We shall see in Section 3.2 that in practice we prefer to use the words acquaintance and acquaintances respectively, using the term neighbour to refer only nodes that are physically located near each other.

A path from v to w is defined by a sequence of edges in E starting at vertex v and ending at vertex w, i.e.:

$$\{v, v_1\}, \{v_1, v_2\}, \{v_2, v_3\}, ..., \{v_{k-1}, v_k\}, \{v_k, w\}$$

If that given path exists we will say that u and w are *connected*, the length of that path will be the number of hops (edges) between them ($k+1$ in this case) and we define the theoretical distance between two nodes as the length of the shortest path connecting them in G. In practical experiences, the shortest path will be defined using the weights of the given edges. The distance for all nodes to themselves is zero.

A cyclic path is where $u = w$, i.e.:

$$\{u, u_1\}, , ..., \{u_{k-1}, u_k\}, \{u_k, v\}, \{v, v_1\}, , ..., \{v_{j-1}, v_j\}, \{v_j, w\}$$

A *simple path* is one path without cycles. The shortest path between two vertices is always simple. A *connected graph* is a graph with paths between all pairs of nodes, otherwise the graph is *disconnected*. In directed graphs, a sub-graph having paths between all its pairs of nodes in both directions is called *strongly connected* (SCC). A connected graph with no cycles is called *acyclic*. When the degrees of the nodes are known, the expected average distance between a pair of nodes can be obtained theoretically (38).

3.3.1 Generating random graphs

A *random graph* is a graph generated by some random process. The study of random graphs has been a relevant tool in the study of the theoretical properties and behaviour of large-scale networks. Nowadays it is applied to the study of Grids and Peer-to-Peer networks.

In 1959, the following model was proposed by Gilbert (60):

1. Fix the graph order n and choose a probability p_e
2. Include each one of the $\binom{n}{2}$ unordered node pairs as an edge in the graph G uniformly at random with probability p_e.

Another approach was presented by Pál Erdös and Alfréd Rényi in 1959 (47):

1. Fix the graph order n and the number of edges m
2. Select m of the possible $\binom{n}{2}$ unordered pairs of nodes uniformly at random.

Even though these uniform random graph models were not intended to capture properties of real-world network problems, as Pál Erdős and Alfréd Rényi reported in 1960 (48), they are useful to capture the existence of certain properties and behaviours of graph algorithms, such as finding the shortest path between nodes. However, the need of real network generation models resulted in wide adoption of uniform random graphs as models of real-world networks, commonly with modifications such as placing the nodes on a plane and using connection probabilities proportional to Euclidean distance (125).

3.3.2 Natural Networks

In 1998, Watts and Strogatz reported observations on natural network data which were in high disagreement with the uniform models (124), and the study of models for real networks was re-opened. The work of Watts and Strogatz studied two properties: the cluster coefficient (average connectivity of nodes), which was reported higher for real data than for random graphs models; and the average length of the shortest path between two nodes, which was reported for real data almost as small than for random graph models. Based on their observations, Watts and Strogatz (WS) suggested the following model (see Figure 3.2):

1. Fix the graph order n and place the nodes in a circle
2. An initial lattice graph is formed by connecting each node to the k nearest nodes along the circle in both sides. We call these edges *short-distance edges*.
3. For each node v choose a small probability p_e, and trace with probability p_e an edge between v and the others $n-2k-1$ nodes in V. We call these edges *long-distance edges*.

The second step of the WS model produces high clustering coefficient and the third step produces small average path length. Even though this model could be considered naive, it serves to show that for small values of p_e the introduction of *long-distance edges* reduces the average path length almost to the expected level of an uniform random graph of the same order and size, but having greater clustering coefficient (therefore, closer to real data). Graphs with both properties (low average path length and high clustering coefficient) are known nowadays as **small-world networks** (124).

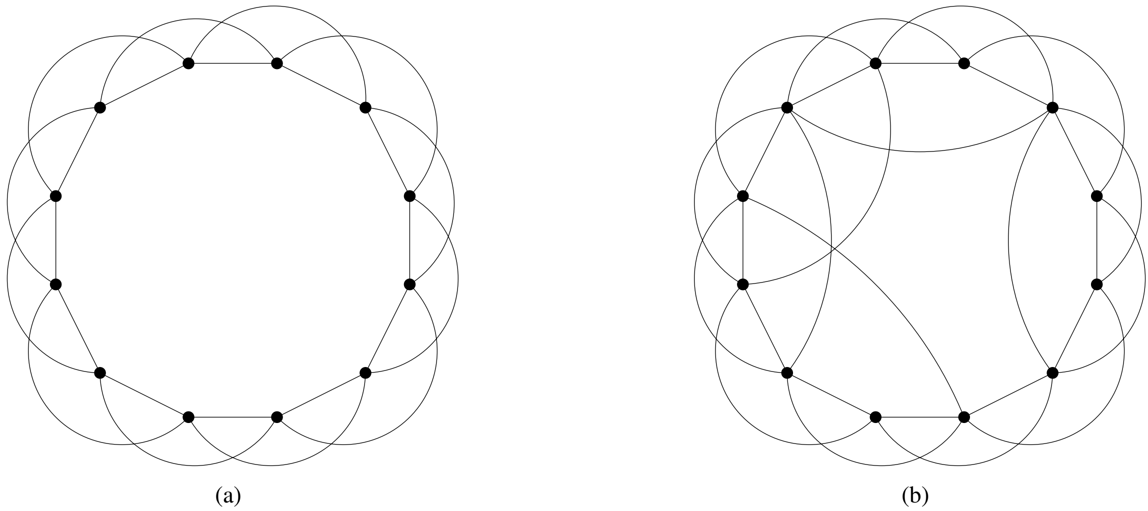


Figure 3.2: (a) step two of Watts and Strogatz model with $n = 12$ and $k = 2$; (b) step three with small p_e

In 1999, Albert-László Barabási and Réka Albert reported an observation which disagrees again with the models of networks, even with the WS model and its variations (9). The observation they made deals with the degree of nodes in natural networks. For uniform random graphs, the degree of a node follows the binomial distribution:

$$\mathrm{Deg}(v) \sim \mathrm{Binom}(n-1, p_e) \tag{3.1}$$

Yielding for the number of vertices with given degree k a Poisson's distribution:

$$\text{Poisson}\left(\binom{n}{k} p_e^k (1-p_e)^{(n-1)-k}\right) \tag{3.2}$$

Nevertheless, Barabási and Albert discovered that all the observed distributions had a persistent right tail with a fast decreasing but without vanishing, and when they plotted the real data on a $\log\log$ scale, practically all of them had could be approximated with straight lines with almost the same slope. Therefore, data show that in all natural networks there are few special nodes with high-degree, which are called *hubs*. The straight line distribution in $\log\log$ scale is called *scale-free* and it could be approximated by a *power law* of the form

$$P(\text{Deg}(v) = k) \sim k^{-\gamma} \tag{3.3}$$

That is, the probability that a randomly chosen node has degree k is proportional to $k^{-\gamma}$. For that reason, *scale-free networks* are also known as *power-law graphs*. Examples of *scale-free networks* are Peer-to-Peer networks as Gnutella (with a reported $\gamma = 2.3$ (Gnutella measurement project)) and the router topology of internet in 1995 ($\gamma = 2.48$ (49)).

Although the concept of the small-world phenomenon was introduced already in 1960's by Stanley Milgram (87), the theory of small-world networks was initiated by the seminal paper of Watts and Strogatz (124) and quickly followed by the work of Jon Kleinberg (71). Kleinberg presents another model for small-world networks, where the network is constructed in the following manner:

1. using a $n \times n$ matrix to represent the nodes
2. defining the *lattice distance* between a pair of nodes (i, j) and (k,l) as $d[(i, j),(k,l)] = (|i-k| + |j-l|)$.
3. giving a constant $p \geq 1$, a node u has a directed edge to all other nodes at distance lower than p. Connected nodes are known as *local contacts*.

4. giving two constants $q \geq 0$ and $r \geq 0$, a node u has a directed edge to q other nodes (*long-distance contacts*) using independent random trials, where the ith directed edge from u has endpoint v with probability proportional to $d[(i,j),(k,l)]^{-r}$

In the same work (71), Kleinberg shows that the optimal exponent for this implementation is $r = 2$. We are very interested in the model of Kleinberg because it is easy of implement in C language and as we will see in following sections, we will exploit this model to develop fast simulations of large-scale networks.

A good introduction and explanation of natural networks is the work of Elisa Schaeffer (103; 120).

Chapter 4

State of the Art on Load-Balancing

"Idleness is not doing nothing. Idleness is being free to do anything". (Floyd Dell)

Imagine that you are in a supermarket, pushing your shopping cart full of groceries to the register. When you look in front of you there are more people pushing their shopping carts and only one who is served by the cashier ((1) in the Figure 4.1) . You look back and you see more people coming with their carts and following you. Together with the other people with shopping carts, you form a *queue* ((2) in the Figure 4.1). In a queue, those who arrived first are served before those who arrived later. Also, every now and then, new clients join the queue. The number of carts arriving in a given time unit is called the *incoming rate* (Figure 4.1 (3)). You look to the register and note that the time it takes the cashier to attend a customer depends on how many items are in the shopping cart. The number of carts attended on a given unit of time is known as *service rate* (Figure 4.1 (4)).

Suddenly, you look around you and see more queues. One of the cashiers seems to work faster than the others: the service rate of that queue is greater than that of the other queues. Therefore, you think "*if I change to that queue, would I be served before than if I keep my place here?*". That

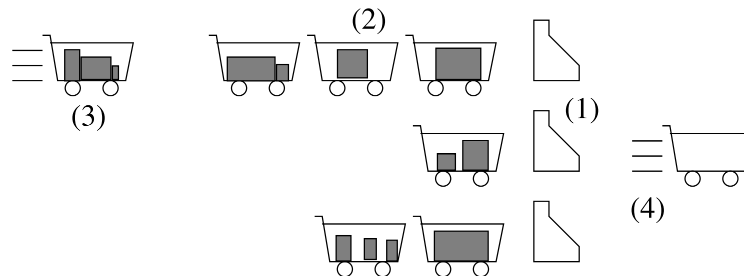


Figure 4.1: A supermarket

question represents the main principle of *load-balancing*: to move tasks (carts) among processors (registers) to reach a given objective (in this case, to minimise the time spent in queue).

Even more, you would think "but, what if everybody thinks the same than me?", "how many people would be seen that queue is faster than others?", "what if when I arrive to the another queue it becomes slower?". Those questions are related to the model and implementation of the load-balancing algorithms themselves. In this chapter, we study the possible response to such questions.

4.1 Static Load-Balancing

Suppose that in the previous example we know exactly the number of registers that the supermarket has, the service ratio of all cashiers, the incoming ratio of all queues, and the number of groceries that each shopping cart will has when it enters the queue. Having all that information and a given objective we can pre-compute how to optimally distribute the shopping carts among the queues even before the arrival of the first cart. The computation of such distributions is known as *static load-balancing*.

Static load-balancing is a well-studied issue in literature. Casavant and Kuhl propose on their *Taxonomy of Scheduling* (36) four categories for static task-distribution algorithms:

1. **Solution-space enumeration and search**: Defining a *cost function* which represents the maximum time for a task to complete its execution and communication in all the processors and a *minimax criterion* based on which both minimisation of inter-processor communication and balance of processor loading can be achieved (105).

2. **Graph theoretic**: Using graph partitioning for minimising execution, communication and reassignment costs (127). Or, having the interconnection pattern of the tasks in a tree form, an algorithm minimises the sum of execution and communication costs for arbitrarily connected distributed systems with arbitrary numbers of processors finding the minimum spanning tree (20).

3. **Mathematical Programming**: Modelling the environment as a system of equations and transforming the scheduling problem in an optimisation problem (66).

4. **Queuing theoretic**: Using Markov chains to model the system, as was done by Mitzenmacher in his PhD Thesis (89). He modelled the system using a supermarket abstraction where arriving customers has to choice their queue and they can not change their decision once enqueued.

Casavant and Kuhl (36) also consider the *heuristics* approach to solve this kind of problems. That is, to make use of some special parameters which could have indirect influence over system performance. For instance, clustering communication-intensive parallel tasks.

4.2 Dynamic Load-Balancing

Suppose that for the problem presented in Figure 4.1, every customer has to decide upon arrival in which queue he wants to join (as in the model of Mitzenmacher (89)), but now registers could open or close every time. If we know the schedule of registers before starting the customers distribution, this problem still can be solved in a static way. But, if we do not know in advance at least one of the parameters such as the incoming ratio, the service ratio or the number of registers; the problem of finding an optimal distribution becomes intractable. However, good approximations

for optimal distributions can be done having only partial information. The process of determining the distribution of the clients in the queues based on information obtained at runtime is known as *dynamic load balancing*.

The design of a balancing strategy is directly related to the objective of the distribution. Riedl and Richter presents in their work a list of primary objectives (99), concerning:

- Service performance for tasks: waiting time, service time, response time, availability of services.
- Physical distance between tasks and data
- Service performance of resources: throughput, cache or communication times.
- Equalisation of load among processors.
- Minimisation of processor idle time.

Starting with these objectives, a *metric* (measuring unit for determine load imbalances) can be chosen to determine the *system performance*.

Considering the objectives, we can study the performance from two perspectives: that of the system and that of the parallel application. For the parallel application point of view, commonly the metric used is the individual process completion time (also known as *makespan*). And, from the system point of view, commonly the metric used aims to a maximisation (or fairly distribution) of resource usage. A *trade-off* can be achieved by choosing the metric that incorporates both view-points, because applications will try to use the available resources to improve their performance and systems will aim at fair distribution of the resources.

Casavant and Kuhl work (36) propose in addition two properties for consideration in evaluating a load distribution mechanism:

1. **Performance**: the quantitative measure of the improvement of the parallel application when the mechanism manage the resources.

2. **Efficiency**: the costs produced by the resource manager.

A *perfect* load-balancing algorithm is that which performs the best performance possible with minimal cost.

Thomas Kunz described in (76) some requirements which proved to be important for a general purpose load-balancing strategy:

1. no a priori knowledge about incoming task requirement

2. no assumptions about the underlying network (topology, homogeneity, size, etc.)

3. dynamic, physically distributed and cooperative decision making (we draw the same conclusion in Section 5.3).

4. Minimisation of average/worst response time of tasks as performance criteria. Defining response time as the time between a task is received by a parallel application and it is finished by the processor.

Nevertheless, Kunz (76) conducted his study in 1991 with heterogeneous networks without varying geographical distance. Nowadays, with the utilisation of *Internet* and large-scale networks for parallel computing, the second requirement proposed by Kunz has becomes inapplicable. As we will see in Section 5.4.2, a knowledge in the underlying network has high importance in modern load-balancing algorithms.

Another important requirement in load-balancing algorithms is their level of complexity. Mirchandaney, Towsley and Stankovic reported in their work (88) that:

- simple load distribution yields dramatic performance improvement compared to a setup without load-balancing
- complex policies, which try to make the best selection, do not offer further improvements

A load-balancing algorithm should aim to minimise work transfer among processors. When the system is under heavy load, above-average transfer delays may be expected, reducing in the performance of the algorithms. Only a small amount of work has to be transferred in order to achieve effective load-balancing (13; 74).

4.3 Components of a Load-Balancing Algorithm

Typically, a load-balancing scheme consists of a load index and a set of policies based on the index. Commonly, the policies can be classified into one of the following categories (59). An **information-sharing policy**, defines *what information* has to be used and *how* it has to be collected and shared. A **transfer policy**, determines *which* work has to be balanced and *when* to do it. And, a **localisation policy**, determines *where* the shared work as to be balanced. There exist two kinds of localisation policies: migration and placement. The former directs the migration of work in execution time and the latter directs the first placement of a parallel application. While in this thesis we focus in the migration policy, we will see all along this work that the first placement is a key issue in load-balancing of active-objects.

The decisions of *when*, *where* and *which* tasks have to be transferred are critical, and therefore the load information has to be accurate and up to date (91). In *dynamic load-balancing*, the balance decisions will strictly depend on the information collected from the system.

4.3.1 Load Index

A key issue for all load monitors is the definition of a good load index. Ferrari and Zhou proposed in (52) that a good load index should:

- correlate well with task response-time, because it is used to predict the performance of a task if it is executed at some particular node;
- aid in predicting the load in the near future, since the response time of a task will be more influenced by future load than by present load;
- be relatively stable. Note that this point is influenced by the load index and the periodicity of the load measurement.
- relatively cheap to compute.

Several load indices have been proposed in the literature (52; 76): CPU queue length, I/O queue length, used memory, CPU utilisation, etc. Ferrari and Zhou (52) proposed a linear combination of resources queue lengths as a load index, using the time t_j that a task requires from a resource r_j which has a queue length q_j and including N different resources:

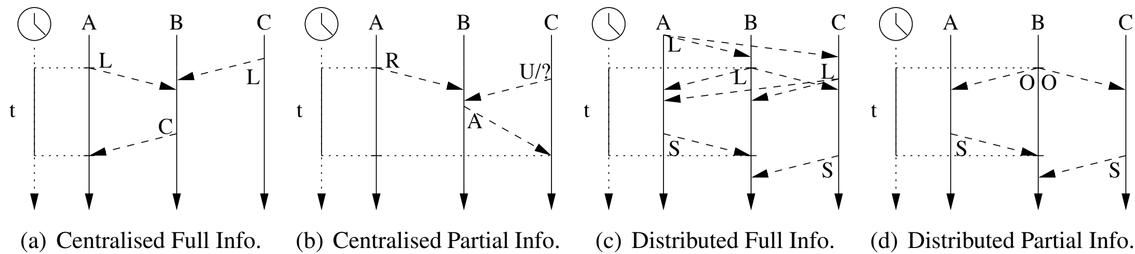
$$\text{load} = \sum_{j=1}^{N} (q_j \times t_j) \tag{4.1}$$

Nevertheless, to know all the tasks requirements is hard in real environments, and Kunz (76) proposed to avoid that requirement. Ferrari, Zhou and Kunz conclude that the CPU queue length is the predominant resource in the studied hosts. That suggests that CPU queue length to be one of the most adequate as load index, because it determines the behaviour of the machine, is relatively stable and cheap to compute. A similar conclusion was reached by Olivier Dalle in its PhD thesis (40).

4.3.2 Information-Sharing Policy

An *information-sharing policy* is responsible of which information will be used in the load-balancing process and how it will be shared. Load information can be shared among processors periodically or "on demand", using centralised or distributed information collectors (114). Also, information-sharing policies can be *full* or *partial*, the former policies share all information and the latter policies share their information only for certain states (values) of the load metric. These policies are defined as follows:

- **Centralised Full Information**: Nodes share all their load information with a central server. Figure 4.2 (a) presents an example with three nodes: nodes A and C send their load information L to the server B periodically. The server collects that information and keeps the system balanced (in the figure, ordering A to balance with C). This policy is widely used on systems such as Condor (62; 86) and middlewares such as Legion (37). Theoretical and practical studies report this policy as non-scalable (2; 35; 80; 114).
- **Centralised Partial Information** There is partial information (such as a *state change*) sharing among the nodes through central server. Figure 4.2 (b) presents an example using three nodes which share information only when they are overloaded. A node A registers on the server B when it enters an "overloaded state" (that is, the "load metric" is above a given threshold), and node C unregisters from the server because it exits the "overloaded state". At the same time C asks the server for overloaded nodes, the server chooses one node from its registers and starts the load-balancing between them.
- **Distributed Full Information** Nodes share all their information using broadcast. Figure 4.2 (c) shows an example using three nodes: Each node broadcasts its load to the others periodically. The nodes use the information for load-balancing (22). Then, A and C realise they can share B's load and send the balance message S. The figure also shows the main problem of this policy: there is no control on the number of balance messages an overloaded node might receive.
- **Distributed Partial Information** There is partial information-sharing among the nodes using broadcast. Figure 4.2 (d) presents an example for the *overloaded* case: a node B broad-



(a) Centralised Full Info. (b) Centralised Partial Info. (c) Distributed Full Info. (d) Distributed Partial Info.

Figure 4.2: Examples of information-sharing policies

casts its load only when changing to the overloaded state, requesting a load balance. Using this information, A and C reply to the request S, but unlike in the previous policy, only the reply from A is considered. In practise, this policy was used in the first load-balancing algorithm developed for ProActive (23).

Also, demand-driven policies can be used, where a node collects information about other nodes only when it wants to make a work transfer; therefore, the information-sharing policy is triggered by the decision policy. We will see in Section 5.4.2 that a demand-driven performs the best performance in the context of load-balancing of active-objects in Peer-to-Peer networks.

4.3.3 Transfer Policy

A *transfer policy* is responsible to determine if a given node *have to* participate in a load-balancing, either as a sender or a receiver. Common policies are based on *thresholds* or based on environment load.

Threshold based policies determines that a given node is a work-sender if its *load index* is greater than a given parameter (threshold) OT or a work-receiver if its *load index* is lower than a given parameter (threshold) UT. A key issue for all transfer policies is the smart selection of both thresholds. Even though some techniques are presented to adapt thresholds to the system load in runtime (96), we will see that fixed parameter behaves very well for load-balancing of active

objects in Section 5.6.

Environment based policies determines if a node has to transfer some of all of its work considering its load and the load of the other nodes in its environment. Nodes will share their work if their load index differ by more than a given threshold (106; 35; 98). Note that a threshold based policy most of the time aims to exploits resource usage, and an environment based policy aims to equalise the workload among nodes.

Transfer policies may be *sender-initiated* (also known as *eager* policies) , *receiver-initiated* (also known as *lazy* policies or *work-stealing*) or *symmetrically-initiated*. In the first case, overloaded nodes initiate the load-balancing process looking for a (set of) candidate(s) to receive it work. In the second case, *underloaded* nodes has to look for an overload node to steal some of its work. Note that, as we will see in Section 5.3, sender-initiated policies have better response time against overloading than receiver-initiated ones, because in real environment the number of underloaded nodes are greater than overloaded nodes, therefore it is more probable that an overloaded node randomly find an underloaded one is greater than the an underloaded node randomly find an overloaded one.

Another issue of a transfer policy is to determine *which* work and how much work to transfer to a new location. We will see in Section 4.4 that if the policy has not access to computer's resources, it is better to send low transfer-cost works which are in a not *on-running* state (12; 18; 37; 117). How much work to send depends on the objectives of load-balancing; for instance, in a sender-initiated scheme the objective could be equalise the work (sending low slices of work) or only avoid overloading (sending the amount of work which produces overloading), and in receiver-initiated schemes, theoretical studies determines that a node has to send at most half of its work (13), stealing only the amount of work which guarantees a long period of working time (18; 117).

4.3.4 Location Policy

Location policy is the responsible of use all the information collected by the information-sharing policy to determine *where* is located the *best* partner to perform a load transfer. A location policy can be *deterministic* (98) (e.g.: "Enumerate n nodes and always send work to node $i+1 mod\ n$"), stochastic (e.g.: random load-balancing schemes (13; 18; 90; 117) or probabilistic (decisions are taken according a set of predefined rules and their probabilities (1; 102)).

4.4 Related Work

The study of load-balancing is always related to what we need to balance and at which level of complexity. For instance, there are some infrastructures which have access to most of the hardware resources and processor schedulers, such as *Condor* (86); thus, it can stop a process in runtime and migrate it completely to another new location. Other infrastructures such as *Legion* (61) and *Cilk* (17) have limited access to hardware resources, so they migrate only inactive entities. In the other side, there are infrastructures built in Java (e.g: *Satin* (118) and *ProActive* (94)), taking advantage of Java portability but having very limited access to hardware resources as the schedulers; therefore, they have to migrate only inactive entities and also handle the lost references.

In this section we will describe the first four architectures (ProActive was described in depth in Section 2.3) and their load-balancing mechanism.

4.4.1 Condor

Condor was first introduced as "*A Hunter of Idle Workstations*" in a work of Michael Litzkow, Miron Livny and Matt Mutka (86). They presented a system able to manage processes in a cluster of workstations using *batch processing*, the main idea was to detect idle resources (CPU, Memory)

and distribute a parallel application among them.

Condor was designed following these principles:

- Batch processing should have no impact on quality and availability of services provided by workstations to their owners.
- *Condor* should have complete control of the resources: locating them for application's jobs, monitoring and informing to user the resource use and job progressing.
- *Condor* should preserve the operating environment of workstations and it should not require special programming to submit parallel applications.

The key point in the infrastructure of Condor is the *Matchmaking* process (97): resources and job requirements are published as a kind of "*classified advertisements (ClassAds)*" (Figure 4.3 (1)) and a central entity makes the matchmaking among *ClassAds* to determine the best pair job-resource (Figure 4.3 (2)). Both (job and resource) are notified of the match (Figure 4.3 (3)) and a claim process (negotiation of undefined variables) begins between them (Figure 4.3 (4)).

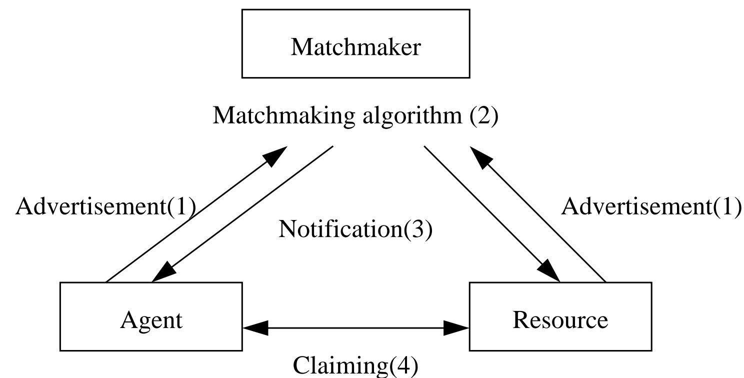


Figure 4.3: Matchmaking process of Condor

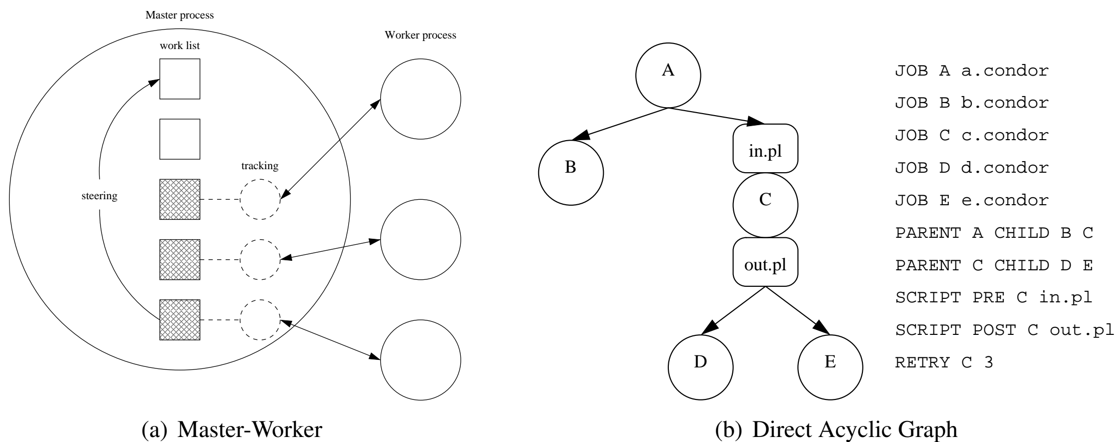
Load-balancing in *Condor* is performed by resource allocation. *Condor* has full access to workstation resources at processor's level; therefore, it may preempt a process if a workstation is

overloaded, find a new location for it and restart the process in a new place. Of course, to perform that kind of migration is very costly in terms of resources; therefore, what Condor really does is to use a checkpointing (85): in case of necessity, it stops a process in a workstation and starts the same process in a new location from the last checkpoint.

Condor is designed to solve two kinds of parallel paradigms: Master-Worker (Figure 4.4(a)) and Direct Acyclic Graphs (Figure 4.4(b)).

In Master-Worker paradigm (62), a central entity (the *Master*) performs the (optimal) division of a big task in several treatable sub-tasks. Those sub-tasks are solved by a set of *independent Workers* and results are returned to the *Master* which use them to build the problem's solution or to produce more sub-tasks. Idle workers are in charge to ask the Master for new sub-tasks.

In Direct Acyclic Graph (DAG) paradigm , tasks are ordered using a Direct Acyclic Graph before execution, providing a structure which allows to know in advance which tasks can be executed in parallel and which ones **must be** sequential. *Condor* provides a semantic to structure this graph using a minimal set of primitives: `JOB`, `PARENT` and `CHILD`. Also, *Condor* allows to declare scripts to pre-process data (`SCRIPT PRE`) and post-process data (`SCRIPT POST`). Finally, a specific primitive to retry in case of node failures is given (`RETRY`).



(a) Master-Worker (b) Direct Acyclic Graph

Figure 4.4: Parallel problems solved by *Condor*

Condor is a powerful tool for distributed computing, but has two disadvantages: its low level management (at Operative System level) reduces its portability (system architecture is a key issue for matchmaking), even that some error handling for Java programs has been published (113); and, even that Master-Worker and DAG paradigms are enough to solve most of parallel programming problems, some new generation parallel applications (e.g: Jem3D (64)) exploit high-speed networks to distribute a task among *dependent workers*, having intensive communication among them and needing mobility to quick react against overloading (24; 27). *Condor* mechanism of checkpointing/restart does not provide mobility for dependent workers.

4.4.2 Legion

Legion is an object-based, meta-systems software project at the University of Virginia. The project began in late 1993 (61), focusing in object-oriented parallel processing, distributed computing, scalability, programming ease, fault tolerance and security. Legion is designed to support large degrees of parallelism in application code and to manage the complexities of the physical system for the user. The first public release was made at Supercomputing '97, San Jose, California, on November 17, 1997.

Legion comprises of independent, address-space disjoint C++ objects that communicate with one another via method invocation. Method calls are non-blocking and may be accepted in any order by the called object. Each method has a signature that describes the parameters and its return value (if any). In the Legion object model, each Legion object belongs to a class, and each class is itself a Legion object. A class object is responsible for creating and locating its instances (non-class objects) and subclasses (other class objects). Further details of Legion's implementation can be found in the work of Mike Lewis and Andrew Grimshaw (78).

We are particularly interested in two *Legion* objects: *Hosts* and *Vaults* (See Figure 4.5). A *Host object* runs on each host that is included in the Legion system. A host handles tasks as instantiating and executing objects on the host, report object exceptions and they encapsulate machine capabilities. Vaults are the generic storage abstraction in *Legion*. All object must have a *Vault* in order to

be executed, and this *Vault* will store the persistent state of the Object (after a set of method calls, the object' state is stored inside the *Vault*), which is used for migration purposes.

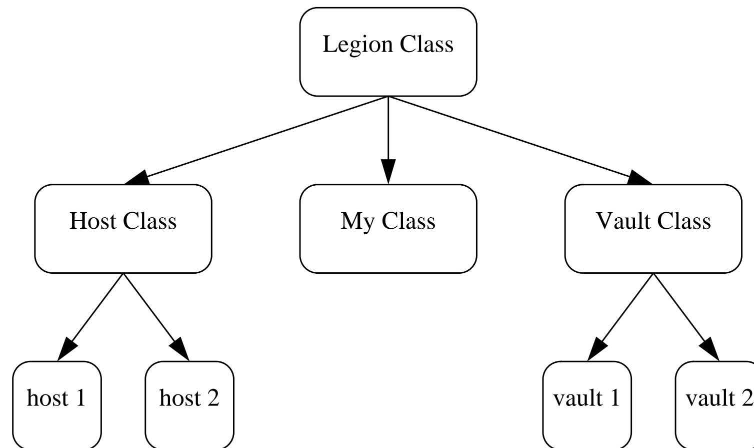


Figure 4.5: Main classes of Legion infrastructure

In *Legion*'s Resource Management Infrastructure (37), three new Legion's object came to light: a *Collection*, which store information of a *set of hosts*; an *Enactor*, which is responsible of the *scheduling* for a given *Collection*, and an execution *Monitor*. Also, a user-defined *Scheduling* to interact with the infrastructure is allowed.

The object placement (and replacement) works as follows (See Figure 4.6):

1. The *Collection* is populated with the information of *Hosts*.
2. The *Scheduler* queries about resources information to the *Collection*
3. Based on the result and knowledge of the application and the answer of the *Collection*, the *Scheduler* performs a mapping of objects to resources.
4. The previous mapping is passed to the *Enactor*.
5. The *Enactor* invokes methods in *Hosts* and *Vaults*.
6. The method call performs reservation in those resources named at the mapping.

7. After the reservation, the *Enactor* confirms the schedule with the *Scheduler*.

8. The approval or rejection is sent to the *Enactor*.

9. Enactor attempts to instantiate the objects through the appropriate class objects.

10. The class objects report success/failure codes.

11. The *Enactor* returns the result to the *Scheduler*.

12. If, during execution, **a resource decides that an object has to be migrated**, it performs an outcall to the Monitor.

13. the Monitor notifies the Scheduler and Enactor that a **rescheduling of that object** has to be performed.

A migration in *Legion* is performed taking out an object from the processing queue, transferring its persistent state to a new location, and rescheduling it. Scalability is achieved because *Collections* are **non-disjoint** sets of resources.

Legion as itself was not finished, but the project team which developed it continued the idea, reporting at the project web-page (http://legion.virginia.edu) that

> *Legion team will not finish Legion but will create an "open" system that allows and actively encourages third-party development of applications, run-time library implementations, and core system components.*

Legion as a model of objects for parallel computing was a very good idea, and some of its features, as the use of non-disjoint sets of resources to achieve scalability and migration of objects in safe-state were take in account in the development of a load-balancing infrastructure for ProActive's active objects, adding the Java natural portability that C++ Legion's objects did not have.

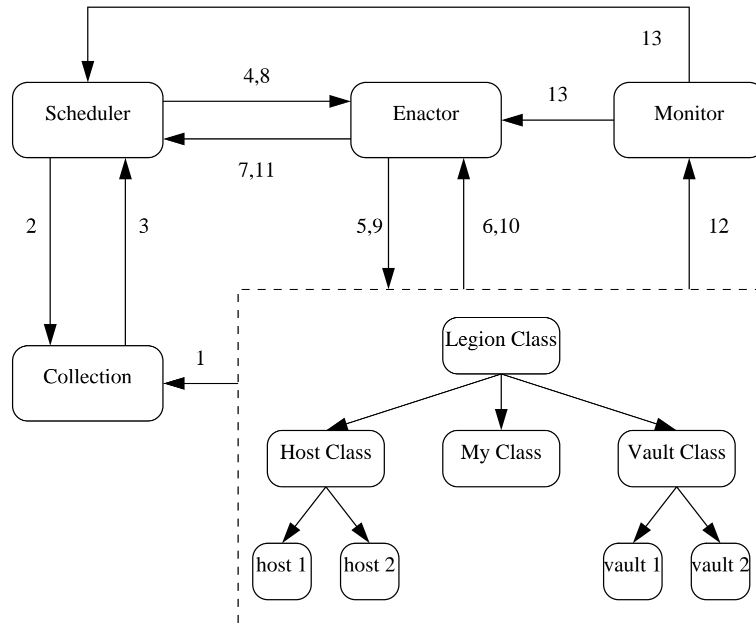


Figure 4.6: Legion Resource Management Infrastructure

4.4.3 Cilk

Cilk is a Middleware for multithreaded parallel programming which is based on ANSI C Language and it was first introduced in 1992 as a model for "Managing Storage for Multithread Computations", the master thesis of Robert Blumofe (16) and as a real implementation in 1995 with the work of Blumofe et al. (17), and winning the *Dutch Open Computer Chess Championship* in November 1996 with CilkChess (77). Finally, in 1998; Frigo, Leiserson and Randall present Cilk5 (58), a new implementation of Blumofe's idea improving its performance reducing overheads of previous versions using distributed shared memory.

The philosophy of *Cilk* is that a programmer should concentrate on structuring the program to expose parallelism and exploiting locality. To achieve that, the programmer has to build an explicit Direct Acyclic Graph as *Condor* (see Figure 4.4(b)) using a primitive called `spawn`. In addition of Condor's DAG, Cilk provides also a primitive to synchronise data dependencies (`sync`). Figure

4.7 presents a DAG built using Cilk's primitives.

Figure 4.7: *Cilk* model: each thread is a circle, grouped in procedures. Each downward arrow is a spawned child, and each horizontal arrow is a spawned successor. Dashed arrows represent data dependency (synchronisations). Also, spawn-levels from the original thread are presented.

Below there is a *Cilk* code example for an (non-optimal) implementation of parallel Fibonacci's function. Note that using a minimal set of primitives (`cilk`, `spawn` and `sync`) a sequential procedure was transformed in a parallel one.

```
cilk int fib (int n)
{
if (n < 2) return n;
else
  {
  int x, y;
  x = spawn fib (n-1);
  y = spawn fib (n-2);
  sync;
  return (x+y);
  }
}
```

As the programmer has the responsibility of make explicit the parallelism in a *Cilk* code, the

Cilk runtime system has the responsibility of scheduling the computation to run efficiently on a given platform. Thus, the *Cilk runtime system* has to take care of details such as load-balancing, paging, and communication protocols. Load-balancing in Cilk is performed by a *Work-Stealing* algorithm (18) which works as follows:

1. Choose a victim to steal
2. If the victim is idle, attempt to steal again
3. Otherwise, steal the first non-executed thread on the lower level and execute it until:
   (a) The thread spawns another thread
   (b) The thread returns/terminates
   (c) The thread reaches a sync point

The previous algorithm was enhanced by Bender and Rabin (12), performing a work stealing and sharing to speed up parallel applications (we arrived to the same conclusion for load-balancing of active objects in Section 5.5). The steps of their modified algorithm are:

1. Choose a victim to steal
2. If the victim has available threads, steal using the previous algorithm
3. If there are not available threads but the victim is working on a thread and it is reported to be β times slower than the *thief* ($\beta > 1$, but β close to 1), then *mug* the thread (*mug* means its thread is migrated to another processor and this processor attempts to work steal).
4. Once a thread is received, work on ituntil:
   (a) The thread spawns another thread
   (b) The thread returns/terminates
   (c) The thread reaches a sync point

(d) The processor is *mugged*

5. Otherwise, there is a failed steal attempt; try to steal again!

Even thought *Cilk* improves the Direct Acyclic Graphs presented in Condor, providing a synchronisation primitive which allows the utilisation of dependent tasks, its performance has been discussed even by its implementers (58). Moreover, its philosophy of give to the programmer the parallelism responsibility (in opposition to Condor and ProActive (94)) may produce poor performance parallel programs with a minimal use of *Cilk*'s primitives (as the Fibonacci code example). Finally, the use of a distributed shared memory makes a hard requirement in the context of large scale networks.

4.4.4 Satin

Satin was first introduced with the work of Robert van Nieuwpoort, Thilo Kielmann, and Henri E. Bal as an extension of Java language with *Cilk*-like primitives for the *Manta* compiler[1], with the goal of efficiently run parallel *divide-and-conquer* applications on *wide-area* hierarchical systems. This work was presented in EuroPar 2000 conference (116).

In 2005, a new implementation of Satin adapted for Grid Computing (118) replaced the *Cilk* `spawn` primitive by an interface which determine that an object *could be* executed in parallel. An example of *Satin* source code is the following (again not-optimal) implementation of Fibonacci's function:

```
interface FiboIter extends satin.Spawnable {
    public long fib (long a);
}

class Fibo extends satin.satinObject
```

[1] *Manta* is a native Java compiler which compiles Java source codes into Intel architecture executables. For details, see the PhD thesis of Robert van Nieuwpoort (115), Chapter 3

```
        implements FiboIter {
    public long fib (long a) {

    if (a < 2) return a;

    long x = fib (a-1); // spawned
    long y = fib (a-2); // spawned
    sync();

    return x+y;
    };

// ...
}
```

The main contribution of *Satin* is its work-stealing algorithm (117). Nieuwpoort et al. (117) presented an experimental study of Cilk-like *Random Work-Stealing* (RS) with existing load-balancing strategies that were believed to be efficient for multi-cluster systems (Random Pushing (106) and two variants of Hierarchical Stealing (6; 8)). They demonstrate that, in practice, these *work-stealing* algorithms perform sub-optimally.

In the same work (117), authors introduce a novel load-balancing algorithm, called *Cluster-Aware Random Stealing* (CRS), which adapts itself to network conditions and job granularities, balancing differently for local nodes (in a cluster or LAN) than for external nodes (accessed through a WAN). *Cluster-Aware Random Stealing* works as follows:

1. Choose a victim to steal
2. If the victim has available threads:
    (a) If the victim is in the same cluster, steal the first non-executed thread on the lower level (this is a synchronous process).

(b) Else, if the *thief* is not performing a *long-distance work-stealing*, it performs an asynchronous *steal requirement*.

A steal requirement may be one of the following:

1. the *thief* sets its *long-distance work-stealing* flag.
2. the *thief* sends a *steal request* to the victim.
3. if the victim has an available thread, the thread is sent to the *thief*; else, a "no available thread" reply is sent.
4. the handler routine for the *long-distance steal* simply resets the flag and, if the request was successful, puts the new thread into the work queue.

Note that while the asynchronous *long-distance work-stealing* is performed, the thief may perform synchronous steal requests to nodes within its own cluster. As long as the flag is set, only local stealing will be performed.

CRS was reported faster than its competitors for 11 out of 12 test applications with various WAN configurations using at most a 4% of overhead in run time compared to normal random stealing on a single, large cluster, even with high wide-area latencies and low wide-area bandwidths. Nevertheless, in Section 6.2 we will show that in large-scale networks, a cluster-awareness has to be complemented with a smart first distribution to improve the performance of a parallel application, else the performance may be worse. Moreover, as we will see in Section 5.3.4, a receiver-initiated load-balancing does not perform quick reaction against overloading; therefore, it has limited applicability in the context of *Desktop Grids*.

Even thought *Satin* behaves better than *Cilk* for *Divide and Conquer* parallel applications in heterogeneous networks, the use of the *Manta* undermines its chances of widespread adoption.

Chapter 5

Setting foundations for Load-Balancing of Active-Objects

“Do you wish to be great? Then begin by being. Do you desire to construct a vast and lofty fabric? Think first about the foundations of humility. The higher your structure is to be, the deeper must be its foundation.” (Saint Augustine)

In this Chapter we present the main contribution of our thesis: foundations for the load-balancing of active objects. The idea is to reduce the overall time of an application developed with active-objects, migrating objects from overloaded to underloaded processors with an increase of application speed regardless of migration time.

5.1 Active-Objects and Processing Idleness

When an active-object is idle (without processing), it can be in one of two states: *wait-for-request* or a *wait-by-necessity* (see Figure 5.1). While the former represents a sub-utilisation of the active-object, the latter means that some of its requests are not served as quickly as they should. The longer waiting time is reflected on a longer application execution time, and thus a lower application performance. Therefore, we focus on a reduction in the *wait-by-necessity* delay.

Even though the balance algorithms will speed up applications such as that on Figure 5.1 (b), they are not the focus of our work, because the time spent in message services is so long that the usage of *futures* would be pointless. In such application designs, asynchronism provided by *futures* will unavoidably become synchronous. Migrating this kind of an active-object to a faster machine will reduce the application's response time but will not correct the application design problem.

Therefore, we focus on the behaviour presented in Figure 5.1 (c), where the active-object on C is delayed because the active-object on B does not have enough free processor time to serve its request. Migrating the active-object from B to a machine with available processor resources speeds up the global parallel application, because the *wait-by-necessity* time of C will become shorter, and B will have fewer active-objects, decreasing its load.

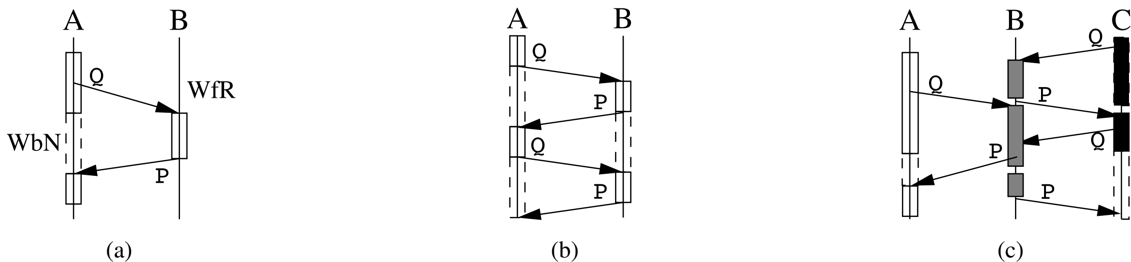


Figure 5.1: Different behaviours for active-objects request (Q) and reply (P): (a) B starts in wait-for-request (WfR) and A made a wait-by-necessity (WfN). (b) Bad utilisation of the active-object pattern: asynchronous calls become almost synchronous. (c) C has a long waiting time because B delayed the answer.

In Section 4.2 we presented several algorithms of load-balancing. Some of them perform batch

processing balance (such as Condor (86)), exploiting their knowledge of the hardware architecture to perform the migration. However, active-objects are also normal objects which are executed on *virtual machines*. That is, virtual environments on real machines where objects can run safely, having no access to kernel calls of hardware resources. Therefore, the study of load-balancing of active objects has to concentrate on those that do not exploit kernel calls for hardware knowledge.

Most of load-balancing schemes perform migration of tasks[1] among queues (Figure 5.2), and those queues are fixed to each processor. Because the stated behaviour of active objects service queues, each service has to be served for the active object on which it was enqueued, unless it does not change the instance variables of the active object. However, to know if the service has the capacity to change the variables while it is enqueued requires similar processing time than serving it, producing that the application runs almost at half the speed than running in a normal behaviour. That is the main reason why a load-balancing algorithm of active-objects has to perform migrations of active-objects instead of tasks (services) (Figure 5.3).

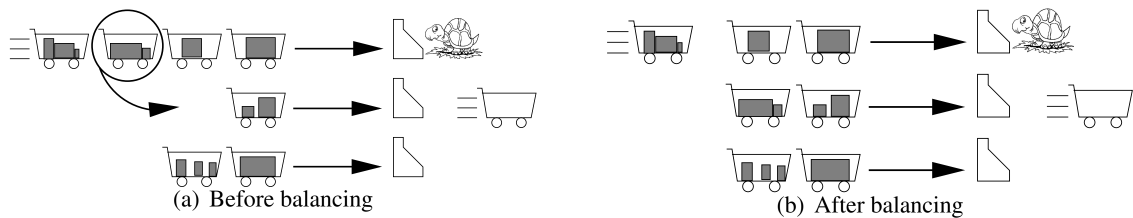

(a) Before balancing (b) After balancing

Figure 5.2: The supermarket abstraction for load-balancing of enqueued tasks.

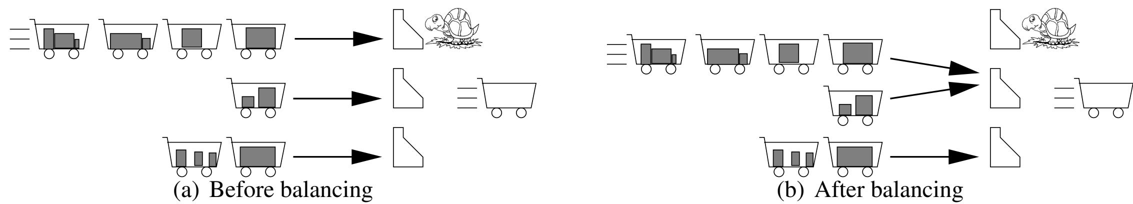
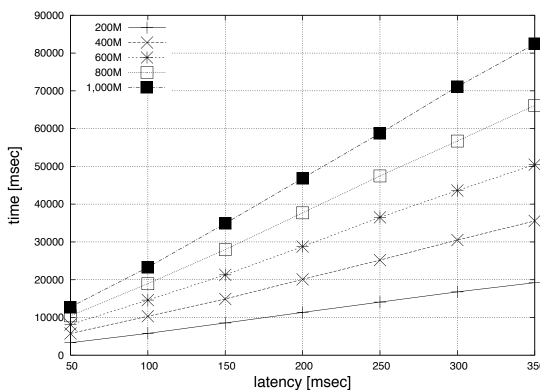

(a) Before balancing (b) After balancing

Figure 5.3: The supermarket abstraction for load-balancing of Active Objects.

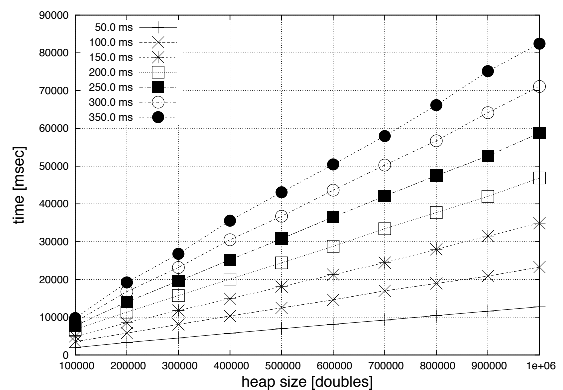
[1]On active-objects there are no notion of task, instead of it, we will use the term *service* (for the service queue).

5.2 Location policy for load-balancing of active-objects

In order to produce a good location policy for load-balancing of active-objects we need a good estimator of the migration costs. Therefore, we measured the migration time between two given nodes, varying the communication latency (which we call the *distance* of the nodes) and the heap-size of the object in *doubles* (one double equals four bytes). The distance parameter was $50,100,150,...,350$ milliseconds (ms) and the object size between $100,000$ and $1,000,000$ doubles. The result was that the migration time corresponds linearly to the size of the object and to the latency of the communication between two nodes. This result is very important for the location policy: an active object will serve no requests while it is migrating; therefore, higher the migration time, slower the performance of the parallel application. Unfortunately, a reliable approximation of the size of the active-object can only be obtained during (or after) a migration, and it may be too late. However, a good estimation of latency can be achieved by network topology (Chapter 3). Therefore, our location policy will aim to locate "*a good, close partner*" (see Section 5.4.2) and even big objects will not have a too high migration time.



(a) Fixed object size



(b) Fixed distance

Figure 5.4: Migration time from the point of view of latency and object' size

5.3 Information and transfer policies for load-balancing of active-objects

The objective of this section is to determine good information-sharing and transfer policies for load-balancing of active-objects. To measure the performance of the different policies we use simulations validated with practical experiences.

In this section, we classify partial-information policies by their transfer policy: *Eager* or *Lazy*. *Eager* policies correspond to the ones where an *overloaded* node triggers the load-balancing, and therefore the shared information corresponds to the underloaded nodes. *Lazy* policies correspond to the ones where the *underloaded* node triggers the load-balancing, and therefore the shared information corresponds to the overloaded nodes.

5.3.1 Modelling ProActive behaviour to test algorithm policies

Each **node** represents a machine (virtual or real) which participates in the balancing. As in (114), we compare centralised and distributed algorithms, adding also partial-information algorithms in our experiments. In ProActive, there is no notion of **tasks** as in parallel batch systems (86) we will use the term **task** to refer to a **service** (94), adding the term **job** for a **set** of services **served** by an active object. In the literature, the word **load** represents a metric such as the CPU queue length, the available memory, a linear combination of both, etc. In this work, **load** represents the number of tasks in the CPU queue modelled with ProActive (see Section 5.3.2). In our study, **response time** is the time since a node entering the *overloaded* state and the beginning of the load-balancing.

Following the recommendations of (13; 35), we simulate the load of each node with a discrete-time population process with birth-rate λ and death-rate μ. The value of λ represents the number of jobs which arrive every second to a node. The job size (in terms of number of tasks) follows an exponential distribution with mean 1. The death-rate μ represents the number of tasks served by a single node per second. In our experiments we use $\lambda = 1, 2, ..., 10$, and in order to maintain the

system stable: $\mu = 10$. Note that this methodology simulates the load balance process and its communications. Simulation data will conclude whether the policies hinder intensive-communicated parallel applications.

Because our experiments have to be comparable for all policies and number of nodes, we calculated the total number of incoming tasks every second (along a period of 60 seconds) for each value of λ. These precomputed values were used for all the experiments.

In our experiments, the nodes are labelled $0, ..., n$ and the value of λ assigned to the node i is $\lambda_i = 1 + i \bmod 10$. Each node used the initial precomputed incoming rate λ_i, and after 60 seconds, the simulation was restarted again with the value of λ_i.

Several studies have shown that on a set of workstations (without load-balancing), more than 80% of the workstations are idle during the day (80; 86; 114). The concept of *occupied* workstations and *overloaded* nodes are similar: processors which want to share work. Therefore, in our study, if no load balance was made, 20% of the nodes had to reach the overloaded state. To achieve this with the previously calculated values for λ, we used the convention:

- Underloaded Node: load < 10.
- Normal Node: $10 \leq$ load < 15.
- Overloaded Node: load ≥ 15.

5.3.2 Implementing the Information-Sharing Policies

When dealing with communication-intensive applications (parallel applications which transfer a large amount of data among processors), the information-sharing policy influences not only the load-balancing decisions but also the communication itself. We studied this problem, because our results can be applied in the context of load-balancing on peer-to-peer networks.

This section describes experiments which measure the response time and bandwidth usage for different information-sharing policies applied by well-known load-balancing algorithms.

Each node is modelled as an *active object* with three principal operations:

- `register`: registers on the communication channel (server, broadcast). This method starts the clock in our experiments.
- `loadBalance`: starts the load-balancing process, to stop the clock in our experiments, and to calculate the response time.
- `addLoad(x)`: adds x tasks to the called object.

Centralised

For this policy, one active object was chosen as a central server which collected and stored load-balance information of each node as: underloaded, normal or overloaded. The policy works as follows:

- Every second, the nodes call the remote `register` execution on the server.
- The *load server* processes incoming method calls. If the call originates from an overloaded node, the server randomly chooses an address of an underloaded node (if any) and calls the method `loadBalance` on the overloaded node with the chosen address.
- The overloaded node performs locally `addLoad(-myLoad/2)` (according to the recommendations of Berenbrink, Friedetzkyand Goldberg (13)) and the underloaded node (remotely) performs `addLoad(myLoad/2)`.

Lazy Centralised

We studied this policy aiming at a reduction of the information transmitted over the network. For this, we included an `unregister` method in the node model. This policy is described as follows:

- When a node reaches the overloaded state, it registers on the central server, and
- When a node leaves the overloaded state, it unregisters (removes its reference) from the server.
- Every second, if a node is underloaded it asks the server for overloaded nodes. When the server receives that query, it randomly chooses the address of an overloaded node (if any), and starts the load-balancing: ordering the overloaded node to balance with the node that originated the query.

Eager Centralised

This policy is similar to the previous one, but underloaded nodes share their information instead of overloaded ones. The nodes register on the server when they reach the underloaded state and unregister when leaving it:

- When a node is in overloaded state, it asks the server for underloaded nodes once per second.
- Upon receiving the query, the server randomly chooses the address of an underloaded node (if any) and begins the load-balancing by ordering the overloaded node that sent the query to balance with the chosen underloaded node.

Distributed

The policy is similar to *Centralised*, but instead of sending the information to a central server, nodes broadcast their information. Therefore, all the nodes are servers, and each node makes its own balance decisions (i.e.: local decisions), using information collected from the communication channel.

Lazy Distributed

This policy is similar to *Lazy Centralised*, but in this case the information is shared through the multicast channel instead of a central server. As in *Distributed* policy, every node is also a server and the decisions are local. We expected this policy to have similar time delay but use less bandwidth than the *Distributed* policy due to the reduction in number of messages sent.

Eager Distributed

This policy is the broadcast version of *Eager Centralised*, and we expected a behaviour similar to the *Lazy Distributed* policy.

5.3.3 Hardware and Software

We simulated the models using the Oasis Team Intranet (93). We tested the policies on an heterogeneous network composed of: 3 Pentium II 0.4 GHz, 10 Pentium III 0.5 - 1.0 Ghz, 3 Pentium IV 3.4GHz and 4 Pentium XEON 2.0GHz for the nodes and a Pentium IV 3.4GHz for the server. We uniformly at random distribute the nodes (active objects) on the processors. For response-time measurements we used the system clock, and for bandwidth measurements we used Ethereal

(Combs) software. The policy methods for nodes and servers were developed using the *ProActive* middleware on Java 2 Platform (Standard Edition) version 1.4.2.

5.3.4 Results Analysis

We tested the policies on $20, 40, 80, 160, 320$ nodes distributed on 20 machines. For each case we took 1000 samples of response times and the bandwidth reports from *Ethereal*. In this section we present the main results of this study. We will first discuss the response time, and then the bandwidth analysis.

Response Time

Figure 5.5 shows the response time for all policies using the model defined in Section 5.3.1. Note that in the *Eager Distributed* policy, overloaded nodes collect the information from underloaded nodes *before* the balancing takes place. Therefore, the response time is near zero, and we omitted this policy from the plot.

According to the recommendations of (91), response time should be less than the periodical update time, and in this study the update time is 1000 ms.

Using this reference, distributed policies presented better response times than centralised policies. Also, policies that sent underloaded information (*Eager* policies) had better performance than policies which shared overloaded information (*Lazy* policies). This happens because in the *Eager* policies, the overloaded nodes generate the load-balancing requests, while in *Lazy* policies overloaded nodes have to wait until an underloaded node initiates the load-balancing.

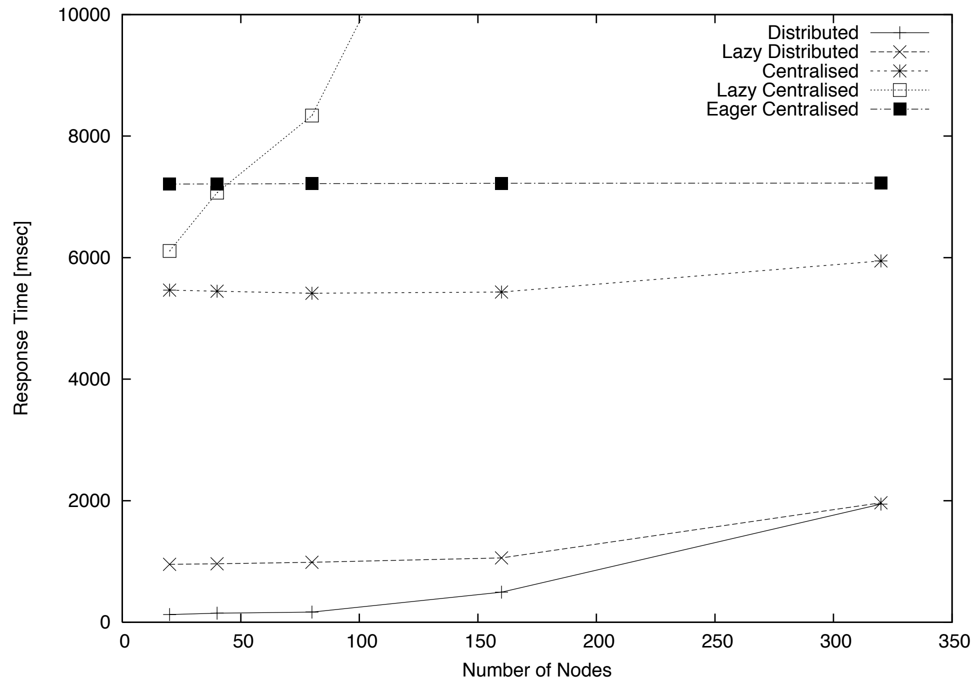


Figure 5.5: Mean response time for all policies

Bandwidth

In this section we tested the policies bandwidth usage. Unfortunately, the underlying implementations introduces an additional difference through resorting to TCP or UDP-based communications (resp. *Centralised* and *Distributed* policies). To avoid having to interpret such bias, we compare performance between *full* and *partial* information policies, developed on centralised and distributed load-balancing algorithms.

Figure 5.6 shows the bandwidth used during the information-sharing phase, counting only messages sent to the server:

1. *Centralised* policies use between 5 (*Eager Centralised*) and 40 times (*Centralised*) more bandwidth than distributed policies. This phenomenon is the result of the different type of

network protocols used, and has been well studied in related-work (Sun Microsystems).

2. For *partial information* schemes with *centralised* policies: when overloaded nodes share their information, less than 20% of the total nodes (see Section 5.3.1) will send register/unregister messages, and more than 80% of them will send queries for registered nodes (every second).

3. When underloaded nodes share their information, more than 80% of the total nodes will send register/unregister messages and less than 20% of them will send queries. This behaviour causes the former approach to consume more bandwidth than the latter.

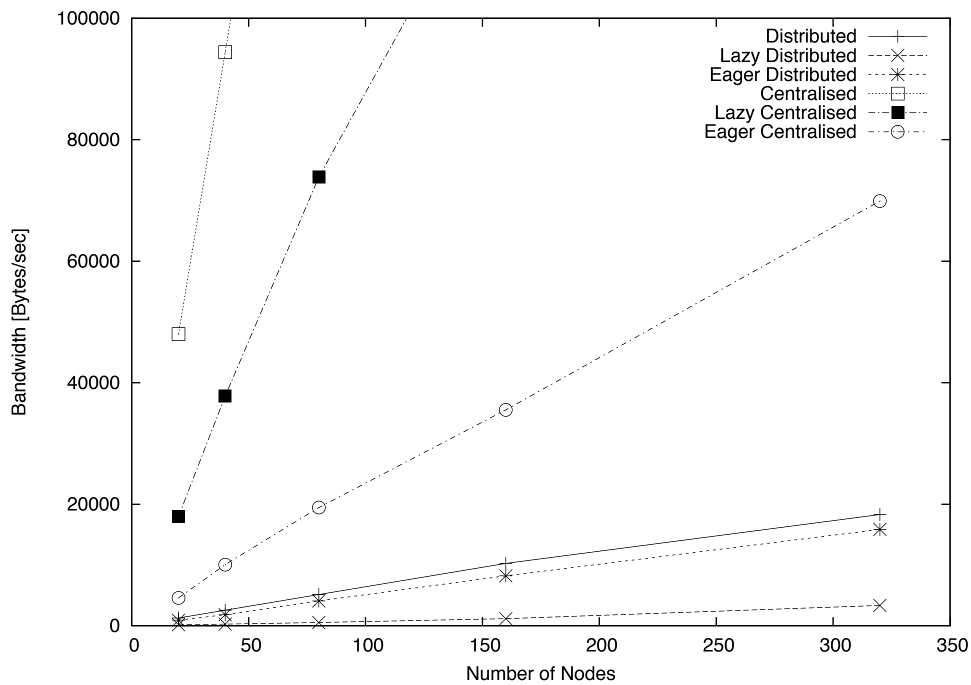


Figure 5.6: Bandwidth usage of coordination policies during the information-sharing phase

Figure 5.7 shows the total bandwidth used by our load model, including the `loadBalance` and `addLoad` messages:

1. *Eager* policies which share *partial* information of underloaded nodes have the lowest band-

width usage for each case (*centralised* and *distributed*).

2. *Lazy Centralised* policies which share *partial* information generate a great increase of the bandwidth usage, because there is no control on how many underloaded nodes send `loadBalance` messages. In the *Lazy Centralised* policy, this behaviour generates a saturation on the communication channel even though the number of messages is half of the *Centralised* policy number. This happens because most of the messages are balance queries, and the server has to choose an overloaded node and send the `loadBalance` message to it.

3. When the service queue of a central server becomes saturated (over 300 nodes on our experiments), the response time increases and the bandwidth usage decreases, because the saturation will cause less messages to be sent over the network. Using a multi-threaded central server can increase the saturation threshold, but it is not a scalable solution because new constraints such as mutual exclusion are generated.

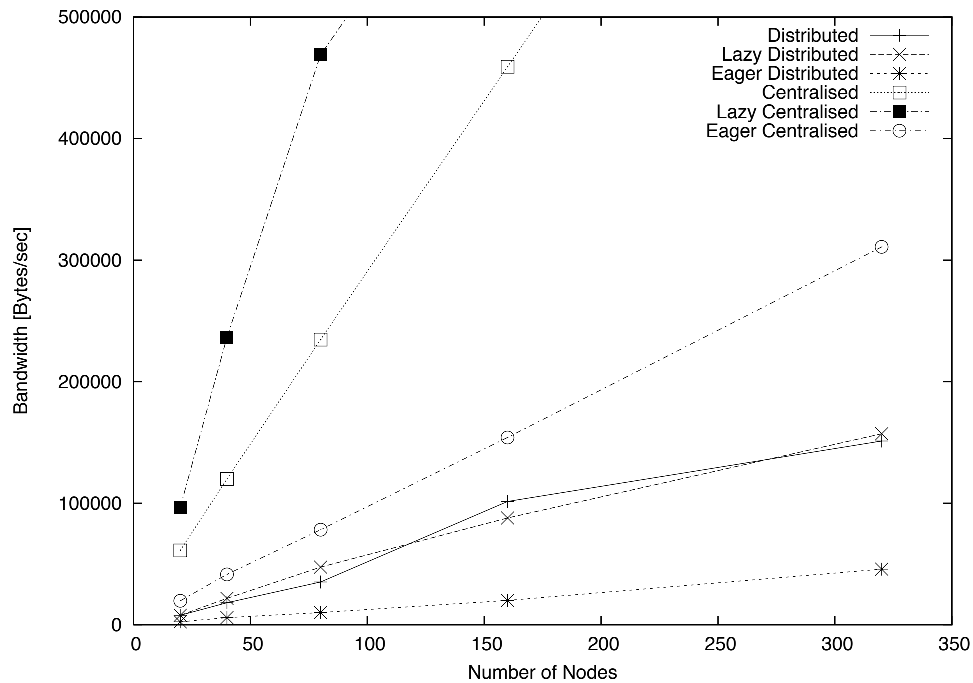


Figure 5.7: Bandwidth usage of coordination policies during all the load-balancing

5.3.5 Testing the impact of Information-Sharing Policies

We tested the impact of the policies with a real application: the calculus of a *Jacobi* matrix. This algorithm performs an iterative computation on a real-valued square matrix. On each iteration, the value of each element is computed using its own value and the value of its neighbours on the previous iteration. We divided a 3600x3600 matrix into 25 disjoint sub-matrices of equal size, each one managed by an active object called "*worker*" (implemented using ProActive). Each worker communicates only with its direct neighbours.

As a reference, all the workers are randomly distributed among 15 machines, using at most two workers per machine. Using this distribution, we measured the mean execution time of performing 1000 sequential calculus of Jacobi matrices (first row of Table 5.1).

To determine the impact of the policies on the *Jacobi* application, we distributed 30 nodes among the 15 machines. We ran the application (placing one load server outside of the simulation machines), and measured the execution time of *Jacobi*. Separately for each policy we measured the CPU cost (in % of busy time) for the 15 machines. The results are in Table 5.1.

Table 5.1: Information-sharing policies and their effects on execution time of a parallel Jacobi application

Policy	Execution Time (sec)	% policy cost (time)	% policy cost (CPU)
None	914.361	—	—
Centralised	1014.960	11.00%	1.3%
Lazy Centralised	995.873	8.91%	1.1%
Eager Centralised	972.621	6.37%	1.1%
Distributed	1004.800	9.89%	10.7%
Lazy Distributed	925.964	1.26%	4.5%
Eager Distributed	915.085	0.08%	4.1%

While *Centralised* policies use less CPU on the "client" side, they use more bandwidth than their distributed equivalents. A special case is the *Distributed* policy, which uses less bandwidth than the *Centralised* policies, but the largest CPU time consumption, and it produces almost 10%

of time delay on the application. So, if this policy is used, the load balancing itself will produce overloading.

We conclude that *Distributed* oriented policies have the best performance using these metrics, and sharing underloaded nodes information (*Eager*), is the best decision. In a load-balancing architecture for communication-intensive parallel applications developed with asynchronous communicated middlewares, we suggest using an *Eager Distributed* policy where overloaded nodes trigger the balancing using previously acquired information, thus avoiding the need for *Centralised* servers. Moreover, if the load index could be updated with a lower frequency than once per second and similar accuracy, the policy would use fewer coordination messages, producing less interference with parallel applications.

5.4 Exploiting the Peer-to-Peer infrastructure: Information on-demand

As we concluded in Section 5.3, the best policy for intensive-communicated parallel applications developed within ProActive, in terms of bandwidth used and response-time, is an *eager* scheme. In this section, we take some ideas from load-balancing studied in Section 4 and design new algorithms for load-balancing of active-objects. Our first approach is a *Robin-Hood eager-centralised* scheme. The algorithm performed a good balance if the initial distribution of the parallel application was near to a local optimum; else, the performance of the algorithm decreased. To improve the algorithm, we study the implementation of the Peer-to-Peer infrastructure of ProActive (see Section 3.2), and develop a new algorithm which exploits its knowledge of the other nodes to find a good selection for balancing.

At the end of this section there are some implementation issues and benchmarking of our algorithm with the Jacobi parallel application.

5.4.1 Robin-Hood Load-Balancing Algorithm

The *Robin-Hood* load-balancing algorithm was the first attempt to perform dynamic load-balancing of ProActive active-objects (23). First it was implemented using a multicast channel but then, given the firewall constraints of multicast channels, it was implemented using a central server (24). The Robin-Hood algorithm uses a central server to store system information, processors can register, unregister and query it for balancing. The algorithm is as follows:

Every `t` units of time

1. if a processor `A` is underloaded, it registers on the central server,
2. if a processor `A` was underloaded in `t-1` and now it has left this state, then it unregisters from the central server,
3. if a processor `A` is overloaded, it asks the central server for an underloaded processor, the server randomly chooses a candidate from its registers and gives its reference to the overloaded processor.
4. The overloaded processor `A` migrates an active object to the underloaded one.

This simple algorithm satisfies the requirements of minimising the reaction time against overloading and, as we explained on Section 5.1, speeds up the application performance. However, it works only for homogeneous networks.

In order to adapt this algorithm to heterogeneous computers, we introduce a function called `rank(A)`, which gives the processing speed of `A`. Note that this function generates a total order relation among processors as the gradient model of Lin and Keller (79).

The function `rank` provides a mechanism to avoid processors with low capacity, concentrating the parallel application on the higher capacity processors. It is also possible to provide the server with `rank(A)` at registration time, allowing the server to search for a candidate with similar or

higher rank, producing the same rank mechanism, with the drawback of adding the search time to reaction time against overloading. In general, any search mechanism of *the best* unloaded candidate in the server will add a delay into server response, and consequently in reaction time.

Before implementing the algorithm, we studied our network and selected a processor `B`[2] as *reference* in terms of processing capacities. Then, we modified the previous algorithm to:

Every `t` units of time

1. If a processor `A` is overloaded, it asks the central server for an underloaded processor, the server randomly chooses a candidate from its registers and gives the reference to the overloaded processor.
2. If `A` is not overloaded, it checks if `load(A,T) < UT*rank(A)/rank(B)`, if true then it registers on the central server. Else it unregisters from the central server.
3. Overloaded processor `A` migrates an active object to the underloaded one.

5.4.2 Robin-Hood over ProActive's Peer-to-Peer Infrastructure

An important issue for load-balancing of active-objects algorithms is the *migration time*, defined as the time interval since the processor requests an object migration, until the objects arrives at the new processor[3]. Migration time is undesirable because the active object is halted while migrating. Therefore, minimising this time is an important aspect at load-balancing.

While several schemes try minimising the *migration time* using distributed memory (17) (hard to implement for ProActive's Active Objects), or migrating idle objects (61) (almost inexistent on intensive-communicated parallel applications), we exploit ProActive's *Peer-to-Peer* architecture (defined on Section 3.2) to reduce the migration time. Using a group call, the first reply will come

[2] Choosing the correct processor `B` requires further research, but for now the median has proved reasonable approach.
[3] In ProActive, an object abandons the original processor upon confirmation of arrival at the new processor.

from the nearest acquaintance, and thus the active object will spend the minimum time travelling to the closest unloaded processor known by the peer. Note that the notion of "nearest acquaintance" is "the node at which active objects can arrive fastest". As seen in Section 5.2, we are trying to minimise latency which is linear to migration time.

We adapted the *Robin-Hood* algorithm, using a subset of peer acquaintances from the *Peer-to-Peer* infrastructure to coordinate the balance. Suppose the number of computers on the P2P network is N, large enough to suppose them load-independents. If p is the probability of having a computer on an underloaded state, and the acquaintances subset size is $n << N$, if we send a request asking for an underloaded node, the probability of having at least k responses is

$$\sum_{i=k}^{n} \binom{n}{k} p^i (1-p)^{n-i} \tag{5.1}$$

Therefore, having an estimation of p, a good selection of the parameter n permits a reduction on the bandwidth used by the algorithm with a minimal addition on reaction time. For instance, using the pairs ($p = 0.8, n = 3$) or ($p = 0.6, n = 6$), one has a response probability greater than 0.99.

The algorithm for P2P networks is: Every `t` units of time

1. If a processor `A` is overloaded, it sends a balance request and the value of `rank(A)` to a subset of n of its acquaintances.
2. When a processor `B` receives a balance request, it checks if `load(B,T) < UT` and `rank(B)` $\geq$ `RB*rank(A)` (RB is a coefficient between [0,1]); if true, then `B` sends a response to `A`.
3. When A receives the first response (from `B`), it migrates an active object to `B`. Further responses for the same balance request can be discarded.

The migration time problem is not the only source of difficulty. There is a second one: the *ping pong effect*. This appears when active objects migrate forwards and backwards between processors.

This trouble is conceptually avoided by our implementation by choosing the migrating active object as the one with *shortest service queue*. During the migration phase, the active object pauses its activity and stops handling requests. For a recently migrated active object, all new requests are waiting in the queue, and will only begin to be treated after the migration has finished. Therefore, a freshly migrated object generally has a longer queue than similar objects on the new processor, thus a low priority for moving. Moreover, on our *Peer-to-Peer* load-balancing algorithm, we chose the target node from a minimal subset of the acquaintances. This random selection reduces the probability of a ping-pong effect.

5.5 Robin-Hood and the Nottingham Sheriff

The main problem of the Robin-Hood algorithm in Peer-to-Peer networks is that the algorithm looks for a proportional equalisation of the system load, and stops the balancing when reaching a local optimum.

To face the algorithm to continue looking for better solutions, we added to the previous algorithm the capacity for an underloaded node to choose another node randomly (a greedy work-stealing approach). If the target node has a lower ranking, an active-object (if any) will be *stolen* from the low ranking node to the higher one. That is:

1. If a processor `A` is underloaded, it randomly chooses one of its acquaintances and it sends to this node a steal message and the value of `rank(A)`.
2. When a node `B` receives a steal request, it checks if it has active-objects. In an affirmative case, if `rank(A) > RS*rank(B)`, `B` migrates an active-object to `A` (RS is a coefficient between [0,1]).

In other words, we have a Robin-Hood algorithm migrating active-objects from overloaded (rich) nodes to underloaded (poor) nodes, and a Ranked Work-Stealing algorithm (the *Nottingham*

Sheriff) trying to collect all active-objects to the best ranked node. We aim to demonstrate that, using a low number of links among nodes and a good selection of parameters, an optimal distribution is reachable.

5.6 Testing algorithms in a real environment

Algorithms were deployed on a set of 25 of INRIA lab desktop computers, having 10 Pentium III 0.5 - 1.0 Ghz, 9 Pentium IV 3.4GHz and 6 Pentium XEON 2.0GHz, all of them using Linux as operating system and connected by a 100 Mbps Ethernet switched network. With this group of machines we used the *Peer-to-Peer* infrastructure to share JVMs. Using our previous experiences (see Section 3.2), we configured the *Peer-to-Peer* infrastructure with: `TTU` (Time-to-Update the acquaintances list) at 10 minutes, `NOA` (Minimal size of acquaintances set for each peer) at 10 peers and `TTL` (depth in hops of the peer searching request) at 5 hops. At first only one peer was chosen as server for the first contact, and other peers used it to join the infrastructure.

Functions `load()` (resp. `rank()`) of Section 4.2 and 5.4.2 were implemented with information available on `/proc/stat` (resp. `/proc/cpuinfo`). load-balancing algorithms were developed using *ProActive* on Java 2 Platform (Standard Edition) version 1.4.2.

In our experience, we used our knowledge of the lab networks to have, in normal conditions, 80% of desktop computers on underloaded state (as it was reported by Litzkow, Livny and Mutka (86)), defining the parameter `UT` of the algorithm as `UT` = 0.3; and, to avoid swapping on migration time, defining `OT` = 0.8.

Since the *CPU speed* (in MHz) is a constant property of each processor and it represents its processing capacity, and after a brief analysis of them on our desktop computers, we define the `rank` function as: $rank(P) = \log_{10} speed(P)$.

When implementing the algorithm, a new constraint appears: all load status are checked each `t`

units of time (called *update time*). If this *update time* is less than migration time, extra migrations which affect the application performance could be produced. After a brief analysis of migration time, and to avoid network implosion, we assume a variable $\tilde{t}$ which follows an uniform distribution and experimentally define the update time as:

$$t_{\text{update}} = 5 + 30\,\tilde{t}(1 - load)[\text{sec}],\ (load \in [0,1]) \tag{5.2}$$

This formula has a constant component (migration time) and a dynamic component which decreases the update time while the load increases, minimising the overload reaction time.

We tested the impact of our load-balancing algorithm over a concrete application: the *Jacobi* matrix calculus. This algorithm performs an iterative computation on a square matrix of real numbers. On each iteration, the value of each point is computed using its value and the value of its matrix neighbours in their last iteration. We divided a 3600x3600 matrix in 36 workers all equivalents, and each worker communicates with its direct matrix neighbours.

Looking for lower bounds in Jacobi execution time, we measured the mean time of Jacobi calculus for 2, 3 and 4 workers by machine, using the computers with higher *rank* and without load-balancing. Horizontal lines on Figure 5.8 are the values of this experience. Note that those values are a good approximation for the static optimal distribution.

Initially, we randomly distributed Jacobi workers among 16 (of 25) machines, measuring the execution time of 1000 sequential calculus of Jacobi matrices. First, we used the central server algorithm defined on Section 4.2 (having a CPU clock of $3GHz$ as reference) and then using the P2P Robin-Hood versions defined on Section 5.4.2 . Measured values of these experiences using $RB = 0.7$ and $RS = 0.9$ can be found in Figure 5.8.

While the central server oriented algorithm produced low mean times for low rate of migrations (an initial distribution near to the optimal), *Peer-to-Peer* oriented algorithm presents better performance while the number of migrations increase. Moreover, considering the addition of mi-

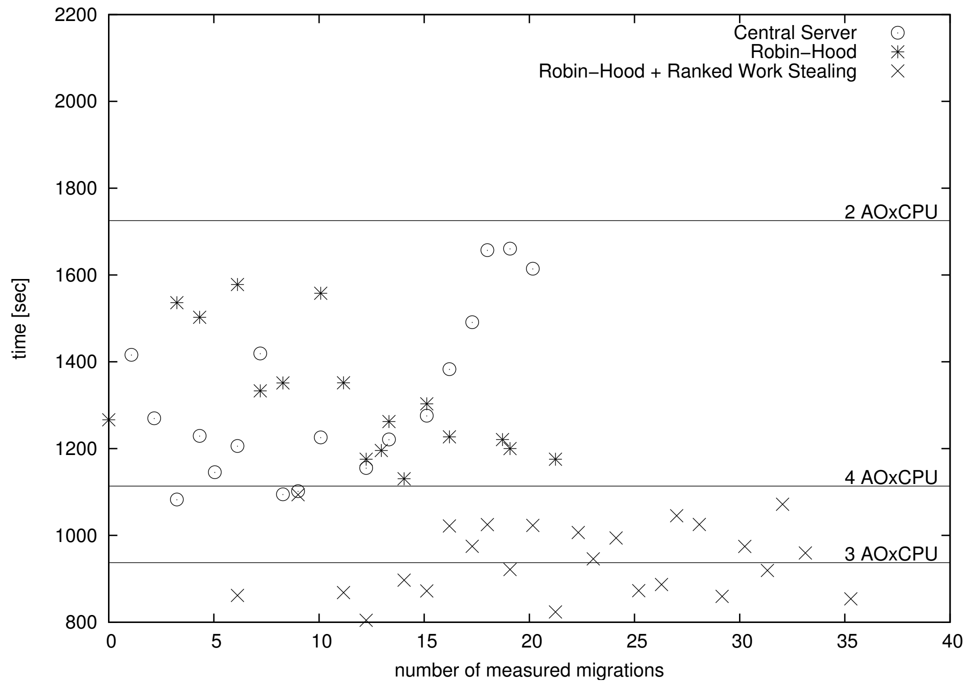


Figure 5.8: Impact of load-balancing algorithms over Jacobi calculus

gration time on Jacobi calculus performance, *Peer-to-Peer* load-balancing algorithms produces the best migration decisions only using a minimal subset of its acquaintances. The use of this minimal subset produces also a minimisation in number of messages for balance coordination. This fact and the acquaintance approach of our P2P network provide automatically scalability conditions for large networks.

However, the plot in Figure 5.8 shows that, for Robin-Hood algorithm, the presence of a local optimal attempts against a good performance of the application; and, for Robin-Hood algorithm using Ranked Work-Stealing, a performance near to the global optimal state is reached for all migration number; that is, for all initial distributions.

Chapter 6

Models, Simulations and Deployment on Large-Scale Networks

"Make everything as simple as possible, but not simpler". (Albert Einstein)

The grid computing research community has started to realise the importance of validated models for simulation work. Therefore, there have been several approaches in the last 2–3 years to model the grid (67; 69; 73; 81; 84). However, to our knowledge, there are no previous attempts to research the characteristics of a part of a grid infrastructure. For instance, the work of Lu and Dinda (81), and of Kee *et al.* (69) focuses on a realistic model for the resources involved within a cluster-based grid, focusing on the model of processors clock speed and number of processors per node. Kondo *et al.* (73) describe a desktop grid environment, in which resources may enter and leave at any moment, focusing on resources availability and provided performance of resources. Medernach (84) analyses the traces of a cluster in a grid computing environment. His work is complemented by the study of Iosup *et al.* (67). The main topics in both these efforts are the characterisation of the main patterns for job submission in their respective environments. Therefore, no other works research about processing capacity on Desktop Grids and Latency on

Institutional-Project Grids.

Our work targets at two main characteristics: processing capacity, presenting a simple but realistic model; and inter-resource communication latency, unstudied yet in a grid environment. Using our Grids models, we simulate our active-objects load-balancing algorithm aiming to select the best behaviour for large-scale Grids.

6.1 Simulating Desktop Grids

In this section we present a contribution on dynamic load balancing for distributed and parallel object-oriented applications. We specially target Desktop Grids and their capability to distribute parallel computation. Using an algorithm for active-object load balancing, we simulate the balance of a parallel application over ProActive's P2P infrastructure. We tune the algorithm parameters in order to obtain the best performance, concluding that our algorithm behaves well and scales to large peer-to-peer networks (around $8,000$ nodes).

This section is organised as follows. First we present the simulated environment of our tests; then, the fine tuning of algorithm parameters, and finally the scalability tests performed over our model of Desktop Grids.

6.1.1 Characterising nodes of Desktop Grids

In the study of load-balancing algorithms, one of the most important characteristics of nodes are their *processing capacity*. A function using this capacity and the amount of work that a node has to perform determine if a node is on an overloaded or underloaded state. To have a reliable model of processing capacity, we made a statistical study of desktop computers registered at the Seti@home project (95). This project aims at analysing the data obtained from the Arecibo Radio telescope, distributing units of data among personal computers and exploiting the processing capacity of

up to 200,000 processors distributed around the world. We analyse the *Mflops* information of Seti@home reported by BOINC (3) benchmarks. We consider *Mflops* as a good metric to determine the processing capacity for parallel scientific calculus, because we are interested in processing balance, not data balance.

We grouped all desktop computers *Mflops* (d_r) in 30 clusters (C_t) using the following formula:

$$d_r \in C_t \; if \; \left\lfloor \frac{r}{10^6} \right\rfloor = t \;\; ; \; therefore \; t = 0, ..., 3000 \tag{6.1}$$

The resultant frequency histogram is shown in Figure 6.1.

Defining a normal distribution $\mathcal{N}(x)$ (equation (6.2)), we compared the real distribution against our model function using Kolmogorov-Smirnov test statistics (*KST*), giving us a value of $KST = 0.0605$ (See Appendix C). Therefore, we can deduce that using a level of significance 0.01, the capacity of processors in a Large-Scale network can be modelled by a normal distribution.

$$\mathcal{N}(x) = 16,000 \times exp\left(\frac{-(x-1,300)^2}{2 \times 400^2}\right) \tag{6.2}$$

6.1.2 Modelling Desktop Grids

Considering a discrete representation of the Euclidean space in which the resources are physically located, we implemented in C a network simulator, using an $n \times n$ matrix for the nodes and an $n^2 \times n^2$ matrix for the edges. We assign the nodes processing capacities (called μ) using a normal distribution $N(1, \frac{1}{9})$ (see Section 6.1.1).

In our simulations, we assume that all active-objects are parts of a parallel application; therefore, we assume all service queues to have equal incoming message ratios λ. Clearly, real Grids run different parallel applications from different sources, having different service queue ratios and workloads. Nevertheless, from the point of view of a given parallel application, we consider other

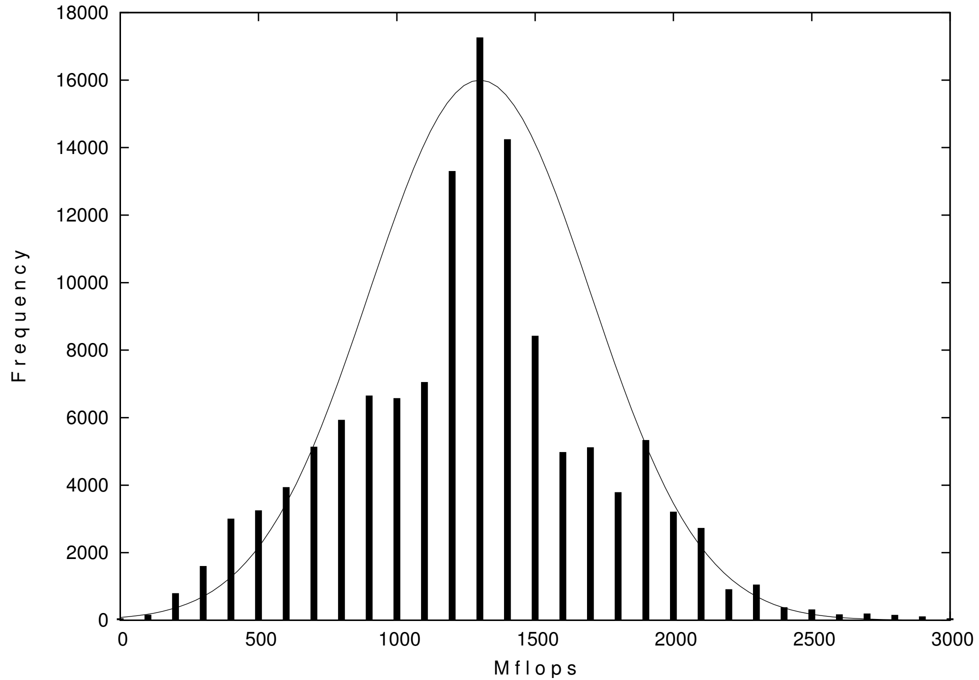


Figure 6.1: Frequency distribution of Mflops for 200,000 processors registered at Seti@home and the normal function which models it.

applications only as a reduction of processing capacity of network nodes for given time periods.

Denoting by j the number of active objects in the node i at a given time, we say that the node i is overloaded if $j\lambda \geq \mu_i$ and underloaded if $j\lambda < T\mu_i$, where T is a given threshold between [0.5, 0.9]. The processor capacity μ_i is also used as the node rank. For consistency with the previous section, we use underload threshold $UT = T \times \mu_i$ and overload threshold $OT = \mu_i$. Each experimental sample is the mean number of 100 repetitions, fixing the parameter set $\{n, m, \lambda, T, RB, RS\}$ (see Table 6.1) and recalculating μ for all nodes in each repetition.

Table 6.1: Parameters and variables used in the simulation

Simulation parameters		Model parameters		Algorithm parameters	
$n \times n$	number of nodes	μ	processor's capacity and ranking	UT	threshold to determine an underloaded state
m	number of active objects	λ	incoming ratio of an active object service queue	OT	threshold to determine an overload state
x, y	initial deployment subset, x is the length and y the high of the network area	T	factor used to determine UT	RB, RS	load-balancing and work-stealing similarity factors

6.1.3 Finding the best processor

We placed 50 active-objects in $(0,0)$ and tested the load-balancing algorithms (with and without stealing), measuring how many of them are capable to traverse all the network until the node with the best capacity $(n-1, n-1)$. Tuning the values of the similarity factors RB and RS, we analyse the final distribution generated by algorithms. In this experience, we define that the node $(0,0)$ has capacity 0, and the node $(n-1, n-1)$ has capacity ∞, testing if active-objects are capable to reach the *best* processor. Our goal is to maximise the number of active-objects in $(n-1, n-1)$; that is, the number of active-objects whom traversed all the network until the node with infinity processing capacity. Note that it is the worst scenario to find the global optimal state.

Each matrix A_i has the number of active-objects per node after the load balancing reaches a stable state (or no active-object can move). Therefore, we repeated the experiment 100 times, each one with different node capacities but equal parameters. Finally, we computed $A = \sum_{i=1}^{100} A_i$. In every matrix A, the number of active-objects per node was normalised by the maximal number of active-objects in a cell. Therefore, each cell in the matrix has values between 0 and 1. To simulate the first response to balancing requests, if there are more than one candidate for balancing, the balance is made with the nearest one.

The objective of this simulation is to demonstrate that using a small number of links a global optimal load balancing (all active-objects on the best processor) can be performed. Then, we tested our algorithms using two different scenario: using fixed links of a "small-world" network and using random links of a ProActive P2P network.

Simulation with fixed links using a *small-world* network

We defined a small world network in Section 3.3.2, showing the model implementation presented by Kleinberg. Considering that the register/forwarding algorithm of the *Peer-to-Peer* infrastructure presented in Section 3.2 uses a probability to accept a fresh peer less than one, and considering that in practice all first contacts will be made in local networks; then, randomly choosing a number of q fixed links from a given acquaintance set, the ProActive's P2P Infrastructure fits with Kleinberg's model for $p = 0$.

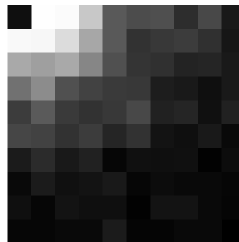
To graphically represent the matrices, we used black for the value 0 and white for 1. If all the objects are concentrated in a single node, only a little box is white and the others are black. If all objects are distributed among the nodes, the matrix will have a grey area. The node $(0,0)$ is the one on the top left and the node $(9,9)$ is the one on bottom right. Also, we measured that, in all final distributions, there are no overloaded nodes.

Figure 6.2 shows the final distribution using a pure Robin-Hood algorithm over the model of Kleinberg for $q = [3,4,5]$, the ponderer $RB = 0.5$, and threshold value $T = 0.5$. We show only those values because similar behaviours are obtained using the values 0.7 and 0.9 for both ponderer and threshold (see matrices in appendix A). We expected that similarity because there exists a correlation between processing capacity and load state: there is a higher probability to find a low capacity node overloaded than underloaded.

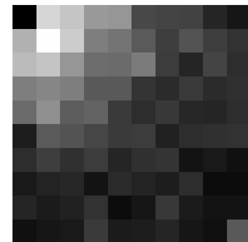
The matrices on Figure 6.2 presents two interesting effects. First, all active-objects leave the node $(0,0)$ and second, the matrices presents the local optimal effect: when no active-object generates overloading, no one of them migrates. Note that low values of the parameter λ generates a

distribution of active objects over the network far from the objective-node $(9,9)$, that phenomenon is explained because the lower the value of λ, the higher the number of active objects which can stay in a node without overloading it.

(a) $\lambda = 0.1, q = 3$



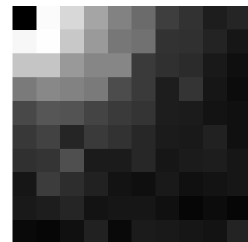
(b) $\lambda = 0.2, q = 3$



(c) $\lambda = 0.3, q = 3$

(d) $\lambda = 0.1, q = 4$

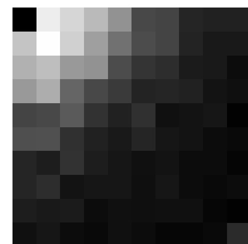
(e) $\lambda = 0.2, q = 4$



(f) $\lambda = 0.3, q = 4$

(g) $\lambda = 0.1, q = 5$

(h) $\lambda = 0.2, q = 5$



(i) $\lambda = 0.3, q = 5$

Figure 6.2: Final distribution for the *Robin-Hood* algorithm only, for $RB = 0.5$ and $T = 0.5$

Adding the *Nottingham Sheriff* step, and using $RS = RB$, the results are significantly different

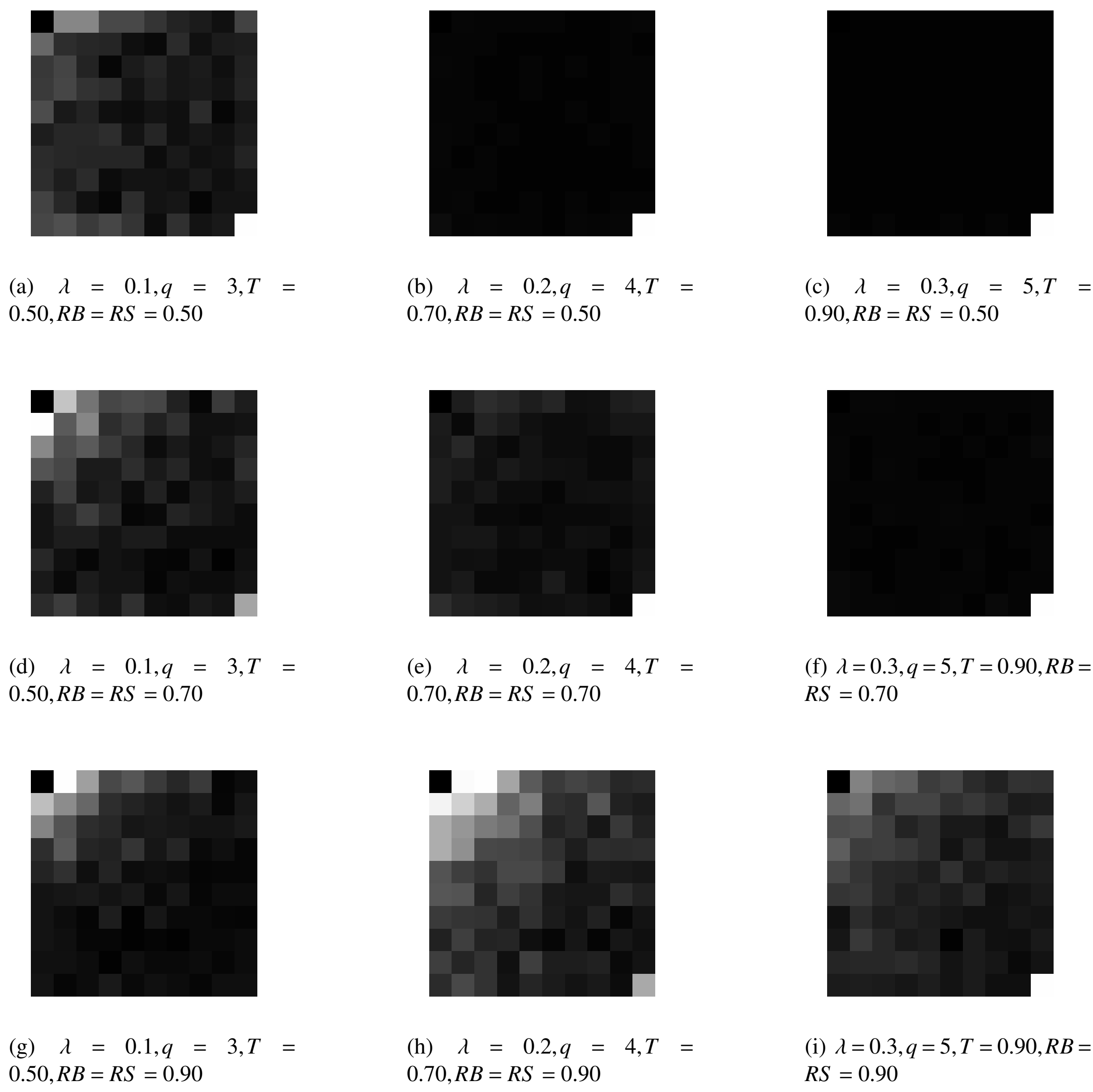
(see Figure 6.3). For active objects with incoming rate λ between 0.2 and 0.3 (Figures 6.3 (b),(c), (d), (f), (i) and (j)), the behaviour is as we expected: the combination of both schemes, *sender* and *receiver* initiated, balances the active objects to the best node $(9,9)$.

Nevertheless, if we consider a low value of λ (near 0.10) and a high value for ponderers RB and RS (Figure 6.3 (g)), it will produce more active objects per node near the initial position $(0,0)$. Active objects would have not the necessity of balancing (because they do not overload the node) or they would not find the path to the best node (because the high value of RB). Moreover, if active-objects cluster near the initial node, and due the fact that on natural networks the edges are between two near nodes, a high value of RS (stealing only if the node is *similar or better* than the target node) does not allow the *Nottingham-Sheriff* step to carry active objects to the best node $(9,9)$. The key point is: what is the cost to carry all those active objects to the best node? if the cost is high, maybe it is not worth to move them there.

It is easy to see that, for low values of λ, if there are 50 active objects and 100 nodes, to use more than $5,000$ migrations will mean that there were be used lots of back-steps (actives-objects returning to previous nodes) during the balance process. Considering the cost of a migration (see section 2.3), all load-balancing algorithm for active-objects will aim at minimising the number of migrations. Figure 6.4 shows in (a) the ratio (percentage) of active objects on the best node $(9,9)$ after a stable state was reached, and in (b) there is the number of migrations used to reach that stable state. Because the results using q from 3 to 6 acquaintances were similar (see matrices in appendix B), only those for $q = 3$ are shown. We can see that the higher the value of RS, the lower the number of migrations and the lower the number active-objects on the best ranked processor. Therefore, using fixed links, there is a low probability to perform an efficient load balancing until an optimal state.

Simulation with randomly chosen links using a Peer-to-Peer network

In the previous experiment we demonstrated that using a low number of fixed links, a global optimal load balance may be performed using a high number of migrations. In this section, we aim



(a) $\lambda = 0.1, q = 3, T = 0.50, RB = RS = 0.50$

(b) $\lambda = 0.2, q = 4, T = 0.70, RB = RS = 0.50$

(c) $\lambda = 0.3, q = 5, T = 0.90, RB = RS = 0.50$

(d) $\lambda = 0.1, q = 3, T = 0.50, RB = RS = 0.70$

(e) $\lambda = 0.2, q = 4, T = 0.70, RB = RS = 0.70$

(f) $\lambda = 0.3, q = 5, T = 0.90, RB = RS = 0.70$

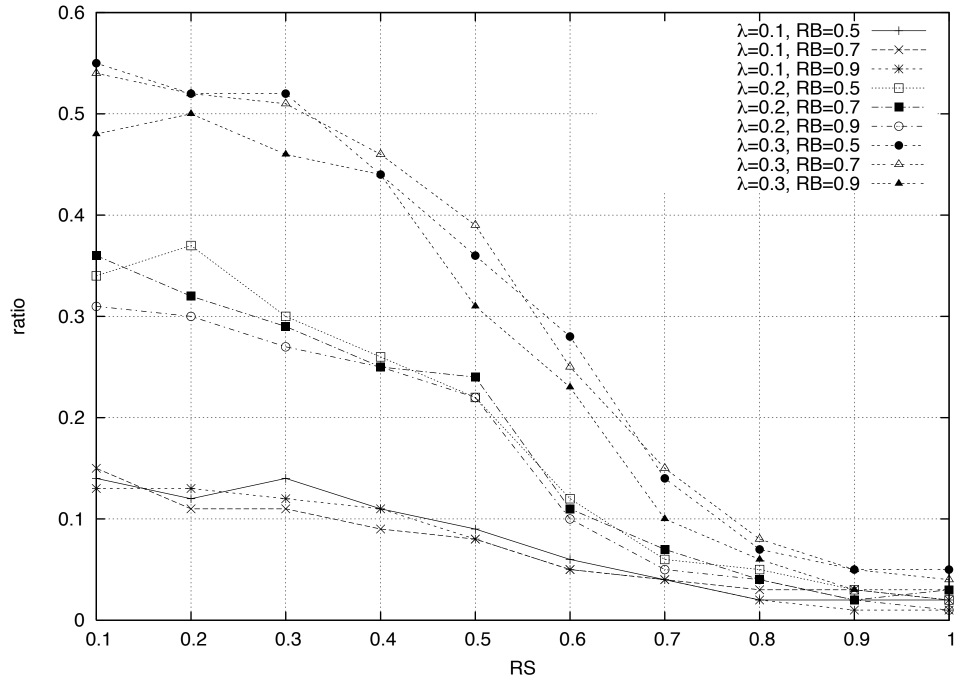
(g) $\lambda = 0.1, q = 3, T = 0.50, RB = RS = 0.90$

(h) $\lambda = 0.2, q = 4, T = 0.70, RB = RS = 0.90$

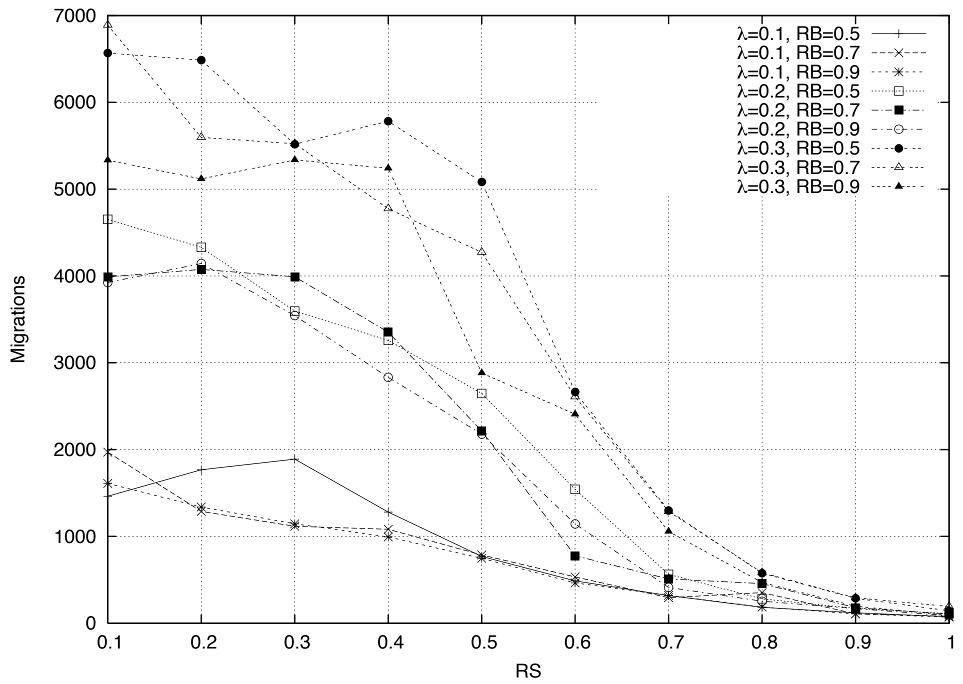
(i) $\lambda = 0.3, q = 5, T = 0.90, RB = RS = 0.90$

Figure 6.3: Final distribution for the *Robin-Hood* + *Nottingham Sheriff*

at demonstrating that using a low number of randomly chosen links, the global optimum can be reached using less number of migrations.



(a)



(b)

Figure 6.4: Tuning for RS considering: a) number of active-objects in (9, 9) per total of active-objects; and b) Number of total migrations reaching a stable state.

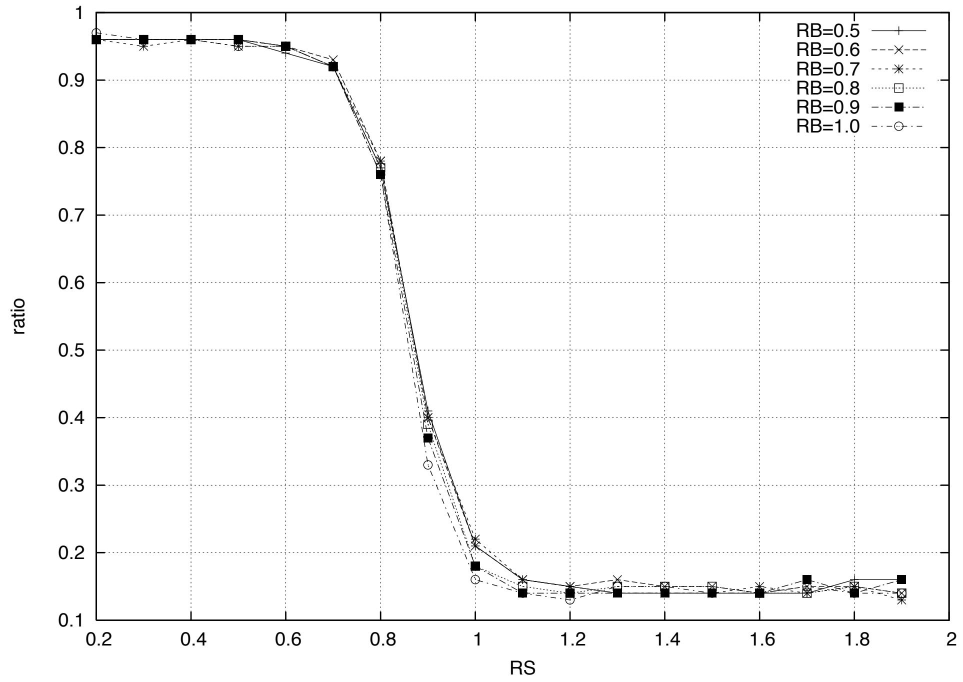
We study the number of active-object until the algorithm reaches its final distribution over the P2P infrastructure (24) having all peers with at least 5 acquaintances. For the *Robin-Hood* algorithm we randomly choose 3 to 6 acquaintances to send the balance request, and for the *Nottingham Sheriff* step we still randomly choose only one acquaintance. The resulting matrices are similar to those obtained in the previous section, therefore in this case we use for illustration 2D plots (see Figure 6.5), having the values for RS on the X-axis and placing on Y-axis: a) ratio of active-objects in $(9, 9)$ per total of active-objects; and b) number of total migrations until a stable state is reached. Our goal is to determine the tuning of the parameters in order to have the maximal numbers of active-objects in the best node using the minimal number of migrations.

Figure 6.5(b) shows that number of migrations if we use a value $RS \leq 1.9$. Figures 6.5(a) and 6.5(b) clearly present a trade-off in the values of RS: if this value is low, most of the active-objects will reach the *optimal node*, but using a high number of migrations (most of them back-steps). If RS is high, no *steal* will be performed. Therefore, considering that in real P2P networks there will be more than one node with high processing capacity (hence, active-objects would have not to traverse through the entire network to find it) we recommend to use the parameter RS with values near 0.9. In our worst scenario a value of $RS = 0.9$ gives that around 35% of active-objects might reach the best ranked node.

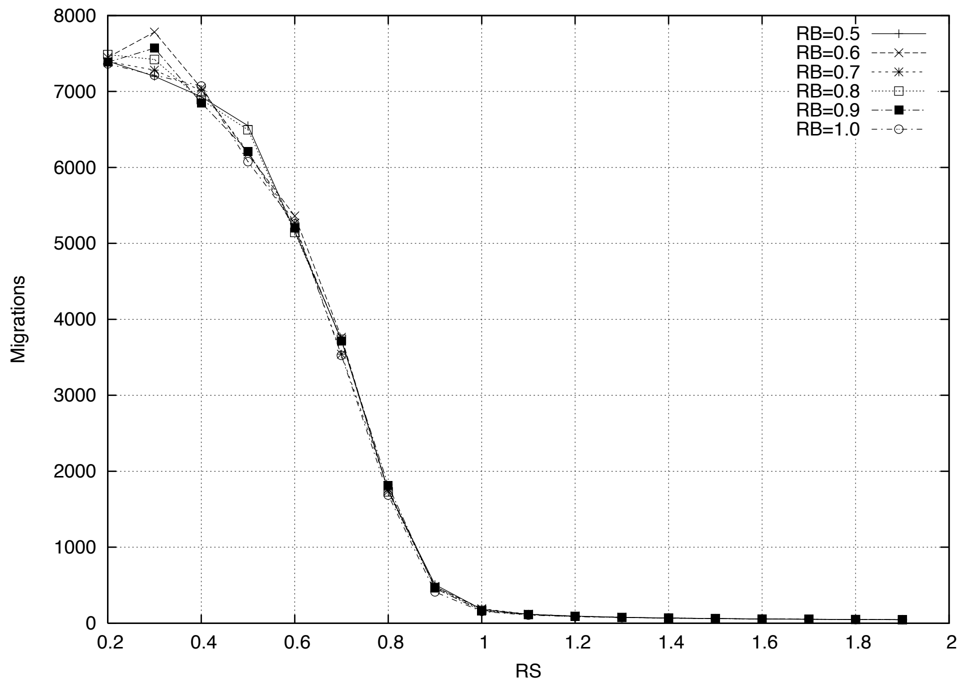
Figure 6.5 shows the same behaviour for values of RB between 0.5 and 0.9. As we can see in previous section, there exists a correlation between processing capacity and load state. For that reason, and to avoid back-steps, we recommend to use RB values in the range $[0.5, 0.9]$. Values lower than 0.5 might produce migrations to very low ranked processors, which could be overloaded with the execution of only one active object, and values higher than 0.9 will reduce the probability to find an underloaded node to perform the balance, increasing the response time of the algorithm.

We also presented that there exists a trade-off between the number of active-objects that can traverse the network to find an optimal node and the number of migrations performed by them. In a middleware such as ProActive, minimising number of migrations (that are costly in processing time) is essential to any load balancing algorithm.

Therefore, we suggest to use a value near 0.9 for the stealing ponderer (RS), which permits



(a)



(b)

Figure 6.5: Tuning for RS considering: a) number of active-objects in (9, 9) per total of active-objects; and b) Number of total migrations reaching a stable state. Because the results using 3 to 6 acquaintances were similar, only those for 3 are shown.

that more than 35% of the active-objects do traverse all the network to find the optimal node, using around 400 migrations.

6.1.4 Scaling towards the "infinite network"

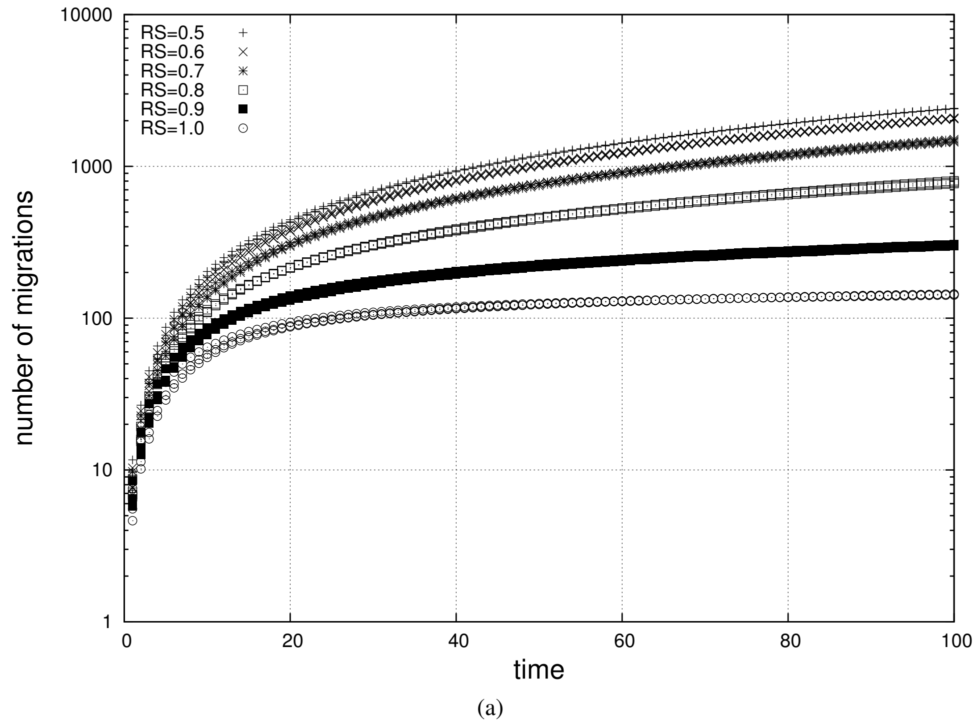
Our goals are to perform a fine-tuning of the constant RS and second to determine whether our algorithm can reach a stable state near to the optimal on large-scale P2P networks using a minimal subset of acquaintances. Even though migration cost seems to be a key issue for load balancing algorithm, it is possible that processors use the blocking or idle time of the parallel application to perform migrations having a low overcost in application total time.

Now we will use a different initial placement: we randomly placed m active objects in $(0+x, 0+y)$ (x and y defined on runtime) and tested the load-balancing algorithm, measuring the total number of migrations and the kind of processors used by the algorithm on each time-step. Each experimental sample is the mean number of 100 repetitions, fixing the parameter set $\{n, m, \lambda, T, RB, RS\}$ (see Table 6.1) and recalculating μ for all nodes in each repetition.

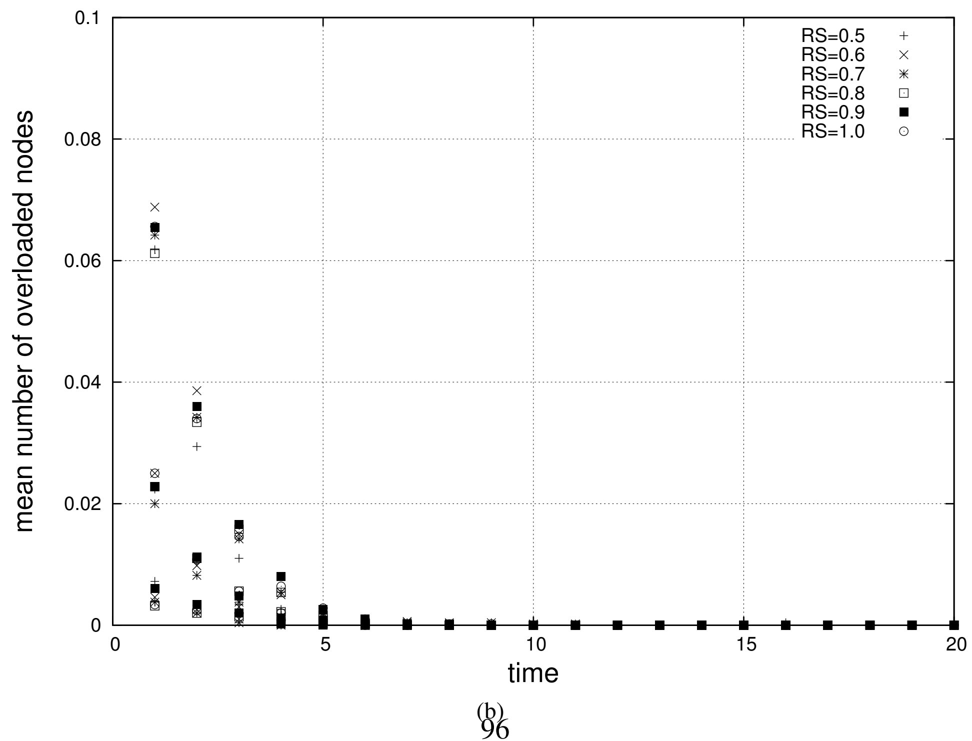
Fine-Tuning

We placed m = 50 active-objects in a simulated P2P network of 100 nodes, measuring the total number of migrations performed by the algorithms until a given time-step (Figure 6.6a) and the number of overloaded nodes per time-step (Figure 6.6b), because it is imperative for all load-balancing algorithms to avoid increasing the number of overloaded nodes. As we expected, a lower value for RS generates a greater number of migrations. It is easy to see that a low value of this factor will produce bad decisions of balance, migrating active objects to underloaded nodes with low processing capacity. Then, those active objects could cause overload in subsequent nodes, or an infinite migration among underloaded nodes.

Figure 6.7a presents the mean number of active-objects in nodes with capacity higher than



(a)



(b)

Figure 6.6: Tuning for RS considering: a) mean number of total migrations until each time-step; and b) mean number of overloaded nodes in each time-step. Using $RB = 0.7$, acquaintances subset size = 3, $|x - y| \leq 3$, $\lambda = 0.1, 0.2, 0.3$ and $T = 0.7$

one per total number of active objects during 100 repetitions, and Figure 6.7b presents the mean number of active objects in nodes with capacity higher than $1\frac{1}{3}$ by total number of active objects during 100 repetitions. Because we are using a normal distribution for the processor capacity μ, 50% of nodes will have $\mu \geq 1$ and 25% of nodes will have $\mu \geq 1\frac{1}{3}$.

Two behaviours are present in Figure 6.7 (a) and (b). First, because our algorithm aims to cluster active-objects on the best processors, for high values of RS, the number of active objects in the best quadrant of the processors increase. Second, for low values of RS, some active objects are stolen by worse processors. We can see from the plots that $RS \geq 0.9$ behaves well, placing all of active objects in nodes with processing capacity greater than one.

Scalability tests

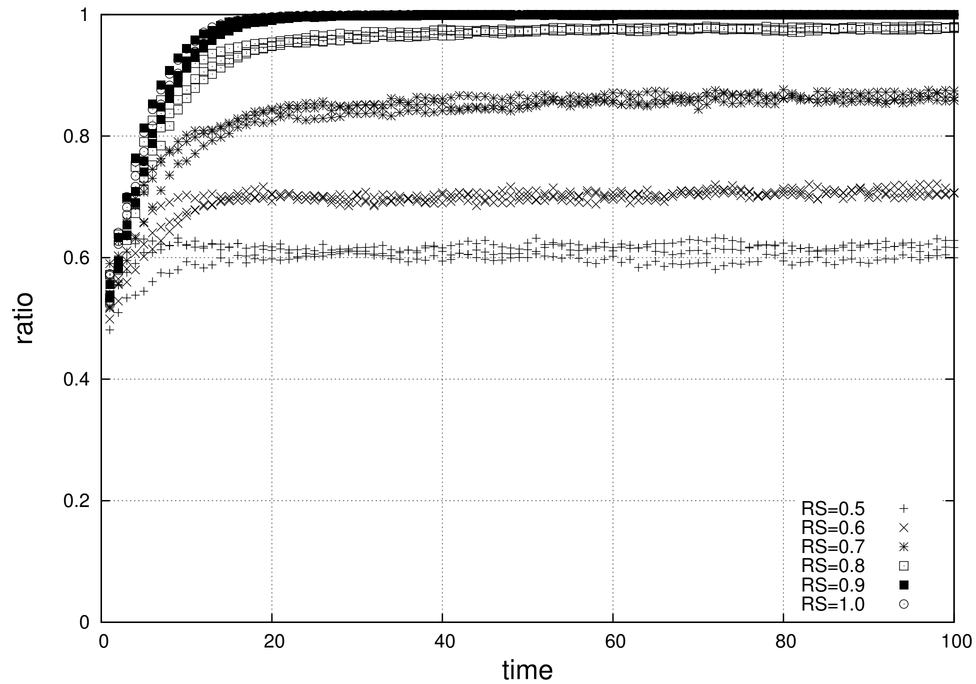
As seen in the previous section, we aimed at optimising the application performance clustering active-objects on the best qualified processors. Therefore, using the values of μ, we sorted the nodes from higher to lower processing capacity and we defined the *optimal subset* as the first OPT nodes that satisfy the condition:

$$\sum_{i=1}^{OPT} \mu_i > m \times \lambda \tag{6.3}$$

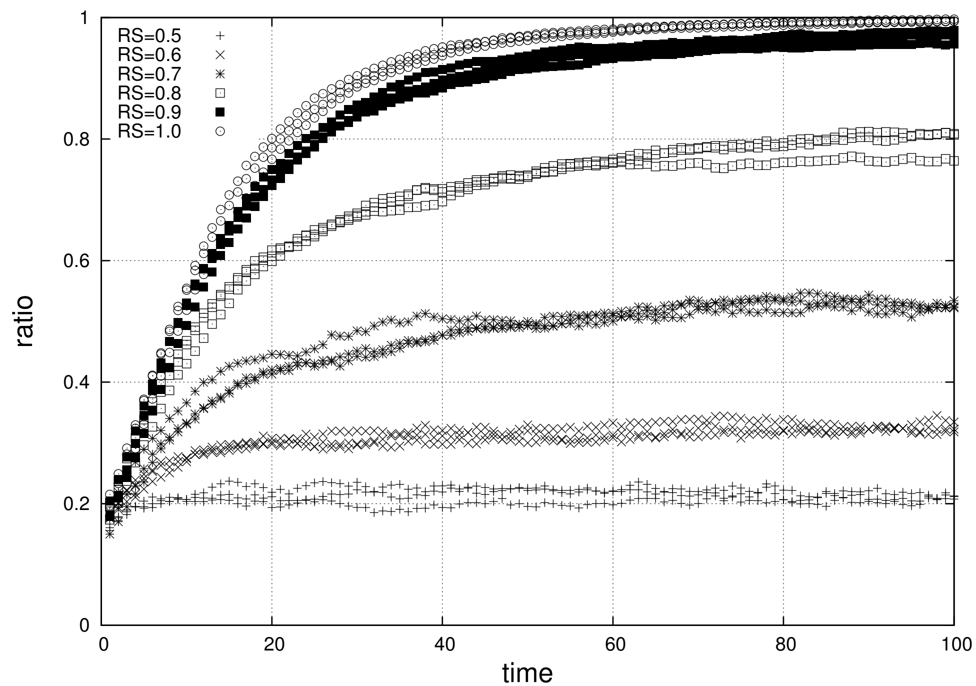
Simulating an application of $m = 100$ active objects using different network sizes ($n \times n$), we have:

- $OPT(n = 10) = 13$,
- $OPT(n = 20, 30) = 11$,
- $OPT(n = 40) = 10$,
- $OPT(n \in [50, 90]) = 9$.

These results of the optimal subset size (OPT) are because we modelled processing capacity following a normal distribution. Therefore, larger the network size, higher the processing capacity of



(a)



(b)

Figure 6.7: Tuning the value of RS considering: a) mean number of active objects on a node with $\mu \geq 1$ per total number of active objects; and b) mean number of active objects on a node with $\mu > 1 + \frac{1}{3}$ per total number of active objects. Using $RB = 0.7$, acquaintances subset size = 3, $|x - y| \leq 3$, $\lambda = 0.1, 0.2, 0.3$ and $T = 0.7$

best nodes, then lower the number of nodes in the optimal subset.

In order to measure the performance of the *Robin-Hood* algorithm for large-scale networks, we define the "Algorithm Optimum" (*ALOP*) ratio as:

$$ALOP = \frac{\text{Number of nodes used by Robin-Hood}}{OPT} \tag{6.4}$$

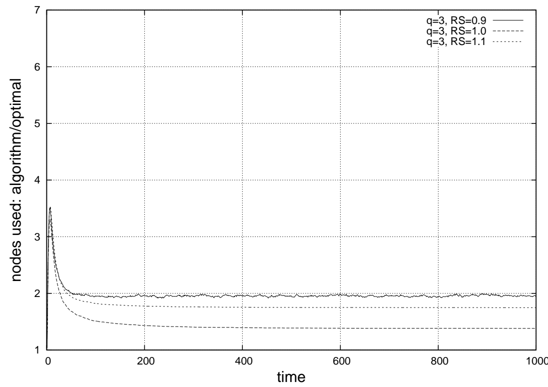
At the same time, we calculate the mean number of accumulated migrations performed by all active objects from time-step 0 until time-step t.

An increase in the acquaintances subset size results in an increase in the probability to find a node to migrate, and hence an increase in the probability to reach the optimal state. Looking for the worst treatable scenario, and following the recommendations of (24), we only show the results for subset-size s= 3.

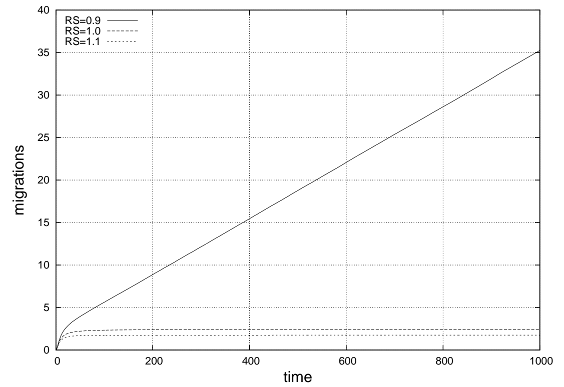
We measured scaling of the *Robin-Hood + Nottingham Sheriff* algorithm in terms of *ALOP* and the number of migrations, for networks of 100 (Figures 6.8 (a) and (b)) and 400 nodes(Figures 6.8 (c) and (d)). Even though in Section 6.1.4, a value of $RS = 0.9$ was promising, these plots show that the total number of migrations generated by this value makes the algorithm not scalable. Scalability in terms of migrations in Figures 6.8 (b) and (c) exists only for values of $RS \geq 1.0$. The optimal scalability, in terms of *ALOP*, in Figures 6.8 (a) and (c) exists for a value of $RS = 1.0$.

Considering that a 20×20 network can still be considered as a small network, we test the scalability in terms of *ALOP* and number of migrations over $n \times n$ P2P networks using $n = [10, 90]$, fixing the parameter RS in 1.0 and RB in 0.7. The results are shown in Figure 6.9.

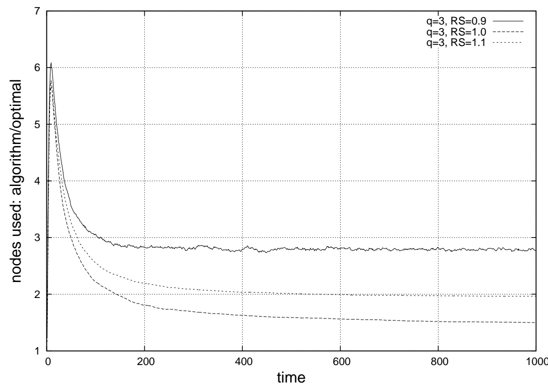
Note that at the beginning, the *Robin-Hood + Nottingham Sheriff* algorithm increases the number of nodes used, because active objects are first placed in a small subset of the network generating a high overload in this subset. Then, the algorithm quickly performs migrations to reduce the overload. Then, only the *work-stealing* step of *Robin-Hood + Nottingham Sheriff* algorithm works, clustering active-objects on the best nodes and thus, reducing the number of nodes used by the algorithm. Experiments report no overloaded nodes over 30 time-steps.



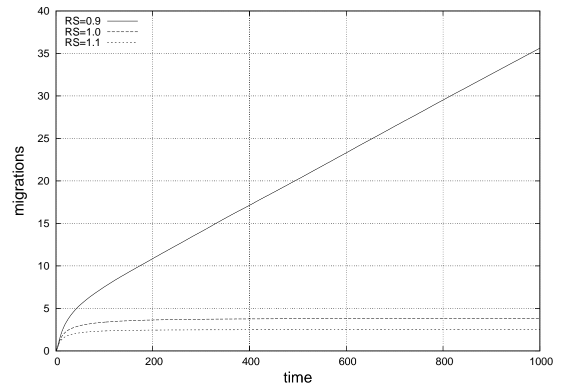
(a) ALOP for a 10x10 network



(b) Migrations for a 10x10 network



(c) ALOP for a 20x20 network



(d) Migrations for a 20x20 network

Figure 6.8: Scalability for a network using $RS = 0.9, 1.0, 1.1$, $RB = 0.7$

Figure 6.9 presents two behaviors at the same time:

1. **Number of nodes used by *Robin-Hood + Nottingham Sheriff* algorithm through time**, because the number of optimal nodes used by a static distribution (OPT) is constant for each

number of nodes ($n \times n$). We aim to cluster all active objects in a minimal set of nodes to avoid communication delays.

2. **ALOP ratio** (number of nodes used by *Robin-Hood* + *Nottingham Sheriff* algorithm versus number of nodes used by an optimal statical distribution *OPT*), evaluating "*how good*" are the minimal subsets found by the *Robin-Hood* + *Nottingham Sheriff* algorithm.

For networks of until 40×40 nodes, *Robin-Hood* algorithm uses less than two times the optimal number of nodes. In other words, the algorithm uses less than 20 nodes from all the network until $1,000$ time-steps. For networks of 50×50 to 70×70 nodes, the algorithm uses less than three times the number of optimal nodes (i.e: 27). For larger networks, the algorithm uses more than three times the optimal number of nodes at time-step $1,000$; nevertheless, the curves seem to decrease before that value.

We expected the previous behaviour, because the distribution of processing capacity μ follows a normal distribution; therefore, values of μ in the subset of the "best X nodes" will be higher for larger values of n (larger the network, smaller the subset size); and, because the *Robin-Hood* algorithm tries to use the nearest nodes while balancing an overloaded node. Therefore, as the network size increase, the probability of finding a node from the optimal subset decreases.

The plot in Figure 6.9 shows how at the first 10 time-steps the algorithm reacts against an overloaded situation, distributing the active objects among the network and then, when a stable state is reached, it begins the clustering of active objects. Similar behaviour can be seen in Figure 6.10, having a high number of *accumulated* migrations at the beginning and then the system becomes stable (for small-size networks) or there are some migrations in order to group the active objects on the "best processors" (large-size networks). Remember that plots present the *mean* number of *accumulated* migrations for m active objects; therefore, the contribution in plots of a each new migration is $1/m$.

For all studied network size, the curves remain under 6.5 migrations per active-object. Moreover, considering only the time-step $1,000$, we can see that the number of migrations is of order $O(\log(n))$. Both are promising results in terms of scalability of the *Robin-Hood* algorithm.

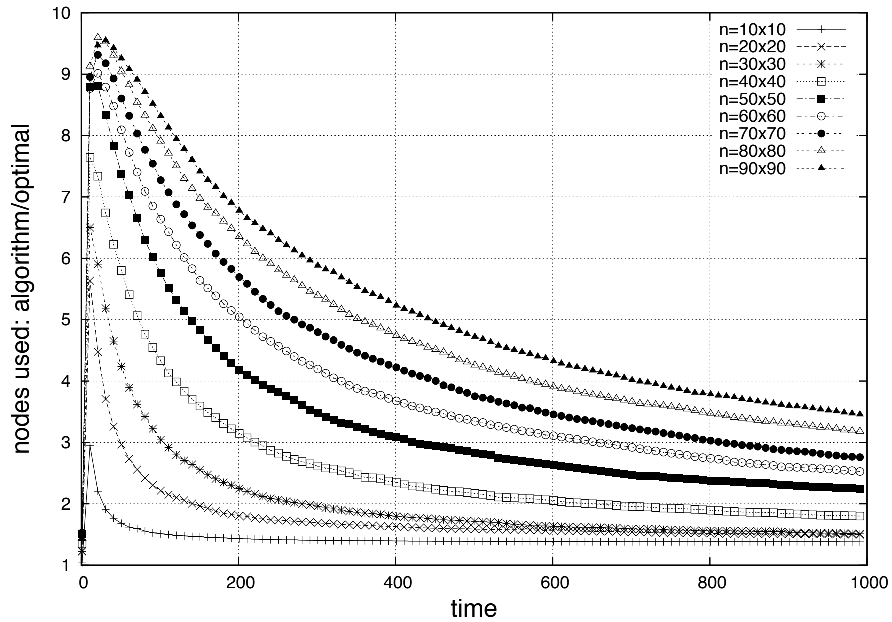


Figure 6.9: Scalability in terms of number of processors used, having $RS = 1.0$

Previous experiment was performed using a fixed number of active objects and an increasing number of nodes, in Figure 6.11 we study another interesting case: having the number of active objects proportional to the number of nodes, distributing uniformly at random sets of active objects on the network. Figure 6.11(a) presents the number of nodes used by the algorithm divided by the number of optimal nodes, noting that in this case the size of the optimal set increase compared to the number of nodes. The behaviour is similar to have the network divided in sub-networks, performing load-balancing only inside the sub-networks (sets of active-objects uniformly distributed at random produce a natural subdivision of the space). As a consequence of the previous behaviour, a constant number of migrations until a stable state is experimentally presented (Figure 6.11(b)).

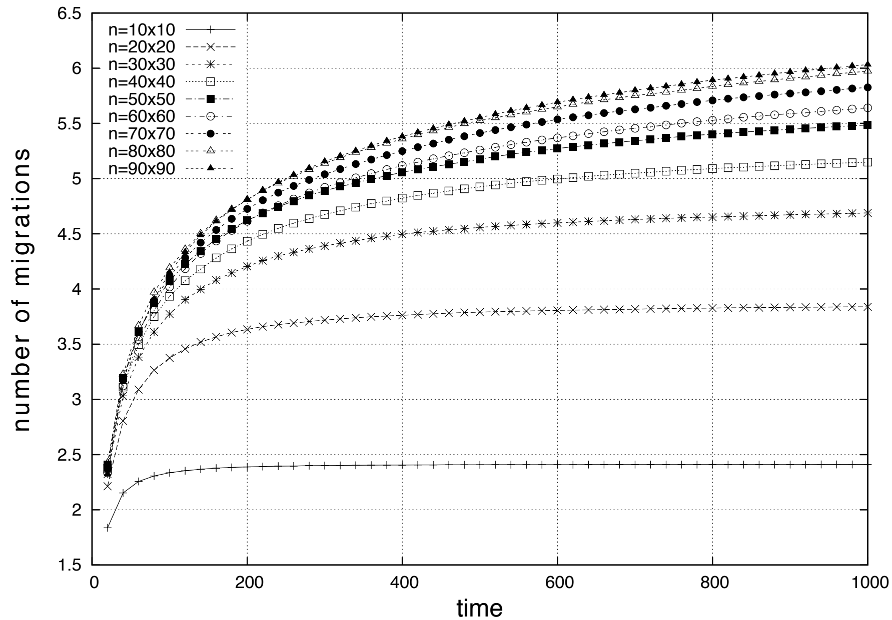
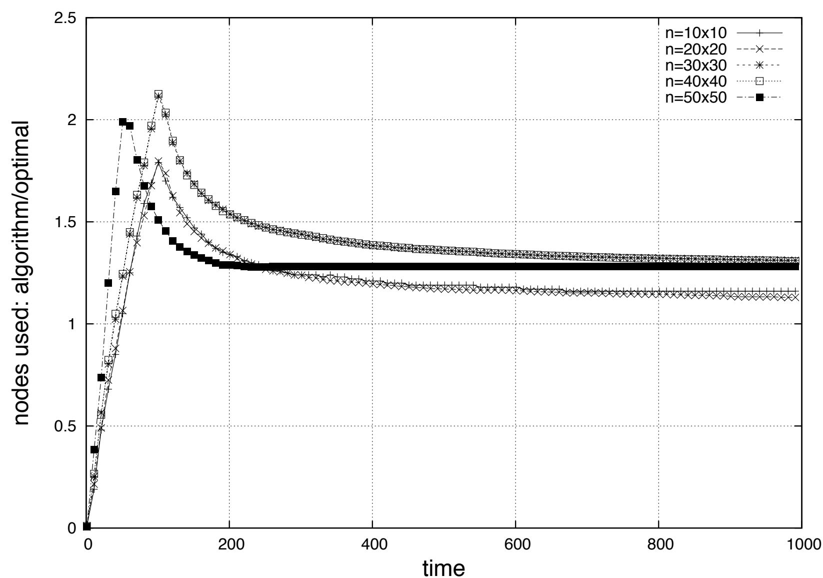


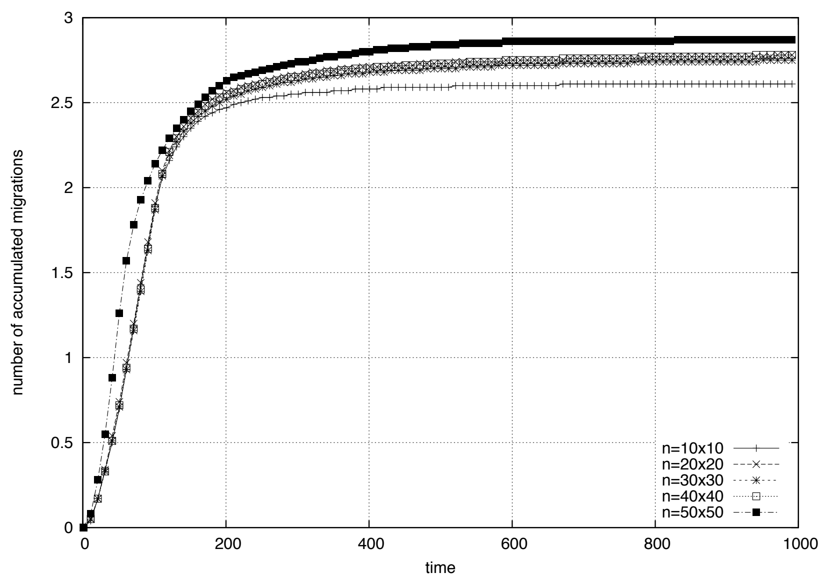
Figure 6.10: Scalability in terms of number of migrations, having $RS = 1.0$. The plot presents, for an active object, the (mean) number of accumulated migrations performed until a time-step $t \in [0; 1,000]$.

6.2 Simulating Project Grids

The grid computing paradigm (the Grid) promises to ease the sharing of heterogeneous resources, and their aggregation into truly global platforms, to be seamlessly used by multiple organisations and independent users alike (56). With the emerging Grid infrastructure starting to fulfil such ambitious promises (14), e.g., the CERN Large Hedron Collider Grid (LCG (112)) encompasses today more than 200 clusters and 40,000 processors at any time, multi-institutional projects are starting to run their applications in dynamically-created (virtual) environments. However, the achieved scale comes at a price: the resources dynamics require that the applications be equipped with environment-awareness, that is, the ability to adapt to the environment's layout and behaviour. In



(a) Number of processors divided by the optimum



(b) Accumulated migrations

Figure 6.11: Scalability, having the number of active objects proportional to the number of nodes

this section we focus on the environment-awareness problem.

The environment-awareness problem is broad; our approach treats the case of active-objects parallel applications (see Section 2.3) running in a multi-institutional project's virtual environment (project Grid, see Section 6.2.1). Our main contributions in this section are:

- A model for project Grids dedicated to running active-objects-based applications, derived from a set of traces and applications coming from a multi-institutional project, namely the ProActive PlugTest (ETSI and INRIA) (Section 6.2.1). To the best of the authors knowledge, ours is the first approach to identify the characteristics of such a project Grid, with specific new insights in the inter-resource communication latency;
- Two environment-aware load-balancing algorithms dedicated to active-objects-based applications, based on a generic concept of clustered resources. Our notion of clustered resources should not to be confused with the notion of physical clusters of resources. Our approach generalises previous cluster-aware load-balancing results, such as the one by van Nieuwpoort *et al.* (117), where clusters must be manually and, most importantly, statically defined. The algorithms are validated experimentally through simulation, and shown to offer better performance when compared to traditional, non-environment-aware, algorithms (Section 6.2.5).

For the Grid case, the environment where the active objects run is usually composed from multiple clusters of resources, e.g., a set of monitor-less machines inter-connected by a high-speed local network. In this case, the load balancing procedure must take into consideration the inter-cluster vs. intra-cluster communication costs, for optimal performance (117). In ProActive, the active objects form a P2P network; the load-balancing algorithm should also take into consideration the topology of this network. Note that for ProActive applications latency is a key performance estimator.

6.2.1 Characterising a Project Grid

The ProActive PlugTests project grid (ETSI and INRIA) is used normally as an environment for the *n*-queens competition: participants program using the ProActive library an application that must solve the largest possible instance of the *n*-queens problem. The infrastructure is provided by the organisers, by several research institutions that use ProActive, and by some of the participants.

We have obtained information pertaining to the 2005 version of the ProActive PlugTests: the characteristics of the resources shared by each participating institution, and the communication latency between each two resources in the project grid. The latency information was obtained as follows. Two sources, one located within the INRIA Sophia-Antipolis Network in France (INRIA), and one located at the Computer Science Department of the University of Chile (DCC), sent 100 ping messages to each participating resource and discarding outliers. The average observed latencies were selected as the representative of the distance between the sources and the participating clusters.

Table 6.2 depicts the characteristics of the PlugTests project grid. The *project leader* provides the FRANCE G5K cluster, which is by far dominating the project grid, by size. The CHINA *contributing institution* offers the best per-node performance. The NETHERLANDS *contributing institution* dedicates 20 of its 72 nodes to this project grid. Several institutions participate with shared resources to the project grid, that is, their resources can also be used by users external to the project, therefore making the actual contribution size variable. For instance, measured real Mflops/node of China contributing institution was around 90 instead of the theoretical 569.92.

Figure 6.12 depicts graphically the latency information shown in Table 6.2. Given the latency values, we define four classes of nodes inter-location:

Close: This class represents nodes located within the same network, with a ping time of around 2.5 ms;

Near: This class represents nodes that are near geographically, with a ping time between 10 and

Table 6.2: Summary of the PlugTests project grid characteristics. The acronyms D and S represent dedicated and shared resources, respectively.

Country	# Nodes	*Mflops*	$\frac{Mflops}{node}$	d(INRIA)	d(DCC)	Type
AUSTRALIA	13	1,658	127.54	394	329	D
BRAZIL	8	2,464	308.00	268	60	D
CHILE I	26	2,917	112.19	299	2.1	D
CHILE II	30	5,103	170.1	388	17.5	S
CHINA	184	104,865	569.92	287	392	S
FRANCE G5K	822	278,647	338.99	2.1	299	D
FRANCE	162	48,298	298.14	2.1	301	S
GREECE	16	4,125	257.81	168	464	D
IRELAND	14	2,147	153.36	42.3	308	S
ITALY I	25	3,465	138.60	58.5	314	D
ITALY II	33	2,385	72.27	39.7	298	D
NETHERLANDS	20	1,346	67.3	32.2	284	D
NORWAY	22	2,328	105.82	51.7	302.67	D
SWITZERLAND	46	3,918	85.17	29.14	288.7	S
U.S.A	22	3,179	144.5	169.1	134.3	D

70 ms;

Far: This class represents nodes located in clusters situated on different continents, with a ping time around 150 ms;

Very Far: This class represents nodes that are poorly connected (for a ProActive application), with a ping time over 250 ms.

The results show that for project grids with resources obtained from institutional partners, there clusters of closely-connected resources are a majority. This contrasts with the situation observed for freely-contributing partners in large P2P networks (68), with which the ProActive applications share the application topology model.

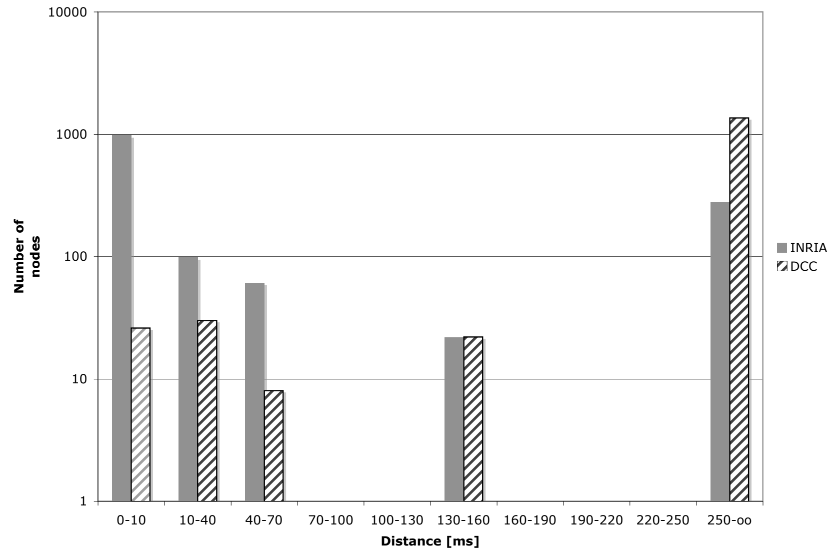


Figure 6.12: Latency between nodes from the PlugTest project grid.

6.2.2 Modelling a Project Grid

We describe in the following paragraphs a constructive procedure for modelling a project grid as a modification of the ProActive's Infrastructure Algorithm presented in Section 3.2 and simulated in Section 6.1.

1. Considering again a discrete representation of the Euclidean space in which the resources are physically located, randomly choose a set of "institutions" by assigning to them random locations (or known locations, if the topology is fixed in advance). For modelling the ProActive PlugTest environment, we have selected a 40×40 matrix, and 10 institutions.

2. The institutions are used as first contacts, and all links created to them receive a distance of

1.

3. Connect resources belonging to the same cluster, and mark all the newly created links with a distance of 1; all resources within a cluster can connect to each other at the lowest (local) cost.

4. Inter-connect resources from different clusters; the distance between nodes from two different clusters is Euclidean[1]. If a resource belongs to several clusters, e.g., because the clustering method is not a one-to-one mapping, randomly assign the resource to one cluster from its belonging-to set of clusters. For ProActive PlugTests, we have assigned the inter-cluster communication latencies extracted from the traces (see Section 6.2.1).

5. For each resource, select a processing capacity corresponding to your model. For our data, we have chose a processing capacity (denoted by μ) from a uniform distribution $U[50,150]$ for each cluster representing a *contributing institution*, and assigned a value of $\mu_i \pm \varepsilon, \varepsilon \in [0,1]$ to all processors in that cluster. We have assigned to the cluster representing the *project leader* (the $FRANCEG5K$ cluster in our data) a capacity of $\mu = 350 \pm \varepsilon$.

As shown by the PlugTests experience, even though an "inter-continental" project grid seems to be a good idea to solve parallel problems using as many resources as possible, regardless of their geographical location, the notion of location has to be exploited by the application to achieve optimal performance.

We define a load balancing algorithm as *environment-aware* if it uses information about the relative distance between two resources to select a destination for its load balancing process. In this work we focus only on latency as a location (or distance) estimator. This decision is based on the fact that latency is a very good distance estimator (65). Since the resources considered in this work typically come from institutional clusters, we use from hereon the terms environment-aware and cluster-aware interchangeably.

[1] $d(\{x_1,y_1\},\{x_2,y_2\}) = |x_1 - x_2| + |y_1 - y_2|$

6.2.3 Environment-aware Algorithms

Environment-aware Robin-Hood (also known as cluster-aware Robin-Hood, or **crh**) corresponds to the environment-aware version of the pure Robin-Hood algorithm presented in Section 5.4.2 (labelled as **rh**).The Robin-Hood algorithm exploits the ProActive's P2P infrastructure to perform efficient load balancing, using a minimal subset of neighbours (commonly on Robin-Hood algorithm a value of $n = 3$ is used). The environment-aware Robin-Hood exploits also the distance knowledge of neighbours to perform efficient load balancing through Project Grids. Cluster-aware Robin-Hood works as follows.

On an overloaded node:

1. Define `Ngb` as list of neighbours, `Nls` the neighbours list size, `Dist` a table with distances to the neighbours and `n`, number of neighbours to use for load-balancing.
2. Sort `Ngb` by (dynamic) distance, ascending (environment-awareness)
3. Every time-step, choose a neighbour N_i from `Ngb[1,Nls]` at random with probability `Dist`$[i]^{-2}$ (71):
   (a) if `isNotLoaded(` N_i `) and Rank(`N_i`)` $= 0.7\times$ `Rank(`*self*`)`, send work unit to N_i.
   (b) exit after n tries.

Environment-aware Work Stealing (also known as cluster-aware Work-Stealing, or **cws**) is a receiver-initiated and ranked algorithm, which corresponds to the environment-aware version of the *Nottingham Sheriff* step presented in Section 5.5 (labelled as **ws**). Nottingham Sheriff step exploits the ProActive's P2P infrastructure to perform short-distance work-stealing, using a minimal subset of neighbours (commonly on Nottingham-Sheriff step a value of $n = 1$ is used). The environment-aware Work Stealing exploits also the distance knowledge of neighbours to perform efficient and trusty short-distance work-stealing through Project Grids. Cluster-aware Work-Stealing works as follow:

1. Define `Ngb` as list of neighbours, `Nls` the neighbours list size,`Dist` a table with distances to the neighbours and `n`, number of neighbours to use for work-stealing.

2. Sort `Ngb` by (dynamic) distance, ascending (environment-awareness)

3. Every time-step, choose a neighbour N_i from `Ngb[1,Nls]` at random with probability `Dist[`i`]`$^{-2}$ (71):

   (a) if `Rank(`N_i`)` $<$ `Rank(`*self*`)`, steal work unit from N_i.

   (b) exit after n tries.

6.2.4 Experimental Setup

We have built a simulator based on the model described in Section 6.2.2.

Similarly to Section 6.1, we modelled active objects as queues, adding this time the capabilities to put active objects in *wait* state, and to introspect the queues. Using the introspection we have added to the application model synchronisation features, communication costs to remote objects, and migration costs.

The setup of each experiment was as follows. We randomly choose a cluster and, using the "institution" neighbour list, we simulate the deployment of 100 active objects with an arrival rate $\lambda = 10$. The arrival process ends after 1,000 time-steps. We group the active objects in 10 sets (the ith active objects belongs to the set $\lfloor \frac{i}{10} \rfloor$) and we define that the $10th$ request is a message to the remote objects on the same set.

Defining a parameter C as the message size, the tenth request has a size of

$$round(\sum C \times d(n_i, n_j)) \; \forall \frac{i}{10} = \frac{j}{10}, i \neq j \; services$$

Defining the cost of the communication between active objects at the same node is zero.

We define M as the active object size; then, the migration cost from a node i to a node j is

$$round(M \times d(n_i, n_j))$$

To simulate previously reported idleness of resources (Litzkow, Livny and Mutka (86) reported that desktop processors are idle 80% of the time; this value is reported up to 90% in 2005 (45)), we randomly choose 25% of the clusters in the Grid, which represent the "Desktop Laboratory" components of the Grid, and on the clusters we randomly choose $\lfloor 10\% \rfloor$ of their processors. Then, at each processor we generate the same number of services than processing capacity (μ) on each time-step.

6.2.5 Simulation Results

Our first goal is the analysis of the influence of remote communication and migration on the performance of algorithms. To this end, we test each step of the algorithm alone (**rh**,**crh**, **ws** and **cws**) and combined (**rh-ws**, **crh-ws**, **crh-cws** and **rh-cws**). Each simulation was performed 100 times; to avoid noise produced by saturated queues, only the experiences in which at the end of 1,000 time-steps there are no overloaded nodes are considered for the reported values.

Not-Synchronised Parallel Applications

Figures 6.13 and 6.14 depict the observed mean number of pending requests in all active objects for low and high message and object size, respectively. Initially queues are long and and the load-balancing algorithm try to distribute them until reaching a minimum (considering that the communication cost is measured in number of services) in a stable state before 1,000 time-steps, note that even the communication cost increase the queue length, when the stable state is reached active objects are placed in processors capable to process all the queue during the time step. The values lead to the conclusion that, for non-synchronised parallel applications based on active objects:

- A Work-Stealing algorithm without cluster-awareness is useless in project grids even if it targets its load balancing at close neighbours (28). Due to migration, two active objects could go from close neighbours to intercontinental distances quickly, for a penalty of over 100 ms of communication latency.
- Robin-Hood algorithm tries to select a near underloaded node to perform the balance using probabilities, but Work-Stealing algorithm tries to perform "stealing" of a near underloaded node using network properties. For this reason, increasing the message size and the object size (from Figure 6.13 to Figure 6.14), we observe that algorithms performing non Environment-Aware Work Stealing have the worst performance. The only exception to this rule is the combination **crh-ws**, which exploits the quick reaction against overloading of the Robin-Hood algorithm (here, active objects are distributed inside the cluster before an external node performs its stealing).
- For high size messages and objects, the best performance is achieved by **rh-cws** and **cws** algorithms. The reason for the former is that this algorithm aims to balance active objects on near nodes only while overloading occurs; for the latter, the reason is that a steal inside a cluster reduces the migration time, and the algorithm aims to equalise the load of the cluster.

Synchronised parallel applications

We now focus on the behaviour of synchronised parallel applications (7),e.g., *Single Program - Multiple Data* (SPMD). We model the synchronisation requirements of an application as follows: every 10 time-steps, a synchronisation request is enqueued in each active object. When the request is served, actives objects switch to a *wait state* until all objects in their set have served the synchronisation request, then all objects in that set can continue serving requests. Figure 6.15 presents the results of the simulation under this new constraint. Note that, as in previous section, even though the communication cost increase the queue length, when the stable state is reached active objects are placed in processors capable to process all the queue during the time step.

Note that a bad decision (as to use non-environment-awareness work stealing) could produce 10 times more work than a good decision. The best performance is achieved by algorithms using

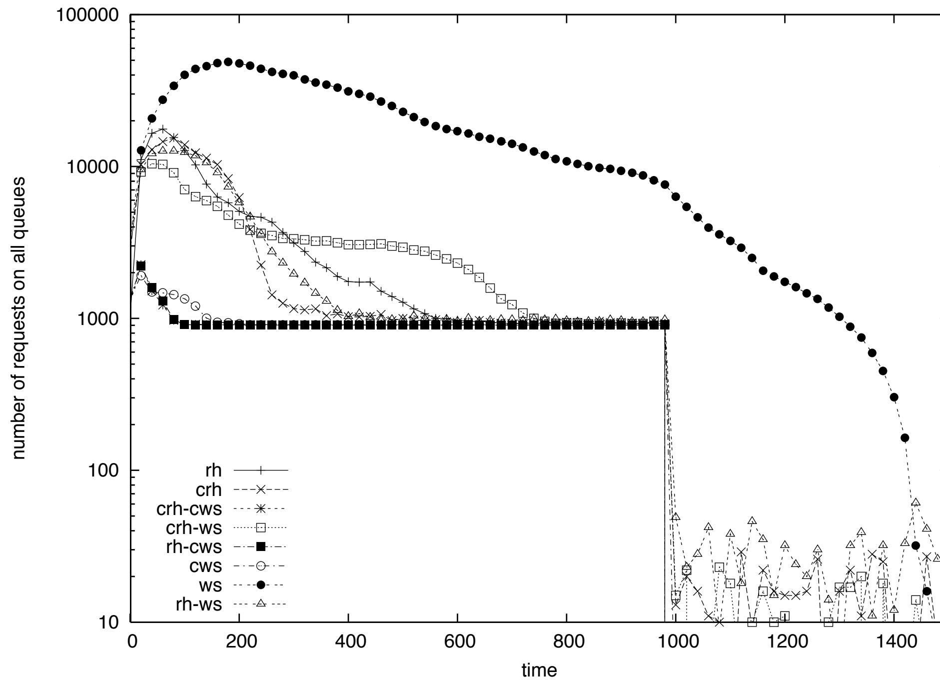


Figure 6.13: Total number of pending requests in all active-objects using message-size $C = 0.1$ and object size $M = 1$, without synchronisation.

Environment-Awareness Work Stealing, because they aim to distribute active objects on the same cluster.

6.2.6 Results Confidence

We define the *results confidence* as the percent of simulation cases, from the total simulation tries, in which there are no overloaded queues after $1,000$ simulation time-steps. These results have not been included in the evaluation described in Section 6.2.5. The higher the results confidence value, the better the presented results describe the true capabilities of the evaluated algorithm.

We measured the results confidence using a message size of 0.1 (to compare results with Fig-

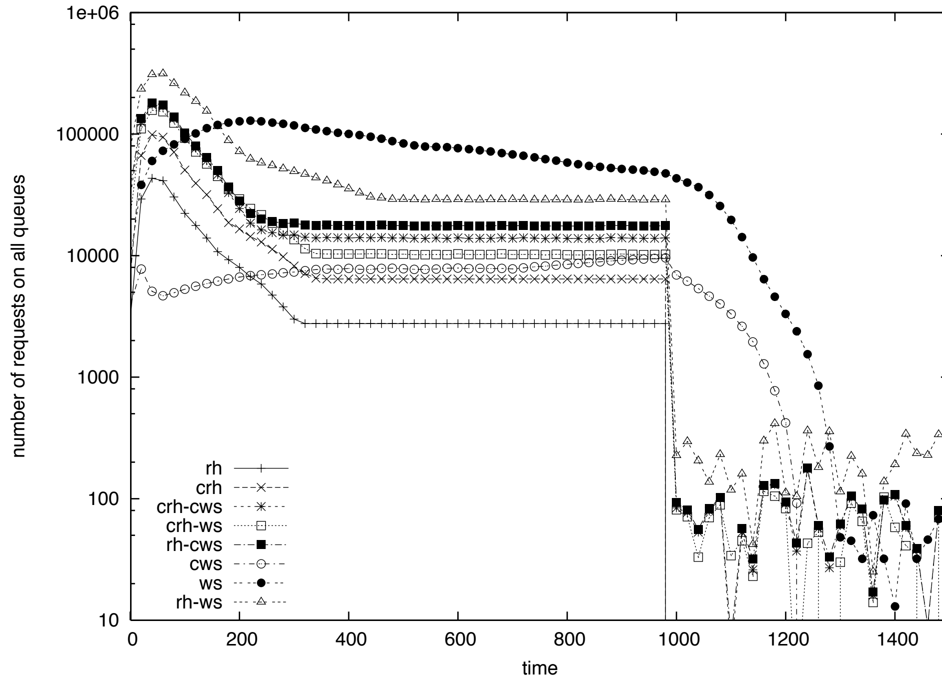


Figure 6.14: Total number of pending requests in all active-objects using message-size $C = 1$ and object size $M = 10$, without synchronisation.

ures 6.13 and 6.15), object sizes 1, 10 and 100 services, with and without synchronisation (Figure 6.16) . From these results we conclude that:

- Most of the time a given cluster was not suitable to process the parallel application. Therefore, algorithms performing only *environment-awareness* steps have low level of confidence.
- Environment-Awareness Robin-Hood algorithm, used alone and in conjunction with both flavours of Work-Stealing, presents a poor performance for Institutional Grids, the search for a "good" partner, inside the cluster only, to send *one* active object every time-step produces bottlenecks and node overloading. Note that to search a good pattern to send and active object may produce less migrations in a given time-step that a set of underloaded nodes stealing from the overloaded node.
- As we noted at Sections 6.2.5 and 6.2.5, to use a work-stealing algorithm without environment-

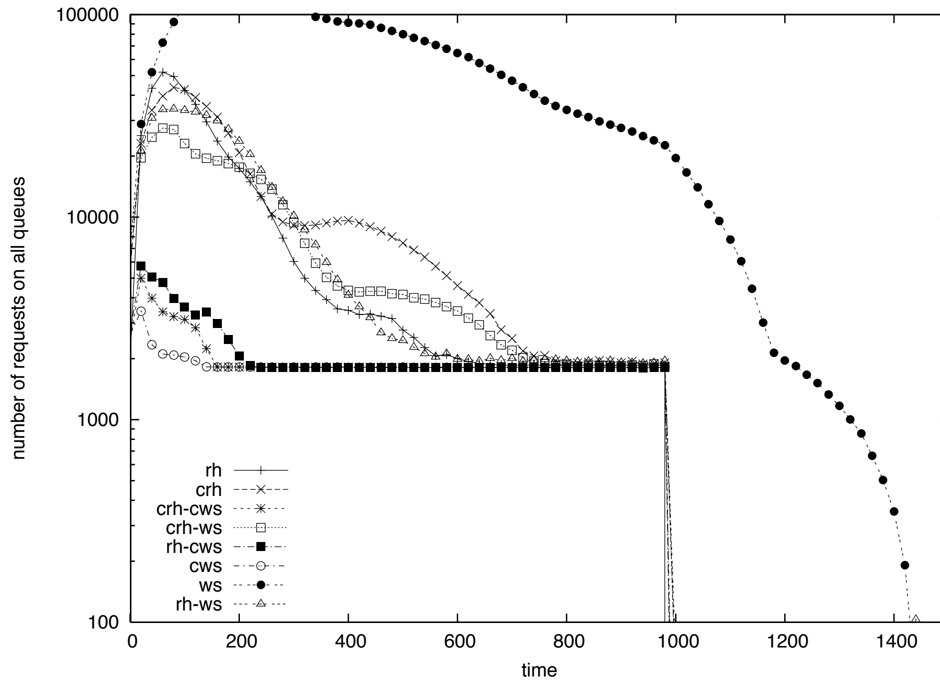
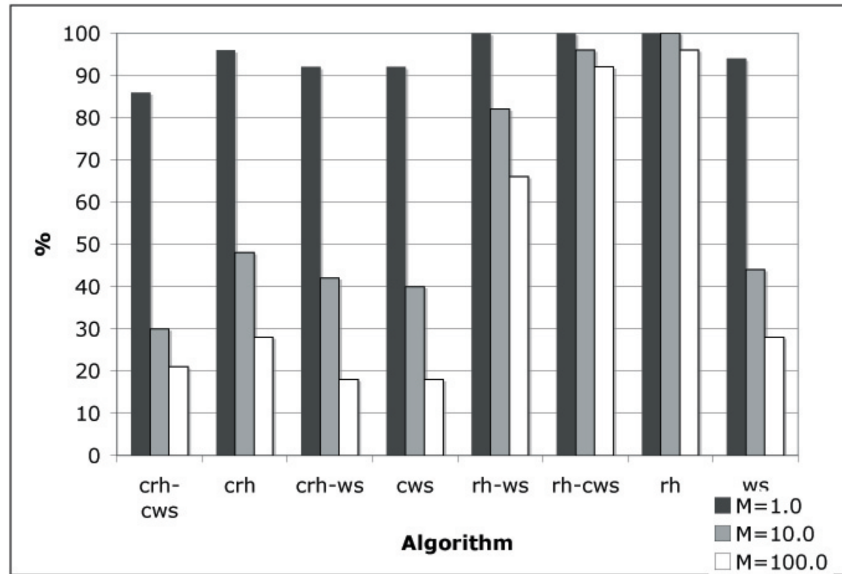


Figure 6.15: Total number of pending requests in all active-objects using message-size $C = 0.1$ *services*, object size $M = 1$ *services* and synchronisation each 10 *time-steps*.

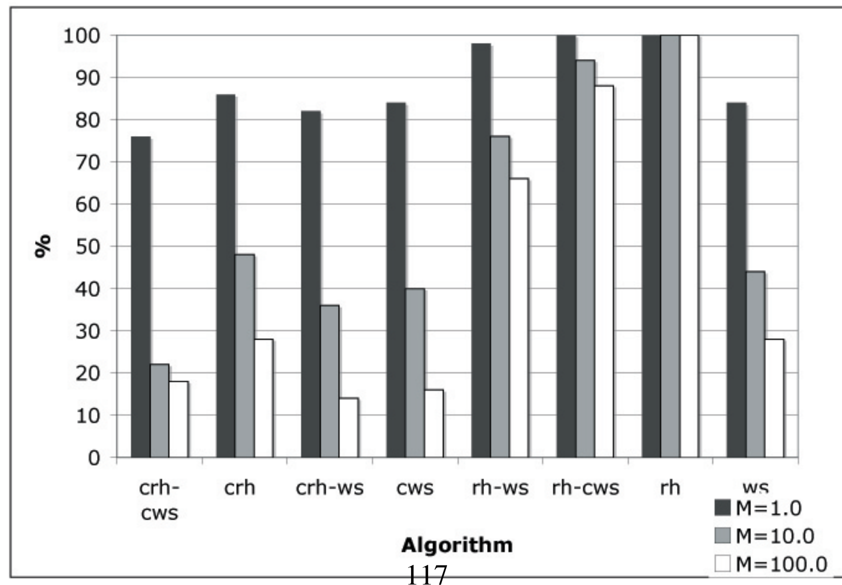
awareness behaves badly, in Figure 6.16 we also note that greater the object size lower its confidence. That was a expected result because the migration cost is strictly dependent on the distance.

- The best confidence is achieved for the symmetrically initiated *Robin-Hood + Environment-Awareness Work Stealing* algorithm. Because the *Robin-Hood* step will distribute active object among near nodes (*neighbouring clusters* in this case) and the *Environment-Awareness Work Stealing* step will distribute active objects on the cluster maintaining the distance property.

Last conclusion is very important, because it can be applied also to Desktop Grids (Section 6.1). Therefore, our Load Balancing algorithm is able to perform efficient balance in both Project and Desktop Grids.



(a) Without synchronisation



(b) With synchronisation

Figure 6.16: % of confidence of load-balancing algorithms, increasing object size (M)

6.3 Where to run parallel applications?

In previous section we noted the confidence of some combinations of algorithms was low because a given cluster was not suitable to process the given application. The problem of finding a suitable Grid infrastructure for an application can be seen as a problem of classified advertisements and matchmaking (97) or a problem of database search like UDDI web services (50). ProActive provides the mechanism for, given a known set of processors (clustered or not), define where and how to deploy a parallel application without code modification (10).

We propose coupling based on contracts as a mechanism to address the problem of exchanging information in a generic way between unfamiliar parties. We aim to couple the deployment of an unfamiliar application with an unfamiliar Grid infrastructure descriptor using ProActive, deploying an application on a Grid infrastructure without modifying or inspecting either.

Nevertheless, unfamiliar parties cannot exchange information with each other in a generic way. A group of typed clauses will then form an interface that will specify *what* information is required and provided by each party. The coupling of the interfaces will yield a contract, that will allow the parts to couple and work together on a common goal. The semantic definition of typed clauses was presented in the work of Mario Leyton *et al.* (26), in this thesis we show how to use typed clauses on ProActive's deployment descriptors to achieve efficient deployment.

6.3.1 Problematic of Applications and Descriptors

In the traditional approach, the application developer and the descriptor developers need to have a previous agreement on the name of the *Virtual Node* (the abstraction of the Grid nodes). This means that the name of the Virtual Node is hardcoded inside the application and the descriptor. If the application wants to use a new descriptor, then either the descriptor or the application has to be modified to agree on the new Virtual Node name.

A possible solution to this problem is passing the Virtual Node name as a parameter to the application. Nevertheless, the problem of figuring out the proper Virtual Node name from the descriptor remains. To find out the name of the Virtual Node, inspection of the descriptor has to be performed, which can be a problem for someone alien with respect to the Grid infrastructure's descriptor.

Furthermore, the Virtual Node name is not the only information sharing problem that the application and descriptor have. For example, a descriptor might be configured to deploy on k nodes, but the application only requires j nodes ($j < k$). Without shared clauses, the descriptor has to be modified to comply with the requirements of the application.

Modifying the application or the descriptor can be a painful task, specially if we consider that the person deploying the application (deployer) may not be the author of either. To complicate things further, the application source may not even be available for inspecting the requirements and performing modifications.

6.3.2 Clauses in ProActive Descriptors

Clauses can be specified using XML tags as shown in the example of Figure 6.17 for the descriptor. To define the clauses, a new section labeled `clauses` has been added at the beginning of the descriptor to hold the `interfaces`. The clauses shown in the example correspond to:

`PROACTIVE_HOME & MAX_NODES` Correspond to descriptor set clauses. The value is set directly in the descriptor, and can be used later on, inside the descriptor or the application.

`VIRTUAL_NODE_NAME` Corresponds to a clause that the descriptor enforces the application to set. If the application doest not set this value, the clause inside the coupling contract will not be valid, and the application will not be allowed to couple with the descriptor. In the example, we force the application to set the name of the Virtual Node.

`LOAD_BALANCING` Corresponds to a clause that the application has set, but the descriptor can

```
<clauses>
  <interface name="descriptor-example-interface">
    <Descriptor name="PROACTIVE_HOME" value="ProActive/"/>
    <Descriptor name="MAX_NODES" value="100/"/>
    <Application name="VIRTUAL_NODE_NAME" value=""/>
    <DescriptorPriority name="LOAD_BALANCING" value="on"/>
    <ApplicationPriority name="NUMBER_OF_NODES" value="1">
      <!--// (NUMBER_OF_NODES>0) && NUMBER_OF_NODES<=MAX_NODES -->
      <integerConstraint>
        <and>
          <biggerThan>0</biggerThan>
          <smallerOrEqualThan>${MAX_NODES}</smallerOrEqualThan>
        </and>
      </integerConstraint>
    </ApplicationPriority>
    <JavaProperty name="USER_NAME" value="user.name"/>
  <interface>
</clauses>
...
  <virtualNodesDefinition>
   <virtualNode name="${VIRTUAL_NODE_NAME}"/>
  </virtualNodesDefinition>
...
<sshProcess class="org.objectweb.proactive.core.process.SSHProcess"
      hostname="example.host" username="${USER_NAME}"/>
```

Figure 6.17: Example of clauses in descriptor.

override. In the example, we imagine that an application is capable of handling, or not, the load balancing. By default the application will assume that no load balancing is provided by the Grid infrastructure (Figure 6.18), and thus handle the load balancing at the application level. Nevertheless, the descriptor is aware if load balancing can be done at the Grid infrastructure level and activate it. The application can then access the contract's clauses to learn if the infrastructure is using the load balancing and disable the application load balancing mechanism.

NUMBER_OF_NODES Corresponds to a clause that the descriptor has set a value, but the application may override. Additionally, the descriptor has set constraints indicating that the value must be an integer between 1 and MAX_NODES.

USER_NAME Corresponds to a clause that is set from the environment. In this case, the username can be specified from the environment as a java property.

Figure 6.17 also shows an example of how the clauses can be used inside descriptors. Note that the value of the clause `VIRTUAL_NODE_NAME` has not been set in the descriptor, since it is of type Application. This means that the value used inside the descriptor will be the one set from the application. Also note, that clauses obtained from the environment can also be used, like the `USER_NAME` clause.

6.3.3 Clauses in ProActive Applications

We have also provided a mechanism for specifying clauses and interfaces from the application. This can be done through an API, or loading the clauses from an external XML file. Since the XML approach has already been shown for the descriptor, Figure 6.18 shows an example using the API. First an *interface* is created, and then the clauses are added to the interface. The interface is then passed as a parameter when parsing the descriptor. The parsing will try to generate a coupling contract using the application's and the descriptor's interfaces.

If the application can be coupled with the descriptor, then the application can retrieve the coupling contract and consult the contract's clauses. For example, using this strategy the application can know if the descriptor activated the infrastructure load balancing, and avoid using the application load balancing.

6.3.4 Constraints

Constraints are boolean expressions that will be evaluated for each clause when the contract is built. The constraints operate on two types: *integer* or *string*. For each constraint the logical operators: *and*, *or*, *xor* are allowed. Also, boolean operators are provided for each type of constraint. The integer operators are: `biggerThan`, `biggerOrEqualThan`, `smallerThan`, `smallerOrEqualThan`, `equals`. The string case sensitive operators are: `subString`, `superString`, `equals`. Figure 6.19 shows the constraint grammar specified using XML Schema(W3C) for the integer type constraints.

```
//Create a new interface
ClausesInterface ci= new ClausesInterface("application-example-interface");

//Set the clauses in this interface
  //set(<type>, <clause name>, <value>, [<constraint>])
ci.set(Application, "VIRTUAL_NODE_NAME", "testnode",);
ci.set(ApplicationPriority, "NUMBER_OF_VIRTUAL_NODES", "16");

// LOADBALANCE="on" || LOADBALANCE="off"
OrConstraint oc = new OrConstraint();
oc.add(new EqualsConstraint("on"));
oc.add(new EqualsConstraint("off"));
ci.set(DescriptorPriority, "LOAD_BALANCING", "off", new StringConstraint(oc));

//Parse and load the descriptor using the coupling interface. If the application and descriptor can
    not be coupled an exception will be thrown
ProActiveDescriptor pad = ProActive.getProactiveDescriptor("descriptor.xml", ci);

//Clauses from the coupling contract can be used in the application
CouplingContract cc = pad.getCouplingContract();
String loadBalancing = cc.getValue("LOAD_BALANCING");

//The application can take decisions based on the clauses
if(loadBalancing.equals("on")){...}
else{...}
```

Figure 6.18: Example of clauses in application.

```
<xs:element name="integerConstraint">
  <xs:complexType>
    <xs:choice>
      <xs:element name="and" type="intConst"/>
      <xs:element name="or" type="intConst"/>
      <xs:element name="xor" type="intConst"/>
    </xs:choice>
  </xs:complexType>
</xs:element>
</xs:complexType>
  <xs:complexType name="intConst">
  <xs:choice minOccurs="1" maxOccurs="unbounded">
    <xs:element name="and" type="intConst"/>
    <xs:element name="or" type="intConst"/>
    <xs:element name="xor" type="intConst"/>
    <xs:element name="biggerThan" type="xs:string"/>
    <xs:element name="biggerOrEqualThan" type="xs:string"/>
    <xs:element name="smallerThan" type="xs:string"/>
    <xs:element name="smallerOrEqualThan" type="xs:string"/>
    <xs:element name="equals" type="xs:string"/>
  </xs:choice>
</xs:complexType>
```

Figure 6.19: Integer Constraint Schema Grammar.

Figure 6.17 shows an example where the clause `NUMBER_OF_NODES` is constrained to be: 0 < `NUMBER_OF_NODES` <= `MAX_NODES`. Note that `MAX_NODES` is defined as a `Descriptor` type clause. Figure 6.18 shows an example using string constraints. The clause `LOAD_BALANCING` is constrained to be either `on` or `off`.

Using the proposed coupling approach of (26), we have shown how coupling contracts can be applied for automated deployment of unfamiliar applications on alien Grids. For this, we have provided mechanisms to specify clauses in the application and the deployment descriptor using the Grid middleware ProActive. As a result, the approach can now be used to couple applications with descriptors, without having to modify or inspect either.

6.4 The real world

To finalise our work, we tested the performance of our algorithm on *Grid5000* network. Grid5000 is a French project which aims to interconnect by a high-speed network a set of $5,000$ processors distributed over the French territory in Institutional Clusters (see Figure 6.20 [2]). Nowadays Grid5000 has around 900 fully operative nodes (see Table 6.3), all of them shared by French scientific world.

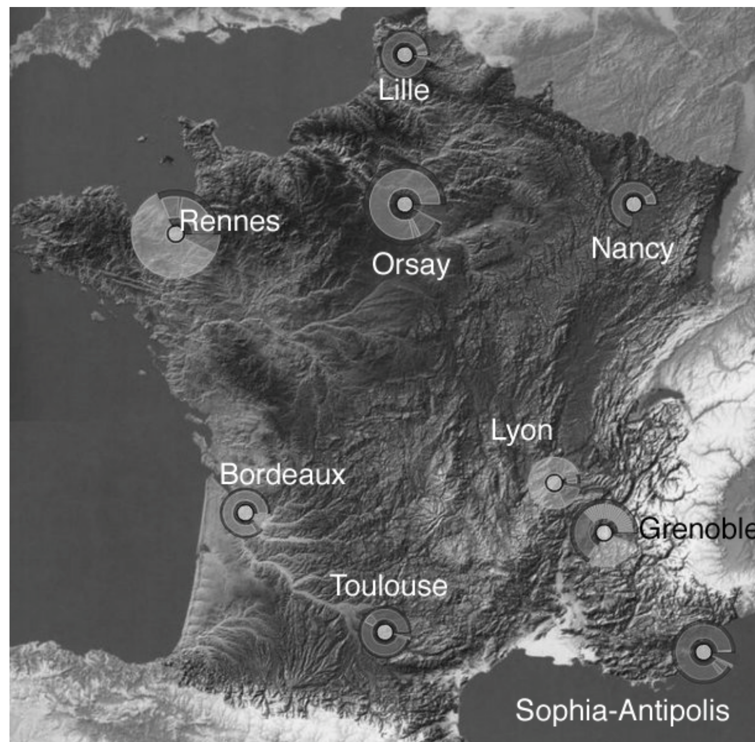


Figure 6.20: Institutional clusters on Grid5000: Bordeaux, Grenoble, Lille, Lyon, Nancy, Orsay, Rennes, Sophia-Antipolis and Toulouse.

We checked the *Robin-Hood + environment-aware work-stealing* algorithm in terms of migration decisions and scalability using the Jacobi parallel calculus defined in Section 5.6. In this

[2]Map from `https://www.grid5000.fr/mediawiki/images/thumb/612px-G5K_geographical_map.jpg`

Table 6.3: Clusters of AMD Opteron processors from Grid5000: on the left the number of nodes by cluster, on the right *Gflops* per processor clock

			Clock Freq (GHz)					
Site	CPUs per node	N_Nodes	1.0	1.8	2.0	2.2	2.4	2.6
Bordeaux	2	48	-	-	-	3.542	-	-
Lille	2	53	1.718	3.022	3.336	3.647	-	-
Lille II	2	15	-	-	-	-	-	4.3
Lyon (capricorne)	2	56	-	-	3.254	-	-	-
Lyon (sagittaire)	2	70	-	-	-	-	3.865	-
Nancy (grillon)	2	47	1.737	3.057	3.379	-	-	-
Orsay (gdx)	2	216	-	-	3.388	-	-	-
Orsay (gdx II)	2	126	1.730	3.065	3.385	3.720	4.040	-
Rennes (paravent)	2	99	1.737	3.059	3.364	-	-	-
Rennes (parasol)	2	64	-	-	-	3.573	-	-
Sophia (azur)	2	105	-	-	3.258	-	-	-
Sophia (helios)	4	56	-	-	-	3.675	-	-
Toulouse	2	58	-	-	-	3.586	-	-

Note: Grenoble has 206 Intel itanium 2 and 64 Intel Xeon IA32
(Source: https://www.grid5000.fr/mediawiki/index.php/Processor#AMD_Opteron)

experiment, we used 100, 200 and 300 Grid5000 nodes to deploy our application and we measured the *application speed* in number of iterations per milliseconds (see Figure 6.21), and mean number of cumulated migrations that an active objects performs (see Figure 6.22); both measurements until $1,000$ Jacobi iterations.

As we expected by our simulations, after an initial redistribution as a reaction against overloading, the mean number of migrations remains almost constant, and similar values are observed by different number of nodes. This result validates our scalability hypothesis of the behaviour of our algorithm, using the nearest nodes where the parallel application fits, regardless of the network size. On the other hand, an increase in the number of nodes used (in this case, more nodes per cluster reserved) produced an increase on application speed compared to the initial distribution. This result validates our hypothesis of clustering of active objects on the best processors.

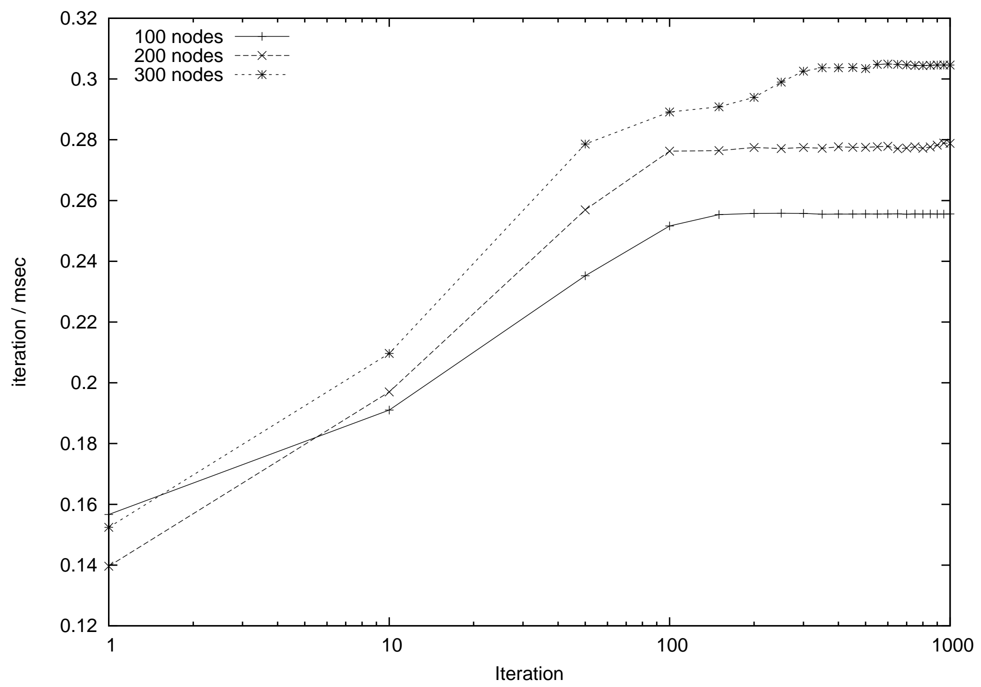


Figure 6.21: Speed of Jacobi parallel application in iterations per milliseconds.

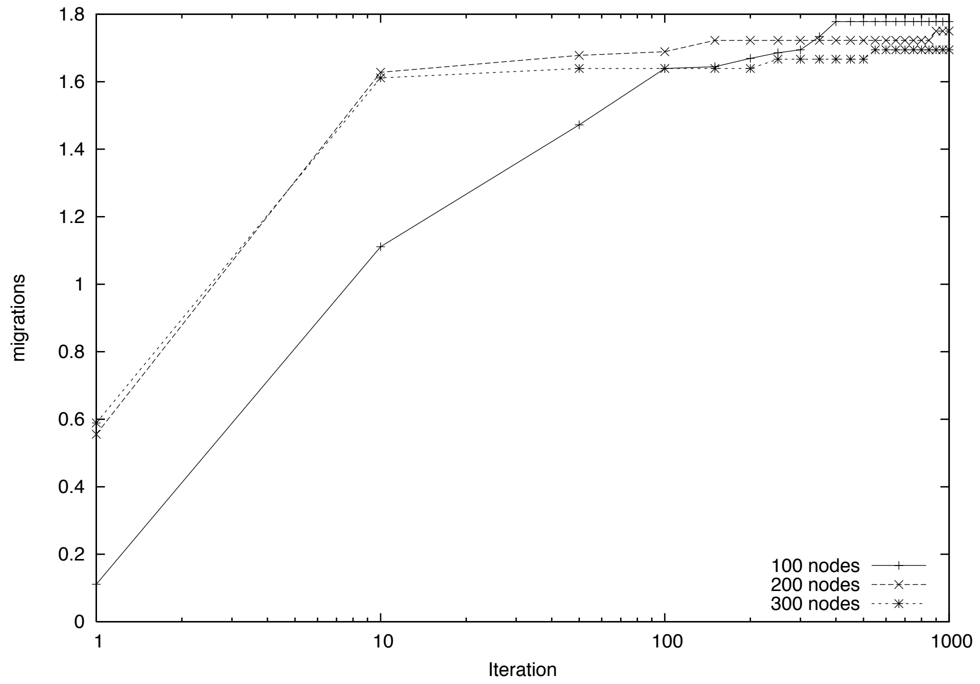


Figure 6.22: Mean number of cumulated migrations that an active object performs during the experience.

Chapter 7

Conclusions and Future Work

> *"And God saw every thing that he had made, and, behold, it was very good..." (Genesis, 1:31)*

In this thesis, a load-balancing algorithm for active objects which belong to a parallel application was developed and studied, setting the foundations for development of load-balancing algorithms for the middleware ProActive. This first approach is called the *Robin-Hood* + *Nottingham Sheriff* load-balancing algorithm.

Initially (Section 5.3) we found that the best policy for intensive-communicated parallel applications developed within ProActive, in terms of bandwidth used and response-time, was an *eager* scheme (called Robin-Hood). Then, we noted that using an eager scheme we did reach stability in balance, but the algorithm did not exploit all the network capabilities (as the best processors of the network). Therefore, we added a ranked work-stealing component (called Nottingham Sheriff), grouping active-objects on a subset of the best qualified processors. Both components exploited a Peer-to-Peer infrastructure developed for ProActive to perform *On-Demand* load-balancing queries (Section 5.4), reducing the number of messages which traverse the network. Thus, bandwidth in-

terference produced by load-balancing algorithm was reduced.

Then, aiming at reaching a near-optimal distribution of active objects using only local information provided by a P2P infrastructure, we studied the *Robin-Hood + Nottingham Sheriff* load-balancing algorithm on Desktop Grids (Section 6.1). Using a simulated P2P network, we showed that, using only a low number of links among nodes and a careful tuning of the algorithm parameters, a near-optimal distribution is reachable even for large-scale networks. We suggested to use a value near 1.0 for the stealing factor, which allows using around 1.7 times the optimal number of nodes for networks until 400 nodes, using less than 5.5 migrations in average per active object. Moreover, the number of migrations appears to be of order $O(\log(n))$ after the first optimal state (without overloaded nodes) is reached.

As seen in Section 6.1.4, the value of RS is a key factor for a low-cost and efficient load balancing and we did experimental fine-tuning for it. RS seems to depend on network topology and we are studying its behaviour to calculate it automatically and dynamically.

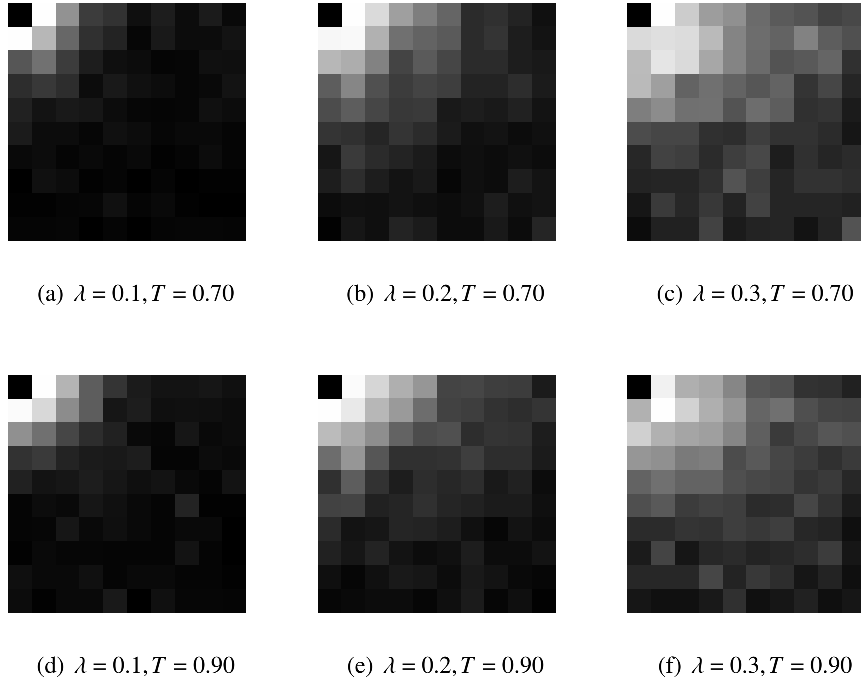
We presented the concept of *Project Grid* (Section 3.1.3) as the virtual organisation which has all the resources used on an Institutional Grid for a given project. Studying the traces collected from the ProActive PlugTests environment, we characterised and presented a model for Project Grids considering the node processing capacity and the latency among nodes (Section 6.2). Using the information of Project Grids, we presented in Section 6.2.3 two environment-aware load balancing algorithms dedicated to active-objects-based applications, aiming for optimising the performance of the *Robin-Hood + Nottingham Sheriff* load-balancing algorithm over project grids. The algorithms are validated experimentally through simulation, and shown to offer better performance when compared to traditional, non-environment-aware, algorithms.

For the future, we plan to extend the work on environment-aware load-balancing algorithms with more metrics: symmetric (e.g., bandwidth in an unrestricted network), asymmetric (e.g., bandwidth in a network with traffic shaping and different quotas for different participants to the project grid), and user-defined (e.g., based on economic principles).

In Section 6.2.6 we noted that confidence of some combinations of algorithms was low because a given cluster was not suitable to process the given application. Therefore, in Section 6.3 we shown how using coupling based contracts to ensure minimal requirements to deploy parallel applications. By now a contract have only two states: *valid* or *invalid*. In the future, we would like to extend this concept by introducing *conformance levels* in coupling contracts. Thus, a minimum conformance level (i.e. minimum set of known clauses) could be provided for basic applications, and higher conformance levels (i.e. a superset of the lower conformance levels) could be used for more advanced features that require more specific clauses. From the Grid infrastructure side, in the future we would like to identify standard interfaces for coupling applications with different types of Grids. The idea is to be able to release applications packaged with interfaces that certify the deployment of an application with a Grid interface.

Appendix A

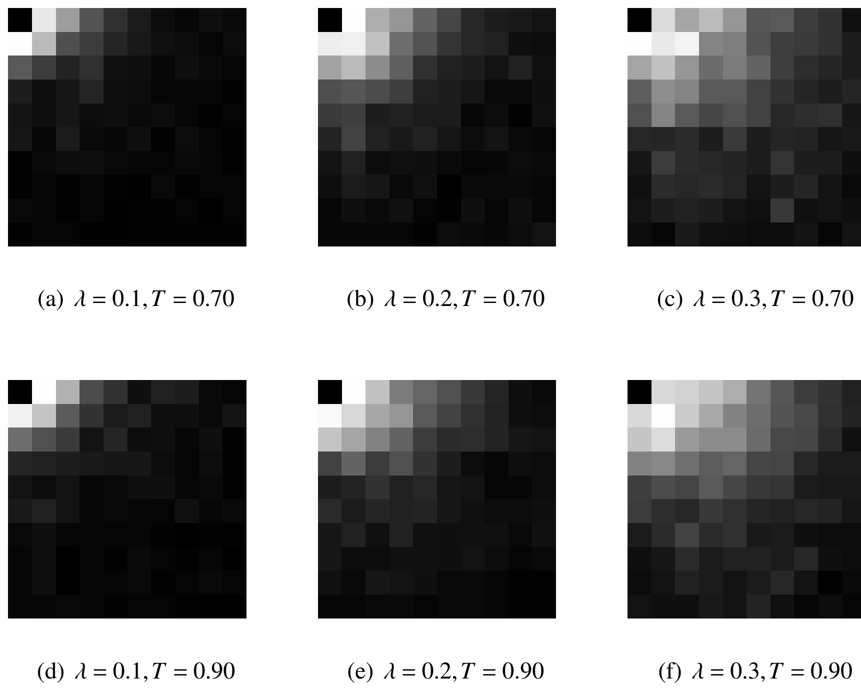
Matrices for Robin-Hood algorithm working alone



(a) $\lambda = 0.1, T = 0.70$ (b) $\lambda = 0.2, T = 0.70$ (c) $\lambda = 0.3, T = 0.70$

(d) $\lambda = 0.1, T = 0.90$ (e) $\lambda = 0.2, T = 0.90$ (f) $\lambda = 0.3, T = 0.90$

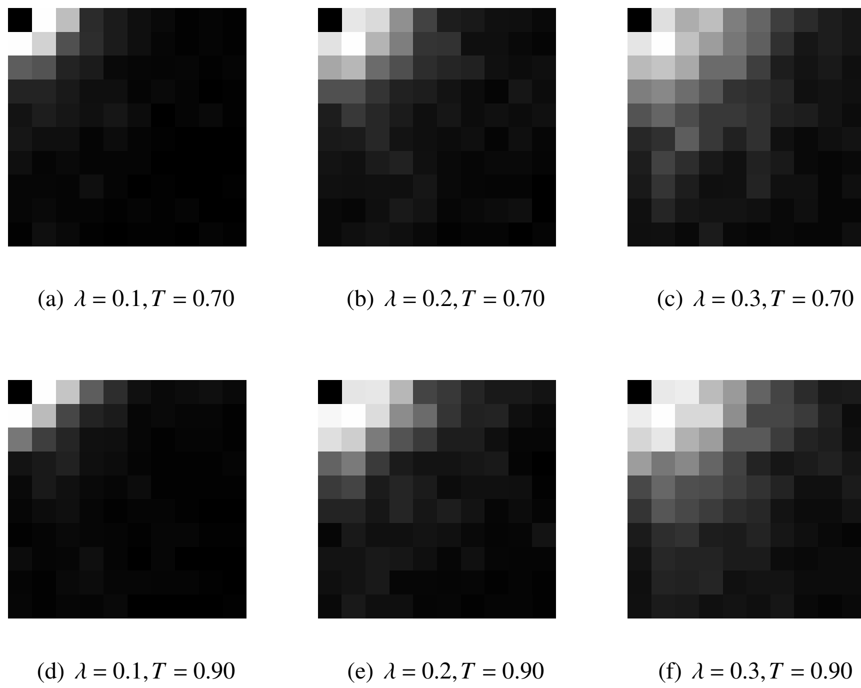
Figure A.1: Final distribution for the *Robin-Hood* algorithm only, for $RB = 0.5$ and $q = 3$



(a) $\lambda = 0.1, T = 0.70$ (b) $\lambda = 0.2, T = 0.70$ (c) $\lambda = 0.3, T = 0.70$

(d) $\lambda = 0.1, T = 0.90$ (e) $\lambda = 0.2, T = 0.90$ (f) $\lambda = 0.3, T = 0.90$

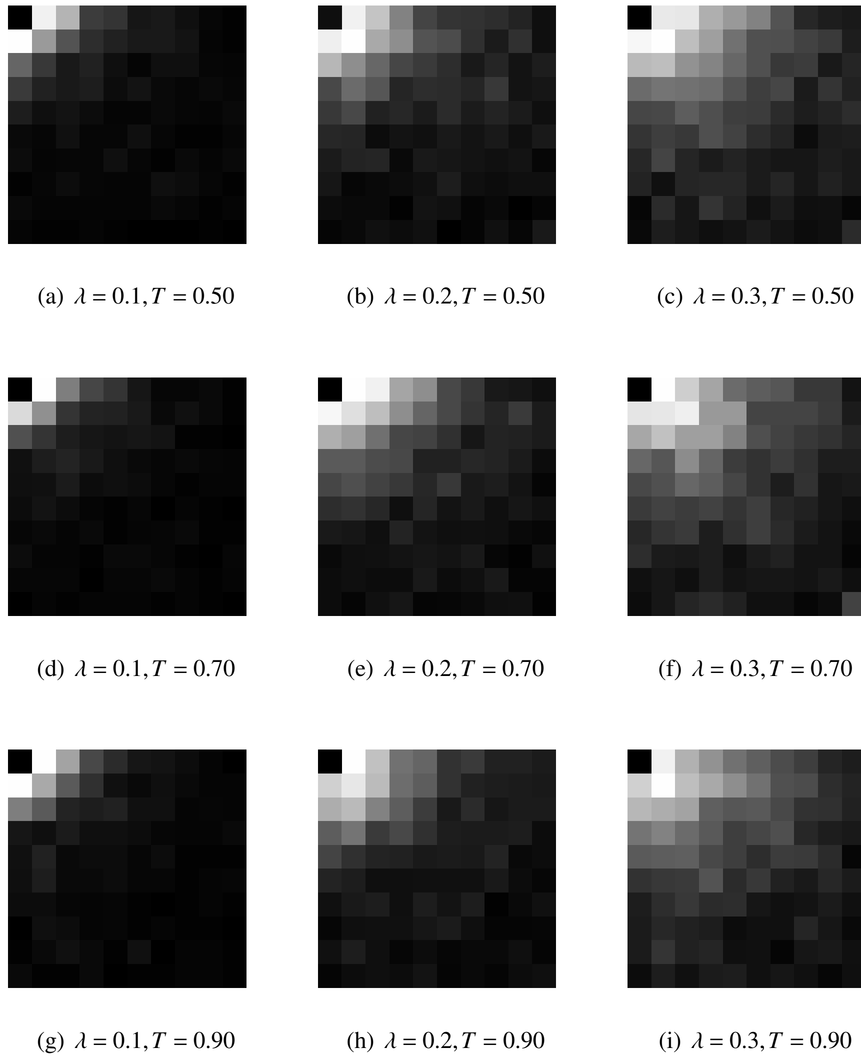
Figure A.2: Final distribution for the *Robin-Hood* algorithm only, for $RB = 0.5$ and $q = 4$



(a) $\lambda = 0.1, T = 0.70$ (b) $\lambda = 0.2, T = 0.70$ (c) $\lambda = 0.3, T = 0.70$

(d) $\lambda = 0.1, T = 0.90$ (e) $\lambda = 0.2, T = 0.90$ (f) $\lambda = 0.3, T = 0.90$

Figure A.3: Final distribution for the *Robin-Hood* algorithm only, for $RB = 0.5$ and $q = 5$



(a) $\lambda = 0.1, T = 0.50$ (b) $\lambda = 0.2, T = 0.50$ (c) $\lambda = 0.3, T = 0.50$

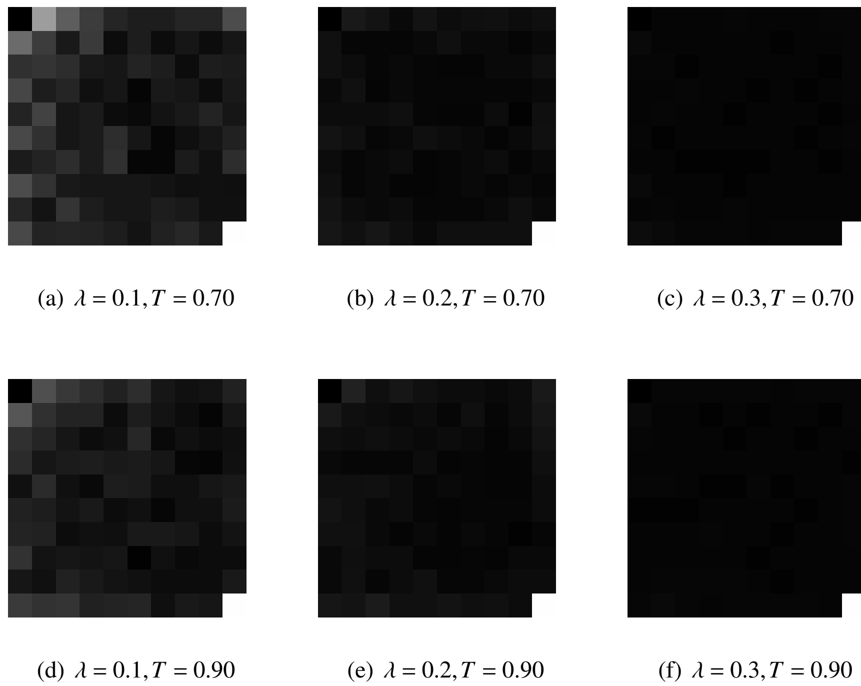
(d) $\lambda = 0.1, T = 0.70$ (e) $\lambda = 0.2, T = 0.70$ (f) $\lambda = 0.3, T = 0.70$

(g) $\lambda = 0.1, T = 0.90$ (h) $\lambda = 0.2, T = 0.90$ (i) $\lambda = 0.3, T = 0.90$

Figure A.4: Final distribution for the *Robin-Hood* algorithm only, for $RB = 0.7$ and $q = 4$

Appendix B

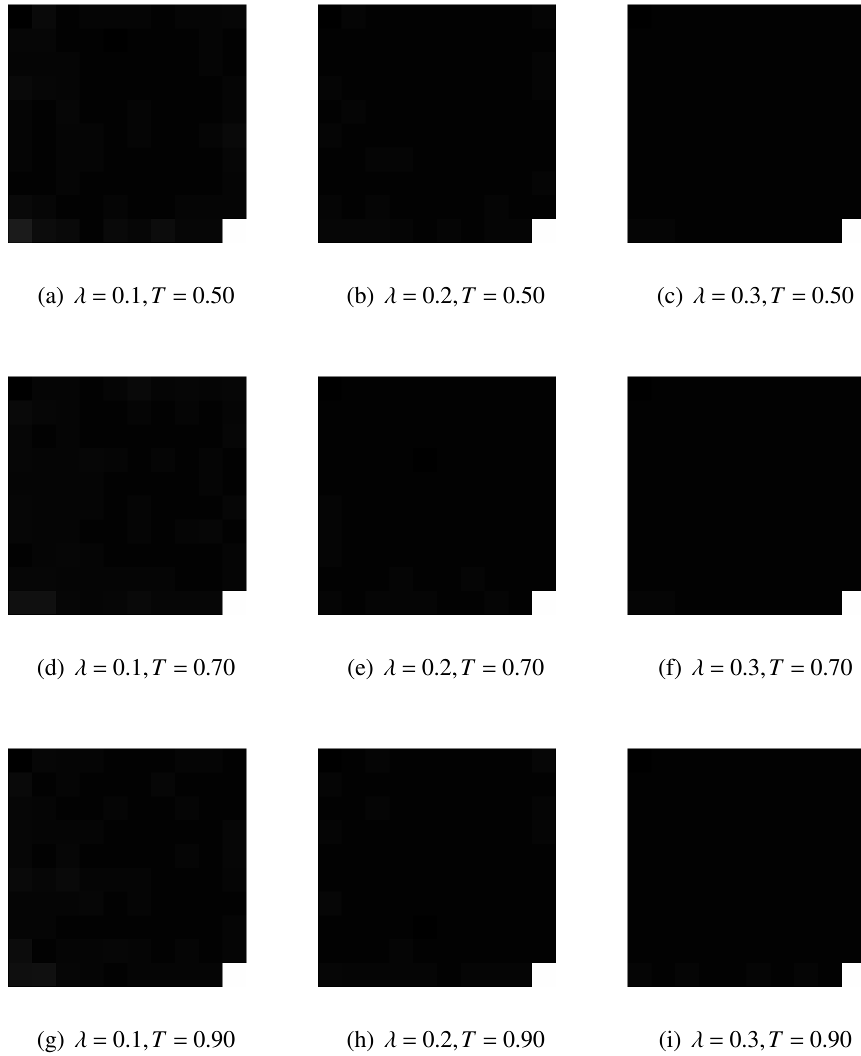
Matrices for Robin-Hood + Nottingham-Sheriff algorithm



(a) $\lambda = 0.1, T = 0.70$ (b) $\lambda = 0.2, T = 0.70$ (c) $\lambda = 0.3, T = 0.70$

(d) $\lambda = 0.1, T = 0.90$ (e) $\lambda = 0.2, T = 0.90$ (f) $\lambda = 0.3, T = 0.90$

Figure B.1: Final distribution for the *Robin-Hood + Nottingham Sheriff* algorithm, for $RB = 0.5$, $RS = 0.5$ and $q = 3$

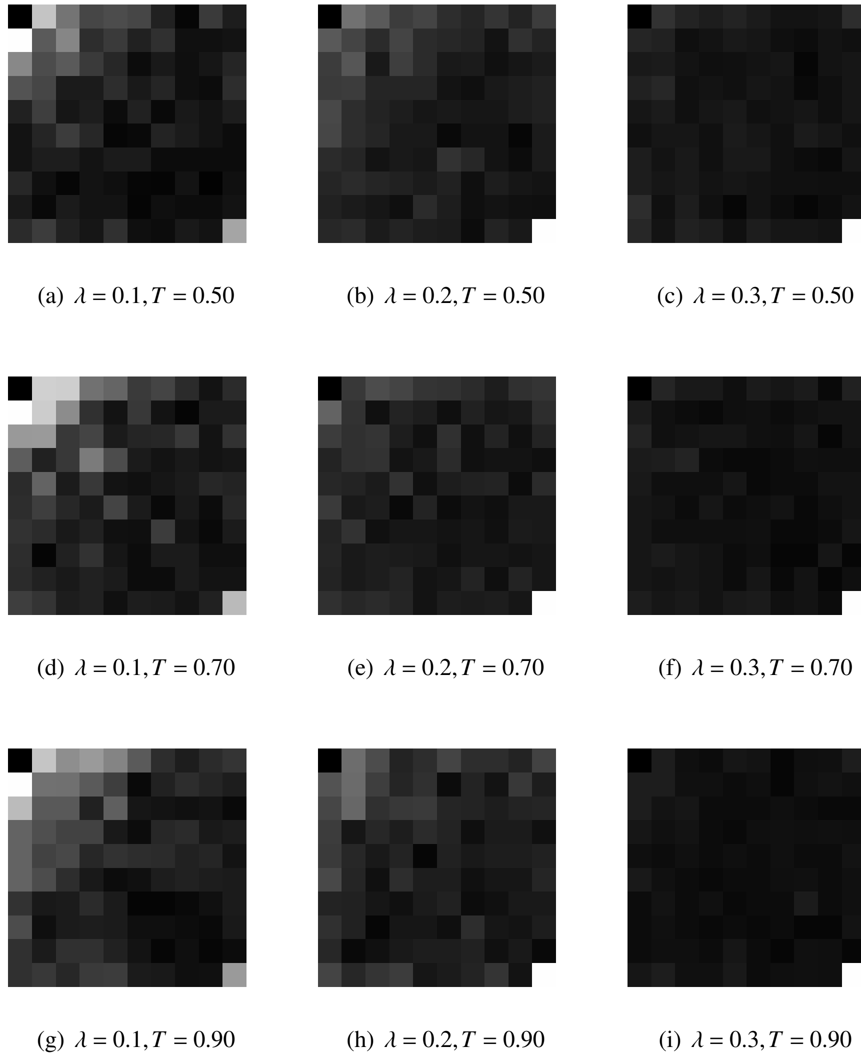


(a) $\lambda = 0.1, T = 0.50$ (b) $\lambda = 0.2, T = 0.50$ (c) $\lambda = 0.3, T = 0.50$

(d) $\lambda = 0.1, T = 0.70$ (e) $\lambda = 0.2, T = 0.70$ (f) $\lambda = 0.3, T = 0.70$

(g) $\lambda = 0.1, T = 0.90$ (h) $\lambda = 0.2, T = 0.90$ (i) $\lambda = 0.3, T = 0.90$

Figure B.2: Final distribution for the *Robin-Hood* + *Nottingham Sheriff* algorithm, for $RB = 0.5$, $RS = 0.5$ and $q = 5$

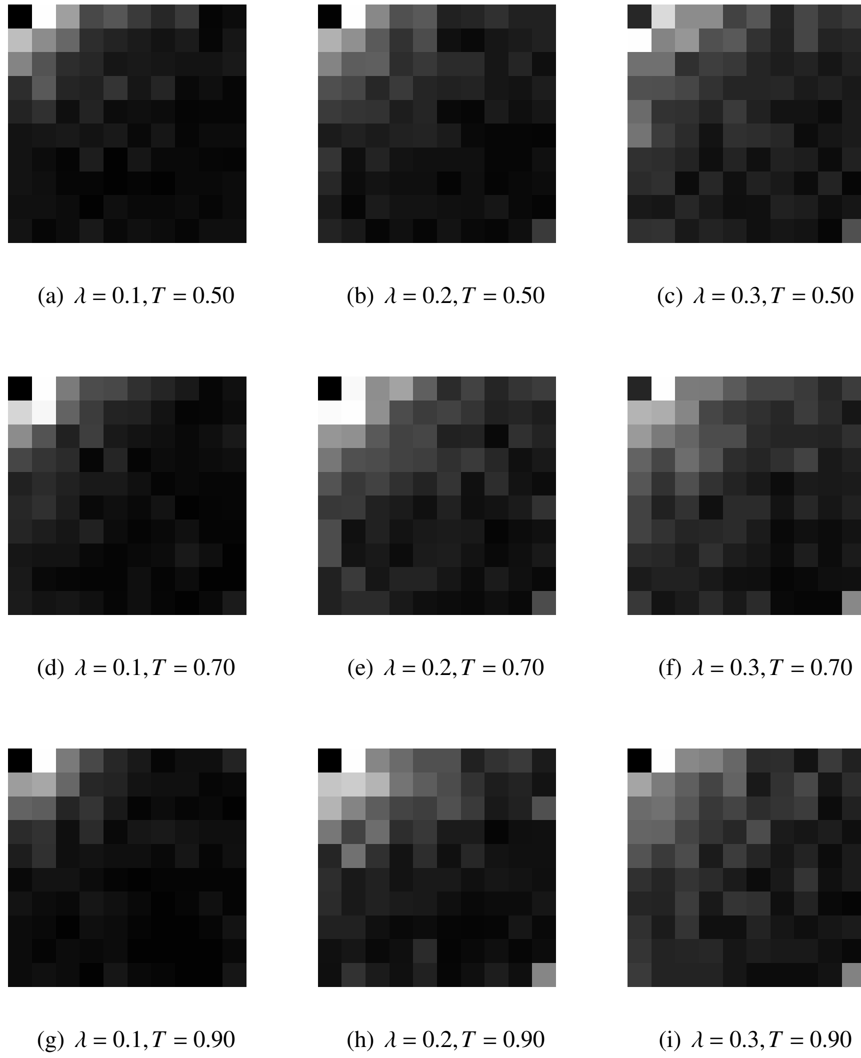


(a) $\lambda = 0.1, T = 0.50$ (b) $\lambda = 0.2, T = 0.50$ (c) $\lambda = 0.3, T = 0.50$

(d) $\lambda = 0.1, T = 0.70$ (e) $\lambda = 0.2, T = 0.70$ (f) $\lambda = 0.3, T = 0.70$

(g) $\lambda = 0.1, T = 0.90$ (h) $\lambda = 0.2, T = 0.90$ (i) $\lambda = 0.3, T = 0.90$

Figure B.3: Final distribution for the *Robin-Hood + Nottingham Sheriff* algorithm, for $RB = 0.7$, $RS = 0.7$ and $q = 3$



(a) $\lambda = 0.1, T = 0.50$ (b) $\lambda = 0.2, T = 0.50$ (c) $\lambda = 0.3, T = 0.50$

(d) $\lambda = 0.1, T = 0.70$ (e) $\lambda = 0.2, T = 0.70$ (f) $\lambda = 0.3, T = 0.70$

(g) $\lambda = 0.1, T = 0.90$ (h) $\lambda = 0.2, T = 0.90$ (i) $\lambda = 0.3, T = 0.90$

Figure B.4: Final distribution for the *Robin-Hood* + *Nottingham Sheriff* algorithm, for $RB = 0.9$, $RS = 0.9$ and $q = 3$

Appendix C

Expected values for Kolmogorov-Smirnov test statistics

	Level of Significance (α) for T = Max $\{F_{\text{Real}}(x) - F_{\text{Theoretical}}(x)\}$				
Sample Size (K)	0.20	0.15	0.10	0.05	0.01
1	0.900	0.925	0.950	0.975	0.995
2	0.684	0.726	0.776	0.842	0.929
3	0.565	0.597	0.642	0.708	0.828
4	0.494	0.525	0.564	0.624	0.733
5	0.446	0.474	0.510	0.565	0.669
10	0.322	0.342	0.368	0.410	0.490
15	0.266	0.283	0.304	0.338	0.404
20	0.231	0.246	0.264	0.294	0.356
25	0.210	0.220	0.240	0.270	0.320
30	0.190	0.200	0.220	0.240	0.290
35	0.180	0.190	0.210	0.230	0.270
> 35	$\frac{1.07}{\sqrt{K}}$	$\frac{1.22}{\sqrt{K}}$	$\frac{1.36}{\sqrt{K}}$	$\frac{1.52}{\sqrt{K}}$	$\frac{1.63}{\sqrt{K}}$

If the calculated ratio is greater than the value shown, the null hypothesis is rejected for the chosen level of confidence.

Bibliography

[1] Aberer, K., Datta, A., and Hauswirth, M. (2005). Multifaceted simultaneous load balancing in DHT-based P2P systems: A new game with old balls and bins. In Babaoglu, Ö., Jelasity, M., Montresor, A., Fetzer, C., Leonardi, S., van Moorsel, A. P. A., and van Steen, M., editors, *Self-star Properties in Complex Information Systems*, volume 3460 of *Lecture Notes in Computer Science*, pages 373–391. Springer Berlin Heidelberg.

[2] Aida, K., Natsume, W., and Futakata, Y. (2003). Distributed computing with hierarchical master-worker paradigm for parallel Branch and Bound algorithm. In *CCGRID '03: Proceedings of the 3rd International Symposium on Cluster Computing and the Grid*, pages 156–163, Washington, DC, USA. IEEE Computer Society.

[3] Anderson, D. P. (2004). Boinc: A system for public-resource computing and storage. In *GRID '04: Proceedings of the Fifth IEEE/ACM International Workshop on Grid Computing (GRID'04)*, pages 4–10, Washington, DC, USA. IEEE Computer Society.

[4] Appelbe, W. and Ravn, A. P. (1984). Encapsulation constructs in systems programming languages. *ACM Transactions on Programming Languages and Systems*, 6(2):129–158.

[5] Attali, I., Caromel, D., and Guider, R. (2000). A Step Towards Automatic Distribution of Java Programs. In *Proceedings of 2nd Formal Methods for Open Object-based Distributed Systems*, pages 141–161, Stanford University, California, USA. Kluwer Academic.

[6] Backschat, M., Pfaffinger, A., and Zenger, C. (1996). Economic-based dynamic load distribution in large workstation networks. In Bougé, L., Fraigniaud, P., Mignotte, A., and Robert, Y., editors, *Euro-Par '96 Parallel Processing, Second International Euro-Par Conference, Lyon,*

France, August 26-29, 1996, Proceedings, Volume II, volume 1124 of *Lecture Notes in Computer Science*, pages 631–634. Springer Berlin Heidelberg.

[7] Baduel, L., Baude, F., and Caromel, D. (2005). Object-oriented SPMD. In *5th International Symposium on Cluster Computing and the Grid (CCGrid 2005), 9-12 May, 2005, Cardiff, UK*, pages 824–831. IEEE Computer Society.

[8] Baldeschwieler, E., Blumofe, R., and Brewer, E. (1996). Atlas: An Infrastructure for Global Computing. In *Proceedings of the Seventh ACM SIGOPS European Workshop: Systems Support for Worldwide Applcations*.

[9] Barabasi, A.-L. and Albert, R. (1999). Emergence of scaling in random networks. *Science*, 286:509–512.

[10] Baude, F., Caromel, D., Huet, F., Mestre, L., and Vayssière, J. (2002). Interactive and Descriptor-Based Deployment of Object-Oriented Grid Applications. In *11th IEEE International Symposium on High Performance Distributed Computing (HPDC)*, pages 93–102, Edinburgh, Scotland. IEEE Computer Society.

[11] Baude, F., Caromel, D., Huet, F., and Vayssière, J. (2000). Communicating Mobile Active Objects in Java. In *Proceedings of the 8th International Conference on High Performance Computing and Networking Europe*, volume 1823 of *Lecture Notes in Computer Science*, pages 633–643, Amsterdam, The Netherlands. Springer Berlin / Heidelberg.

[12] Bender, M. A. and Rabin, M. O. (2000). Scheduling cilk multithreaded parallel programs on processors of different speeds. In *SPAA '00: Proceedings of the Twelfth Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 13–21, New York, NY, USA. ACM Press.

[13] Berenbrink, P., Friedetzky, T., and Goldberg, L. A. (2001). The natural work-stealing algorithm is stable. In *IEEE Symposium on Foundations of Computer Science*, pages 178–187, Washington, DC, USA. IEEE Computer Society.

[14] Berman, F., Fox, G., and Hey, A. J. G. (2003). *Grid Computing: Making the Global Infrastructure a Reality*. John Wiley & Sons, Inc., New York, NY, USA.

[15] Birtwhistle, G., Dahl, O., Myhrhaug, B., and Nygaard, K. (1979). *Simula Begin*. Chartwell-Bratt Ltd.

[16] Blumofe, R. D. (1992). Managing storage for multithreaded computations. Master's thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, Cambridge, USA. Also available as MIT Laboratory for Computer Science Technical Report MIT/LCS/TR-552.

[17] Blumofe, R. D., Joerg, C. F., Kuszmaul, B. C., Leiserson, C. E., Randall, K. H., and Zhou, Y. (1995). Cilk: an efficient multithreaded runtime system. In *PPOPP '95: Proceedings of the Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 207–216, New York, NY, USA. ACM Press.

[18] Blumofe, R. D. and Leiserson, C. E. (1999). Scheduling multithreaded computations by work stealing. *Journal of the ACM*, 46(5):720–748.

[19] Bobrow, D. G., Gabriel, R. P., and White, J. L. (1993). *CLOS in context: the shape of the design space*, pages 29–61. MIT Press, Cambridge, MA, USA.

[20] Bokhari, S. H. (1981). A shortest tree algorithm for optimal assignments across space and time in a distributed processor system. *IEEE Transactions on Software Engineering*, 7(6):583–589.

[21] Booch, G. (1994). *Object-oriented analysis and design with applications (2nd ed.)*. Benjamin-Cummings Publishing Co., Inc., Redwood City, CA, USA.

[22] Bosque Orero, J., Gil, M., and Pastor, L. (2004). Dynamic load balancing in heterogeneous clusters. In *Proceedings of IASTED International Conference on Parallel and Distributed Computing and Networks*. Acta Press.

[23] Bustos-Jiménez, J. (2003). Robin hood: An active objects load balancing mechanism, for intranet. In *Proceedings of Workshop de Sistemas Distribuidos y Paralelismo, Chile*.

[24] Bustos-Jiménez, J., Caromel, D., di Costanzo, A., Leyton, M., and Piquer, J. M. (2005). Balancing active objects on a peer to peer infrastructure. In *Proceedings of the XXV International Conference of the Chilean Computer Science Society (SCCC 2005)*, pages 109–115, Valdivia, Chile. IEEE Computer Society.

[25] Bustos-Jiménez, J., Caromel, D., Iosup, A., Leyton, M., and Piquer, J. M. (2006a). The rocking chair and the grid: Balancing load across project grids. In *Integrated Research in Grid Computing, CoreGrid Integration Workshop*, pages 117–128.

[26] Bustos-Jiménez, J., Caromel, D., Leyton, M., and Piquer, J. M. (2006b). Coupling contracts for deployment on alien grids. In *CoreGRID Workshop on Grid Middleware (in conjunction with EuroPar)*, Lecture Notes in Computer Science. Springer Berlin.

[27] Bustos-Jiménez, J., Caromel, D., Leyton, M., and Piquer, J. M. (2006c). Load information sharing policies in communication-intensive parallel applications. In *ISSADS*, Lecture Notes in Computer Science. Springer Berlin Heidelberg.

[28] Bustos-Jiménez, J., Caromel, D., and Piquer, J. M. (2006d). Toward the infinite network, and beyond. In *Proceedings of 12th Workshop on Job Scheduling Strategies for Parallel Processing (JSSPP)*, volume 4376 of *Lecture Notes in Computer Science*, pages 176–191. Springer Berlin Heidelberg.

[29] Caromel, D. (1989a). A general model for concurrent and distributed object-oriented programming. *Proceedings of the 1988 ACM SIGPLAN workshop on Object-based concurrent programming*, 24(4):102–104.

[30] Caromel, D. (1989b). Service, asynchrony, and wait-by-necessity. *Journal of Object-Oriented Programming*, 2(4):12–22.

[31] Caromel, D. (1993). Toward a method of object-oriented concurrent programming. *Communications of the ACM*, 36(9):90–102.

[32] Caromel, D., Belloncle, F., and Roudier, Y. (1996). The C++// Language. In *Parallel Programming using C++*, pages 257–296. MIT Press. ISBN 0-262-73118-5.

[33] Caromel, D. and Henrio, L. (2005). *A Theory of Distributed Objects*. Springer. ISBN 3-540-20866-6.

[34] Caromel, D., Henrio, L., and Serpette, B. (2004). Asynchronous and Deterministic Objects. In *Proceedings of the 31st ACM Symposium on Principles of Programming Languages*, pages 123–134, Venice, Italy. ACM Press. ISBN 1-58113-729-X.

[35] Casavant, T. and Kuhl, J. (1988a). Effects of response and stability on scheduling in distributed computing systems. *IEEE Transactions on Software Engineering*, 14(11):1578–1588.

[36] Casavant, T. and Kuhl, J. (1988b). A taxonomy of scheduling in general-purpose distributed computing systems. *IEEE Transactions on Software Engineering*, 14(2):141–154.

[37] Chapin, S., Katramatos, D., Karpovich, J., and Grimshaw, A. (1999). The Legion resource management system. In Feitelson, D. G. and Rudolph, L., editors, *Job Scheduling Strategies for Parallel Processing*, Lecture Notes in Computer Science, pages 162–178. Springer Berlin Heidelberg.

[38] Chung, F. and Lu, L. (2005). The average distances in a random graph with given expected degrees. *Internet Mathematics*, 1(1):91–113.

[Combs] Combs, G. Ethereal: The world's most popular network protocol analyzer. `http://www.ethereal.com`.

[39] Czarnecki, K. and Eisenecker, U. (2000). *Generative Programming: methods, tools, and applications*. ACM Press/Addison-Wesley Publishing Co. New York, NY, USA.

[40] Dalle, O. (1999). *Techniques et Outils pour les communicatios et la répartition dynamique de charge dans les réseaux de stations de travail*. PhD thesis, École Doctorale Sciences pour l´ Ingenieur. Université de Nice - Sophia Antipolis, Nice, France.

[41] Dasgupta, P., Richard J. LeBlanc, J., Ahamad, M., and Ramachandran, U. (1991). The clouds distributed operating system. *Computer*, 24(11):34–44.

[42] di Costanzo, A. (2004). Modèle et infrastructure de programmation pair-à-pair. Master's thesis, Nice University France.

[43] Diestel, R. (2000). *Graph Theory*. Springer, New York, USA, second edition.

[44] Dobber, M., Koole, G., and van der Mei, R. D. (2004). Dynamic load balancing for a grid application. In Bougé, L. and Prasanna, V. K., editors, *High Performance Computing - HiPC 2004*, volume 3296 of *Lecture Notes in Computer Science*, pages 342–352. Springer Berlin / Heidelberg.

[45] Domingues, P., Marques, P., and Silva, L. M. (2005). Resource usage of windows computer laboratories. In *ICPP Workshops*, pages 469–476. IEEE Computer Society.

[46] dos Santos, L. P. P. (1996). Load distribution: a survey. citeseer.ist.psu.edu/santos96load.html.

[47] Erdős, P. and Rényi, A. (1959). On random graphs I. *Publ. Mathematica*, 6:290–297.

[48] Erdős, P. and Rényi, A. (1960). On the evolution of random graphs. *Publ. Mathematica. Inst. Hungar. Acad. Sci.*, 5:17–61.

[ETSI and INRIA] ETSI and INRIA. 2nd Grid Plugtests. `http://www.etsi.org/plugtests/GRID.htm`.

[49] Faloutsos, M., Faloutsos, P., and Faloutsos, C. (1999). On power-law relationships of the internet topology. *Computer Communication Review*, 29(4):251–262.

[50] Fensel, D. and Bussler, C. (2002). The web service modeling framework WSMF. *Electronic Commerce Research and Applications*, 1(2):113–137.

[51] Ferber, J. (1989). Computational reflection in class based object-oriented languages. In *OOPSLA '89: Conference proceedings on Object-oriented programming systems, languages and applications*, pages 317–326, New York, NY, USA. ACM Press.

[52] Ferrari, D. and Zhou, S. (1988). An empirical investigation of load indices for load balancing applications. In *Performance '87: Proceedings of the 12th IFIP WG 7.3 International Symposium on Computer Performance Modelling, Measurement and Evaluation*, pages 515–528. North-Holland.

[53] Flynn, M. (1972). Some computer organizations and their effectiveness. *IEEE Transactions on Computers*, C-21:948–960.

[54] Foster, I. (2002). What is the grid? - a three point checklist. *GRIDtoday*, 1(6).

[55] Foster, I. and Kesselman, C. (1999). Computational grids. In *The Grid: blueprint for a new computing infrastructure*, pages 15–51. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

[56] Foster, I., Kesselman, C., and Tuecke, S. (2001). The anatomy of the grid: Enabling scalable virtual organizations. *Int. J. High Perform. Comput. Appl.*, 15(3):200–222.

[57] Frey, J., Tannenbaum, T., Livny, M., Foster, I., and Tuecke, S. (2002). Condor-g: A computation management agent for multi-institutional grids. *Cluster Computing*, 5(3):237–246.

[58] Frigo, M., Leiserson, C. E., and Randall, K. H. (1998). The implementation of the Cilk-5 multithreaded language. In *PLDI '98: Proceedings of the ACM SIGPLAN 1998 conference on Programming language design and implementation*, pages 212–223, New York, NY, USA. ACM Press.

[59] ghazali Talbi, E. (1995). Allocation dynamique de processus dans les systèmes distribués et parallèles : état de l´art. Technical report, Laboratoire d Informatique Fondamentale de Lille.

[60] Gilbert, E. (1959). Random graphs. *Annals of Mathematical Statistics*, 30(4):1141–1144.

[Gnutella measurement project] Gnutella measurement project. Clip 2. `http://www.clip2.com`.

[61] Grimshaw, A., Wulf, W., French, J., Weaver, A., and Reynolds Jr., P. (1994). Legion: The next logical step toward a nation wide virtual computer. Technical Report CS-94-21, University of Virginia.

[62] Heymann, E., Senar, M. A., Luque, E., and Livny, M. (2000). Adaptive scheduling for master-worker applications on the computational grid. In *GRID*, volume 1971 of *Lecture Notes in Computer Science*, pages 214–227. Springer Berlin Heidelberg.

[63] Huet, F. (2002). *Objets mobiles : conception d´un middleware et évaluation de la communication*. PhD thesis, Université de Nice Sophia-Antipolis, Nice, France.

[64] Huet, F., Caromel, D., and Bal, H. (2004). A high performance java middleware with a real application. In *Proceedings of the ACM/IEEE Conference on Supercomputing (SC)*, pages 2–17, Washington, DC, USA. IEEE Computer Society.

[65] Huffaker, B., Fomenkov, M., Plummer, D., Moore, D., and Claffy, K. (2002). Distance metrics in the internet. `http://citeseer.ist.psu.edu/huffaker02distance.html`.

[66] Ichikawa, S. (1998). Mathematical programming approach for static load balancing of parallel PDE solver. In *Proceedings of the 16th IASTED International Conference on Applied Informatics*. Acta Press.

[67] Iosup, A., Dumitrescu, C., Epema, D., Li, H., and Wolters, L. (2006a). An analysis of four long-term grid traces. Technical Report PDS-2006-003, TU Delft.

[68] Iosup, A., Garbacki, P., Pouwelse, J., and Epema, D. (2006b). Correlating topology and path characteristics of overlay networks and the internet. In *CCGRID '06: Proceedings of the Sixth IEEE International Symposium on Cluster Computing and the Grid (CCGRID'06)*, pages 10–18, Washington, DC, USA. IEEE Computer Society.

[69] Kee, Y.-S., Casanova, H., and Chien, A. (2004). Realistic modeling and synthesis of resources for computational grids. In *Proceedings of the ACM/IEEE conference on Supercomputing (SC)*, pages 54–63. ACM Press.

[70] Kiczales, G. and Rivieres, J. D. (1991). *The Art of the Metaobject Protocol*. MIT Press, Cambridge, MA, USA.

[71] Kleinberg, J. M. (2000). The small-world phenomenon: an algorithm perspective. In *Proceedings of the Thirty Second Annual ACM Symposium on Theory of Computing*, pages 163–170, New York, NY, USA. ACM Press.

[72] Kleinoder, J. and Golm, M. (1996). Metajava: an efficient run-time meta architecture for Java. In *Proceedings of the 5th International Workshop on Object Orientation in Operating Systems (IWOOOS)*, pages 54–61, Washington, DC, USA. IEEE Computer Society.

[73] Kondo, D., Taufer, M., Brooks III, C. L., Casanova, H., and Chien, A. A. (2004). Characterizing and evaluating desktop grids: An empirical study. *International Parallel and Distributed Processing Symposium*, 01:26–35.

[74] Kremien, O. and Kramer, J. (1992). Methodical analysis of adaptive load sharing algorithms. *IEEE Transactions on Parallel and Distributed Systems*, 3(6):747–760.

[75] Kronenberg, N. P., Levy, H. M., and Strecker, W. D. (1986). Vaxcluster: a closely-coupled distributed system. *ACM Transactions on Computer Systems*, 4(2):130–146.

[76] Kunz, T. (1991). The influence of different workload descriptions on a heuristic load balancing scheme. *IEEE Transactions on Software Engineering*, 17(7):725–730.

[77] Kuszmaul, B. (1995). The StarTech Massively Parallel Chess Program. *Journal of the International Computer Chess Association*, 18(1):3–19.

[78] Lewis, M. and Grimshaw, A. (1996). The core legion object model. In *Proceedings of the Fifth IEEE International Symposium on High Performance Distributed Computing*, pages 551–561, Syracuse, New York, USA. IEEE, IEEE Computer Society.

[79] Lin, F. and Keller, R. (1987). The gradient model load balancing method. *IEEE Transactions on Software Engineering*, 13(1):32–38.

[80] Lo, M. and Dandamudi, S. (1996). Performance of hierarchical load sharing in heterogeneous distributed systems. In *Proceedings of International Conference on Parallel and Distributed Computing Systems, Dijon, France*, pages 370–377.

[81] Lu, D. and Dinda, P. A. (2003). Synthesizing realistic computational grids. In *Proceedings of the ACM/IEEE Conference on Supercomputing (SC)*, page 16, Washington, DC, USA. IEEE Computer Society.

[82] Maes, P. (1987). Concepts and experiments in computational reflection. In *OOPSLA '87: Conference proceedings on Object-oriented programming systems, languages and applications*, pages 147–155, New York, NY, USA. ACM Press.

[83] Malenfant, J., Jaques, M., and Demers, F. (1996). A tutorial on behavioral reflection and its implementation. In Kiczales, G., editor, *Proceedings of the Reflection 96 Conferencext*, pages 1–20.

[84] Medernach, E. (2005). Workload analysis of a cluster in a grid environment. In Feitelson, D. G., Frachtenberg, E., Rudolph, L., and Schwiegelshohn, U., editors, *Proceedings of 11th Workshop on Job Scheduling Strategies for Parallel Processing (JSSPP)*, volume 3834 of *Lecture Notes in Computer Science*, pages 36–61, Cambridge, USA. Springer Berlin.

[85] Meehean, J. and Livny, M. (2005). A service migration case study: Migrating the Condor schedd. In *Midwest Instruction and Computing Symposium*.

[86] Michael Litzkow, M. L. and Mutka, M. (1998). Condor - a hunter of idle workstations. In *Proceedings of 8th International Conference on Distribuited Computing Systems*, pages 104–111.

[87] Milgram, S. (1967). The small world problem. *Psychology Today*, 2(1):60–67.

[88] Mirchandaney, R., Towsley, D., and Stankovic, J. A. (1990). Adaptive load sharing in heterogeneous distributed systems. *Journal of Parallel and Distributed Computing*, 9(4):331–346.

[89] Mitzenmacher, M. (1996). *The power of two choices in randomized load balancing*. PhD thesis, University of California, Berkeley, California, USA. Chair-Alistair Sinclair.

[90] Mitzenmacher, M. (1997). On the analysis of randomized load balancing schemes. In *SPAA '97: Proceedings of the ninth annual ACM symposium on Parallel algorithms and architectures*, pages 292–301, New York, NY, USA. ACM Press.

[91] Mitzenmacher, M. (2000). How useful is old information? *IEEE Transactions on Parallel and Distributed Systems*, 11(1):6–20.

[92] Murata, K., Horspool, N., Manning, E., Yokote, Y., and Tokoro, M. (1995). Unification of active and passive objects in an object-oriented operating system. In *Proceedings of the 4th International Workshop on Object-Orientation in Operating Systems (IWOOOS)*, pages 68–71, Washington, DC, USA. IEEE Computer Society.

[93] Oasis Group at INRIA Sohpia-Antipolis (1999). Oasis: Active objects, semantics, internet, and security. `http://www-sop.inria.fr/oasis`.

[94] Oasis Group at INRIA Sohpia-Antipolis (2002). Proactive, the java library for parallel, distributed, concurrent computing with security and mobility. `http://proactive.objectweb.org`.

[95] Paul, P. (2002). Seti @ home project and its website. *Crossroads*, 8(3):3–5.

[96] Pulidas, S., Towsley, D., and Stankovic, J. (1987). Design of efficient parameter estimators for decentralized load. Technical Report UM-CS-1987-079, University of Massachusetts, Amherst, MA, USA.

[97] Raman, R., Livny, M., and Solomon, M. (1998). Matchmaking: Distributed resource management for high throughput computing. In *HPDC '98: Proceedings of the The Seventh IEEE International Symposium on High Performance Distributed Computing*, pages 140–146, Washington, DC, USA. IEEE Computer Society.

[98] Renard, H. (2005). *Équilibrage de charge et redistribution de données sur plates-formes hétérogénes*. PhD thesis, École normale supérieure de Lyon, France.

[99] Riedl, R. and Richter, L. (1996). Classification of load distribution algorithms. In *PDP '96: Proceedings of the 4th Euromicro Workshop on Parallel and Distributed Processing (PDP '96)*, pages 404–413, Washington, DC, USA. IEEE Computer Society.

[100] Ritter, J. (2001). Why Gnutella can't scale. No, really. `http://www.darkridge.com/~{}jpr5/doc/gnutella.html`.

[101] Roussopoulos, M. and Baker, M. (2002). Practical load balancing for content requests in peer-to-peer networks. *The Computing Research Repository*, cs.NI/0209023.

[102] Roy Chowdhury, S. and Gupta, B. (1994). A probabilistic dynamic load balancing algorithm for homogeneous distributed systems (with extension to hypercubes). In *CSC '94: Proceedings of the 22nd annual ACM computer science conference on Scaling up: Meeting the challenge of complexity in real-world computing applications*, pages 165–172, New York, NY, USA. ACM Press.

[103] Schaeffer, S. E. (2006). *Algorithms for nonuniform networks*. PhD thesis, Helsinki University of Technology, Espoo, Finland.

[104] Schollmeier, R. (2001). A definition of peer-to-peer networking for the classification of peer-to-peer architectures and applications. In *2001 International Conference on Peer-to-Peer Computing (P2P2001)*, Department of Computer and Information Science Linkopings Universitet, Sweden. IEEE Computer Society.

[105] Shen, C.-C. and Tsai, W.-H. (1985). A graph matching approach to optimal task assignment in distributed computing systems using a minimax criterion. *IEEE Transactions on Computers*, 34(3):197–203.

[106] Shivaratri, N. G., Krueger, P., and Singhal, M. (1992). Load distributing for locally distributed systems. *Computer*, 25(12):33–44.

[107] Shurkin, J. (1984). *Engines of the Mind: A History of the Computer*. W. W. Norton & Co.

[108] Smith, B. C. (1982). *Reflection and Semantics in a Procedural Language*. PhD thesis, Massachusetts Institute of Technology, Cambridge, USA.

[109] Sterling, T., Savarese, D., Becker, D., Dorband, J., Ranawake, U., and Packer, C. (1995). BEOWULF: A parallel workstation for scientific computation. In *Proceedings of the 24th International Conference on Parallel Processing*, volume 1, pages 11–14, Oconomowoc, WI. CRC Press.

[Sun Microsystems] Sun Microsystems. *RMI Architecture and Functional Specification*. `http://java.sun.com/j2se/1.4.2/docs/guide/rmi/spec/rmiTOC.html`.

[110] Sunderam, V. (1990). PVM: a framework for parallel distributed computing. *Concurrency: Pract. Exper.*, 2(4):315–339.

[111] Tanenbaum, A. S., van Renesse, R., van Staveren, H., Sharp, G. J., and Mullender, S. J. (1990). Experiences with the amoeba distributed operating system. *Commun. ACM*, 33(12):46–63.

[112] Team, E. (2004). `http://lcg.web.cern.ch/LCG`.

[113] Thain, D. and Livny, M. (2002). Error scope on a computational grid: Theory and practice. In *HPDC '02: Proceedings of the 11 th IEEE International Symposium on High Performance Distributed Computing HPDC-11 20002 (HPDC'02)*, pages 199–208, Washington, DC, USA. IEEE Computer Society.

[114] Theimer, M. and Lantz, K. (1989). Finding idle machines in a workstation-based distributed system. *IEEE Transactions on Software Engineering*, 15(11):1444–1458.

[115] van Nieuwpoort, R. V. (2003). *Efficient Java-Centric Grid-Computing*. PhD thesis, Vrije Universiteit, Amsterdam, The Netherlands.

[116] van Nieuwpoort, R. V., Kielmann, T., and Bal, H. E. (2000). Satin: Efficient parallel divide-and-conquer in java. In *Proceedings of EuroPar 2000*, Lecture Notes in Computer Science, pages 690–699, Munich, Germany. Springer Berlin / Heidelberg.

[117] van Nieuwpoort, R. V., Kielmann, T., and Bal, H. E. (2001). Efficient load balancing for wide-area divide-and-conquer applications. *SIGPLAN Notices*, 36(7):34–43.

[118] van Nieuwpoort, R. V., Maassen, J., Kielmann, T., and Bal, H. E. (2005). Satin: Simple and efficient java-based grid programming. *Scalable Computing: Practice & Experience*, 6(3):19–32. AGridM 2003, Workshop on Adaptive Grid Middleware, New Orleans, Louisiana, USA.

[119] Veldhuizen, T. L. and Gannon, D. (1998). Active Libraries: Rethinking the roles of compilers and libraries. In *Proceedings of the SIAM Workshop on Object Oriented Methods for Interoperable Scientific and Engineering Computing (OO'98)*, Philadelphia, Pennsylvania, USA.

[120] Virtanen, S. E. (2003). Properties of nonuniform random graph models. Research Report A77, Helsinki University of Technology, Laboratory for Theoretical Computer Science, Espoo, Finland.

[W3C] W3C. XML schema: Formal description. `http://www.w3.org/TR/xmlschema-formal`.

[121] Waldo, J. and Arnold, K. (2000). *The Jini Specifications, Second Edition*. Addison-Wesley.

[122] Wand, M. and Friedman, D. P. (1986). The mystery of the tower revealed: a non-reflective description of the reflective tower. In *LFP '86: Proceedings of the 1986 ACM conference on LISP and functional programming*, pages 298–307, New York, NY, USA. ACM Press.

[123] Watanabe, T. and Yonezawa, A. (1988). Reflection in an object-oriented concurrent language. In *OOPSLA '88: Conference proceedings on Object-oriented programming systems, languages and applications*, pages 306–315, New York, NY, USA. ACM Press.

[124] Watts, D. and Strogatz, S. (2005). Collective dynamics of “small world” networks. *Nature*, 393(6684):440–442.

[125] Waxman, B. (1988). Routing of multipoint connections. *IEEE Journal on Selected Areas in Communications*, 6(9):1617–1622.

[126] Wegner, P. (2005). Concepts and paradigms of object-oriented programming. *SIGPLAN OOPS Messenger*, 1(1):7–87.

[127] Yan, J. C. and Lundstrom, S. F. (1989). The post-game analysis framework - developing resource management strategies for concurrent systems. *IEEE Transactions on Knowledge and Data Engineering*, 1(3):293–309.

[128] Yokote, Y. and Tokoro, M. (1987). Concurrent programming in concurrent smalltalk. In *Object-oriented concurrent programming*, pages 129–158, Cambridge, MA, USA. MIT Press.