THÈSE DE DOCTORAT

École Doctorale
« Sciences et Technologies de l'Information et de la Communication »
de Nice - Sophia Antipolis

Discipline Informatique

UNIVERSITÉ DE NICE - SOPHIA ANTIPOLIS
FACULTÉ DES SCIENCES

BRANCH-AND-BOUND WITH PEER-TO-PEER FOR LARGE-SCALE GRIDS

---

par

Alexandre DI COSTANZO

Thèse dirigée par **Denis CAROMEL**

au sein de l'équipe OASIS,
équipe commune de l'I.N.R.I.A. Sophia Antipolis, du C.N.R.S. et du laboratoire I3S
soutenue le 12 octobre 2007

Jury:

Président du Jury	Franck CAPPELLO	Directeur de recherche, LRI-INRIA Futurs Orsay
Rapporteurs	Raphaël COUTURIER	Professeur, IUT de Belfort Montbéliard
	Dennis GANNON	Professeur, Indiana University
	El-Ghazali TALBI	Professeur, LIFL-INRIA Futurs Lille
Examinateur	Manuel CLERGUE	Maître de Conférences, I3S UNSA-CNRS
Invité industriel	François LABURTHE	Directeur R&D, Amadeus
Directeur de thèse	Denis CAROMEL	Professeur, Université de Nice Sophia Antipolis

To Magali,
to my family,
– Alexandre

Acknowledgments

A PhD thesis is a great experience for working on very stimulating topics, challenging problems, and for me perhaps the most important to meet and collaborate with extraordinary people. This remarkable research environment wasn't possible without the determination of Isabelle Attali and Denis Caromel who both created the best conditions for my PhD. Unfortunately, Isabelle tragically died with her two young sons in the tsunami, Sri Lanka, December 2004. I thank you for the great opportunities you and Denis gave to me.

First and foremost, I thank my advisor, Denis Caromel for accepting me in the OASIS project. I am grateful to him for letting me pursue my research interests with sufficient freedom, while being there to guide me all the same. I thank also Françoise Baude for advices she gave me along my thesis.

My greetings also go to Raphaël Couturier, Dennis Gannon, and El-Ghazali Talbi who honored me by accepting to review this thesis. I am also very grateful to the other members of the jury, Franck Cappello, Manuel Clergue, and François Laburthe.

I would like again to greatly thank Dennis Gannon for reviewing this thesis and for accepting to be a member of the jury, it is an immense honor for me. Of course, I am very thankful to him for his kindness when he invited me at Indiana University to work with his team. During this visit, I met great people and learned a lot. My sojourn gave me an extraordinary motivation to accomplish my thesis. I am still thinking a lot about this experience.

I am grateful, too, to the legions of OASIS people who are not only colleague but friends. In a random order, I thank Laurent Baduel, Arnaud & Virginie Contes, Christian G. Delbé, Matthieu Morel, Ludovic Henrio, Fabrice Huet, Cédric Dalmasso, Clément Mathieu, and others.

A special thanks to all OASIS's Chilean people Tomas Barros, Marcela Rivera, Javier Bustos, and specially Mario Leyton and Antonio Cansado who reviewed some parts of this thesis.

Also I am grateful to Ian Stokes-Rees for his incredible review of my thesis.

I say good luck to the next: Guillaume, Mario, Antonio, Paul (my ping-pong professor), and Marcela.

Without forgetting assistants who during years help me to solve all administrative annoyances. Thanks to Claire Senica, Patricia Lachaume, Christiane Gallerie, and Patricia Maleyran.

During my PhD my happiness was to meet many people some of the them are now

more than friends.

A special acknowledgment goes to Guillaume Chazarain – alias *the Geek* – my office-mate. I learned a lot with your research/technical discussion and thanks for having taking care of my desktop computer. And good luck because you are the next PhD. I keep the hope than one day you leave the Linux side for the Mac side.

I would like to thank Vincent Cavé who I have a debt. Thank you Vincent, to have taken care of the n-queens while I was staying in the USA.

Many thanks to Igor Rosenberg for his unconditional friendship, his support, and to remember me to focus on finishing my PhD.

This thesis could not have been written without Magali's help and support during these last months. For that I am deeply grateful to her.

Last but certainly not the least, I owe a great deal to my family for providing me with emotional support during my PhD. My parents have been very supportive; I must acknowledge them for their help and support.

Thanks, folks!

Contents

List of Figures

List of Tables

Part I

Thesis

Chapter 1

Introduction

The main goal of this thesis is to propose a framework for solving combinatorial optimization problems with the branch-and-bound algorithm in Grid computing environments. Because we target large-scale Grids, this thesis also proposes an infrastructure, based on a peer-to-peer architecture. For Grids that allows to mix desktop machines and clusters.

1.1 Problematics

These last years, computational Grids have been widely deployed around the world to provide high performance computing tools for research and industry. Grids gather large amount of heterogeneous resources across geographically distributed sites to a single virtual organization. Resources are usually organized in clusters, which are managed by different administrative domains (labs, universities, *etc.*).

The branch-and-bound algorithm is a technique for solving problem such as combinatorial optimization problems. This technique aims to find the optimal solution and to prove that no ones are better. The algorithm splits the original problem into sub-problems of smaller size and then, for each sub-problem, the *objective function* computes the lower/upper bounds.

Because of the large size of handled problems (enumerations size and/or NP-hard class), finding an optimal solution for a problem can be impossible on a single machine. However, it is relatively easy to provide parallel implementations of branch-and-bound. Thanks to the huge number of resources Grids provide, they seem to be well adapted for solving very large problems with branch-and-bound.

In parallel of Grid computing development, an approach for using and sharing resources called peer-to-peer networks has also been deployed. Peer-to-peer focuses on sharing resources, decentralization, dynamicity, and resource failures.

Grid users have usually access to one or two clusters and have to share their computation time with others; they are not able to run computations that would take months to complete because they are not allowed to use all the resources exclusively for their experiments. At the same time, these researchers work in labs or institutions, which have a large number of desktop machines. These desktops are usually under-utilized and are only available to a single user. They also are highly volatile (*e.g.* shutdown, reboot,

failure). Organizing such desktop machines as a peer-to-peer network for computations or other kinds of resource sharing is now increasingly popular.

However, existing models and infrastructures for peer-to-peer computing are limited as they support only independent worker tasks, usually without communications between tasks. However peer-to-peer computing seems well adapted to applications with low communication/computation ratio, such as parallel search algorithms. We therefore propose in this thesis a peer-to-peer infrastructure of computational nodes for distributed communicating applications, such as our branch-and-bound framework for Grids.

Furthermore, Grid computing introduces new challenges that must be handled by both infrastructure and framework. These challenges may be listed as follows:

- *Heterogeneity*: Grids gather resources from different institutional sites (labs, universities, *etc.*). This collection of gathered resources implies resources from multiple hardware vendors, different operating systems, and relying on different network protocols. In contrast, each site is usually composed of a cluster, which is an homogeneous computing environment (same hardware, same operating system, same network for all cluster machines).
- *Deployment*: the large amount of heterogeneous resources complicates the deployment task in terms of configuration and connection to remote resources. Deployment sites targeted may be specified beforehand, or automatically discovered at runtime.
- *Communication*: solving combinatorial optimization problems even in parallel with a very large pool of resources may be very hard. Nevertheless, using communications between distributed processes may increase the computation speedup. However, Grids are not the most adapted environment for communicating because of issues such as heterogeneity, high latency between sites, and scalability.
- *Fault-tolerance*: Grids are composed of numerous heterogeneous machines which are managed by different administrative domains, and the probability of having faulted nodes during an execution is not negligible.
- *Scalability*: this is one of the most important challenge for Grids. This is also one of the most difficult task to handle: first, for the large number of resources that Grids provide; and second, for the wide-distribution of resources leading to high-latency and bandwidth reduction.

1.2 Objectives and contributions

This work belongs to the research area that focuses on Grid infrastructures and branch-and-bound frameworks for Grids, and our main objective is to *define an infrastructure and a framework for solving combinatorial optimization problems that address the requirement of Grid computing*.

Grids gather large amount of heterogeneous resources across geographically distributed sites to a single virtual organization. Thanks to the huge number of resources Grid provide, they seem to be well adapted for solving very large problems with branch-and-bound. Nevertheless, Grids introduce new challenges such as deployment, heterogeneity, fault-tolerance, communication, and scalability.

In this thesis, we consider that some of these challenges must be handled by the underlying Grid infrastructure, particularly deployment, scalability, and heterogeneity; and that the other challenges and scalability must be handled by the framework. In the proposed Grid infrastructure, we aim at easing the access to a large pool of resources. Moreover, with these resources we aim at providing a framework that takes the maximum benefit of all these computational power. Furthermore, both infrastructure and framework have to hide all Grid difficulties.

The main contributions of this thesis are:

- an analysis of existing peer-to-peer architectures, and especially those for Grid computing;
- an analysis of existing branch-and-bound frameworks for Grids;
- a peer-to-peer infrastructure for computational Grids, that allows to mix desktop machines and clusters, the infrastructure is decentralized, self-organized, and configurable;
- an operating infrastructure that was deployed as a permanent desktop Grid in the INRIA Sophia lab, with which we have achieved a computation record by solving the n-queens problem for 25 queens; and
- a branch-and-bound framework for Grids, which is based on a hierarchical master-worker approach and provides a transparent communication system among tasks.

1.3 Overview

This document is organized as follows:

- In Chapter 2, we position our work in the context of the Grid computing. First, we provide an overview of existing Grid systems; this allows us to point out what must be enhanced. Then, we define the notion of peer-to-peer systems and we show that peer-to-peer can provide more dynamic and flexible Grids. We also position our work in the context of peer-to-peer systems for Grids. Afterward, we present existing branch-and-bound models for Grids and we relate our work to existing frameworks, which target Grids. Finally, we present the active object model and the ProActive middleware that we based our work on.
- In Chapter 3, we propose a desktop Grid infrastructure based on a peer-to-peer architecture. With this infrastructure, we are the first to have solved the n-queens problem with 25 queens: the computation took six months to complete.
- In Chapter 4, we describe our branch-and-bound framework for Grids to solve combinatorial optimization problems. We also report experiments with the flow-shop problem on a nationwide Grid, the french *Grid'5000*.
- Chapter 5 describes an extension of our peer-to-peer infrastructure to mix desktops and clusters. With this large-scale Grid, we report experiments with both n-queens and flow-shop.
- Chapter 6 describes the deployment framework provided by ProActive. We also present some improvements of this framework: nodes localization (used by our branch-and-bound framework implementation to optimize communications), deployment of non-functional services (such as fault-tolerance or load-balancing),

and a mechanism to describe requirements of the application to deploy. Then, we present a load-balancing mechanism based on our peer-to-peer infrastructure.

- In Chapter 7, we give an overview of our current work and of the enhancements we foresee.
- Finally, Chapter 8 concludes and summarizes the major achievements of this thesis.

Chapter 2

Principles and Positioning

In this chapter, we justify our choice of a peer-to-peer system for acquiring computational resources on Grids, by evaluating existing Grid systems and peer-to-peer architectures. We also evaluate existing branch-and-bound frameworks for Grids and we point out what requirements are not addressed by those frameworks, to motivate our model for Grid branch-and-bound.

This chapter is organized as follows: first, we define Grid computing and we present an overview of Grid systems; then, we show a classification of peer-to-peer systems and we compare both approaches, Grid and peer-to-peer; next, we evaluate existing peer-to-peer systems for Grids and from this analysis we point out what requirements are needed to provide a peer-to-peer infrastructure for Grids.

The chapter finishes with a presentation of branch-and-bound, which is an algorithmic technique for solving combinatorial optimization problems, and we discuss existing frameworks for Grids. Based on this discussion, we show the main properties that our Grid branch-and-bound framework requires.

Finally, we describe the active object model and the ProActive middleware that we based our work on.

2.1 Grid: Principles and Positioning

In this section, we define the concept of the Grid and then we position our contribution with respect to general Grid architecture.

2.1.1 Principles of Grid computing

With the recent explosion of interest in the Grid from industry and academic, it is now harder to precisely define the concept of Grid. We read about Computational Grids, Data Grids, Campus Grids, Cluster Grids, Bio Grids, and even Equipment Grids. Hence, in this section we define the Grid within the context of the work done in this thesis.

Grid computing was first popularized with the book *"The Grid: Blueprint for a New Computing Infrastructure"* [FOS 98]. The concept of Grid has since matured and may now be defined as follows [LAS 05]:

Definition 2.1.1 *A production Grid is a shared computing infrastructure of hardware, software, and knowledge resources that allows the coordinated resource sharing and problem solving in dynamic, multi-institutional virtual organizations to enable sophisticated international scientific and business-oriented collaborations.*

Virtual organizations are defined in the Grid foundation paper [FOS 01] as:

Definition 2.1.2 *Virtual organizations enable disparate groups of organizations and/or individuals to share resources in a controlled fashion, so that members may collaborate to achieve a shared goal.*

In other words, Grids gather large amounts of heterogeneous resources across geographically distributed sites for use by virtual organizations. Resources are usually organized in groups of desktop machines and/or clusters, which are managed by different administrative domains (labs, universities, companies, etc.). Virtual organizations are then set up following discussions and agreements between involved partners.

Grids introduce new challenges such as:

- *Distribution*: resources are distributed on different sub-networks with different bandwidth, sub-networks are connected to each other by shared nationwide networks, resulting in significant latency and bandwidth reduction.
- *Deployment*: the large amount of heterogeneous resources complicate the deployment task in terms of configuration and connection to remote resources. Deployment sites targeted may be specified beforehand, or automatically discovered at runtime.
- *Multiple administrative domains*: each domain may have its own management and security policies.
- *Scalability*: Grid applications aim to use a large number of resources, which are geographically distributed. Hence, Grid frameworks have to help applications with scalability issues, such as providing parallelism capabilities for a large number of resources.
- *Heterogeneity*: each computing centre is likely owned by a different institution. Thus, Grids gather resources from different hardware vendors, running with different operating systems, and relying on different network protocols. In contrast, each site is usually composed of homogeneous resources, often organized in a single cluster, which provides a high performance environment for computations and communications.
- *High performance and communication*: applications that target Grids, cannot be executed on a common dedicated computing infrastructures, such as clusters. These kind of applications aim to solve computation-hungry problems and are designed to efficiently use parallelism and communications.
- *Dynamicity*: applications may need more resources at runtime. Also, the large number of resources that are distributed on different domains implies a high probability of faults, such as hardware failures, network down time, or maintenance, all the Grid infrastructure, Grid applications, and users need to be aware of to respond appropriately.

- *Programming model*: a Grid gathers geographically disperse and administratively independent computational sites into a large federated computing system with common interfaces. New programming models have to be defined to abstract away these complexities from programmers.
- *Fault-tolerance*: Grids are composed of numerous heterogeneous machines and which are managed by different administrative domains, and the probability of having faulted nodes during an execution is not negligible.

Finally, Grid computing aims at providing transparent access to computing power and data storage from many heterogeneous resources in different geographical locations. This is called *virtualization of resources*.

2.1.2 Positioning with respect to general Grid architecture

In the context of this thesis, we aim to define a framework for solving optimization problems with the branch-and-bound technique (branch-and-bound is defined in Section 2.3.1). This framework has the goal of taking advantage of the large number of resources provided by Grids to minimize the computation time. In addition, another goal of this thesis is to offer a peer-to-peer network to provide a large-scale computational infrastructure (see the next Section).

First, from a user point of view, Grid application developers may be classified in three groups, as proposed in [GAN 02]. The most numerous group are *end users* who build packaged Grid applications by using simple graphical or Web interfaces; the second group is *programmers* that know how to build a Grid application by composing them from existing application ”components” and Grid services; the third group consists of *researchers* that build individual components of a distributed application, such as simulation programs or data analysis modules.

In this thesis, we essentially address the group of researchers, by providing them *a complete branch-and-bound framework for solving optimization problems*. We also address the second group, programmers, with *a peer-to-peer infrastructure for dynamically acquiring a large number of resources*.

Then, from the software point of view, the development and the execution of Grid applications involve four concepts: *virtual organizations*, *programming models*, *deployment*, and *execution environments*. All these concepts may be organized in a layered view of Grid software, shown in Figure 2.1.

- At the bottom, the *Grid fabric*, which is all resources gathered by the Grid. These resources consist of computers, databases, sensors, and specialized scientific instruments. They are accessed from operating systems (and virtual machines), network connection protocols (rsh, ssh, etc.), and cluster schedulers (PBS, LSF, OAR, etc.). At this layer, resources are federated through virtual organizations.
- Above, layer 2, is the *Grid middleware infrastructure*, which offers core services such as remote process management and supervision, information registration and discovery, security and resource reservation. Various frameworks are dedicated to these aspects. Some of them are global frameworks providing most of these services, such as the Globus toolkit [FOS 05] which includes software services and libraries for resource monitoring, discovery, and management, plus security and

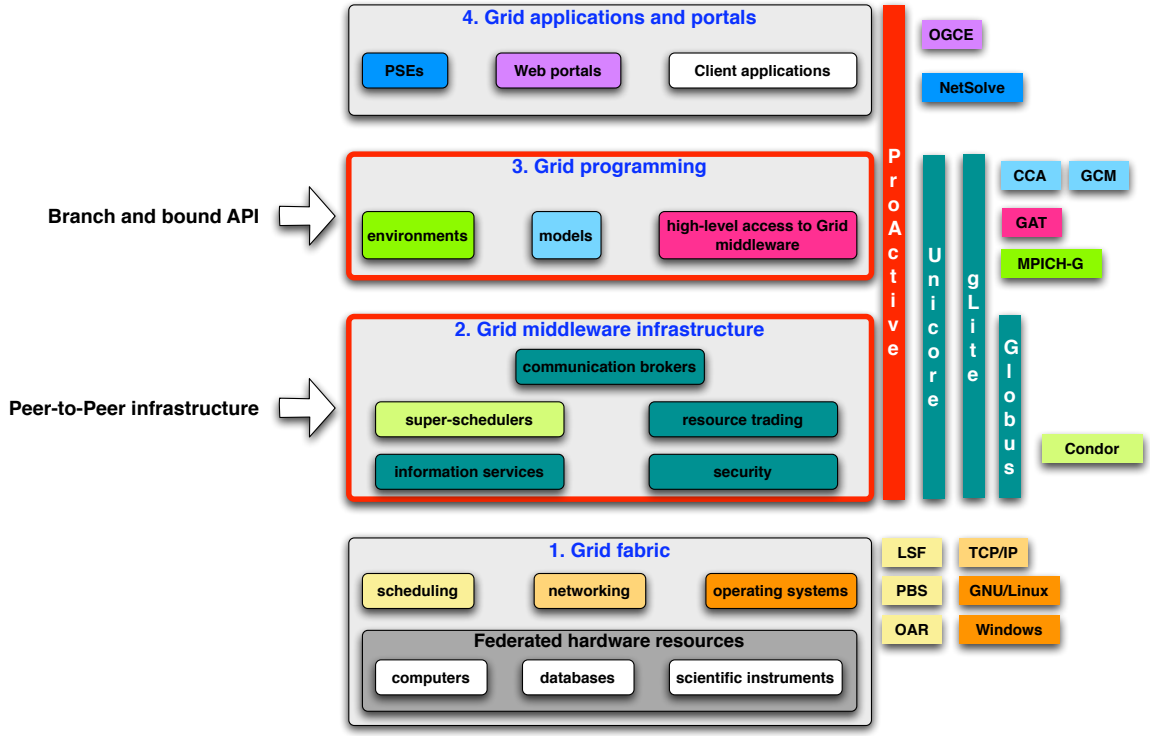

Figure 2.1: *Positioning of this thesis within the Grid layered architecture*

file management. Nimrod [Dav 95] (a specialized parametric modeling system), Ninf [SEK 96] (a programming framework with remote procedure calls) and Condor [THA 05] (a batch system) take advantage of the Globus toolkit to extend their capabilities towards Grid computing (appending the -G suffix to their name).
The deployment and execution environments for Grid entities are provided by the Grid middleware layer.

- Layer 3 is the *Grid Programming* layer, which includes programming models, tools, and environments. This layer eases interactions between application and middleware, such as Simple API for Grid Applications (SAGA) [GOO 05, GGF04] an initiative of the Global Grid Forum (GGF). This initiative is inspired by the Grid Application Toolkit (GAT) [ALL 05] and Globus-COG [LAS 01] which enhances the capabilities of the Globus Toolkit by providing workflows, control flows and task management at a high level of abstraction. As reported by some users [BLO 05] however, some of these abstractions still leave the programmer with the burden of dealing with explicit brokerage issues, sometimes forcing the use of literal host names in the application code.
  One of the goals of this thesis is to hide all these issues from the programmer by providing an abstraction of resources.
  This layer includes programming models, such as MPICH-G [KAR 03] a grid-enable implementation of MPI that use services from Globus Toolkit. Common Component Architecture [ARM 99] and Grid Component Model [OAS 06] are both programming models for the Grid based on the component paradigm.
- The top layer is the *Grid Application* layer, which contains applications developed

using the Grid programming layer. Web portals are also part of this layer, they allow users to control and monitor applications through web applications, such as OGCE [OGC] or GridSphere [Gri c]. The layer includes Problem Solving Environment (PSE), which are systems that provide all facilities needed to solve a specific class of problems, NetSolve/GridSolve [SEY 05] are complete Grid environment that help programmers for developing PSEs.

Some Grid middlewares cover more than a single layer, it is notably the case for Unicore [UNI] and gLite [LAU 04], which provide Grid programming features in addition of Grid middleware properties, for instance access to federated Grid resources, with services including resource management, scheduling and security. ProActive is also one of these, we describe in detail ProActive in Section 2.4.

The contributions of this thesis belong to the Grid programming layer and the Grid middleware infrastructure. The branch-and-bound framework is a Grid programming environment and the peer-to-peer infrastructure is a Grid middleware infrastructure (resource trading). Related work to branch-and-bound and to peer-to-peer are considered in more details in the next sections.

The academic community is gathering distributed resources to build Grids. Many of these Grids are now large infrastructure nationally distributed. Below we describe several of the most important of these projects:

- *EGEE* [EGE 04]: is an European project that aims to provide a production quality Grid infrastructure spanning more than 30 countries with over 150 sites to a myriad of applications from various scientific domains, including Earth Sciences, High Energy Physics, Bio-informatics, and Astrophysics.
- *TeraGrid* [TER]: is an open scientific discovery infrastructure combining leadership class resources at nine partner sites in USA to create an integrated, persistent computational resource.
- *Open Science Grid* (OSG) [GRI a]: is a USA Grid computing infrastructure that supports scientific computing via an open collaboration of science researchers, software developers and computing, storage and network providers.
- *Grid'5000* [CAP 05]: this project aims at building a highly reconfigurable, controllable and monitorable experimental Grid platform gathering nine sites geographically distributed in France. Most of this thesis experiments use this platform.
- *NorduGrid* [SMI 03]: is a Norway Grid research and development collaboration aiming at development, maintenance and support of the free Grid middleware, known as the Advance Resource Connector (ARC).
- *ChinaGrid* [JIN 04]: aims to provide a nationwide Grid computing platform and services for research and education purpose among 100 key universities in China.
- *Naregi* [NAR]: aims to conduct R&D on high-performance and scalable Grid middleware to provide a future computational infrastructure for scientific and engineering research in Japan.
- *DAS* [BAL 00]: The Distributed ASCI Supercomputer (DAS) is a wide-area distributed computers, distributed over four Dutch universities.

In addition to all these projects, global interaction between these infrastructures may also be achieved, resulting in even larger virtual organizations, such as during the

Grid Plugtests events [ETS 05b, ETS 05a, ETS 06].

Most current Grids have static infrastructures and use dedicated clusters. The inclusion of new machines to virtual organizations are scheduled and budgeted a long time before, *i.e.* it is impossible to dynamically add spare resources from a site to temporarily increase the computational power of the Grid. Nearly all these Grids are, for the moment, experimental platforms for helping researchers to prepare the next generation of platforms. NorduGrid, EGEE, and TeraGrid are production Grids.

Although these projects are called Grids their lack of dynamic infrastructure, among others to fulfill the given definition of Grid (see Definition 2.1.1 and Definition 2.1.2). In the context of this thesis, we propose using a peer-to-peer infrastructure to solve the issue of dynamically including new machines into a Grid. The next section presents peer-to-peer systems.

2.2 Peer-to-Peer: Principles and Positioning

Grid computing aims at building virtual organizations composed of shared resources from different institutions. Another concept which targets the same goal is *Peer-to-Peer*. In this section, we first define the notion of peer-to-peer, then we compare peer-to-peer and Grid computing, and finally we position our contribution with respect to peer-to-peer systems.

2.2.1 Principles of peer-to-peer systems

The first work on peer-to-peer (P2P) systems appeared in the early 1990s [SIM 91, YOU 93]. However, it was not until the early 2000s that P2P systems were popularized with Napster [NAP 99], a file sharing application, and SETI@home [AND 02], a CPU-cycle stealing application. The key idea of these applications is to take advantage of under-exploited desktop machines at the edges of the Internet.

In the distributed computing research community, there are many definitions about what P2P systems are: decentralized and non-hierarchical node organization [STO 03], or even much like a Grid from the book *"Peer-to-Peer: Harnessing the Power of Disruptive Technologies"* [ORA 01]:

> *Peer-to-Peer is a class of applications that take advantage of resources – storage, cycles, content, human presence – available at the edges of the Internet.*

Many of those definitions are similar to other distributed systems, such as client-server architecture or master-worker paradigm. Napster and SETI@home are both based on a centralized system, which is master-worker. E. Heymann *et al.* define the master-worker paradigm as follows [HEY 00]:

Definition 2.2.1 *The Master-Worker paradigm consists of two entities: a master and multiple workers. The master is responsible for decomposing the problem into small tasks (and distributes these tasks among a farm of worker processes), as well as for gathering the partial results in order to produce the final result of the computation. The worker processes execute in a very simple cycle: receive a message from the master with the next task, process the task, and send back the result to the master.*

Figure 2.2 show an example of master-worker architecture.

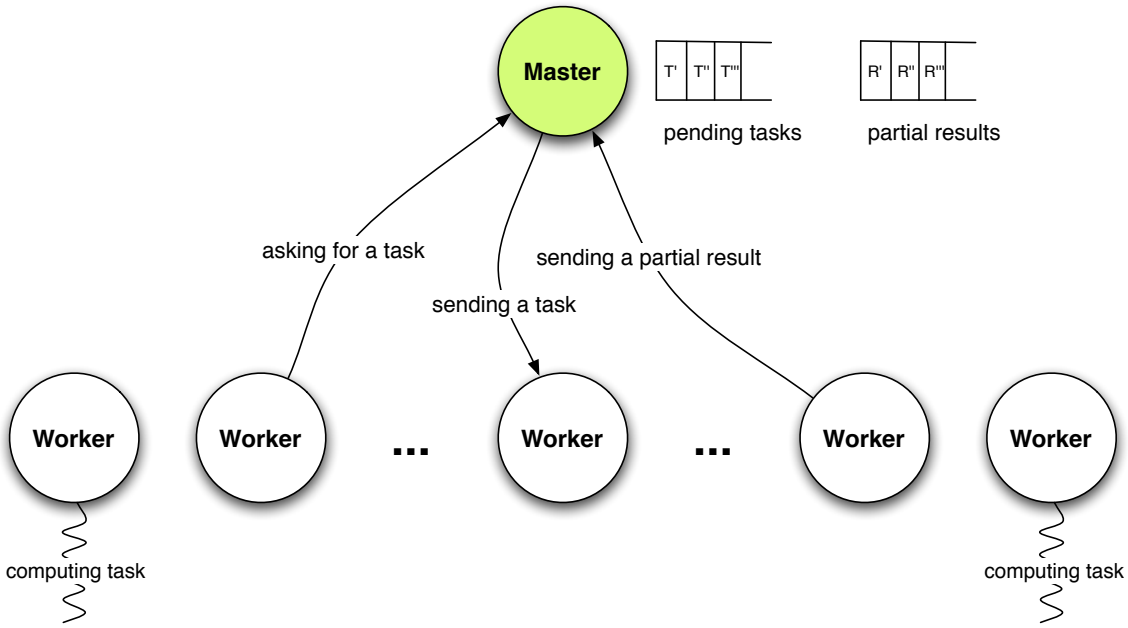

Figure 2.2: *Master-worker architecture*

In SETI@home, the master distributes computational tasks to workers, which are desktop machines at the edges of Internet. Then, workers send back results to the master that checks the computation for doctored results. For Napster, the master maintains a global list of all connected clients' files and when a client searches for a file the master connect it directly to the client which stores the requested file.

SETI@home and Napster are considered as the *first-generation* of P2P systems, based on a centralized approach to take advantage of desktops at the edges of the Internet.

The *second-generation* is based on *decentralized approach*, such as Gnutella [GNU 00], Freenet [CLA 00], and KaZaA [KAZ 00] which are all file sharing applications. These applications are the first to propose a new architecture different from master-worker. R. Schollmeier [SCH 01] defines peer-to-peer as:

Definition 2.2.2 *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, etc.). 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).*

The key ideas behind P2P are *sharing* resources, *point-to-point* direct communications between entities, and that all entities are at the same time *providers and requesters* of resources. The entities or participants of P2P networks are named *peers*.

In other words, a peer of a P2P network shares its resources with other peers and can directly access theirs shared resources.

In addition to defining P2P, Schollmeier [SCH 01] also proposes a classification of P2P systems. He distinguishes two types of P2P networks, which:

- *"Pure"* peer-to-peer network defined as:

  Definition 2.2.3 *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 Definition 2.2.2 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.*

- *"Hybrid"* peer-to-peer network defined as:

  Definition 2.2.4 *A distributed network architecture has to be classified as a "Hybrid" Peer-to-Peer network, if it is firstly a Peer-to-Peer network according to Definition 2.2.2 and secondly a central entity is necessary to provide parts of the offered network services.*

In contrast to the first-generation, pure P2P networks have no central entities. As shown by Figure 2.3, each peer maintains a list of connections to other peers, called neighbors or acquaintances. Due to the lack of structure, there is no information about the location of resources, therefore peers broadcast queries through the network, with a method called *flooding*. In this example, the peer A queries for the resource Y, it floods the network and then when resource Y is found, A directly accesses the requested resource hosted by peer B.

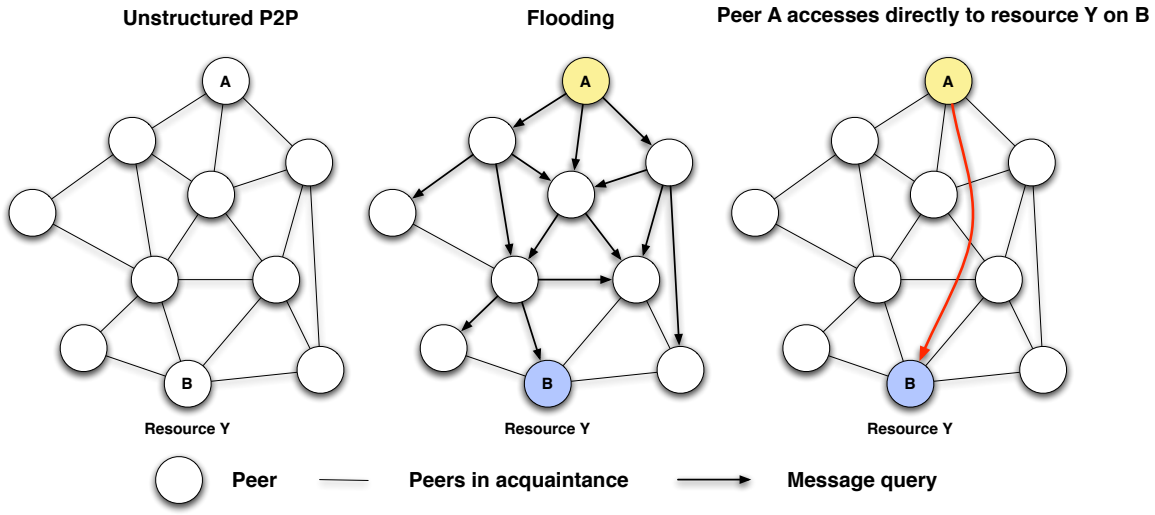

Figure 2.3: *Unstructured peer-to-peer network*

The drawback of pure P2P is that flooding generateslarge messaging volumes in order to find the desired resources. Ritter shows that Gnutella does not scale well [RIT]. In order to limit the cost of flooding many mechanisms have been proposed: Dynamic Querying [Dyn], which dynamically adjusts the TTL of queries; or by dynamically adapting the search algorithms [CHA 03].

Hybrid P2P combines the features of centralized P2P (master-worker) and pure P2P. It defines several super-peers. Each super-peer acts as a server to a small portion of network. Hybrid P2P has been used by Gnutella (version 0.6) to solve the flooding cost problem, by using central servers to maintain global lists of shared files.

Figure 2.4 shows an example of hybrid P2P, the peer A requests resource Y from a super-peer and the super-peer then floods the super-peer network to find that Y is hosted by peer B.

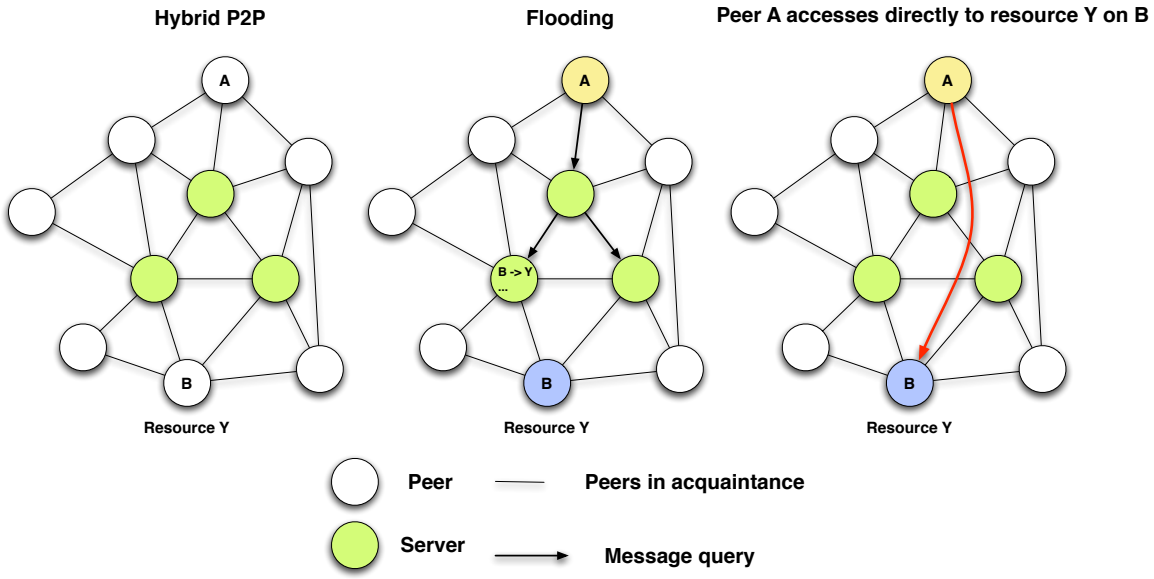

Figure 2.4: *Hybrid peer-to-peer network*

Compared to pure P2P, hybrid P2P reduces the message traffic and saves the bandwidth. More generally, all P2P systems based on super-peer network [YAN 03] are hybrid P2P, such as JXTA [SUN 01], which is a set of open protocols based on P2P.

Hybrid P2P reduces cost and has better scalability than pure P2P, however it is still a centralized approach. The *third-generation* of P2P aims to solve both scalability and centralization issues. This lastest generation is based on *distributed hash tables* (DHT) [STO 03].

DHTs do indeed enable pure P2P as defined by Definition 2.2.3, but they differ from the first-generation in the method they use to find resources on the network. DHTs do not use flooding, they organize peers in a structured network. DHTs are also often called *structured networks*. On the contrary, pure P2P networks of the second-generation that use flooding are called *unstructured networks*. The organization result of the network topology and/or the data distribution is that looking for a resource requires $O(\log n)$ steps, whereas in comparison unstructured P2P networks require $O(n)$ steps.

Figure 2.5 shows an example of a DHT based on Chord [STO 03]. Chord organizes peers on a ring, each peer has an identifier (ID) of m-bit size, and thus the ring cannot have more than 2^m peers. Each peer knows m other peers on the ring: successor finger i of n points to node at $n + 2^i$; acquaintances are stored in the finger table. A key is generated for the resource (typically a file), the key is also named the hashcode. The key is mapped to the first peer which ID is equal to or follows the key. Each peer is

responsible for $O(\frac{K}{N})$ keys, so $O(\frac{K}{N})$ keys must be reallocated when a peer joins or leaves the ring. In the figure the resource has a key $K54$, and the peer $N8$ has three hops to find the resource.

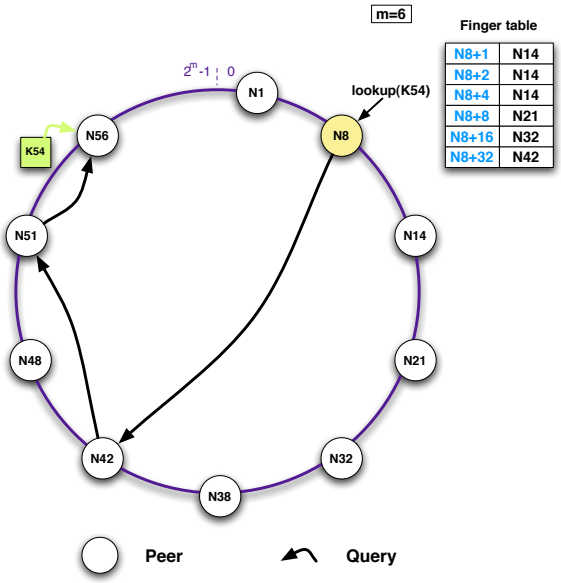

Figure 2.5: *Structured with DHT peer-to-peer network*

The ring topology is one technique for a DHT. It is also used by Pastry [ROW 01]. The Content-Addressable Network (CAN) [RAT 01] system uses another method, which consists in organizing peers in a virtual multi-dimensional cartesian coordinate space. The entire space is partitioned among all peers and each peer owns a zone in the overall space.

DHTs are now used by the most recent file sharing P2P applications, such as the popular BitTorrent [COH 03]. This lastest generation of P2P systems solves scalability and failure issues without introducing central points. However, DHTs have earned some criticisms for their high maintenance cost due to high churn [CHA 03], where the system has to discover failures, re-organize lost data and pointers, and then manage data re-organization when the failed peers return to the network.

In less than a decade, P2P systems have matured from centralized approaches to fully decentralized approaches. The original goal of P2P was to take advantage of under-utilized desktops located at the network edge. The goals have now shifted to manage scalability, failure, and performance of systems that are composed of a large number of peers.

Currently, P2P focuses on sharing resources, decentralization, and stability. However, current research aims at sharing data using, for example, DHTs. In the context of this thesis, we present a decentralized P2P infrastructure for sharing computational nodes.

Grids and P2P are both environments for large-scale applications, they also target similar goals, such as resource sharing and dynamicity. The next section compares in details P2P and Grid.

2.2.2 Comparing peer-to-peer and Grid computing

In the past few years, new approaches to distributed computing have emerged: Grid computing and peer-to-peer (P2P). Both approaches have a similar objective: the pooling and coordinated use of large sets of distributed resources. However, the two technologies are based on different user communities and, at least in their current designs, focus on different requirements.

The first difference between them is the user base, *i.e.* the community, as reported by [FOS 03, GOU 00a]. Current Grids provide many services to moderate-sized communities composed of scientists. In contrast, P2P usually provides a single service, typically file sharing, to a large community of Internet users.

As Figure 2.6 depicts, development of Grids and P2P are driven differently. In the P2P community, users demand applications, because P2P infrastructure are generally mono-application. The P2P application/service writer has to develop a new infrastructure or adapt an existing one for the new service. Grids are more adapted to provide multi-application/services on the same infrastructure, because infrastructures are built over standards, such as the Global Grid Forum (GGF) [FOR 04] and the Open Grid Services Architecture (OGSA) an integrated generic computational environment [FOS 02]. Thus Grid users, scientists, demand new services or applications from application writers, who can directly provide requested service or, if possible to implement, pass on to Grid infrastructure architects.

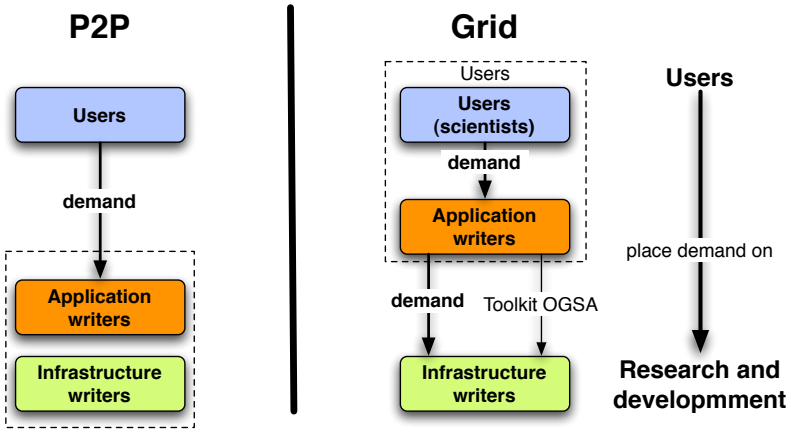

Figure 2.6: *Peer-to-Peer and Grid computing communities drive technology development*

Table 2.1 summarizes the basic differences between Grid computing and P2P. These differences have been reported by Ledlie *et al.* [GOU 00a], and by Foster *et al.* [FOS 03].

Although Grids and P2P systems emerged from different communities to serve different needs and provide different features, both constitute successful resource sharing paradigms. Ledlie *et al.* [GOU 00a] and by Foster *et al.* [FOS 03] argue that Grids and P2P will converge. They argue that they could each benefit from the other.

The *Peer-to-Peer Grid* [FOX 03] is one of the first that understood the benefit of mixing Grids and P2P concepts to provide a global computing infrastructure. Thereby, peer-to-peer Grid proposes mixing structured services (concept from Grids) with dynamic services (concept from P2P). The infrastructure is composed of services and provides collaborative services to users, hence the infrastructure is based on Web Services. Grids and P2P concepts are clearly separated: centralized servers are used at the core of the

Table 2.1: *The basic differences between Grid computing and P2P*

In terms of ...	Grid computing	Peer-to-Peer
Users	Scientific community	Any Internet users
Computing	Dedicated clusters and large SMP machines	Common desktop machines
Network	High speed dedicated network	Internet TCP connections
Administration	Centralized and hierarchical	Distributed
Applications	Complex scientific applications, large-scale simulations, data analysis, *etc.*	File sharing, CPU-cycle stealing, real-time data streaming, *etc.*
Scalability	Small number of specialized sites (moderate size)	Any desktop at the edges of Internet (large size)
Security	Secure services for job submission and interactive execution (high)	Poor
Participation	Static and take time to add new nodes	High churn
Trust	Identified and trusted users	Anonymous users, untrusted
Standards	GGF, OGSA, Web Services	No standard

infrastructure (as Grids) and machines in labs are organized as a P2P network.

Grids are based on centralized approaches and resources are statically shared, *i.e.* resources are shared between institutions but are dedicated to the Grid. Also, the inclusion of new resources in the infrastructure needs previous discussions and agreements between Grids members. Furthermore, Grid resources are usually high performance environments, such as clusters. Thus, many of current Grid infrastructures, such as Grid'5000, do not completely fulfill the Definition 2.1.1.

On the contrary, P2P networks are decentralized and can handle dynamic inclusion of resources. Unlike Grids, current P2P applications, such as Gnutella, focus on file sharing and desktop machines. In the context of this thesis, we propose a P2P infrastructure for building large-scale Grids. This infrastructure handles the dynamic inclusion of both desktop and cluster resources. Unlike the other P2P infrastructures, the proposed one is not dedicated to file sharing but it is dedicated for computing.

In the next section, we give an overview of existing approaches that can be considered as P2P infrastructures for Grids.

2.2.3 Overview of existing peer-to-peer systems

The main goal of Grids is to provide large pools of heterogeneous resources gathered in virtual organizations. Resources are geographically distributed around the world.

In this section, we present some P2P systems for computation, which fulfill both definitions of Grid and P2P. First, we present global computing platforms that are first P2P generation. Then, we introduce decentralized P2P systems.

2.2.3.1 Global computing platforms

Global computing platforms, also named *Volunteer computing* or even *Edge computing*, aim at using desktop computers that are located at the edges of the Internet. With the recent explosion of CPU performance and easy access to the Internet in industrialized countries, millions of desktop machines are inter-connected and are under-exploited by their owners. Global computing consists of a technique, named *cycle stealing*, which uses idle CPU cycles of desktop machines connected to the Internet. These cycles would otherwise be wasted at night, during lunch, or even in the scattered seconds throughout the day when the computer is waiting for user input or slow devices.

Global computing was popularized beginning in 1997 by *distributed.net* [dis 97] and later in 1999 by *SETI@home* to harness the power of desktops, in order to solve CPU-intensive research problems. The main idea of these projects is to grow a community of users around the world donating the power of their desktops to academic research and public-interest projects. For instance, distributed.net started an effort to break the RC5-56 portion of the RSA secret-key challenge, a 56-bit encryption algorithm that had a $10,000 USD prize available to anyone who could find the key.

In the rest of this section we describe the famous SETI@home platform and the generalization of its platforms with BOINC and XtremWeb.

SETI (Search for Extra-Terrestrial Intelligence) is a set of research projects that aim to explore, understand and explain the origin, nature and prevalence of life in the universe. One of these projects is SETI@home [AND 02], which is a scientific experiment that uses Internet-connected computers to analyze radio telescope data.

SETI@home is based on a centralized approach, master-worker (see Definition 2.2.1). A radio telescope records signals from space. Next, recorded data are divided into work units, with each piece being sent to workers.

Workers are desktops located at the edges of the Internet, and each work unit is processed by the worker during its idle time. The result of the work unit is then sent back to the server and the worker gets a new work unit. The worker is deployed via a screen-saver that Internet users can install on their desktop. This screen-saver approach is an important part of the SETI@home success.

In 2002, SETI@home was considered the world's fastest supercomputer with a total of 27 TeraFLOPS [AND 02]. Now, with over 1.36 million computers in the system, as of March 2007, SETI@home has the ability to compute over 265 TeraFLOPS [ZUT]. The Lawrence Livermore National Laboratory (LLNL) currently hosts the most powerful supercomputer [PRO 07], which has a theoretical power of 367 TeraFLOPS and a measured power of 280 TeraFLOPS, as reported by the TOP500 [SIT] project, which ranks and details the 500 most powerful publicly-known computer systems in the world.

BOINC As a result of the success of the SETI@home platform, the platform has been open to other scientific projects such as mathematics, medicine, molecular biology, climatology, and astrophysics. The platform is now known as the *Berkeley Open Infrastructure for Network Computing* (BOINC) [AND 04] and makes it possible for researchers to use the enormous processing power of personal computers around the world. SETI@home is now one of the applications of the BOINC platform. This platform can be considered as the most powerful supercomputer with a total of 536 TeraFLOPS [ZUT].

XtremWeb [GIL 01] is another global computing platform. The main goal of this platform is to provide a generic system for volunteer computing. XtremWeb provides a complete API to develop applications. Participants contribute in two different ways: they may be *users* by submitting jobs to the platform, or they may be *providers* by sharing their idle cpu-cycles with the rest of the system.

The XtremWeb platform aims principally to run master-worker applications. Users submit jobs to master and workers download tasks from a server, then when tasks are completed, results are sent back to the server. Thus, users have no direct control on workers. Furthermore, users have to provide different compiled versions of their tasks to be able to use workers, running on different architectures, different operating systems, *etc.*

The master may be not a single entity running on a single machine, it may be distributed on several machines to reduce bottlenecks. For instance, the interface with users can run on a separate machine than the machine which runs the task allocation, and a third machine can handle worker monitoring and fault-tolerance.

Communications rely on the Remote Procedure Call (RPC) model. In order to be able to pass through firewalls, users and workers initiate all communications to the master, *e.g.* job submission and tasks requests.

XtremWeb provides a generic and complete platform for scientific applications relying on global computing platforms.

Condor [CON 93] is a specialized workload management system for compute-intensive jobs. Like other full-featured batch systems, Condor provides a job queueing mechanism, scheduling policy, priority scheme, resource monitoring, and resource management. Condor can be used to build Grid-style computing environments that cross administrative boundaries. Condor allows multiple Condor compute installations to work together. It also incorporates many of the emerging Grid-based computing methodologies and protocols. For instance, Condor-G [FRE 01] is fully interoperable with resources managed by Globus. Condor uses a mechanism called *matchmaking* for managing resources between owners and users. This mechanism is mainly based on a matchmaking algorithm, which compares two descriptors called *classified advertisements* (ClassAds); one describes the task and one describes the compute node, *i.e.* there are two kinds of ClassAds: Jobs ClassAd and Machines ClassAd. Users have to provide this Job ClassAd with the query description of which resources are needed. Condor is well suited for deploying batch jobs, which need to run on specific resources, such as on a given architecture type.

Beside the large power that global computing can gather, resources are shared with their normal owners, thus desktops can be available for a small part of time. Global computing platforms have to manage high volatility.

2.2.3.2 Peer-to-Peer platforms

In addition to global computing platforms, which are considered the first generation of P2P systems and centralized, we now present decentralized systems, such as pure or hybrid P2P. Thus, these systems are considered the second generation of P2P architectures.

OurGrid [AND 05a] is a complete solution for running Bag-of-Tasks applications on computational Grids. These applications are parallel applications whose tasks are independent. Furthermore, OurGrid is a cooperative Grid in which labs donate their idle computational resources in exchange for accessing other labs idle resources when needed. It federates both desktops and dedicated clusters. The architecture of this middleware is composed of three main components: *MyGrid*, a scheduler, which is the central point of the Grid and provides all the necessary support to describe, execute, and monitor applications; *Peers* organize and provide machines that belong to the same administrative domain (*i.e.* they are machine providers for the scheduler); and *User Agents* which run on each Grid machine and provide access to the functionality of the machine (User Agents are registered in Peers). As with Condor, OurGrid users have to provide a Job descriptor file, which queries resources and describes all tasks of the application to deploy. Hence, OurGrid is an hybrid P2P network.

JXTA Project [SUN 01] is a set of open protocols that allow any connected device on the network ranging from cell phones and wireless PDAs to PCs and servers to communicate and collaborate in a P2P manner. JXTA creates a virtual network where any peer can interact with other peers and resources directly even when some of the peers and resources are behind firewalls and NATs or are relying on different network transports. JXTA is a low-level specification/system for P2P communications. The communication protocol is based on an XML data structure. Indeed all communications are handled by the JXTA P2P network in order to pass through firewalls. JXTA can be seen as a network layer, like TCP/IP sockets, on which developers build their applications. The core of JXTA network is composed of super-peers that manage the rest of the infrastructure, thus JXTA is a hybrid P2P systems.

JaceP2P [BAH 06] is a decentralized framework for running parallel iterative asynchronous applications on Grids.

A parallel iterative asynchronous algorithm is a technique that solves problems by iterations, rather than doing it in one step. The technique proceeds by successively refining a solution. In parallel execution, communications must be performed between processes after each iterations in order to satisfy all the computing dependencies. Synchronizations can be done in an asynchronous manner to hide idle times required by sharing dependencies.

The JaceP2P architecture is a hybrid P2P system, where super-peers manage the introduction of new peers in the network. Peers are able to directly communicat between each other, asynchronously, for exchanging data and results while the computation continues. The framework focuses on fault-tolerance issues of iterative asynchronous applications by proposing a mechanism based on check-points, which are shared between peers. JaceP2P provides an API to implement iterative asynchronous applications and a P2P infrastructure designed to run these applications.

2.2.3.3 Synthesis

Table 2.2 summarizes and compares the main P2P infrastructures. The table is split in two parts: the first for global computing platforms, and the second for decentralized platforms.

We choose to compare these systems by selecting several criteria, which are the P2P architecture (*Kind*), the type of shared resources (*Resources*), the use of direct or point-

to-point communication between peers (*PtP com.*), and which kind of applications can be deployed (*Applications*).

Table 2.2: *Evaluation of the general properties of evaluated systems*

Systems	Kind	Resources	PtP com.	Applications
BOINC	Centralized	Desktops	no	Scientific Applications
XtremWeb	Centralized	Desktops	no	Scientific Applications
Condor	Centralized	Desktops & Clusters	no	Bag-of-Tasks
OurGrid	Hybrid P2P	Desktops & Clusters	no	Bag-of-Tasks
JXTA	Hybrid P2P	Everything	yes*	Collaborative/Sharing
JaceP2P	Hybrid P2P	Desktops	yes	Iterative algorithm

** communications are handled by the JXTA network, thus the communication is not direct between peers and may pass by unknown hops.*

All global computing platforms are centralized and principally target desktops located at the Internet edge. BOINC has been historically designed for a single application, SETI, and then generalized for embarrassingly parallel applications.

Providing point-to-point communication between peers is important for performance and scalability issues. Only JXTA and JaceP2P, which are hybrid P2P, provide communication without passing through a center point, even if JXTA does not allow direct communications.

Furthermore, only JXTA is able to deploy any kind of applications, others handle bag-of-tasks applications. Bag-of-tasks applications are composed of independent tasks that do not need to communicate with each other to complete their computation. For OurGrid, a task may be a parallel application written in MPI for instance, that requires a cluster to be executed.

No third generation, DHT, frameworks are presented here, because they focus on data sharing and generally for sharing anything than may be represented by a hash. The goal is to minimize the number of message to find the peer that owns the resource represented by the hash. These systems are not designed for CPU intensive applications, such as combinatorial optimization problems targeted by this thesis.

2.2.4 Requirements for computational peer-to-peer infrastructure

In the context of this thesis, we aim to propose a framework based on the branch-and-bound technique to solve combinatorial optimization problems. The distinguishing feature of this work is the ability to incorporate large number of dynamic heterogeneous resources that Grids can provide.

We previously presented the Grid layered architecture; with that model, the framework developed here is at Layer 3: Grid programming, and relies on Layer 2: Grid middleware infrastructure. In the previous section, we presented some Grid infrastructures and have identified some missing requirements. We also introduced P2P concepts and argued that P2P and Grids have similar goals: managing and providing a large pool of dynamic resources.

With all these observations, we now define requirements for a P2P infrastructure that is able to manage and provide a large number of computational resources. In other words, we propose a new Grid infrastructure relying on P2P concepts.

Specific Grid requirements We previously introduced several challenges for Grids. Here, we outline how our infrastructure addresses those challenges:

- *Distribution*: the infrastructure has to take into account significant latency and bandwidth reduction. To cover these issues, the P2P infrastructure may use asynchronous communication between peers. Thus, peers are not blocked while exchanging messages.
- *Deployment*: the large amount of heterogeneous resources complicate the deployment task in terms of configuration and connection to remote resources. All deployments problems must be hidden by the infrastructure. Also, the infrastructure has to provide a simple and unified way for acquiring resources.
- *Multiple administrative domains*: the infrastructure has to be flexible and easily adaptable to handle different management and security policies.
- *Scalability*: P2P already demonstrated the capabilities of scalability through a decentralized overlay network.
- *Heterogeneity*: using Java as programming language simplifies the problem of heterogeneity.
- *High performance*: resources are managed by the infrastructure, but once applications acquire them, the infrastructure has to provide full control to the application, *e.g.* communications between two application nodes may not be handle by the infrastructure but must be direct.
- *Dynamicity*: the topology of Grids is dynamic, resources are volatiles. Hence, the infrastructure must be dynamic.
- *Programming model*: the access to resources and the P2P infrastructure has to be as easy as possible for user.
- *Fault-tolerance*: the infrastructure has to handle resource failure in order to do fault the whole infrastructure.

Now, we detail more P2P related requirements.

Specific P2P requirements In this thesis, P2P follows the definition of *Pure Peer-to-Peer* as defined by Definition 2.2.3, meaning that it focuses on sharing resources, decentralization, and stability. The most important point of that definition is that a single arbitrarily chosen peer can be removed from the network without having the network suffering any loss of network service. However, removing a peer of the network leads to a linear degradation of the service provided by the network, *e.g.* having n less peers has effect of having n fewer available computational resources to improve the application speed.

The objective here is to provide a Grid infrastructure for computations. Existing Grid infrastructures do not address the dynamic aspect of virtual organizations. Also, for most of them it is harder to combine several different Grids into a bigger one. On the other hand, P2P infrastructures are designed to manage dynamicity and share similar objectives to Grids, which is providing virtual organizations. However, P2P are mono-application and P2P architectures vary according to their for different goals. The first generation allowed only master-worker applications without communication between workers; the second and the third generation are widely used for file sharing.

Although DHTs avoid flooding, this approach cannot be used here. The main goal of our infrastructure is to provide computational nodes to applications, where applications ask for a desired number of nodes and then the infrastructure broadcasts the request to find any available nodes to satisfy the request. Here queries do not target a specified resource, such as data identified by a hash, but instead try to find a number of available peers. In addition, DHTs earned some criticisms for their high maintenance cost with high churn [CHA 03], where the system has to discover failures, re-organize lost data and pointers, and then manage data re-organization when the failed peers return to the network. Unlike DHTs, our infrastructure is an unstructured P2P network.

Like other unstructured approaches we propose that each peer maintains a list of acquaintances and also some modifications to the basic flooding protocol to make the system scalable. Unlike all these systems we do not focus on data sharing, instead we propose some modifications in order to adapt the flooding to find available nodes for computation.

Unlike global computing platforms (BOINC/SETI@home and XtremWeb), Condor, and OurGrid, we do not provide any job schedulers. Applications connect to our P2P infrastructure and request nodes. Nodes are then returned in a best-effort way by the infrastructure. Unlike the others, applications dynamically interact with the infrastructure to obtain new nodes. The infrastructure works as a dynamic pool of resources. Once applications get nodes, there are no restrictions on how they are used. This property allows applications to communicate easily in arbitrary ways. Application communications are not handled by our infrastructure, unlike other P2P networks. Most other P2P infrastructures require the overlay network to be used for all communications. This limitation is highlighted by JXTA, which has very poor communication performance [HAL 03]. With our approach, applications can freely use different communications transport layers.

One of the problems raised by P2P infrastructures is the issue of crossing firewalls. Many choose to disallow communications, or another strategy is to always use the same way (worker to master), and even routing communications between peers. Our infrastructure must address this issue. A solution may to use direct communication between peers when it is allowed and to rely on forwarders at the level of firewalls.

Summary of requirements Infrastructure requirements:

- **Asynchronous communication** between peers
- **Simple and unified** way for acquiring resources
- **Flexible** and easily **adaptable** to handle sites
- **Heterogeneity**: Java and the inclusion of interface with most used Grid middlewares/infrastructures/schedulers
- **Resources usability**: applications must have the full usage of acquired resources
- **Dynamicity**: unstructured P2P network
- **Crossing firewalls**

With all these requirements, we propose, in this thesis, a Grid infrastructure based on a P2P architecture. This infrastructure is more than a common Grid infrastructure because it may include resources from desktops, clusters, and other Grid infrastructures. In other words, we propose a meta-Grid infrastructure.

Furthermore, our P2P infrastructure is the bottom layer, Grid infrastructure, of our branch-and-bound framework.

2.3 Branch-and-Bound: Principles and Positioning

The branch-and-bound algorithm is a technique for solving search problems and NP-hard optimization problems, such as the *traveling salesman problem* [LAW 85] or *job shop scheduling problem* [APP 91]. Branch-and-bound aims to find the optimal solution and to prove that no better one exist.

In this section, we define branch-and-bound and especially parallel branch-and-bound as applied to Grids. Then, we position our contribution with respect to branch-and-bound systems.

2.3.1 Principles of branch-and-bound algorithm

The branch-and-bound (B&B) method was first introduced by A. H. Land and A. G. Doig in 1960 [LAN 60] for solving *discrete programming problems*. The discrete optimization problems are problems in which the decision variables assume discrete values from a specified set; when this set is a set of integers, it is an *integer programming problem*. In 1965, the B&B technique was improved by R. J. Dakin [DAK 65] for solving integer programming problem.

B&B is also used to solve *combinatorial optimization problems* [PAP 98]. The combinatorial optimization problems are problems of choosing the best combination out of all possible combinations.

From the work of [LAN 60, DAK 65, PAP 98], we define B&B as:

Definition 2.3.1 *Branch-and-bound is an algorithmic technique for solving discrete and combinatorial optimization problems. This technique proceeds to a partial enumeration of all feasible solutions and returns the guaranteed optimal solution.*

In other words, B&B is a technique to find the optimal solution by keeping the best solution found so far. If a partial solution cannot improve the best, it is abandoned. B&B guarantees that the find solution is the best and none are better.

B&B algorithm proceeds to split the original problem into sub-problems of smaller sizes, *i.e.* the set of feasible solutions is partitioned. Then, the *objective function* computes the lower/upper bounds for each sub-problem. The objective function, or quality function, or even goal function, can be defined as [ATA 99]:

Definition 2.3.2 *The objective function constitutes the implementation of the problem to be solved (also referred to as search space). The input parameter is the search space. The output is the objective value representing the evaluation/quality of the individual. In a multi-criteria optimization problem, the objective function returns multiple objective values. The objective value is often referred to as fitness.*

Thus, for an optimization problem the objective function determines how good a solution is, *e.g.* the total cost of edges in a solution to a traveling salesman problem.

B&B organizes the problem as a tree of sub-problems, called *search tree* or *solution tree*, *i.e.* the search tree is a representation of the search space. The root node of this tree is the original problem and the rest of the tree is dynamically constructed by sequencing two operations: *branching* and *bounding*.

Branching consists in recursively splitting the original problem in sub-problems.

Each node of the tree is a sub-problem and has as ancestor a branched super-problem. Thereby, the original problem is the parent of all sub-problems: it is named the root node.

Bounding computes for each tree node the lower/upper bounds.

The upper bound is the worst value for the potential optimal solution, and the lower bound is the best value. Therefore, if V is the optimal solution for a given problem and $f(x)$ the objective function of this problem, then $lower\ bound \leq f(V) \leq upper\ bound$.

Furthermore, a *global optimal bound* is dynamically maintained for the whole tree. The optimal bound can be the best current lower or upper bound, depending on whether the problem aims to maximize or to minimize the solution. In this thesis, we assume that problems minimize the function. Hence, for us the global optimal bound is a a *global upper bound* (GUB), which is the best upper bound of all tree's nodes.

Thus, nodes with a lower bound higher than GUB can be eliminated from the tree because branching these sub-problems does not lead to the optimal solution, this action is called *pruning*.

The pruning operation helps to reduce the size of the space search to a computationally manageable size. Pruning removes only branches, where we are sure that the optimal solution cannot be at any one of these nodes. Thus, B&B is not a heuristic, or approximating procedure, but it is an exact optimizing procedure that finds an optimal solution.

Note that the GUB changes during the exploration of the search space, because the search tree is dynamically generated. Thus, GUB is improved until the optimal solution is found. Figure 2.7 shows an example of search tree.

Termination condition The last point about B&B is the termination condition. The termination happens when the problem is solved, *i.e.* the optimal solution is found. B&B terminates when all sub-problems have been explored or eliminated (pruned).

In the next section, we illustrate the B&B algorithm with a concrete example.

2.3.1.1 Solving the traveling salesman problem with branch-and-bound

The traveling salesman problem [LAW 85] (TSP) is a problem in combinatorial optimization. TSP is also member of the NP complete [PAP 77] class. In this section, we briefly show how to solve TSP with branch-and-bound.

The TSP problem Given a number of cities and the costs of traveling from any city to any other city, what is the cheapest round-trip route that visits each city exactly once

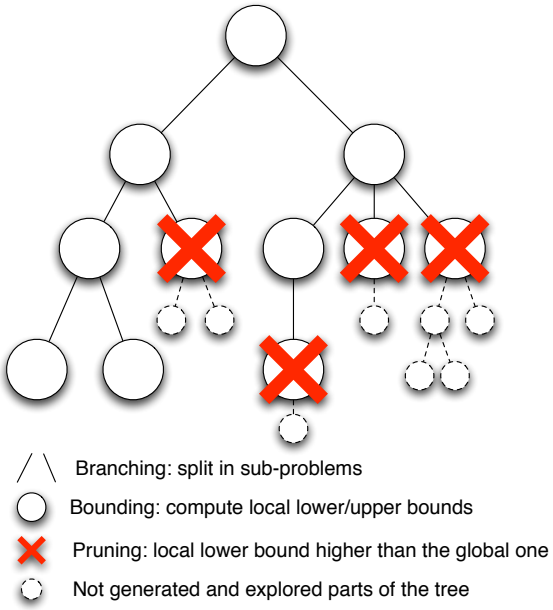

Figure 2.7: *A branch-and-bound search tree example*

and then returns to the starting city?

In other words, TSP consists in figuring out a Hamiltonian cycle with the least weight. A Hamiltonian cycle [ORE 60] is a cycle in an undirected graph which visits each vertex exactly once and also returns to the starting vertex.

The size of the solution space is $\frac{(n-1)!}{2}$, where n is the number of cities. This is also the total number of cycles with n different nodes that one can form in a complete graph of n nodes.

In this example, we deal with the symmetric TSP. Symmetric TSP is defined as follows: for any two cities A and B, the distance from A to B is the same as that from B to A.

Figure 2.8 shows the complete graph with five cities that we use in this example. The start and finish city is A.

This problem has to minimize an objective function. In order to simplify this example, we assume that there is a method for getting a lower bound on the cost of any solution among those in the set of solutions represented by some subset. If the best solution found so far costs less than the lower bound for this subset, we do not need to explore this subset at all.

Let S be some subset of solution. Let:

- $L(S) =$ a lower bound on the cost of any solution belonging to S.
- $C(route) =$ the cost of the given route.
- $BS =$ cost of the best solution found so far.
- If $BS \leq L(S)$, there is no need to explore S because it does not contain any better solution.
- If $BS > L(S)$, then we need to explore S because it may contain a better solution.

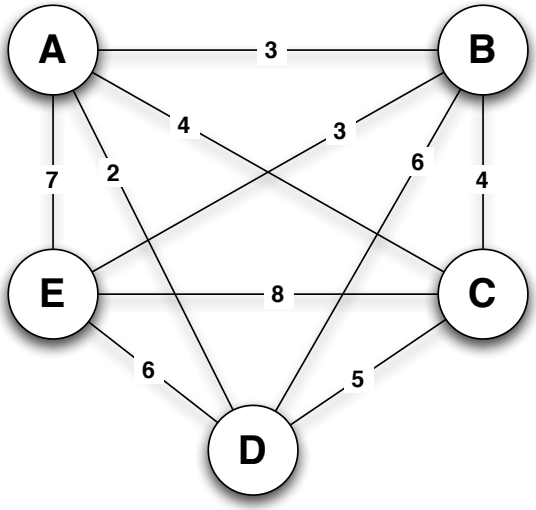

Figure 2.8: *TSP example of a complete graph with five cities*

Figure 2.9 shows an example of a binary solution tree for this example of TSP. This is one way to explore the solution space of this given TSP instance.

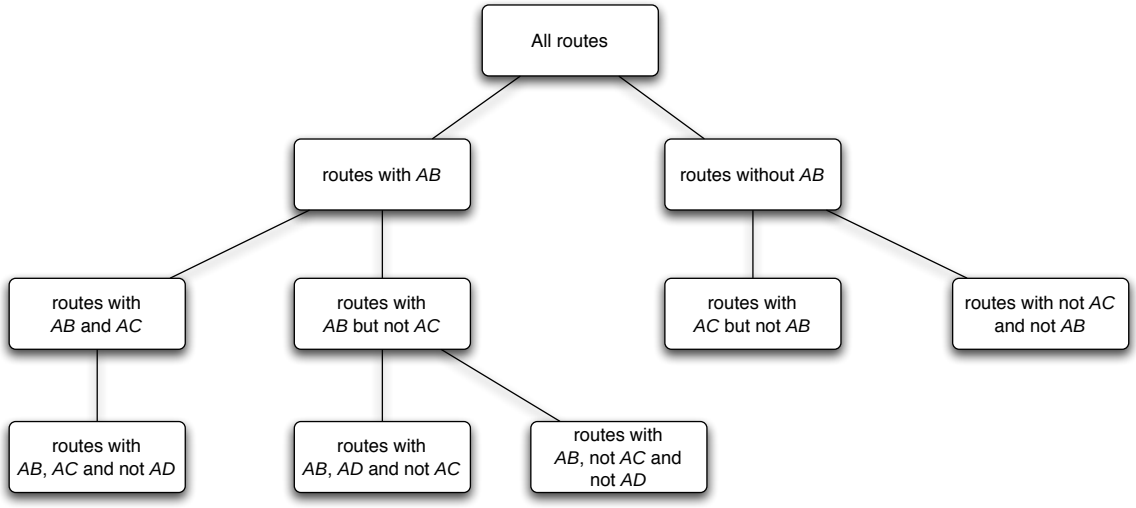

Figure 2.9: *A solution tree for a TSP instance*

Figure 2.10 shows branch-and-bound applied to this instance of TSP. When the branching is done, the lower bounds of both children is computed. If the lower bound for a child is as high or higher than the lowest cost found so far for BS, that child is pruned and its descendants are not constructed.

2.3.1.2 Parallel branch-and-bound

Because of the large size of typical handled problems (enumeration size and/or NP-hard class), finding an optimal solution for a problem can be impossible on a single machine.

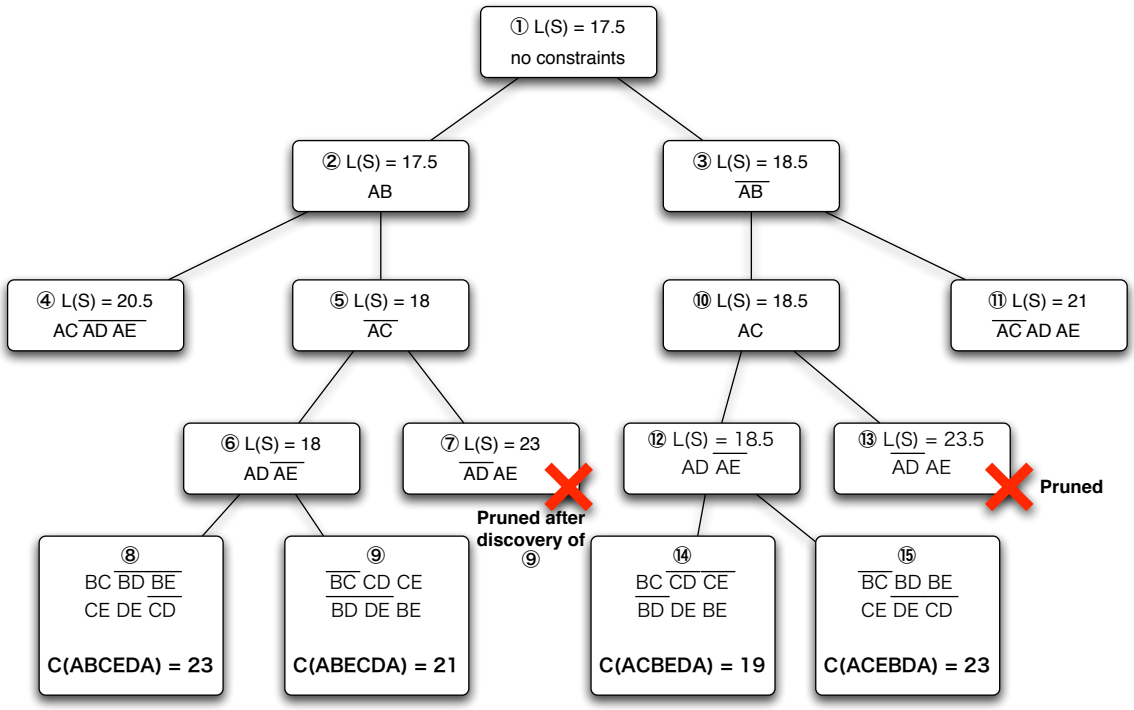

Figure 2.10: *Branch-and-bound applied to a TSP instance*

However, many studies show that it is relatively easy to design parallel B&B algorithms.

The survey [GEN 94] reports three main approaches in designing parallel B&B algorithms:

1. **Parallelism of type 1** (or *node-based*) introduces parallelism when performing the operations on generated sub-problems. For instance, it consists of executing the bounding operation in parallel for each sub-problem to accelerate the execution.
   Thus, this type of parallelism has no influence on the general structure of the B&B algorithm and is particular to the problem to be solved.
2. **Parallelism of type 2** (or *tree-based*) consists of building the solution tree in parallel by performing operations on several sub-problems simultaneously.
   Hence, this type of parallelism may affect the design of the algorithm.
3. **Parallelism of type 3** (or *multisearch*) implies that several solution trees are generated in parallel. The trees are characterized by different operations (branching, bounding, and pruning), and the information generated when building one tree can be used for the construction of another.
   Like approach 2, this type of parallelism may affect the design of the algorithm.

More recently, another survey [CRA 06] completes and updates this previous one. The survey reports that approaches of type 1 and 2 are now considered as classics. Other strategies are also described, such as *domain decomposition*, which decomposes the feasible domain and uses B&B to address the problem on each of the components of the partition.

These strategies are not mutually incompatible. For instance, when problem instances are particularly large and hard, several strategies may be combined to improve the problem's resolution speedup.

From all these approaches, the tree-based strategy (type 2) is the one that has had the most important research effort. Issues related to this strategy are well defined by [AUT 95]:

- Tasks are dynamically generated.
- The solution tree is not known beforehand.
- No part of the tree may be estimated at compilation.
- Task allocations to processors must be done dynamically.
- Distributing issues, such as load-balancing and sharing information.

Much previous work deals with parallel B&B as reported in [GEN 94, CRA 06]. The following some examples of frameworks for parallel B&B:

- PUBB [SHI 95]
- BOB++ [CUN 94]
- PPBB-Lib [TSC 96]
- PICO [ECK 00]
- MallBa [ALB 02]
- ZRAM [BRÜ 99]
- ALPS/BiCePS [XU 05]
- Metacomputing MW [GOU 00b]
- Symphony [RAL 04]

These frameworks differ from the kind of problems that they are able to solve, type of parallelism, programming language API, and targeted execution machines. Table 2.3 reports these differences.

These main differences are organized as follow: algorithms, parallel architecture, and targeted machines.

Algorithms represents the B&B technique used by the framework, *low-level* is a divide-and-conquer approach applied to explore the solution tree, *basic B&B* is the tree-based approach as previously described, *mixed-integer linear programming (LP)* uses linear programming for bounding, and *branch&price&cut* are a specialization of B&B for solving mixed-integer linear problems.

Parallelization is the parallel architecture that implements frameworks, *master-worker* (see Definition 2.2.1), *hierarchical master-worker* is a master-worker with sub-masters to manage scalability, and *single program multiple data (SPMD)* is a parallelism technique that consists to have the same subroutine running on all processors but operate on different data.

Table 2.3: *Summary of main parallel branch-and-bound frameworks*

Frameworks	**Algorithms**	**Parallelization**	**Machines**
PUBB	Low-level Basic B&B	SPMD	Cluster/PVM
BOB++	Low-level Basic B&B	SPMD	Cluster/MPI
PPBB-Lib	Basic B&B	SPMD	Cluster/PVM
PICO	Basic B&B Mixed-integer LP	hier. master-worker	Cluster/MPI
MallBa	Low-level Basic B&B	SPMD	Cluster/MPI
ZRAM	Low-level Basic B&B	SPMD	Cluster/PVM
ALPS/BiCePS	Low-level Basic B&B Mixed-integer LP Branch&Price&Cut	hier. master-worker	Cluster/MPI
Metacomputing MW	Basic B&B	master-worker	Grids/Condor
Symphony	Mixed-integer LP Branch&Price&Cut	master-worker	Cluster/PVM

Machines all these frameworks target clusters, but they differ on the approach to share the memory between distributed processes. Some of them use MPI [GRO 96] that is a library to send message between processes, and the others use PVM [GEI 94] that provides a virtual memory accessible by all processes. Only Metacomputing MW is based on a different middleware, Condor [CON 93], that enables use of Grids.

Despite the plentiful resources made available by grids, there has been limited work done on parallel B&B in this domain [GOU 00b]. In the context of this thesis, one of the objectives is to provide a B&B framework adapted to Grids. The next section complete this related work on parallel B&B with frameworks focused on Grids.

2.3.2 Overview of existing branch-and-bound framework for Grids

Grids gather large amount of heterogeneous resources across geographically distributed sites to virtual organization. Resources are usually organized in clusters, which are managed by different administrative domains (labs, universities, etc.). Thanks to the huge number of resources Grids provide, they seem to be well adapted for solving very large problems with B&B. Nevertheless, Grids introduce new challenges such as deployment, heterogeneity, fault-tolerance, communication, and scalability.

As we have showed in the previous section, there are numerous frameworks for parallel B&B, but only few of them target Grid environments [GOU 00b]. However, some Grid programming paradigms, such as farm skeleton and divide-and-conquer, can also be used to implement B&B for Grids, even if they are not originally made for B&B.

2.3.2.1 Branch-and-bound

Much of the work reported by the previous section is based on a centralized approach with a single manager which maintains the whole tree and hands out tasks to workers. This kind of approach clearly does not scale for Grid environments. Furthermore, many of these works focus on parallel search strategies. For example, in the work of Clausen *et al.* [CLA 99], which proposes to compare best-first search and lazy depth-first search.

Aida *et al.* [AID 03] present a solution based on hierarchical master-worker to solve scalability issues. Workers do branching, bounding, and pruning on sub-problems, which are represented by tasks. The supervisor handles the sharing of the best current upper bound, the root master. Supervisor and sub-masters gather results from workers and are in charge of hierarchically updating the best upper bound on all workers.

In [AID 05] Aida and Osumi propose a study of the work in [AID 03]. They implement their hierarchical master-worker framework using GridRPC middleware [SEY 02], Ninf-G [TAN 03], and Ninf [SAT 97]. Moreover, the authors discuss the granularity of tasks, notably when tasks are fine-grain the communication overhead is too high compared to the computation of tasks.

The work of Iamnitchi and Foster [IAM 00] also proposes a solution to do B&B over Grids. Their approach differs from others because it is not based on the master-worker paradigm, but on a decentralized architecture that manages resources through a membership protocol. Each process maintains a pool of problems to solve. When the pool is empty, the process asks for work from other processes. The sharing of the best upper bound is handled by circulating a message among processes. The fault-tolerance issue is addressed by propagating all completed sub-problem to all processes. Their approach may result in significant overhead, in terms of both duplicated work and messages. Typical problems solved with B&B are often very difficult, sometimes taking weeks of computation on a dedicated cluster to solve single problem. For that reason, the overhead resulting from duplicating work and messages must be considered.

Mezmaz *et al.* [MEZ 07a, MEZ 07b] present a load-balancing strategy for Grid based B&B algorithm. Their approach is based on the master-worker paradigm. In order to avoid the bottleneck that can result from the master, they propose to manage intervals instead of search tree nodes. In other words, the master keeps unexplored intervals and communicates exploration intervals to workers. The tree is explored using the depth-first search strategy, and they apply a particular numbering of the tree nodes, allowing a very simple description of the work units distributed during the exploration. Such description optimizes the communications involved by the huge amount of load-balancing operations. When a worker improves the upper bound, the new value is sent to the master. With this approach, they were the first to exactly solve the Taillar's instance Ta056 [TAI 93] of the flow-shop problem.

ParallelBB [BEN 07] is a parallel B&B algorithm for solving optimization problems. This work is also based on the master-worker paradigm, and it has for originalities to be implemented with ProActive and to use the peer-to-peer infrastructure proposed in this thesis (see Chapter 3). Like the B&B presented in our work (see Chapter 4), ParallelBB proposes direct communications between workers in order to avoid master bottleneck. Hence, workers directly share the upper bound. Unlike this thesis work, Grid related

communication issues between workers are not addressed. However, the use of ProActive for the implementation and of our peer-to-peer infrastructure validites of our choice of using a peer-to-peer approach for managing Grid infrastructures.

ParadisEO [CAH 03] is an open source framework for flexible parallel and distributed design of hybrid meta-heuristics. Moreover, it supplies different natural hybridization mechanisms mainly for meta-heuristics including evolutionary algorithms and local search methods. All these mechanisms can be used for solving optimization problem. The Grid version of ParadisEO is based on the master-worker paradigm. ParadisEO splits the optimization problem into sub-tasks to solve. Then, the task allocation is handled by MW [GOU 00a] a tool for scheduling master-worker applications over Condor [LIT 88], which is a Grid resource manager. ParadisEO just provides mechanisms for searching algorithms.

2.3.2.2 Skeletons

The common architecture used for B&B on Grids is the master-worker one. A master divides the entire search tree in a set of tasks and then distributes these tasks to workers. For parallel programming, the master-worker pattern is called *farm skeleton* [COL 91]. Muskel [DAN 05] is a Java skeleton framework for Grids that provide farm.

Skeleton frameworks usually provide task allocation and fault-tolerance. Thus, skeletons seem well adapted for implementing B&B for Grids. Users have just to focus on the implementation of the problem to solve while all other issues related to the Grid and task management are handled by the framework. Nevertheless, tasks in farm skeletons cannot share data, such as a global upper bound to prune more promising branches of the search tree so as to more rapidly find the optimal solution.

In addition, another skeleton that fits the B&B algorithm is the *divide-and-conquer skeleton*. This skeleton allows dynamic splitting of task, *i.e.* branching, but like farm skeleton it is not possible to share the global upper bound between task.

2.3.2.3 Divide-and-conquer

Conceptually, the B&B technique fits the divide-and-conquer paradigm. The search tree can be divided into sub-trees, and each sub-tree is then assigned to an available computational resource. This is done recursively until the task is small enough to be solved directly.

Satin [NIE 04] is a system for divide-and-conquer programming on Grid platforms. Satin expresses divide-and-conquer parallelism entirely in the Java language itself, without requiring any new language constructs. Satin uses so-called marker interfaces to indicate that certain method invocations need to be considered for parallel execution, called "spawned". A mechanism is also needed to synchronize with spawned method invocations.

Satin can be used directly to implement B&B. Thus, users can mark branching methods to be executed in parallel. Satin is in charge of distributing sub-problems through Grids. Nevertheless, Satin does not provide any mechanisms for sharing a global upper

bound and more generally no mechanism for communication between parallel executed sub-problems.

2.3.3 Requirements for Grid branch-and-bound

In the context of this thesis, we propose a parallel B&B framework for Grids. To motivate this, the framework must address all of the challenges presented by Grids, as listed in Section 2.1.

In addition to the Grid challenges, our framework must take into account analysis the experience, capabilities, and shortcomings of B&B frameworks and parallel B&B principles as documented in the previous section.

With these considerations, this section defines requirements for Grid B&B that our framework must fulfill.

Specific Grid requirements We previously introduced some key Grid challenges. Several of these challenges are in the Grid middleware infrastructure layer (see Layer 2 in Section 2.1) which are addressed directly by the incorporation of a P2P infrastructure, as described in Section 2.2 and will be returned to in detail in Chapter 3. Remaining challenges are in Layer 3, Grid programming, and must be treated by the B&B framework.

Grid challenges that the framework has to address:

- *Distribution*: involves significant latency and bandwidth reduction. Hence, the framework must hide these issues from users. A solution may be to use *asynchronous communications* between distributed processes. Asynchronous communications protect against network problems because processes are not blocked during message exchanges.
- *Deployment*: the P2P infrastructure provides a solution for deployment issues. Resource configuration, connection, etc. are hidden by the infrastructure. Finally, application deployment may be facilitating by using a Grid middleware, such as ProActive (see next section).
- *Multiple administrative domains*: this point is also solved by the P2P infrastructure.
- *Scalability*: this is one of the most important issues that the framework has to handle. The framework must be able to manage a large number of resources. In the related work, previously presented, we analyzed several B&B frameworks for Grids, and most of them use the *hierarchical master-worker* approach for scalability.
- *Heterogeneity*: relying on a Java Grid middleware can answer this point.
- *High performance*: the framework has to *efficiently use parallelism* and especially *communications* in order to get the maximum value from the large pool of resources provided by Grids.
- *Dynamicity and Fault-tolerance*: the acquisition of new resources is handle by the P2P infrastructure. However, in Grids faults, failures, maintenance, etc. are common; thus the framework has to be *fault-tolerant*.

- *Programming model*: the framework must *hide* all Grid issues and ideally all distribution & parallelism complexities.

Specific B&B requirements In this thesis, the B&B framework aims at helping users to solve problems and in particular, combinatorial optimization problems.

In the previous section, we have showed how many existing frameworks use the master-worker approach. Conceptually, the master-worker paradigm, based on Definition 2.2.1, fits B&B. The optimization problem to solve is represented as a dynamic set of tasks. A first task, which is the root node of the problem search tree, is passed to the master and branching is performed. The result is a new set of sub-tasks, or sub-problems, to branch and bound. Thereby, a *master* manages the sub-problem distribution to *workers*, and workers do branching and bounding operations. Both master and workers can handle pruning.

In addition, master-worker is well adapted to implement the three main approaches for designing parallel B&B; even if in this thesis we focus more on the trees because related problems have already been identified [AUT 95], thus it is easier to adapt this approach to Grids.

Even in parallel, generating and exploring the entire search tree leads to significant performance issues. Parallelism allows branch and bound a large number of feasible regions at the same time, but the pruning action seriously impacts the execution time. The efficiency of the pruning operation depends on the global upper bound (GUB) updates. The more GUB is close to the optimal solution, the more sub-trees are pruned. The improvement of GUB is determined by how the tree is generated and explored for a given problem. The exploration technique of the search tree for B&B is the key to rapidly solve problems. Therefore, a framework for Grid B&B has to propose several exploration strategies such as *breadth-first search* or *depth-first search*.

Other issues related to the pruning operation in Grids are concurrency and scalability. All workers must share common global data, which for B&B is only the GUB. GUB has multiple parallel accesses in read (get the value) and write (set the value) modes. A solution for sharing GUB is to maintain a local copy on all workers and when a better upper bound than GUB is found the worker broadcasts the new value to others.

In addition, for Grid environments, which are composed of hundreds of heterogeneous machines and which are managed by different administrative domains, the probability of having faulty nodes during a long execution is high. Therefore, a B&B for Grids has to manage fault-tolerance. A solution may for instance be that the master handles worker failures and the state of the search tree is frequently serialized to persistent storage.

Summary of requirements In this thesis we present *Grid'BnB*, a parallel B&B framework for Grids. *Grid'BnB* aims to hide Grid difficulties from users. Especially, fault-tolerance, communication, and scalability problems are addressed by *Grid'BnB*. The framework is built on a master-worker approach and provides a transparent communication system between tasks. Local communications between processes are used to optimize the exploration of the problem solution space. *Grid'BnB* is implemented in Java within the ProActive [BAD 06] Grid middleware and also relies on the P2P infras-

tructure. The ProActive middleware and infrastructure provide a flexible and efficient deployment framework for Grids, and allow asynchronous communications.

Framework design/architecture requirements:

- **Asynchronous communications**
- **Hierarchical master-worker**
- **Dynamic task splitting**
- **Efficient parallelism and communications**:
  - organizing workers in groups to optimize inter-cluster communications
  - sharing the current best lower/upper bound
  - communication between workers
- **Fault-tolerance**

Users requirements:

- **Hidden parallelism and Grid difficulties**
- **Combinatorial optimization problems**
- **Ease of deployment** multiple administrative domains, heterogeneity, acquisition of new resources (dynamicity)
- **Principally tree-based** parallelism strategy, but also may use others
- **Implementing and testing search strategies**
- **Focus on objective function**

With all these requirements for parallel B&B, and more specifically on Grid environments, we propose, in this thesis, *Grid'BnB* a complete Java API for using parallel B&B technique with Grids.

We also present a mechanism based on communications between workers to share GUB. Thereafter, we propose a system to dynamically organize workers in communication groups. This organization aims to control worker topology to maximize scalability (more details in Chapter 4 and Section 6.2).

2.4 Context: ProActive Grid middleware

ProActive is an open source Java library for Grid computing. It allows concurrent and parallel programming and offers distributed and asynchronous communications, mobility, and a deployment framework. With a small set of primitives, ProActive provides an API allowing the development of parallel applications, which may be deployed on distributed systems and on Grids.

The active object model and the ProActive library are used as a basis in this thesis for developing a peer-to-peer infrastructure, a branch-and-bound framework, and performing large-scale experiments.

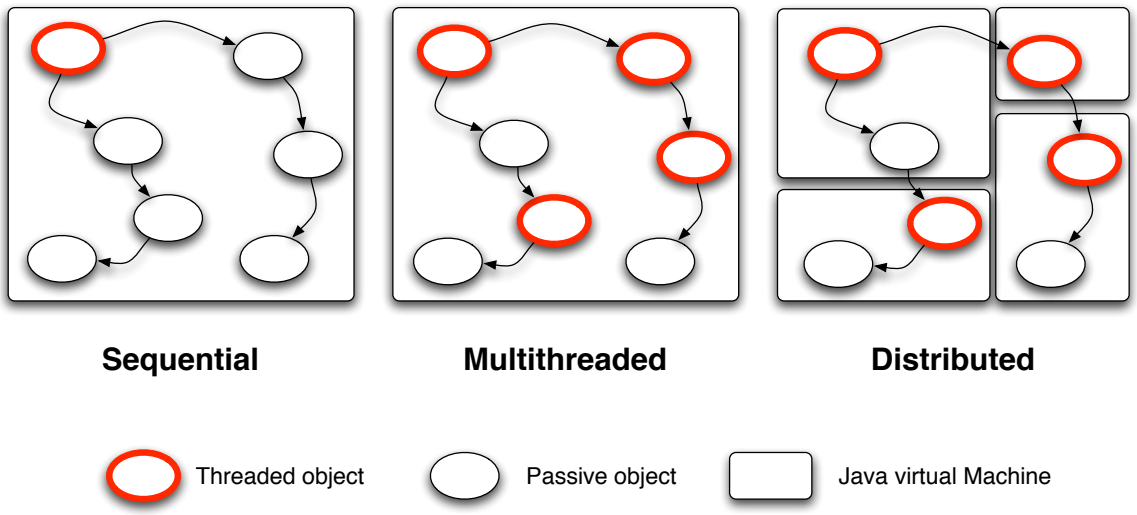

Figure 2.11: *Seamless sequential to multithreaded to distributed objects*

2.4.1 Active objects model

ProActive is based on the concept of an *active object*, which is a medium-grained entity with its own configurable activity.

A distributed or concurrent application built using ProActive is composed of a number of medium-grained entities called active objects (Figure 2.11). 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. Method calls sent to active objects are asynchronous with transparent future objects and synchronization is handled by a mechanism known as wait-by-necessity [CAR 93]. There is a short rendezvous at the beginning of each asynchronous remote call, which blocks the caller until the call has reached the context of the callee. All of this semantics is built using meta programming techniques, which provide transparency and the ground for adaptation of non-functional features of active objects to various needs.

Explicit message-passing programming based approaches were deliberately avoided: one aim to enforce code reuse by applying the remote method invocation pattern, instead of explicit message-passing.

The active object model of ProActive guaranties determinism properties and was formalized with the ASP (Asynchronous Sequential Processes) calculus [CAR 04].

2.4.2 The ProActive library: principles, architecture, and usages

The ProActive library implements the concept of active objects and provides a deployment framework in order to use the resources of a Grid.

2.4.2.1 Implementation language

Grids gather large amount of heterogeneous resources, different processor architectures and operating systems. In this context, using a language which relies on a virtual ma-

chine allows maximum portability. ProActive is developed in Java, a cross-platform language and the compiled application may run on any operating system providing a compatible virtual machine. Moreover, ProActive only relies on standard APIs and does not use any operating-system specific routines, other than to run daemons or to interact with legacy applications. There are no modifications to the JVM nor to the semantics of the Java language, and the bytecode of the application classes is never modified.

2.4.2.2 Implementation techniques

ProActive relies on an extensible meta-object protocol architecture (MOP), which uses reflective techniques in order to abstract the distribution layer, and to offer features such as asynchronism or group communications.

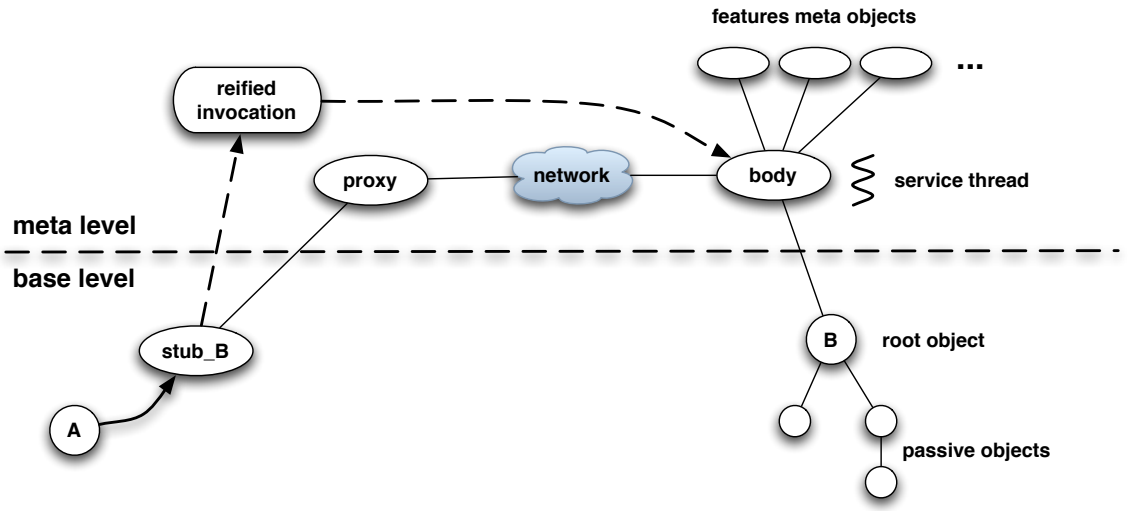

Figure 2.12: *Meta-object architecture*

The architecture of the MOP is presented in Figure 2.12. An active object is concretely built out of a root object (here of type B), with its graph of passive objects. A body object is attached to the root object, and this body references various features meta-objects, with different roles. An active object is always indirectly referenced through a proxy and a stub which is a sub-type of the root object. An invocation on the active object is actually an invocation on the stub object, which creates a reified representation of the invocation, with the method called and the parameters, and this reified object is given to the proxy object. The proxy transfers the reified invocation to the body, possibly through the network, and places the reified invocation in the request queue of the active object. The request queue is one of the meta-objects referenced by the body. If the method returns a result, a future object is created and returned to the proxy, to the stub, then to the caller object.

The active object has it own activity thread, which is usually used to pick-up reified invocations from the request queue and serve them, *i.e.* execute them by reflection on the root object. Reification and interception of invocations, along with ProActive's customizable MOP architecture, provide both transparency and the ground for adaptation of non-functional features of active objects to fit various needs. It is possible to add cus-

tom meta-objects which may act upon the reified invocation, for instance for providing mobility features.

Active objects are instantiated using the ProActive API, by specifying the class of the root object, the instantiation parameters, and a possible location information:

```
// instantiate active object of class B on node1 (a possibly remote location)
B b = (B) ProActive.newActive("B", new Object[] {aConstructorParameter}, node1);

// use active object as any object of type B
Result r = b.foo();

// possible wait-by-necessity
System.out.println(r.printResult());
```

2.4.2.3 Communication by messages

In ProActive, the distribution is transparent: invoking methods on remote objects does not require the developer to design remote objects with an explicit remoting mechanism (like Remote interfaces in Java RMI). Therefore, the developer can concentrate on the business logic as the distribution is automatically handled and transparent. Moreover, the ProActive library preserves polymorphism on remote objects (through the reference stub, which is a subclass of the remote root object).

Communications between active objects are realized through method invocations, which are reified and passed as messages. These messages are serializable Java objects which may be compared to TCP packets. Indeed, one part of the message contains routing information towards the different elements of the library, and the other part contains the data to be communicated to the called object.

Although all communications proceed through method invocations, the communication semantics depends upon the signature of the method, and the resulting communication may not always be asynchronous.

Three cases are possible: synchronous invocation, one-way asynchronous invocation, and asynchronous invocation with future result.

- *Synchronous invocation*:
  - the method returns a non reifiable object: primitive type or final class:
    `public boolean foo()`
  - the method declares a thrown exception:
    `public void bar() throws AnException`

  In this case, the caller thread is blocked until the reified invocation is effectively processed and the eventual result (or exception) is returned. It is fundamental to keep this case in mind, because some APIs define methods which throw exceptions or return non-reifiable results.
- *One-way asynchronous invocation*: the method does not throw any exception and does not return any result:
  `public void gee()`
  The invocation is asynchronous and the process flow of the caller continues once the reified invocation has been received by the active object (in other words, once the rendezvous is finished).

- *Asynchronous invocation with future result*: the return type is a reifiable type, and the method does not throw any exception:
  `public MyReifiableType baz()`
  In this case, a future object is returned and the caller continues its execution flow. The active object will process the reified invocation according to its serving policy, and the future object will then be updated with the value of the result of the method execution.

If an invocation from an object A on an active object B triggers another invocation on another active object C, the future result received by A may be updated with another future object. In that case, when the result is available from C, the future of B is automatically updated, and the future object in A is also update with this result value, through a mechanism called *automatic continuation* [CAR].

2.4.2.4 Features of the library

As stated above, the MOP architecture of the ProActive library is flexible and configurable; it allows the addition of meta-objects for managing new required features. Moreover, the library also proposes a deployment framework, which allows the deployment of active objects on various infrastructures.

The features of the library are represented in Figure 2.13.

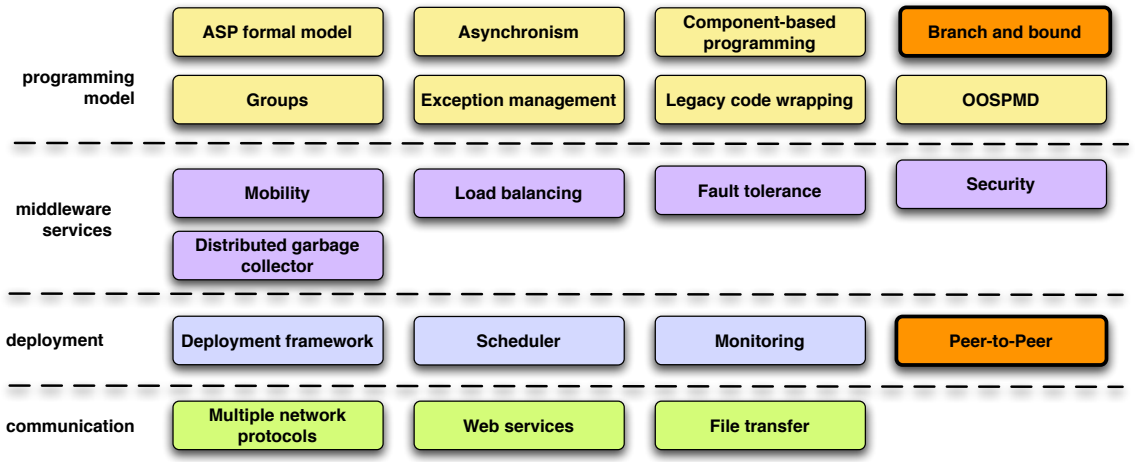

Figure 2.13: *Layered features of the ProActive library*

The active object model is formalized through the ASP calculus [CAR 05b], and ProActive may be seen as an implementation of ASP. The library may be represented in three layers: programming model, non-functional features, and deployment facilities.

The programming model consists of the active objects model which offer asynchronous communications, typed group communications [BAD 02], and the object-oriented SPMD programming paradigm [BAD 05]. In addition to the standard object oriented programming paradigm, ProActive also proposes a component-based programming paradigm, by providing an implementation [BAU 03] of the Fractal component model [BRU 02] geared at Grid computing. In the context of this thesis, we propose a new programming model for ProActive: a branch-and-bound framework.

Non-functional features include a transparent fault-tolerance mechanism based on a communication-induced check-pointing protocol [BAU 05], a security framework for communications between active objects [ATT 05], migration capabilities for the mobility of active objects [BAU 00], a mechanism for the management of exceptions [CAR 05a], a complete distributed garbage collector for active objects [CAR 07b], and a mechanism for wrapping legacy code, notably as a way to control and interact with MPI applications.

The deployment layer includes a deployment framework [BAU 02]; it is detailed in Section 6.1, and it allows the creation of remote active objects on various infrastructures. A scheduler is also proposed to manage the deployment of jobs. The load-balancing framework uses the migration capabilities to optimize the placement of the distributed active objects [BUS 05]. In the context of this thesis, we propose a new deployment infrastructure for ProActive: peer-to-peer infrastructure.

In the communication layer several protocols are provided for the communication between active objects: Java RMI as the default protocol, HTTP, tunneled RMI. It is also possible to export active objects as web services, which can then be accessed using the standard SOAP protocol. A file transfer mechanism is also implemented; it allows the transfer of files between active objects, for instance to send large data input files or to retrieve results files [BAU 06].

2.5 Conclusion

In this chapter, we positioned the work of this thesis in the context of Grid computing. We showed that both communities, peer-to-peer and Grid, share the same goal: pooling and coordinating use of large sets of distributed resources. In addition, we demonstrated the validity and adequacy of using a peer-to-peer approach to provide a Grid infrastructure. We also justified requirements needed by our branch-and-bound framework for Grids. We then introduced the ProActive Grid middleware.

This thesis focuses on the Grid middleware infrastructure layer, with the peer-to-peer infrastructure, and on the upper layer, which is Grid programming, with the branch-and-bound framework. This work addresses all Grid challenges that we pointed out: distribution, deployment, multiple administrative domains, scalability, heterogeneity, high performance, dynamicity, and programming model.

We argue that Grid infrastructures have to be dynamic, especially to support the inclusion of new sites in the Grid. Hence, we propose a peer-to-peer infrastructure to share computational resources. In this thesis, we also propose a branch-and-bound framework adapted to Grids. Finally, ProActive is a strong Grid middleware and provides features that help for Grid development, and we augment ProActive with this thesis work.

Chapter 3

Desktop Peer-to-Peer

In this chapter, we present the first part of this thesis contribution, which is an infrastructure for managing Grids [CAR 07a]. The proposed infrastructure is indeed based on a peer-to-peer architecture. The main goal of this infrastructure is to manage a large-scale pool of resources; and owing to the infrastructure, applications have an easy access to these resources.

In chapter 2, we identified the specificities of Grid computing. Moreover, we argued that Grid and peer-to-peer share the same goal, and thus peer-to-peer architectures can be used as infrastructures for Grids.

We introduce in details the proposed infrastructure and report experiments on a lab-wide Grid. With which we have achieved a computation record by solving the n-queens problem for 25 queens. Large-scale experiments are reported in Chapter 5.

3.1 Motivations and objectives

These last years, computing Grids have been widely deployed around the world to provide high performance computing tools to research and industrial fields. Those Grids are generally composed of dedicated clusters. In parallel, an approach for using and sharing resources called Peer-to-Peer (P2P) networks has also been deployed. In P2P networks, we can discern two categories: Edge Computing or Global Computing, such as SETI@home [AND 02], which takes advantage of machines at the edges of the Internet; and P2P files sharing, such as Gnutella [GNU 00], which permits Internet users to share their files without central servers.

In the previous chapter, we identified many definitions of a P2P network: decentralized and non-hierarchical node organization [STO 03], or taking advantage of resources available at the edges of the Internet [ORA 01]. In this thesis, a P2P network follows the definition of a “Pure Peer-to-Peer Network”, as in Definition 2.2.3, meaning that it focuses on sharing resources, decentralization, and peer failures.

A Grid is an infrastructure that gathers shared resources, from different institutions, in a virtual organization. Resources may be of anything that can be connected to a computer network; resources are for instance, desktop machines, clusters, PDA, GSM, sensors, *etc.*

In this thesis, we consider that resources can be of two kinds: desktop machine and cluster; they are the most common on Grids. For us, a desktop machine or desktop

is a computer, which has a single user. A cluster is typically a group of similar fast computers, connected to a dedicated high-speed network, working together. Thus, a cluster may be seen as a unique resource.

Usually, institutions have only one or two clusters, thus cluster's users have to share their computation slots with others; they are also not able to run computations that would take months to complete because they are not allowed to use all the resources exclusively for their experiments. On the other hand, institutions have several desktops, which are under-utilized and are only available to a single user. The idea of gathering all desktops of an institution in a virtual organization was firstly popularized by Entropia [CHI 03]. This kind of Grid is know as *desktop Grid*.

We define desktop Grid as a virtual organization, which gathers desktops of a same and single institution. Note that, the institution may be composed of several sites geographically distributed. In other words, desktop Grids is a Grid of desktops that are in the same network.

However, existing models and infrastructures for P2P computing are limited as they support only independent worker tasks, usually without communications between tasks. However P2P computing seems well adapted to applications with low communication/-computation ratio, such as parallel search algorithms. We therefore propose in this thesis a P2P infrastructure of computational nodes for distributed communicating applications.

The proposed infrastructure is an unstructured P2P network, *e.g.* Gnutella [GNU 00]. In contrast to others P2P approaches for computing, which are usually hierarchical or master-salve, our approach is original in the way that an unstructured P2P network commonly used for file sharing can be also used for computing.

The P2P infrastructure has three main characteristics. First, the infrastructure is decentralized and completely *self-organized*. Second, it is flexible, thanks to parameters for adapting the infrastructure to the location where it is deployed. Finally, the infrastructure is portable since it is built on top of Java Virtual Machines (JVMs). Thus, the infrastructure provides an overlay network for sharing JVMs.

The infrastructure allows applications to transparently and easily obtain computational resources from Grids composed of both clusters and desktops. The burden of application deployment is eased by a seamless link between applications and the infrastructure. This link allows applications to be communicating, and to manage the resources' volatility. The infrastructure also provides large-scale Grids for computations that would take months to achieve on clusters.

In the previous chapter, we defined the concept of Grid computing; and we also presented peer-to-peer (P2P) architectures. Both, Grid and P2P, concepts share the same goal. From this observation and our analysis, we established a list of several requirements that our infrastructure has to fulfill:

- **Asynchronous communication** between peers;
- **Simple and unified** way for acquiring resources;
- **Flexible** and easily **adaptable** to handle sites;
- **Heterogeneity**: Java and the inclusion of interface with most used Grid middlewares/infrastructures/schedulers;
- **Resources usability**: applications must have the full usage of acquired resources;

- **Dynamicity and volatility**: unstructured P2P network;and
- **Crossing firewalls**.

In this chapter, we present the first part of this thesis main contribution, which is a Grid infrastructure. We also describe a permanent desktop Grid, in our lab, managed by our infrastructure, with which we experiment long-running computation.

In summary, the main features of the P2P infrastructure are:

- an unstructured P2P overlay network for sharing computational resources;
- building Grids by mixing desktops and clusters;
- deploying communicating applications; and,
- achieving computations that take months on clusters.

3.2 Design and architecture

We now present in detail the design and the architecture of the P2P infrastructure.

3.2.1 First contact: bootstrapping

A well-known problem of P2P networks is the bootstrapping problem, also called the first contact problem. This problem can be solved by many different network protocols, such as JINI [WAL 00]. It can be used for discovering services in a dynamic computing environment. At first this protocol seems to be perfectly adapted to solve the bootstrapping problem. However, there is a serious drawback for using this protocol: JINI needs to be deployed on a network with IP multicast communications allowed. That means JINI cannot be widely distributed.

Therefore, a different solution for the bootstrapping problem was chosen, inspired from super-peer networks [YAN 03]. A fresh peer has a list of "registry addresses". These are peers that have a high potential to be available; they are in a certain way the P2P network core. The fresh peer tries to contact each registry within this list. When a registry is responding, it is added to the fresh peer list of known peers (acquaintances). When the peer has connected to at least one registry, it is a member of the P2P Network. Figure 3.1 shows an example of a fresh peer that tries to connect itself to the infrastructure.

In a certain way, the P2P infrastructure may be considered as a hybrid network, because registry peers propose a bootstrapping service that other peers do not provide. However, we just explained that a protocol, such as JINI, may be used to provide the first contact service. The infrastructure is open enough to use any kind of bootstrapping protocol. Hence, we consider that the infrastructure is an unstructured pure P2P network.

3.2.2 Discovering acquaintances

The main problem of the infrastructure is the high volatility of peers because those peers are desktop machines and clusters nodes, possibly available for a short time.

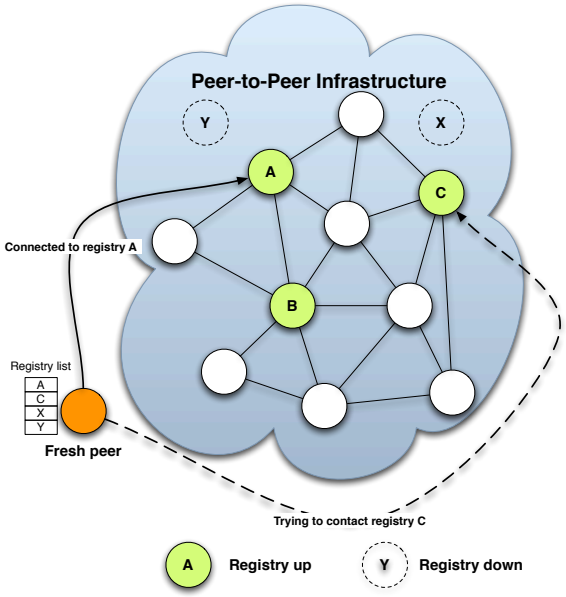

Figure 3.1: *Bootstrapping problem*

Therefore, the infrastructure aims at maintaining an overlay network of JVMs alive; this is called *self-organizing*. When it is impossible to have external entities, such as centralized servers, which maintain peer databases, all peers must be capable of staying in the infrastructure by their own means. The strategy used for achieving self-organization consists of maintaining, for each peer, a list of acquaintances.

The infrastructure uses a specific parameter called *Number of Acquaintances* (NOA): the minimum number of known acquaintances for each peer. Peers update their acquaintance list every *Time to Update* (TTU), checking their own acquaintance list to remove unavailable peers, *i.e.* they send heartbeat messages to them. When the number in the list is less than NOA, a peer will try to discover new acquaintances. To discover new acquaintances, peers send exploring messages through the infrastructure by flooding the P2P network.

The exploring message is sent every TTU until the length of the list is greater than the NOA value. This message is sent with a unique identifier, with a reference to the sender, and with the *Time To Live* (TTL) in number of hops. The TTL and the unique identifier limit the network flooding.

When a peer receives an exploring message, it has to:

1. check the unique identifier: if it is an old message, drop it and do nothing;
2. store the unique identifier;
3. if the requester is not already in its acquaintance list: use a function to determine if the local peer has to answer. This function is for the moment a random function, in a future work we would like to improve the network organization.
4. then if the TTL decremented is greater than 0, broadcast the message to all its acquaintances.

Finally, NOA, TTU, and TTL are all configurable by the administrator, who has deployed the P2P infrastructure. Each peer can have its own value of those parameters and the values can be dynamically updated.

3.2.3 Asking resources

For the infrastructure and the application, all resources are similar. The infrastructure is *best-effort*, applications ask for computational nodes, JVMs, but the infrastructure does not guarantee that applications requests can be satisfied (not enough available nodes, *etc.*). Usually applications request nodes from the infrastructure and then the infrastructure returns node references back to the application. All requests from applications are in competition to obtain available resources. An available resource is a node shared by a peer, which is free. In other words, a free node is the node that the peer does not yet share with an application.

In order to satisfy node queries faster, we distinguish three cases.

Asking one node The application needs *only one node*, then the query node message uses a random walk algorithm, which means that the next hop is randomly chosen. The message is forwarded peer by peer until a peer has a free node to share or until TTL reaches zero. While no free nodes have been found the application re-sends the message at each TTU increasing eventually the TTL. Figure 1 shows the message protocol.

Message Protocol 1 Asking one node to the P2P infrastructure. This protocol shows what is done by a peer when it receives a **one node** request:

Require: A remote reference on the node requester: $Node\_Requester$,
the request TTL
Ensure: A free node for computation
if $Node\_Requester$ is $Myself$ **then**
Forward the request to a randomly chosen acquaintance
else if I have a free node **then**
return the free node
else if $TTL > 0$ **then**
Forward the request to a randomly chosen acquaintance with $TTL = TTL - 1$
else
drop the request
end if

Asking n nodes The application needs *n nodes* at once, for that case the resource query mechanism used is similar to the Gnutella [GNU 00] communication system, which is based on the Breadth-First Search algorithm (BFS). Messages are forwarded to each acquaintance, and if the message has already been received or if its TTL reaches 0, it is dropped. The message is broadcasted by the requester every TTU until the total number of requested nodes is reached or until a global timeout occurs. Also, we have added a kind of transactional commit. When a peer is free, it sends a reference on its node to the requester. Before forwarding the message the current peer waits for an acknowledgment from the requester because the request could have already been fulfilled. After an expired timeout or a non-acknowledgment, the peer does not forward

the message. Otherwise, the message is forwarded until the end of the TTL or until the number of requested nodes reaches zero. The acknowledgment message from the requester is indeed the total number of nodes still needed by the requester. We can distinguish three states for a peer: free, busy, or booked. This mechanism is specified in Message Protocol 2.

Message Protocol 2 Asking for n nodes at once from the P2P infrastructure. This protocol shows the response by a peer when it receives an **n nodes** request:

Require: A remote reference on the node requester $Node\_Requester$,
the request TTL,
the request $UniqueID$, and
the requested number of nodes n
Ensure: At most n free nodes for computation
if $Node\_Requester$ is $Myself$ **or** $allPreviousRequests$ contains $UniqueID$ **then**
drop the request
else
Add $UniqueID$ in $allPreviousRequests$
if I have a free node **then**
Send the node to $Node\_Requester$ {$Node\_Requester$ receives the node and decides whether or not to send back an ACK to the peer, which has given the node}
while not $timeout$ reached **do**
wait for an ACK from $Node\_Requester$ {ACK is the number of still needed nodes by the requester, $ACK = 0$ means NACK}
if ACK received **and** $TTL > 0$ **and** $ACK > 1$ **then**
Broadcast the message to all acquaintances with $TTL = TTL - 1$ and $n = ACK$
end if
end while
end if
end if

Asking MAX nodes The application may ask for all available nodes, the message protocol is close to the previous one but does not need to wait for an acknowledgment and the message is broadcast every TTU until the application end. Message Protocol 3 shows the message protocol.

3.2.4 Peer and node failures

The infrastructure itself is stable, according to the Definition 2.2.3, as each peer manages its own list of acquaintances by the heartbeat mechanism (see Section 3.2.2). Therefore the infrastructure can maintain a network of peers until all peers are down, *i.e.* a peer failure is not a problem for the infrastructure.

The issue is at the application level. The infrastructure broadcasts the node request of the application through itself (see Section 3.2.3); when a peer has an available node, it returns directly (point-to-point) to the application a reference to the node. Once the application has received the reference there is no guarantee that the node is still up or if the node will be up for all the time that the application needs it. Therefore it is the

Message Protocol 3 Asking maximum nodes to the P2P infrastructure. This protocol shows what is done by a peer when it receives a **MAX nodes** request:

Require: A remote reference on the node requester $Node\_Requester$, the request TTL, the request $UniqueID$
Ensure: Free nodes for computation
  if $Node\_Requester$ is $Myself$ **then**
    do nothing
  else
    if I have a free node **then**
      Send the node to $Node\_Requester$
    end if
    if $allPreviousRequests$ **not** contains $UniqueID$ and $TTL > 0$ **then**
      Add $UniqueID$ in $allPreviousRequests$
      Broadcast the message to all my acquaintances with $TTL = TTL - 1$
    end if
  end if

application's responsibility to manage node failures.

However the P2P infrastructure is implemented with the ProActive Grid middleware, which proposes a mechanism for fault-tolerance (see Section 6.3.3.1). With this particular use case we have started to work on a mechanism to deploy non-functional services, such as fault-tolerance or load balancing, on a Grid [CAR 06b]. This mechanism allows users to deploy theirs applications on the P2P infrastructure and to have fault-tolerance automatically applied.

Furthermore, the infrastructure does not work as a job/task scheduler, it is just a node provider. Therefore the application has to manage all node failures and its own failures. In a future work we plan to provide a job scheduler for the infrastructure, see Chapter 7.

3.3 Integration within ProActive

The P2P infrastructure is implemented with the ProActive library, which is described in Chapter 2.4. Thus, the shared resources are not JVMs but ProActive Nodes.

The infrastructure is implemented with classic ProActive active object model and especially with ProActive typed group communication [BAD 02] for broadcasting communications between peers. A peer is an independent entity that works as a server with a FIFO request queue; it is also a client which sends requests to others peers, thus the active object model can be easily representing our idea of what is a peer in our infrastructure. The main active object is *P2PService*, which serves all register requests or resource queries, such as Nodes or acquaintances.

ProActive supports different communication protocols, such as RMI or HTTP; then the infrastructure implemented at the top of the ProActive library allows each peer to use different communication protocols. For example, a peer, which is a desktop machine, accepts RMI communication but uses RMI/SSH to communicate with a peer inside a cluster.

The node-sharing mechanism is an independent activity from the P2P service. This activity is handled by the *P2PNodeManager* (PNM) active object. The PNM manages the sharing status, free for computation or not, of the peer JVM. The peer with its PNM can also share JVMs from an XML Deployment Descriptor. That is useful to share some JVMs from a cluster. For example, a peer at a fixed time can deploy JVMs on a cluster and shares them with the infrastructure for a few hours, in the way to do not monopolize the cluster for a long usage.

The node-asking mechanism is allowed by the *P2PNodeLookup* (PNL), this object is activated by the P2P Service when it receives an node-asking request from an application. The PNL works as a node broker for the application. The PNL aims to find the number of nodes requested by the application. It uses our previously described message protocols to frequently flood the network until it gets all nodes or until the timeout is reached. However, the application can ask the maximum number of nodes, in that case the PNL asks for nodes until the end of the application. The next code example shows how to get access to the PNL from applications codes:

```
// Starting a local Peer:
StartP2PService startServiceP2P = new StartP2PService(peerListForFirstContact);
startServiceP2P.start();

// Get the reference on the P2P Service
P2PService serviceP2P = startServiceP2P.getP2PService();

// Asking n nodes to the infrastructure
P2PNodeLookup p2pNodeLookup = p2pService.getNodes(200);

// Get some nodes by the broker
Node[] someNodes = p2pNodeLookup.giveMeNNodes(50);
Node[] theRestOfNodes = p2pNodeLookup.giveMeAllNodes();

// End of the application free used nodes
 p2pNodeLookup.killAll();
```

Finally, the access to the P2P infrastructure is fully integrated with the ProActive deployment framework, which is based on an XML descriptor (more details on deployment in Chapter 6). The next example shows an example of an XML descriptor file with nodes acquisition by the P2P infrastructure:

```
...
<jvm name="workers">
  <acquisition>
    <aquisitionReference refid="p2pLookup"/>
  </acquisition>
</jvm>
...
<infrastructure>
 <acquisition>
  <acquisitionDefinition id="p2pLookup">
   <P2PService NodesAsked="MAX" NOA="10" TTL="3" TTU="60000" acq="rmissh">
    <peerSet>
     <peer>rmi://registry1:3000</peer>
     <peer>rmi://registry2:3000</peer>
    </peerSet>
   </P2PService>
```

```
  </acquisitionDefinition>
 </acquisition>
</infrastructure>
...
```

We have introduced a new tag for the XML descriptor, which is acquisition. This tag allows to acquire Nodes for a Virtual Node from a P2P infrastructure. The parameter NodesAsked of the tag P2PService is the number of Nodes asked to the P2P infrastructure, this number could take a special value MAX for asking the maximum available Nodes in the P2P infrastructure. The tag peerSet contains the list of peers, which are already in the P2P infrastructure. The next example shows a sample code from users applications for deploying an application over the P2P infrastructure. It also shows how applications can use an events/listeners mechanism to dynamically obtain new nodes:

```
ProActiveDescriptor pad = ProActive.getProactiveDescriptor("myP2PXmlDescriptor.xml");

// getting virtual node "p2pvn" defined in the ProActive Deployment Descriptor
VirtualNode vn = pad.getVirtualNode("p2pvn");

// adding "this" or any other class has a listener of the "NodeCreationEvent"
((VirtualNodeImpl) vn).addNodeCreationEventListener(this);

//activate that virtual node
vn.activate();

...
// The method to implement for the listening nodes acquisition
public void nodeCreated(NodeCreationEvent event) {
  // get the node
  Node newNode = event.getNode();
 // now you can create an active object on your node.
}
```

3.4 Long running experiment

This section describes the experiments achieved for testing the P2P infrastructure. Experiments have been done with the n-queens problem. We first describe the n-queens problem and then we present the *INRIA Sophia P2P Desktop Grid*, which is a permanent desktop Grid infrastructure managed by our infrastructure. Finally, we report a long running experiment result, with which we have achieved a computation record by solving the n-queens for 25 queens.

3.4.1 The n-queens problem

The n-queens problem consists in placing n queens on a $n \times n$ chessboard so that no two queens are on the same vertical, diagonal, or horizontal line (*i.e.* attack each other). We aim to find all solutions with a given n.

The chosen approach to solve the n-queens problem was to divide the global set of permutations in a set of independent tasks. Then a master-worker model was applied to distribute these tasks to the workers, which were dynamically deployed on a desktop

Grid. The master frequently saves the computation status, *i.e.* returned results, on the disk in order to continue the computation after a failure of the master's host. Workers failures are handled by master, the master re-submit the failed task to another worker.

3.4.2 Permanent desktop Grid: INRIA Sophia P2P Desktop Grid

In order to run our experiments, the *INRIA Sophia P2P Desktop Grid* (InriaP2PGrid) has been deployed on about 260 desktop machines of the INRIA Sophia Antipolis lab; this Grid is now a permanent Grid managed by our P2P infrastructure.

All these desktop machines are running various GNU/Linux distributions or Microsoft Windows XP as operating systems, on Intel CPUs, from Pentium 2 to dual-Pentium 4. As to not interfere with daily work, the JVMs, Sun 1.4.2 or Sun 1.4.1, are started with the lowest system priority.

Because these machines are used by their normal users, the INRIA lab direction has authorized us to use machines during nights and weekends for P2P experiments. However some of them may be run 24 hours a day. Thus machines are organized in two groups:

INRIA-ALL joins the infrastructure during the night, 8:00pm to 8:00am, and during weekend, Friday 8:00pm to Monday 8:00am.

INRIA-2424 is a sub-group of INRIA-ALL, and these machines are always members of the infrastructure. This group counts 53 machines. They are selected in regard to their CPU power, thus they are the fastest one.

Also, the users can interrupt the computation if ever our experiments bother them. The INRIA-2424 peers are used as registries (all registries use themselves as registries); and at certain fixed moments the rest of INRIA-ALL machines join the P2P infrastructure by contacting those registries. Figure 3.2 shows the InriaP2PGrid structure.

The repartition of CPU frequencies of all desktop machines are summarized in Figure 3.3.

Finally, the values of the P2P infrastructure parameters used by the InriaP2PGrid are:

- **NOA=30 acquaintances**: each sub-network contains on average 20 machines, so a peer discovers some acquaintances outside of its sub-network.
- **TTU=10 minutes**: INRIA-2424 machines are highly volatiles, we have observed on this group that on average 40 machines are available out of 53. Every 10 minutes, one peer out of 30 becomes unavailable (host down, JVMs killed by users *etc.*). It usually joins back the infrastructure some time later.P2P JVMs restart at fixed time or when the host is up again.
- **TTL=3 hops**: it is the diameter of the network, as shown by Figure 3.2.

This permanent infrastructure has been used for long running experiment and, in the next chapter, for large-scale experiments by mixing desktops and cluster machines.

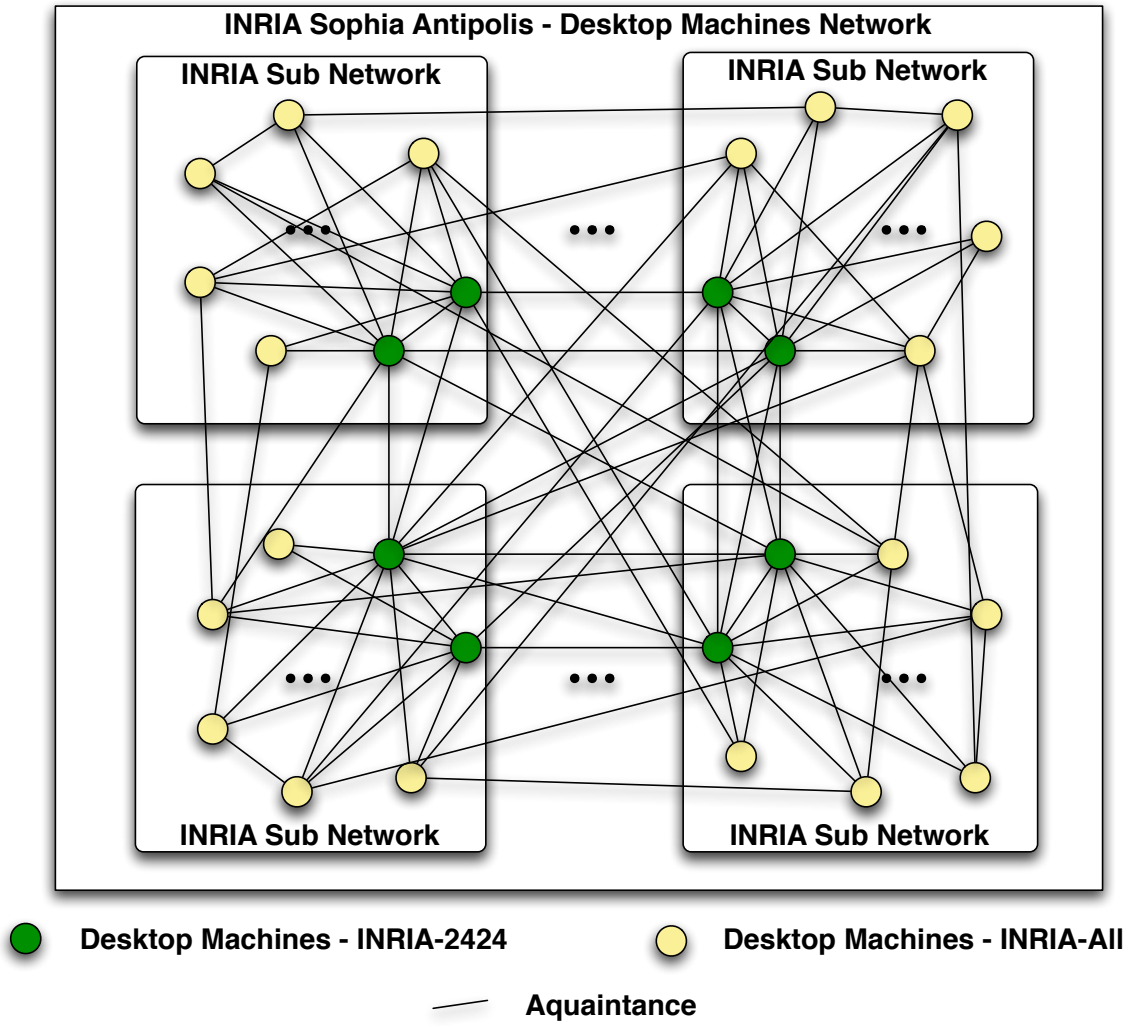

Figure 3.2: *Desktop experiments:* INRIA Sophia P2P Desktop Grid *structure*

3.4.3 Long running experiment

With the InriaP2PGrid managed by our P2P infrastructure, we are the first, as referenced by [SLO 05], to solve the n-queens problem with 25 queens. All the results of the n-queens experiment are summarized in Table 3.1. The experiment took six months for solving this problem instance. The result was later confirmed by Pr. Yuh-Pyng Shieh from the National Taiwan University.

Moreover, Figure 3.4 shows the number of peers, which participated to the n-queens computation over time. This graph does not report all the experiment period time, only three months out of six. During the experiment, the infrastructure counted 260 different desktop machines and a top of 220 machines working at the same time. As shown in Figure 3.4 the infrastructure was down 3 times, owing to 3 global lab power cuts. Once the power back, the computation succeeded to start again by its own. Figure 3.4 shows some troughs where the infrastructure provided less peers for the computation; these

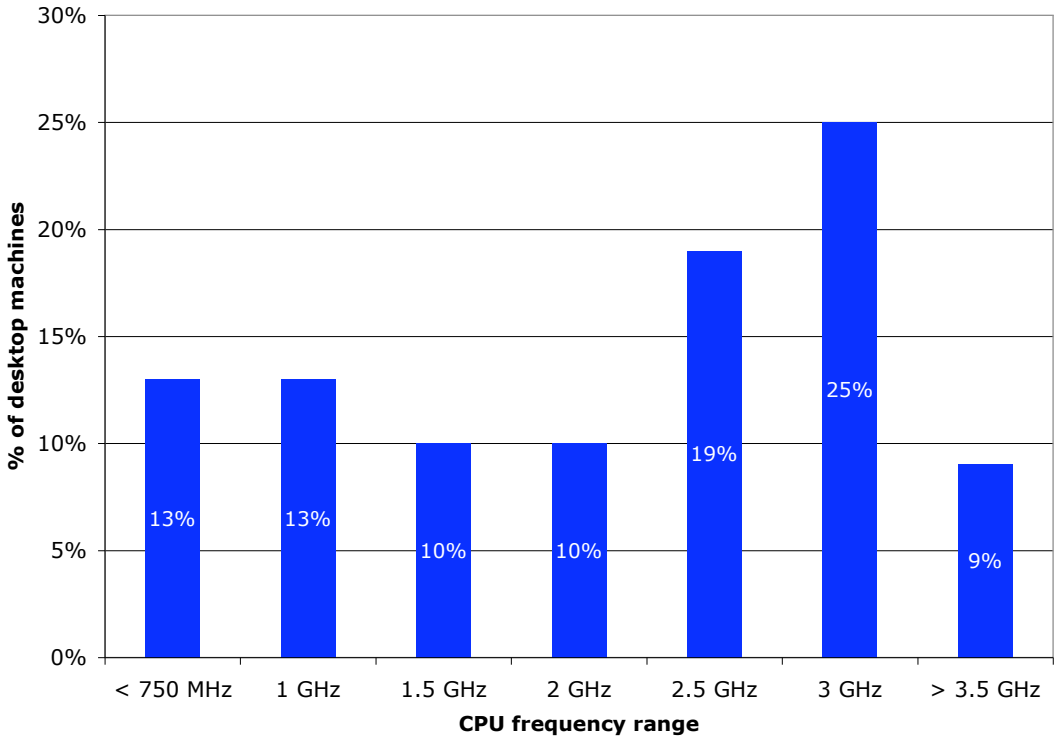

Figure 3.3: *Desktop Grid: CPU frequencies of the desktop machines*

Table 3.1: *Desktop experiments: n-queens experiment summary with n=25*

Total of Solution Found	$2,207,893,435,808,352$ $\approx 2\ quadrillions$
Total # of Tasks	$12,125,199$
Total Computation Time	$4,444h54m52s$ $\approx$ **185 days**
Average Time of One Task Computation	$\approx 2m18s$
Equivalent Single CPU Cumulated Time	$464,344h35m33s$ $\approx$ **53 years**
Total # of Desktop Machines	260 (max of 220 working concurrently)

troughs result from some network hardware failures (switch, router, *etc.*) with consequence of removing some sub-networks.

Figure 3.5 shows the percentage of tasks computed by the workers. To plot this graph we first sort all machines according to the number of tasks computed. Then we calculated the percentage of tasks computed by those workers. We observe that 10% of all workers (26 workers) have computed 27.78% of total tasks and that 20% have computed

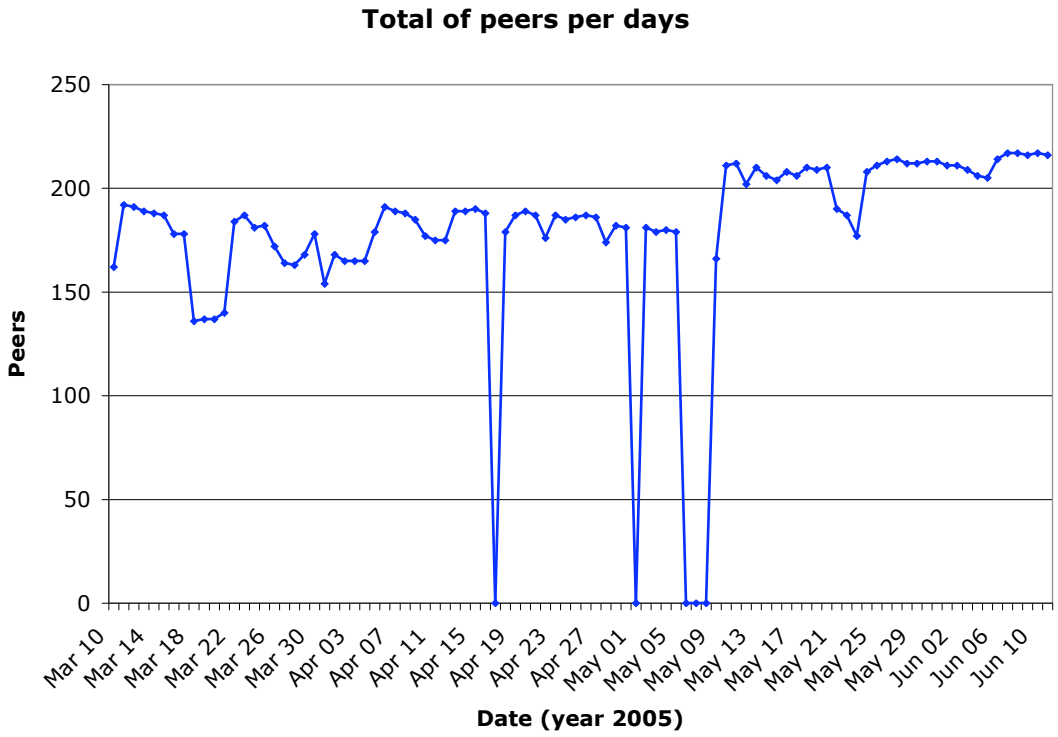

Figure 3.4: *Desktop experiments: number of peers per days, which have participated in the n-queens computation*

44.21% of total tasks. We also observed that the first 28 workers ($\approx 11\%$) of this graph are all members of the group *INRIA-2424*. We also note that $\approx 50\%$ have done $\approx 25\%$ of the total work. It is a normal observation because these machines are the most powerful and work 24 hours a day. However, it is only a half of this group (out of 53 machines). The JVMs of the second part ran on machines over-loaded because of regular use of their users, or even the JVMs were often killed by users, suffered hardware failures, or ran on more unstable systems, for instance.

The INRIA-2424 are selected in regard to their CPU power, thus it is normal that they computed a large number of tasks. Also, the INRIA-ALL is composed of a large number of less powerful machines. Figure 3.3 shows that about 46% of machines have CPUs of speed less that 2.5GHz, *i.e.* machines at least 2 years old. Nevertheless, these machines contributed with 34% of the total number of tasks.

To conclude, all this experimentation and figures show that it is hard to forecast which machines we have to choose for improving the total computation time.

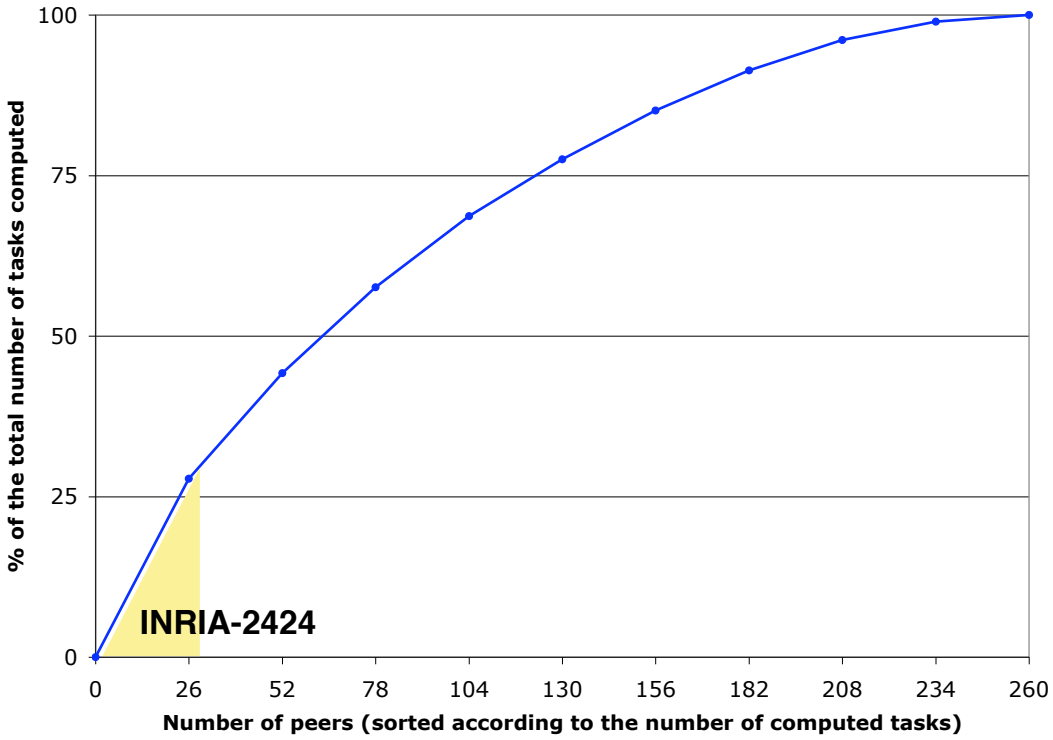

Figure 3.5: *Desktop Grid: Percentage of tasks computed by all peers*

3.5 Application to numerical hydrodynamic simulation

At the end of the year 2005, the *OASIS project* of INRIA Sophia lab started a collaboration with the *Société du Canal de Provence* company (SCP$_{id}$). The goal was to port a numerical hydrodynamic simulation software, *TELEMAC-2D*, to desktop Grids.

In this section, we present the results of that collaborative effort [CAR 06a]. First, we describe the work of SCP$_{id}$ on river modeling using finite element method. Second, using ProActive and especially our P2P infrastructure, we demonstrate how to design and program a distributed version of TELEMAC-2D, wrapping legacy code and deploying it on desktop Grids. Then, we report experiments on the *INRIA Sophia P2P Desktop Grid* (InriaP2PGrid).

3.5.1 Motivations and context

In order to review the plan of flood risk management all along the river Durance, the French government asked SCP$_{id}$ to build a numerical model of part of the river to study flood expansions and propagations [C. 05]. The concerned area represents more than 50 km of river between the Cadarache dam and the Mallemort dam. Also are present many levees or dikes that are important in flood propagation, which must be included in the model.

Two-dimensional mathematical models are increasingly used for the simulation of flood propagation in river plains. This type of model are now preferred to the former 1D network model with the conceptualization of the flood plain by a series of ponds, which necessitate adjusting a number of parameters in order to describe the conceptualized hydraulic transfer between ponds.

Finite elements description is particularly well adapted to such topography because of the possibility to refine the mesh where it is important to have a good description of the field. Hence, the obtained mesh of the considered Durance river area account more than 20,000 calculation nodes. The computation of one flood algorithm can then takes up to 38 hours on a dedicated workstation, which prevents the engineers from taking advantage of all the benefit that the model can provide.

In the past decade, hydroinformatics have seen the apparition of more powerful tools in order to improve the knowing of natural phenomenon. Some of them are two dimensional free surface flow models which are especially useful for floodplain modeling. The SCP_{id} uses a software called *TELEMAC 2D* to build such hydraulic models. This is a program for the solution of the two dimensional Saint-Venant equations in their depth-velocity form using triangular finite elements. It has been developed by *EDF-DRD*.

The objective of this work is to provide a desktop Grid environment for TELEMAC-2D. The work achieved by the OASIS project in that collaboration was:

- wrapping TELEMAC-2D code;
- adapting the P2P infrastructure for requesting specific resources, such as running Windows OS; and
- deploying TELEMAC-2D on desktop Grid.

3.5.2 River modeling using finite element method

The Saint-Venant equations The model uses the depth-velocity form of the 2D Saint-Venant equations (continuity and momentum equations):

$$\frac{\delta h}{\delta t} + \mathbf{U} \cdot \nabla h + h \nabla \cdot \mathbf{U} = S \tag{3.1}$$

$$\frac{\delta \mathbf{U}}{\delta t} + \mathbf{U} \cdot \nabla \mathbf{U} = -g \nabla Z + h\mathbf{F} + \nabla \cdot (h v_e \nabla \mathbf{U}) + \frac{S}{h}(\mathbf{U}_S - \mathbf{U}) \tag{3.2}$$

Where h is the water depth, Z is the free surface elevation, $\mathbf{U}$ is the velocity vector, $\mathbf{F}$ is the friction force vector, S is the bottom source term, $\mathbf{U}_S$ is the source term velocity vector, v_e is the effective viscosity, which includes the dispersion and the turbulence contributions.

The friction force is given by the Strickler formulae:

$$\mathbf{F} = -\frac{1}{cos\alpha} \frac{g}{h^{4/3} K^2} \mathbf{U} U \tag{3.3}$$

Where α is the steepest slope at the point and K is the Strickler coefficient.

Boundary conditions Physically two types of boundary conditions are distinguished: the solid boundaries and the liquid boundaries.

In solid boundaries there exists an impermeability condition: no discharge can take place across a solid boundary. In order to take account of friction the following relation is imposed:

$$\frac{\delta \mathbf{U}}{\delta n} = a\mathbf{U} \tag{3.4}$$

Where a is the boundary friction coefficient and n is the normal vector.

Liquid boundary condition assumes the existence of fluid domain that does not form part of the calculation domain. Four types of liquid boundaries are distinguished:

- Torrential inflow: velocity and depth prescribed
- Fluvial inflow: velocity prescribed and free depth
- Torrential outflow: free velocity and depth
- Fluvial outflow: free velocity and prescribed depth

Initial conditions In order to obtain realistic initial conditions, the simulation starts from a fictive reasonable state satisfying all the boundary conditions. Keeping the upstream discharge equal to the initial value of the flood hydrograph, the calculation is performed, and a final stationary state is obtained, with water only in the main channel. It is this stationary state which is used as initial state for the flood study.

The algorithm The Saint-Venant equations are solved in two steps. The first step allows computing the convection term with the characteristic method. In the second step, the program solves the propagation, diffusion, and source term of the equations with a finite elements scheme. Variationnal formulation associated with time and space discretization changes continuous equations in discreet linear system where unknown quantities are depth and velocity at each node. Iterative method is then used to solve this system [JM. 03].

Results The results of such a model are depth and velocity field at each node and at each time step. The analysis is done through time or space variations or maximum values of quantities. Figure 3.6 is an analysis of the flood dynamic. It represents the flow on small part of the modelled area. As the software compute the complete velocity field, it is possible to distinguish main flow from secondary flow like overtopping or weir flow above dikes. It is also possible to evaluate the modification in first bottom flow when breaches occur. Figure 3.7 represents the depth field. Places where water depth is more than one meter are identified in black while places where water depth is more than half a meter are identified in grey.

Running time of the software A calculation on such a model requires a running time of the same order than the propagation time of the flood on a dedicated workstation with two 2.4 GHz processors and 1 GB RAM. Therefore, it is difficult to optimize the model as time between one modification and its result is very long.

The Durance model is the largest one built in SCP_{id}. SCP_{id} has also built up to 10,000 calculation nodes, for running time of about 10 hours. This model has been used for the first tests of desktop Grid computing.

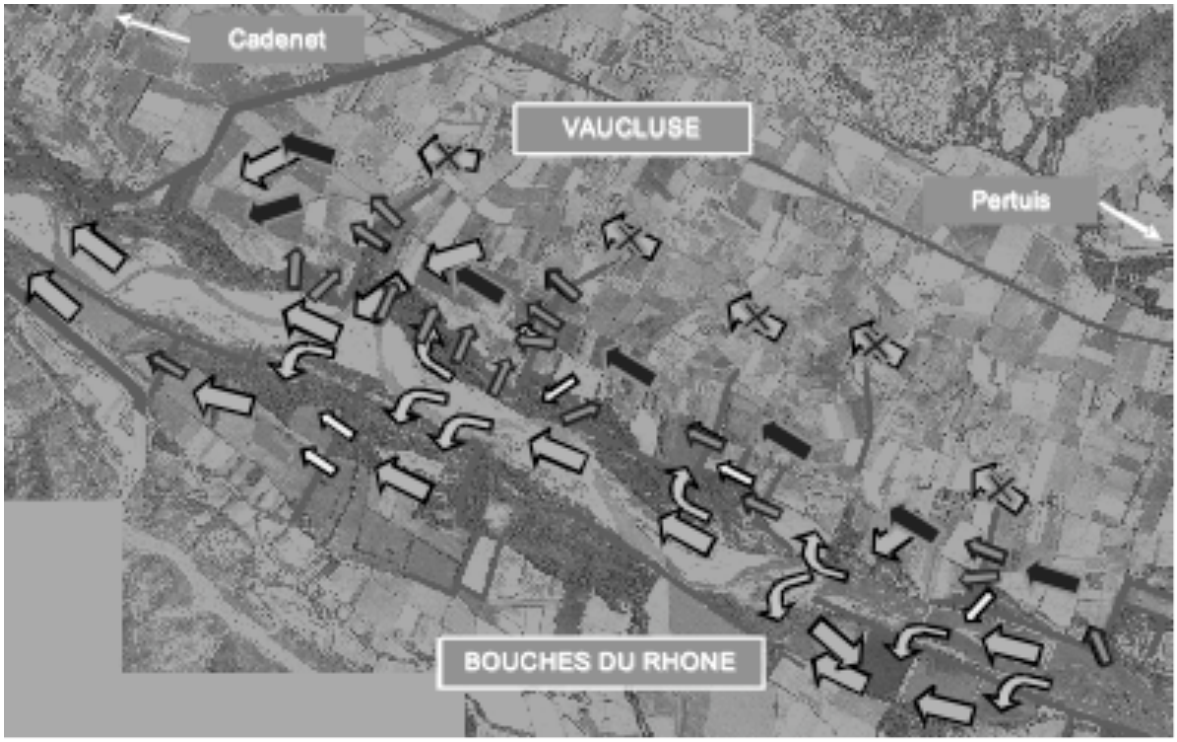

Figure 3.6: *Dynamic analysis of flood*

3.5.3 The TELEMAC system parallel version with ProActive

The parallel version of the TELEMAC System is based on the Single Program Multiple Data (SPMD) programming model, which focuses on distributing the same program on different computers, each program computes some parts of data. The TELEMAC System uses the Message Passing Interface (MPI) [GRO 96] to distribute the computation among several computers.

In order to distribute computations, an MPI process needs, before to start computing, to have on the targeted host an MPI Daemon (MPD) previously running, *i.e.* a runtime for MPI processes. The deployment of the MPI program is handled by the MPI job launcher (MPIrun), which deploys MPI processes on specified MPDs.

Thus, TELEMAC-2D users have to follow steps for deploying theirs simulations, these steps are:

1. Editing the TELEMAC-2D simulation configuration file to specify which parallel mode to use and how many CPUs are needed.
2. Starting MPDs on each host.
3. Writing MPI configuration files with addresses of MPDs for each MPI process.
4. Starting the TELEMAC-2D computation: prepare simulation data and run the MPI job launcher.

All these steps imply for users to modify two configuration files and to start process (MPDs) on all involved hosts. Furthermore, users have to run all these steps each time

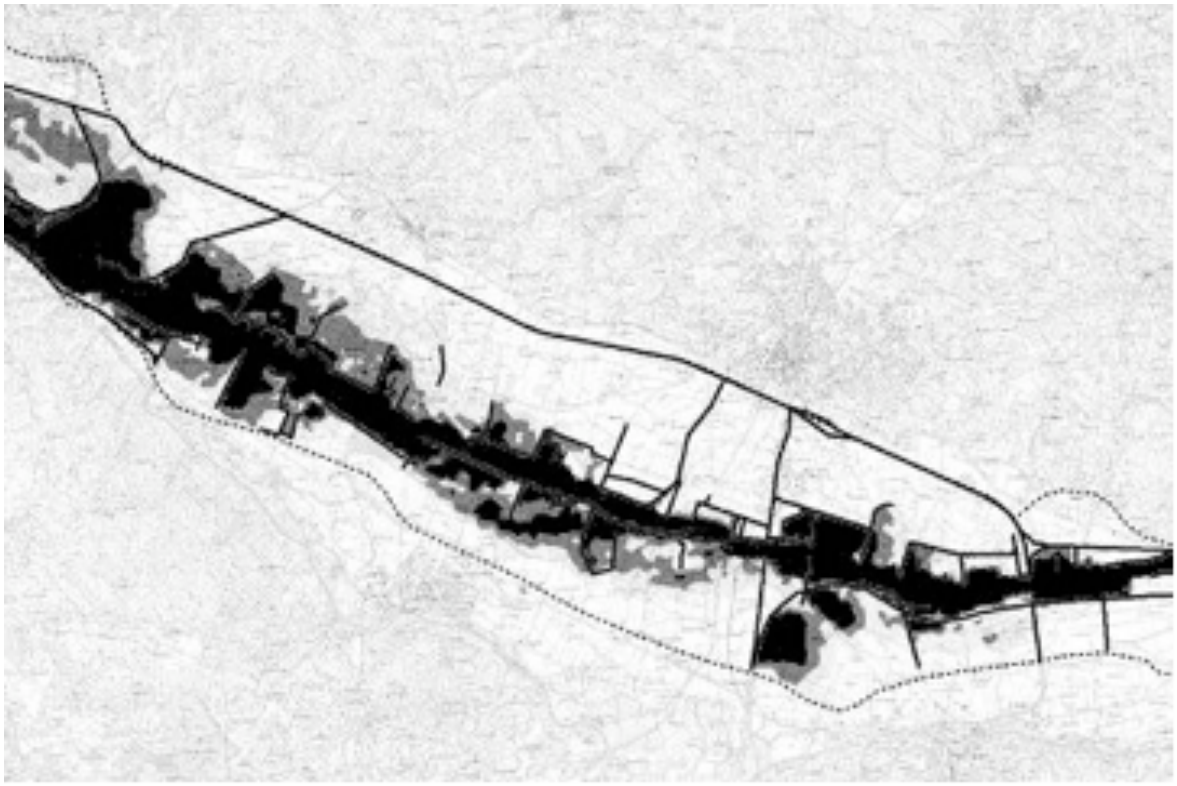

Figure 3.7: *Depth field*

they have to perform a simulation.

Users with good computer background can easily write scripts for automating these burden steps. Nevertheless, regular TELEMAC-2D users are not friendly with command-line tools, scripts, *etc.* Therefore, in this work we propose to automate all these painful steps.

Our solution turns this static deployment into a dynamic and automated one, which does not require special skills from users. The main points of the deployment mechanism which we will focus are:

- automating configuration file edition and
- providing a dynamic desktop infrastructure for deploying simulations.

Therefore, we propose a seamless system, which is composed of a graphical client interface for managing simulations and of a wrapping of TELEMAC-2D. This legacy wrapping aims to deploy simulations on a desktop Grid managed by our P2P infrastructure. In this work, we focus on the usage of the infrastructure.

TELEMAC-2D manager We have developed a graphical job management application, TELEMAC-2D manager, for seamlessly deploying TELEMAC-2D simulations. Users only have to follow some basic steps:

1. Starting the job management application.

2. Selecting a TELEMAC-2D simulation file.
3. Selecting the desired resources from the desktop Grid.
4. Starting computation.

Thus the application entirely hides connections to the desktop Grid. The application manages the acquisitions of resources and the remote MPD deployment using active objects, which are started by the application on acquired resources. It also automates all the MPI configuration file creation and modification edition process. Thereby the application eliminates all the burden steps required with the classical TELEMAC-2D.

In addition of helping to deploy simulations, the manager controls the computation. Users are able to stop, pause, and restart their simulations. TELEMAC-2D uses a check-point mechanism to frequently backup the simulation state. Also, it does not require to re-start all the simulation when a process failed; other process are paused until a new process is started.

This fault-tolerant mechanism is very important in the context of desktop Grids, because machines are unstables. Hence, TELEMAC-2D manager continually monitors resources on which the simulation is executing, and when a process is done it asks for a new resource to the P2P infrastructure, starts MPD, and then restarts the simulation from the last check-point.

Adaptation of the P2P infrastructure The infrastructure is the same as previously described in this chapter. However, TELEMAC-2D is a closed-commercial application, which runs only on Windows powered machines. In the other words, we do not have access to the application's code and we cannot modify or re-compile it.

The TELEMAC-2D manager that we have developed is able to deploy TELEMAC-2D on machines running Windows OS. The deployment is indeed creating a specific active object on machines, and call a method on the active object for locally starting a MPD process. Then, the graphical client can start the simulations.

The desktop Grid is managed by the P2P infrastructure, thus the TELEMAC-2D manager acquires resources from the infrastructure. One of the main particularities of the infrastructure is that all resources are identical, *i.e.* the infrastructure differentiates only the state of nodes (free or not). In addition, Java does not allow to collect a complete description of the system on which the JVM is running. However, the single requirement of TELEMAC-2D is the operating system; Java provides the system architecture, the name of the operating system and its version.

Thus, we have modified the resource request protocol of the P2P infrastructure to take into account the operating system requirement. The request include a new optional parameter, which is the name of the operating system. Protocols, previously presented, are still the same, except for the case of the peer has a free node to share. Before returning the node, the peer compares the operating system parameter with the local operating system name. If they match, the node is returned to the requester, otherwise the protocol runs as if the peer has no free nodes.

This improvement of the request protocols is used for constrained deployments, more details in Chapter 6.

Figure 3.8 shows the *INRIA Sophia P2P Desktop Grid* running TELEMAC-2D.

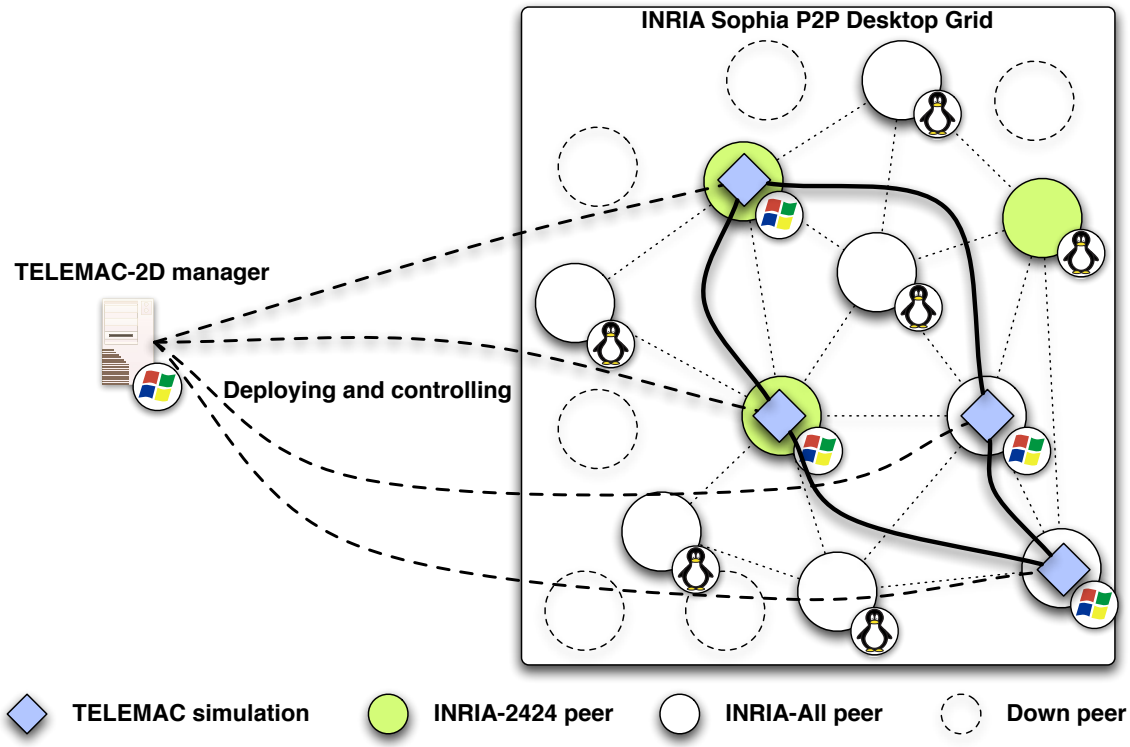

Figure 3.8: *TELEMAC-2D desktop Grid*

3.5.4 Experiments

Experimentation results for the Siagne river model using the parallel version of the TELEMAC-2D are shown by Figure 3.9. These experiments have been run out using a desktop Grid deployed at INRIA Sophia Antipolis lab; this Grid is composed of Windows desktop workstations running Pentium 4 at 3.6 GHz. We notice that the execution times are very close for both two CPUs and four CPUs. We believe that this phenomenon is due to the parallelization technique that uses sub-domain decomposition. Therefore when sub-domains are too small, workstations are communicating more than they compute, thus increasing the overall computation time.

3.5.5 Analysis

In this work we show that an MPI application, TELEMAC-2D, with a static-based deployment requires significant efforts to be deployed on a desktop Grid. Thus it involves end-users in burden tasks such as managing available resources and editing configuration files, both of them manually. Therefore we have developed a lightweight TELEMAC-2D job management application. This job manager addresses previous problems without requiring any source code modification of the original MPI application. In addition we believe that this tool is not only limited to TELEMAC-2D and can be commonly used to deploy and manages other MPI SPMD-like applications.

The P2P infrastructure helped in order to maintain a dynamic set of resources, and to select appropriate resources (running Windows). Unfortunately, we regret to do not have more Windows desktops in order to run simulation with more CPUs.S

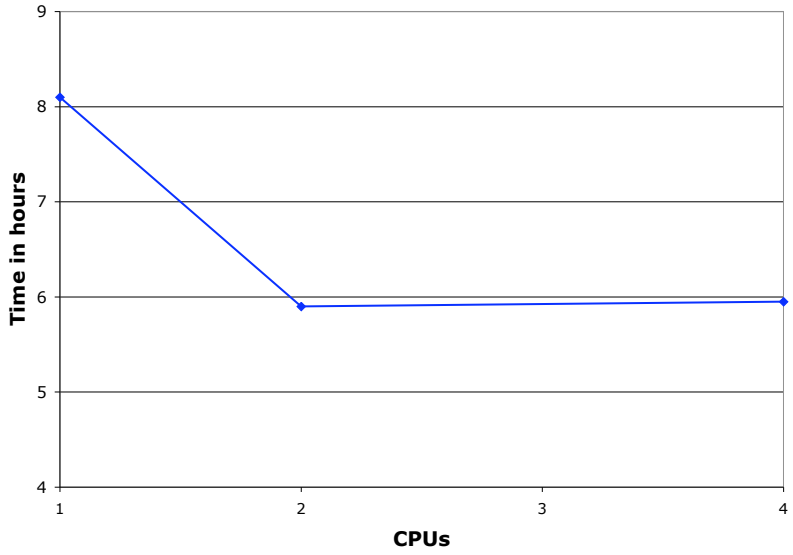

Figure 3.9: *Benchmarks of the parallel version of TELEMAC deployed on a desktop Grid*

Nevertheless, experimentation shows that even with a small desktop Grid there is a real execution time gain while running the simulation.

3.6 Conclusion

In this chapter we described our P2P infrastructure for building Grids. This infrastructure is an unstructured P2P overlay network for sharing computational resources. It also allows to deploy and to complete computations that would take months to achieve on clusters.

In order to validate our approach, we have deployed a permanent desktop Grid, managed by our P2P infrastructure, in our lab. This Grid federates under-exploited desktops of the INRIA Sophia center. With this experimental infrastructure, we are the first to solve the n-queens problem with 25 queens. This computation took six months for solving this problem instance; and thus validated that the infrastructure can be used for long-running computations.

Furthermore, we also show the capability of the infrastructure to handle and to deploy non-Java applications, such as TELEMAC-2D that is a MPI SPMD-like application.

In the next chapter, we present a parallel branch-and-bound framework for Grids, named *Grid'BnB*. We then report large-scale experiments of *Grid'BnB* with the P2P infrastructure. In these experiments, the infrastructure allows us to build a Grid composed of the *INRIA Sophia P2P Desktop Grid* and clusters that are nationally-distributed in France.

Chapter 4

Branch-and-Bound: A Communicating Framework

Previously in Chapter 2, we introduced principles about parallel branch-and-bound, and we also identified requirements that a B&B framework for Grids has to fulfill. With these requirements, we now present the second part of this thesis contribution, a parallel B&B framework for Grids. This framework is named *Gird'BnB* [CAR 07c].

First, we remember the motivations and the objectives of this framework. Second, we describe the architecture and the implementation. Then, we report experiments on a cluster and a nation-wide Grid.

4.1 Motivations and objectives

In Chapter 2, we identified the requirements that a B&B framework must fulfill, from the users point of view:

- **Hiding parallelism and Grid complexities**. Parallel programming is complex and especially in Grid computing, which adds new programming challenges to manage. The main goal of this framework is to hide all parallelism and Grid complexities from the users. Thus, users only need yo code the algorithms for solving their optimization problems.
- **Combinatorial optimization problems**. B&B is an algorithmic technique for solving several kind of problems. In this thesis work, we test our framework with combinatorial optimization problems.
- **Ease of deployment**. Deployment of applications on Grids is a complex task. Using an adequate underlying Grid middleware helps users to easily deploy their applications on Grids.
- **Principally tree-based** parallelism strategy, we presented several techniques for paralleling B&B and identified that the tree-based is the most studied. For that reason, we choose to based the framework on this one. However, users can easily implement other.
- **Implementing and testing search strategies**, a generic search strategy cannot be used to solve all combinatorial problems. Depth-first search may be good for a given problem and bad for another. Hence, the framework proposes several strategies, and allows users to provide their own.

- **Objective function** is the problem to solve. It is what users have to implement and to optimize. The framework must be enough well designed for that performance issues result from the objective function and not from the framework's core.

In addition of these user-level requirements, we defined the characteristics that a parallel B&B framework for Grids as to satisfy:

- **Asynchronous communications** hides latency and bandwidth reduction involved by Grid environments. Also, asynchronous communications between distributed processes avoid computation pauses for synchronization. Thus, the framework can efficiently use parallelism and takes the maximum profit from the large pool of resources provided by Grids.
- **Hierarchical master-worker**, Aida *et al.* [AID 05] show that running a parallel B&B application based on a hierarchical master-worker architecture scales on Grids.
- **Dynamic task splitting**, this requirement results from the choice of using the master-worker paradigm. The solution tree is indeed generated as a set of tasks, each task representing a sub-part of the tree. As the tree is unknown at the beginning, tasks are dynamically produced as long as branching is operated.
- **Efficient parallelism and communications**, both asynchronous communication and master-worker are not enough to provide an efficient parallel framework. We already identified that the productivity of the pruning operation depends on how the global lower/upper bound is updated on each process. Thus, impacting on the overall computation performance.
  - *sharing the current best lower/upper bound with communications*, each process keeps a local copy of the best bound. Then, when a better bound is found, the process send the new value to other.
  - *communications between workers*, because we choose master-worker paradigm, processes are workers. For scalability and performance issues, using central communication (through the master) is not possible with a large number of workers.
  - *organizing workers in groups to optimize inter-cluster communications*, due to the large number of worker, a communication from a worker to all can take a while. On the other hand, Grids are usually built of clusters, which are high speed communication environments, inter-connected by shared networks. Thus, communications inside cluster are very efficient but communications between clusters are slower. We propose to organize workers in groups to optimize the update of the best bound.
- **Fault-tolerance**, the probability of failures is dramatically high for Grid systems: a large number of resources imply a high probability of failures of one of those resources. Hence, the framework must be able to handle node failures.

With all these requirements for parallel B&B, and more specially on Grid environments, we propose, in this thesis, *Grid'BnB* a complete Java API for using parallel B&B technique with Grids. *Grid'BnB* aims to hide Grid difficulties to users. Especially, fault-tolerance, communication, and scalability problems are solved by *Grid'BnB*. The framework is built over a master-worker approach and provides a transparent communication system among tasks. Local communications between processes are used to optimize the

exploration of the problem to solve. *Grid'BnB* is built on top of the ProActive Grid middleware (fully described in Section 2.4).

We also present a mechanism based on communications between workers to share the best upper bound (GUB). Thereafter, we propose a system to dynamically organize workers in groups of communication. This organization aims to control communication for scalability.

In summary, *Grid'BnB* main features are:

- master-worker architecture with communication between workers;
- dynamic task splitting;
- different search tree algorithms;
- sharing the current best upper bound with communications;
- organizing workers in group to optimize inter-cluster communications; and
- fault-tolerance.

4.2 Design and architecture

We now present in detail the design and the architecture of *Grid'BnB*, the parallel B&B framework for Grids.

4.2.1 Entities and their roles

Grid'BnB is based on a hierarchical master-worker architecture. We already defined master-worker in Definition 2.2.1. Hierarchical master-worker introduces a new entity comparing to simple master-worker. The third entity is the *sub-master*. Its role is similar to the master: distributing tasks to workers and gathering partial results. The difference between the master and the sub-master is that the sub-master acts like an intermediary. In other words, the sub-master requests tasks to the master and sends results to the master. Figure 4.1 shows an example of hierarchical master-worker architecture. Note that some workers may directly interact with the master, without passing through a sub-master.

The hierarchical master-worker paradigm is one of solutions to avoid performance degradation in classical master-worker paradigm for Grids. The advantages of this architecture are: first, to reduce communication overhead by organizing worker in groups; and second, to avoid a single over-loaded master becomes a performance bottleneck.

Because in our model, workers have to communicate with each other in order to share the GUB, we have to adapt the hierarchical master-worker paradigm for this requirement. Thus, *Grid'BnB* is composed of four kind of entities: *master*, *sub-master*, *worker*, and *leader*. We consider that the sub-master cannot handle communication because sub-masters are deployed by the users, thus sub-masters do not necessarily reflect the Grid topology.

We now define and describe in detail the role of each entities:

- **Master** is the unique entry point: it receives the entire problem to solve as a single task (it is the *root task*). At the end, once the optimal solution is found, the master returns the solution to the user. Thus, the master is responsible for branching the

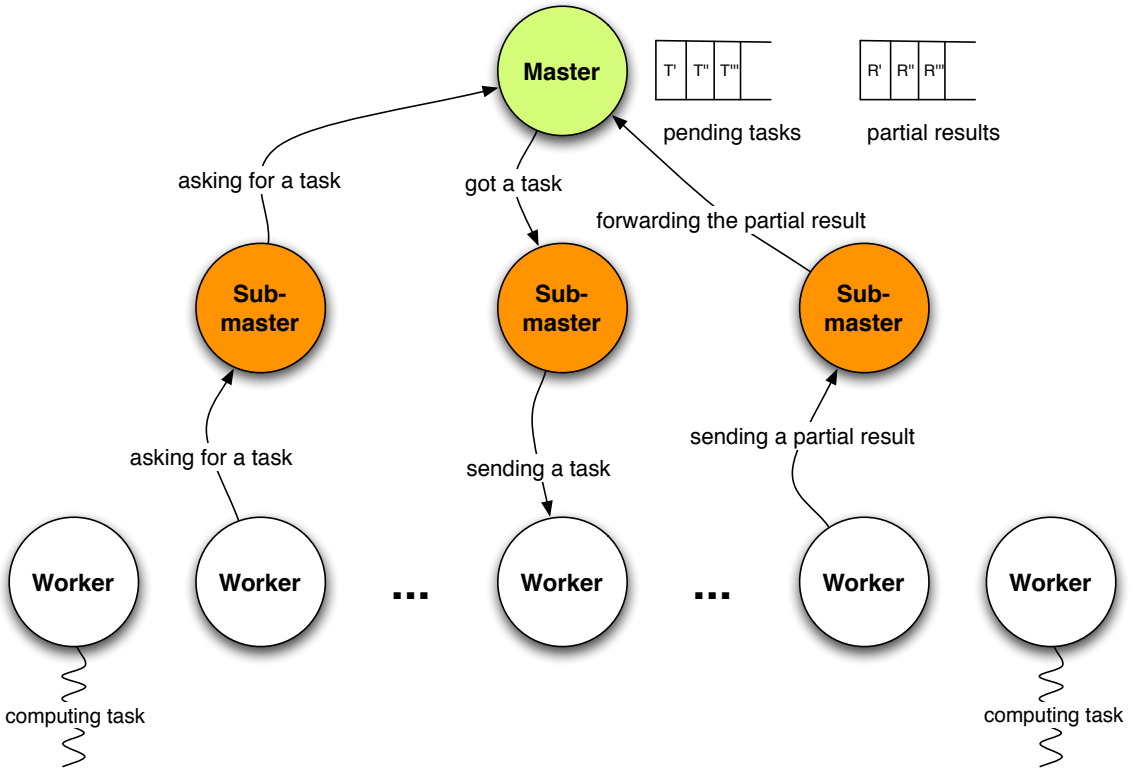

Figure 4.1: *Hierarchical master-worker architecture*

root task, managing task allocation to sub-masters and/or workers, and handling failures.

- **Sub-master** are intermediary entities whose role is to ensure scalability. They are hierarchically organized and forward tasks from the master to workers and vice versa by returning results to the master (or their sub-master parent).
- **Worker** is to execute tasks. They are also the link between the tasks and the master. Indeed when a task does branching, sub-tasks are created into the worker that sent them to the master for remote allocation.
- **Leader**: is specific role for workers. Leaders are in charge of forwarding messages between clusters (more details in Section 4.2.5).

Figure 4.2 shows the global architecture of *Grid'BnB*. Sub-master and leader entities are indeed specific roles of master and worker: a master can be *sub-master* when it manages workers and has a master parent, and a worker can be *leader* when it is in charge of forwarding communications between clusters.

Sub-masters and leaders look similar because they are both an entity, which manages workers. However, they have two clearly different functions. The sub-master forwards tasks and results between the master and workers, its role is to improve scalability of the architecture. On the other hand, leaders are not mandatory to the architecture,*i.e.* without leaders the framework is still able to solve user problems and also to scale. They aim to improve communication performances between workers in order to share GUB. The leader is more detailed in Section 4.2.5.

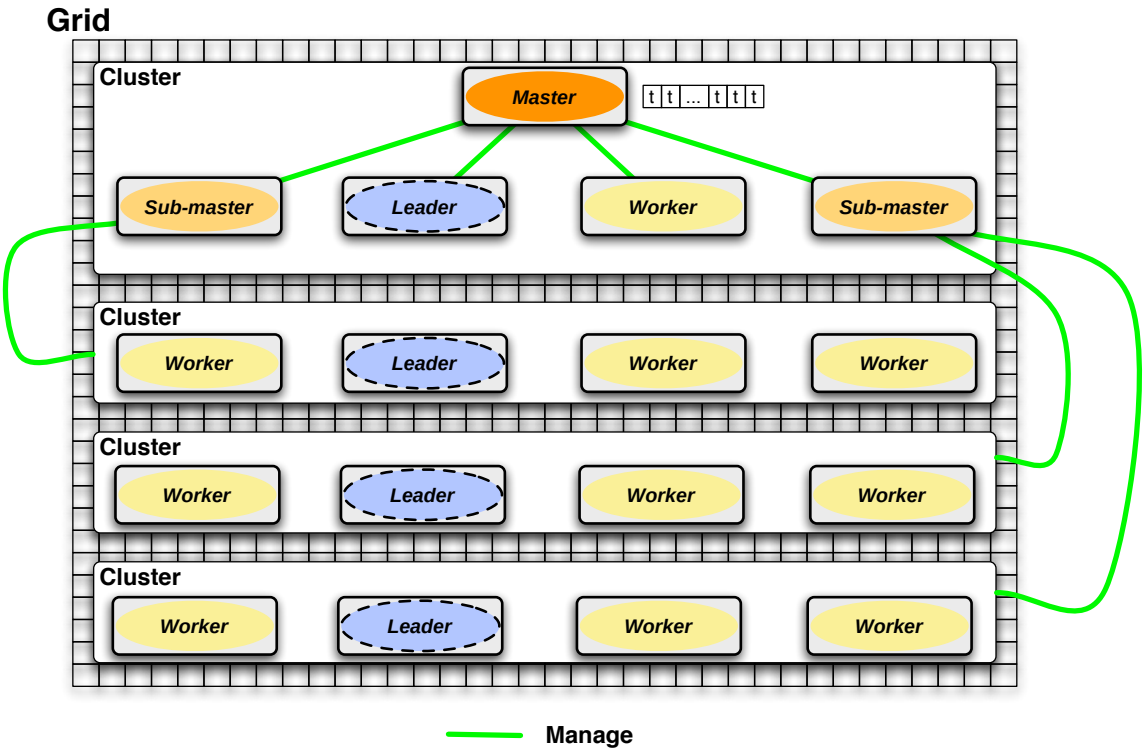

Figure 4.2: *Global archictecture of Grid'BnB*

4.2.2 The public library

Users who want to solve combinatorial optimization problems have to implement the task interface provided by the *Grid'BnB* API. Figure 4.3 shows the task interface provided by the framework API.

```
public abstract class Task<V> {
    protected V GUB;
    protected Worker worker;

    public abstract V explore(Object[] params);

    public abstract ArrayList<?extends Task<V>> split();

    public abstract void initLowerBound();

    public abstract void initUpperBound();

    public abstract V gather(V[] values);
}
```

Figure 4.3: *The task Java interface*

The task interface contains two fields:

- GUB is a local copy of the global upper bound; and
- worker is a reference on the associated local process, handling the task execution.

The main method that users have to implement is explore, this is the objective function. The result of this method, of type V, must be the optimal solution for the feasible region represented by the task. V is a Java 1.5 generic type: the user defines the real type.

The branching operation is implemented by the split method. In order to not always send to the master all branched sub-problems, the *Grid'BnB* framework provides, via the worker field, the method availableWorkers, which allows users to check how many workers are currently available. Depending on the result of this method, users can decide to do branching and to locally continue the exploration of the sub-problem. In other words, this method helps users to program tasks and to dynamically determine the granularity of the tasks.

To help users to structure their codes, we introduce two methods to initialize bounds: initLowerBound and initUpperBound. These two methods are called for each task just before the objective function, explore, and they are not mandatory. The last method to implement is gather: the (sub-)master calls this method when all its tasks are solved. The method returns the best results from all tasks, *i.e.* the optimal solution.

Figure 4.4 shows the interface of the worker. All the interface's methods are implemented by the framework itself. Users call these method from their task objects, which implement the previous presented interface Task.

```
public interface Worker <T extends Task, V>{
        public abstract IntWrapper availableWorkers();

        public abstract void sendSubTasks(ArrayList<T> subTasks);

        public abstract void newBestBound(V betterBound)
}
```

Figure 4.4: *The worker Java interface*

The method availableWorkers has been already presented, however the return type is a special object: IntWrapper. This object encapsulates a Java int. Because we implement the framework with the active object programming model provided by the ProActive Grid middleware, this model allows to call methods in asynchronous way except when methods return a Java primitive types, such as int. Thus, we have to use a primitive type wrapper provided by the middleware.

Like availableWorkers all worker interface methods have to be call from the task. The method sendSubTasks send the result of the split method to the master or sub-master. The last method, newBestBound, update the GUB in all workers, more details in Section 4.2.5.

A complete example of this API is described in Section 4.4.

4.2.3 The task life-cycle

The master, which is the entry point, takes a single task from the user. This task implement the task interface previously presented. This task is the *root task*, it represent the whole problem to solve. This section presents the execution of the root task, *i.e.* the solving of the problem.

The root task is passed to the master that performs the first branching, which provides sub-tasks. Then when a task is allocated to a worker, the worker starts to explore it. Figure 4.5 shows the state diagram of the task during its execution. As soon as a worker is available, a new task can be allocated. The worker starts by heuristic methods to initialize lower/upper bounds for the current feasible region, then it calls the objective function. Within the objective function the user can decide whenever it is needed to branch the current region with the help of the availableWorkers method, which returns the current number of free workers.

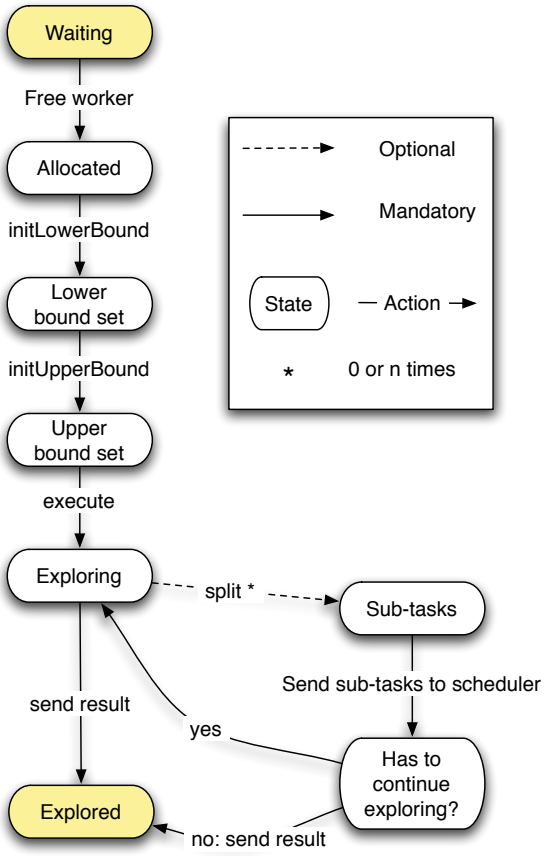

Figure 4.5: *Task state diagram*

4.2.4 Search tree strategies

The tasks allocation is handled by the master, and it is indeed the search tree strategy; thereby the master works as a queue for task scheduling. The exploration algorithm of the search tree is important regarding performances. Therefore, *Grid'BnB* allows

users to choose the best adapted algorithm to solve their problems. We propose four algorithms:

- *breadth-first search* explores the tree in larger;
- *depth-first search* explores all branches one by one;
- *first-in-first-out* (FIFO) explores the tree following the order tasks have been sent to the master; and
- *priority* explores in priority branches that updated the GUB the most frequently.

If none of those algorithms satisfy the problem, users can implement their owns. The framework provides a public interface for this.

4.2.5 Communications

In order to minimize the execution time, the framework proposes to share the best current lower/upper bound, here the upper bound (GUB).

The tasks produce new GUB candidates while they are computed by workers. The GUB must be available to all tasks to prune the maximum of none promising branches of the search tree. The strategy for sharing GUB is to use a local copy of GUB on all workers and to broadcast updated value. Figure 4.6 shows the process of updating GUB when a worker finds a new better upper bound.

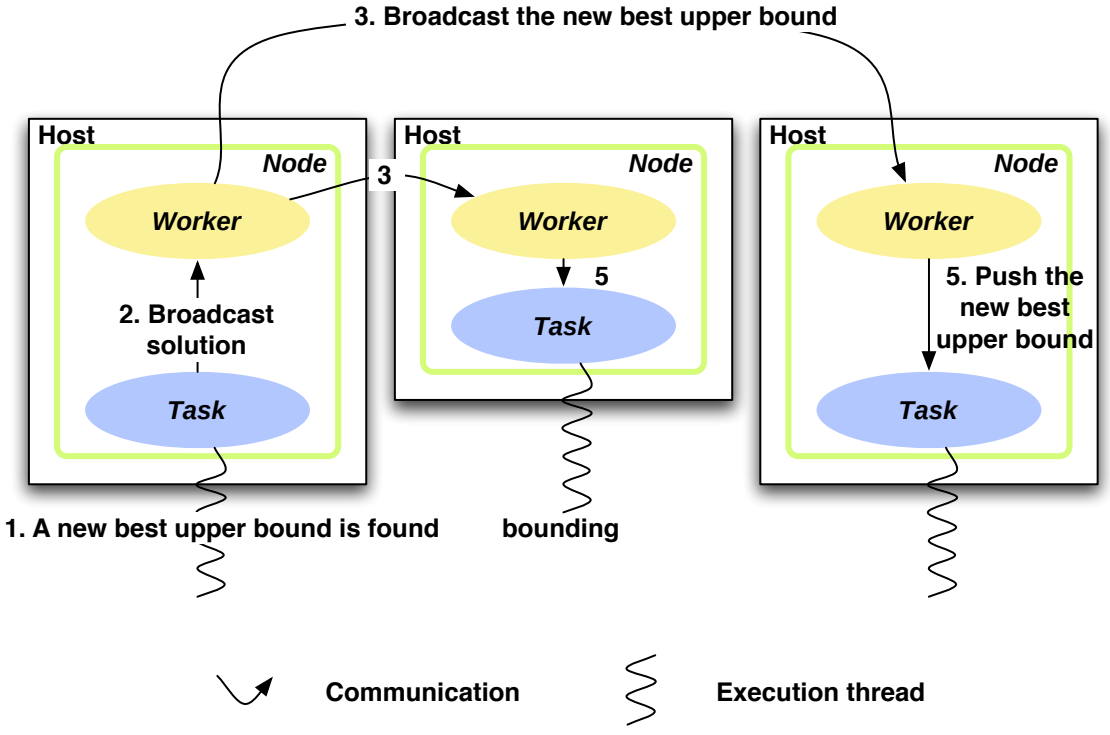

Figure 4.6: *Update the best global upper bound*

To be efficient, a B&B framework has to broadcast the GUB as fast as possible. With a large number of workers, directly broadcasting GUB to every worker cannot scale. For that reason *Grid'BnB* organizes workers in groups.

Groups are sets of workers, which can efficiently broadcast GUB between them. The master is in charge of building groups. Thus, the main criterion to put workers in the same group is their localization on the same cluster. Clusters usually provide a high performance environment for communication. Thereby there is no scalability issue to broadcast GUB within the same cluster. The master elects a worker as *leader* in each group. This leader has a reference to all other group leaders. When a leader receives a communication from outside its group, it broadcasts the communication to its group. Inversely when the leader receives a communication from a member of its group, it broadcasts the communication to the other leaders. To limit useless broadcast, leaders forward communication only if the new upper bound is better than their own GUB value. Figure 4.7 shows an example of broadcasting GUB between groups.

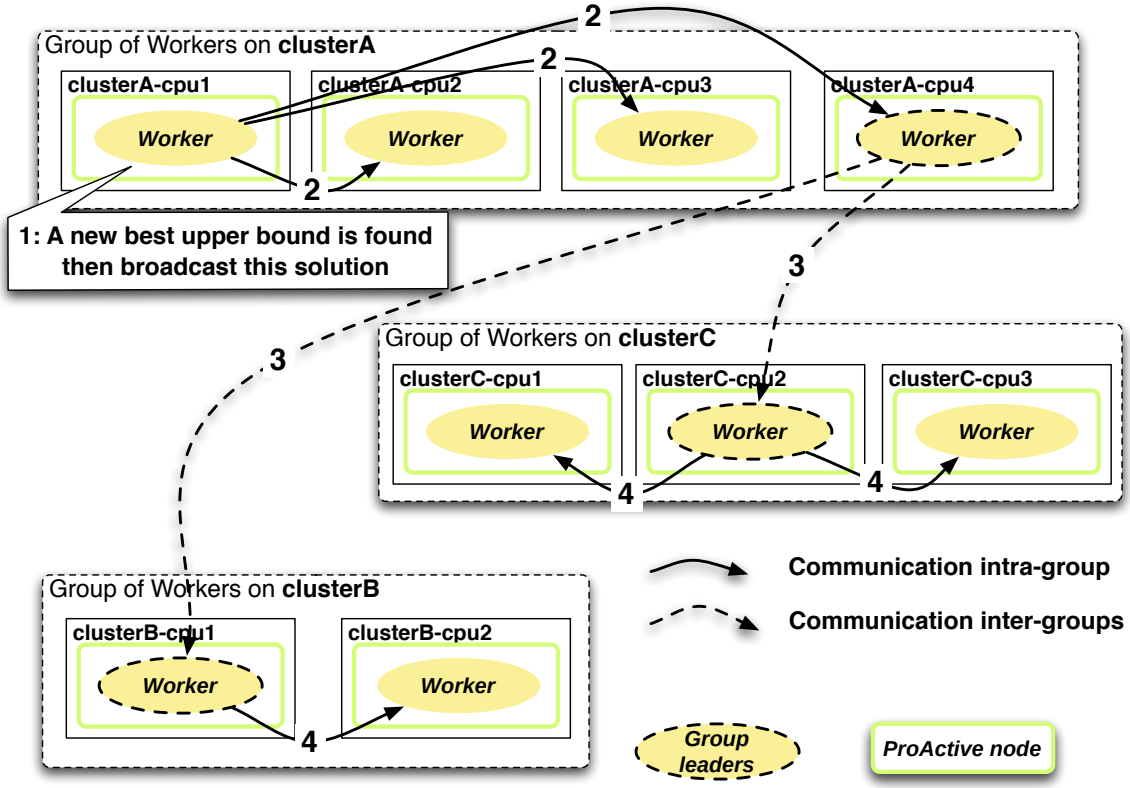

Figure 4.7: *Broadcasting solution between group of workers*

The localization of workers is done by an extension of the ProActive deployment mechanism (described in Section 6.1). This improvement is one of this thesis contribution and aims at localizing computational resources on Grids. With an abstract deployment, we are able to detect architectural locality at runtime. The goal of this contribution is to provide the Grid topology to applications in order to improve scalability and performance. The localization mechanism is presented in Section 6.2.

4.2.6 Load-balancing

Even if load-balancing comes naturally from master-worker paradigm, it is important to explain how it works. Especially due to higher possibility of failure, tasks can become

very imbalanced.

The load-balancing is indeed not transparent. The framework provides a method to help users to take maximum advantage of the large number of workers. This method is availableWorkers, which returns the number of current available workers (waiting for tasks). Ideally, users have to branch tasks until the number of available workers reaches zero, then wait for free workers before branching again.

The second problem with load-balancing is when a task fail, failures and fault-tolerance are treated in the next section. Once the fault is detected by the framework, the master reallocates the task as soon as a worker is available for computation. The task is executed before the other, which are waiting to be executing.

4.2.7 Exceptions and fault-tolerance

We distinguish two kind of failures: *user failures* and *infrastructure failures*.

User failures Within the users code, errors can occurs, such as uncaught exceptions. Users exceptions are handled by workers. When a worker catches an exception, the worker forwards it to the master, and then the master stops the whole computation and returns the exception to the user.

Infrastructure failures The last feature of *Grid'BnB* is the fault-tolerance. It is an important issue on Grid environments; the large number of resources that are distributed on different administration domains implies a high probability of faults, such as hardware failures, networks down time, or maintenance.

Master and sub-masters hierarchically manage infrastructure failures, such as host failures. The master monitors sub-masters and the sub-masters monitor workers. The monitoring consists of frequently pinging monitored entities. When the ping call fails (communication timeout, network errors, *etc.*), the remote host is considered as unreachable and down.

When a worker is unreachable, the master re-allocates the task to the next available worker. If for the same task several results are returned to the master (worker considered down for network problem and come back), only the first one is kept, others are flushed. The master handles the fault of sub-masters: if a sub-master does not answer to a ping call, the master chooses a free worker and re-instantiates it as a sub-master. The master handles the host fault of leaders; the master frequently pings leaders. When a leader is unreachable, the master elects a new leader in the group.

The master has to be deployed on a stable machine, because it is at the top of the monitoring hierarchy. As opposed to sub-masters and workers, master host failures cannot be dynamically handled by the framework but require users intervention. The status of the current execution (current GUB and all tasks) is frequently serialized on the disk of the master. Thus for long-running problem, if the host of the master faults the user can restart the solving at the last state of the execution.

Grid'BnB provides a high level programming model for solving problems with parallel B&B. From the users points of view, the framework handles all issues related to distribution/parallelism and fault-tolerance. The API is also very simple: it consists only in the Task interface. Thus, users can focus exclusively on their objective function.

In the next sections, we describe the framework implementation within ProActive and present an user example.

4.3 Implementation

Grid'BnB is implemented within the Java ProActive Grid middleware, described in Section 2.4. ProActive is based on the active object programming model.

Master, sub-masters, and workers are active objects. Each active object serves remote calls in FIFO order. The master manages futures on current executing tasks.

Then, groups of workers are ProActive groups. Leaders are also member of a ProActive group. Thereby, hierarchical ProActive groups represent workers. A hierarchical group is indeed a group of groups.

Finally, in order to optimize communication between workers to solve more rapidly problems, the management of workers in groups relay on the ProActive deployment framework. ProActive features a system for the deployment of applications on Grids. The Chapter 6.2 explains the deployment mechanism and how we extended it to manage organization of workers in groups of communications.

4.4 Grid'BnB user code example: solving flow-shop

In this section, we illustrate *Gird'BnB* with a complete example of solving a combinatorial optimization problem, the flow-shop.

4.4.1 The flow-shop problem

Flow-shop is a NP-hard [GAR 76] permutation optimization problem. The flow-shop problem consists in finding the optimal schedule of n jobs on m machines. The set of jobs is represented by $J = \{j_1, j_2, \ldots j_n\}$, each j_i is a set of operations $j_i = \{ o_{i1}, o_{i2}, \ldots o_{im} \}$ where o_{im} is the time taken on machine m and the set of machines is represented by $M = \{m_1, m_2, \ldots m_m\}$.

The operation o_{ij} must be processed by the machine m_j and can start on machine m_j if it is completed on m_{j-1}. The sequence of jobs are the same on every machines, *e.g.* if j_3 is treated in position 2 on the first machine, j_3 is also executed in position 2 on all machines.

We consider the mono-objective case, which aims to minimize the overall completion time of all jobs, *i.e. makespan*. The makespan is the total execution time of a complete sequence of jobs. Thus, the mono-objective goal is to find the sequence of jobs that takes the shortest time to complete.

Figure 4.8 shows an example of flow-shop permutation problem schedule.

4.4.2 Solving flow-shop with Grid'BnB

4.4.2.1 The algorithm

For this example, we apply a simple B&B algorithm to solve the flow-shop problem. The technique consists of enumerating all possible permutations of jobs, and of pruning non-promising partial enumerations.

This algorithm is far from being the fastest algorithm to solve flow-shop with B&B. We use this algorithm because of its simplicity as an example. A potential improvement

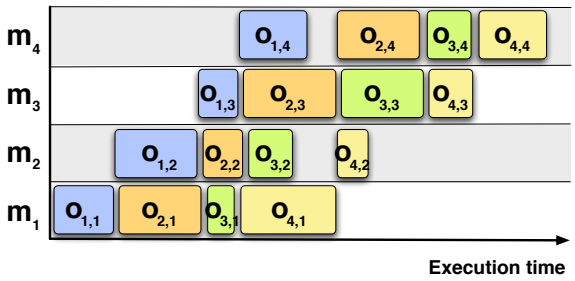

Figure 4.8: *Example of flow-shop permutation problem schedule*

of this example could to use the algorithm proposed by Lageweg [LAG 78].

The search tree is dynamically generated and it is represented by tasks. The root task represents the whole tree. *Gird'BnB* firstly branches the root task, for this example, the root task splits in n sub-tasks, where n is the total number of Jobs.

When a task is executed by a worker, the task enumerates all permutations of its tree part. The task frequently checks the availability of free workers, by calling the method availableWorkers. When workers are free, the task does the branching operation by splitting itself in sub-tasks. The task may improve the current value of GUB, in this case the task informs others by calling the method newBestBound. The task is also responsible for pruning bad tree branches of its part. Figure 4.9 shows the decomposition of the search tree in tasks.

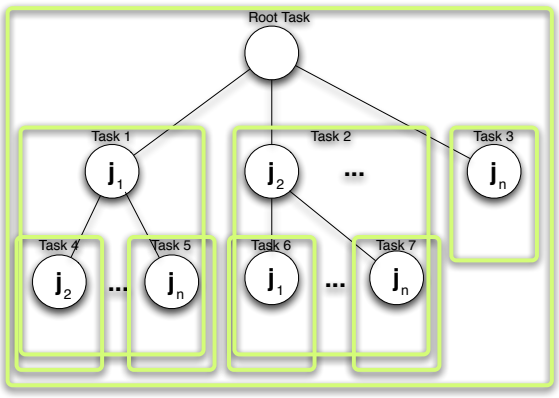

Figure 4.9: *Flow-shop search tree decomposed in task*

4.4.2.2 The root task

The root task is implemented by the user and represents the whole problem to solve.

The flow-shop task is the interface that the user has to implement:

```
public class FlowShopTask extends Task<FlowShopResult> {

 /**
  * Construct the root task with the flow-shop instance to solve.
```

```
   * This instance can be a sub-part of the whole instance to solve.
   */
  public FlowShopTask(FlowShopProblem p) {
    this.flowshopProblem = p;
  }
}
```

The user must implement several methods in the task.

Initializing bounds for the sub-part of the search tree represented by the task. For this example, we do not initialize the lower bound. The upper bound is initialized with the makespan of a permutation randomly chosen in set of permutation of the task, if better than GUB, otherwise it is GUB.

```
// Compute the lower bound
public void initLowerBound() {
  // nothing to do
}
// Compute the upper bound
public void initUpperBound() {
  this.upperBound = this.computeUpperBound(this.p);
}
```

Branching: split method Split the task in ten sub-tasks.

```
public Vector split () {
  // Divide the set of permutations in 10 sub-tasks
  int nbTasks = 10;
  Vector tasks = new Vector(nbTasks);
  for (int i = 0 ; i < nbTasks ; i++){
    tasks.add(new FlowShopTask(this, i, nbTasks));
  }

  return tasks;
}
```

The objective function and pruning are implemented by the execute method:

```
FlowShopResult fsr;

public Result execute() {

  if (! this.iHaveToSplit()) {
  // Test all permutation
    while((FlowShopTask.nextPerm(currentPerm)) != null) {
        int currentMakespan;
        fsr.makespan = ((FlowShopResult)this.bestKnownSolution).makespan;
        fsr.permutation = ((FlowShopResult)this.bestKnownSolution).permutation;
        if ((currentMakespan = FlowShopTask.computeConditionalMakespan(
            p, currentPerm,
            ((FlowShopResult) this.bestKnownSolution).makespan,
            timeMachine)) < 0) {
        //bad branch
```

```
            int n = currentPerm.length + currentMakespan;
            FlowShopTask.jumpPerm(currentPerm, n, tmpPerm[n]);
            // ...
          } else {
            // better branch than previous best
            fsr.makespan = currentMakespan;
            System.arraycopy(currentPerm, 0, fsr.permutation, 0,
                             currentPerm.length);
            r.setSolution(fsr);
            this.worker.newBestBound(r);
          }
        }
      } else {
      // Using the Stub for an asynchronous call
        this.worker.sendSubTasksr(
          ((FlowShopTask) ProActive.getStubOnThis()).split());
      }

          // ...

      r.setSolution(bestKnownSolution);
      return r;
    }
```

The main method :

```
Task task = new FlowShopTask(flowShopInstance);
Manager manager = ProActiveBranchNBound.newBnB(task,
                          master_node);
manager.addNodesForWokers(workerNodes);
// Start solving flow-shop
FlowShopResult r = manager.start(); // this call is asynchronous
// Do something
System.out.println(r);
```

4.5 Experiments with flow-shop

This section describes the experiments achieved for testing *Grid'BnB*. Experiments have been done with a permutation optimization problem, the flow-shop. We first present and analyze experiment results on a single cluster. Finally, we present experiment results on a large-scale Grid, named *Grid'5000*.

4.5.1 Single cluster experiments

These experiments aim to choose the best search strategy with flow-shop and to determine the impact on performances of dynamically sharing GUB with communications. We use the *Nef* cluster of INRIA Sophia lab, composed of 32 nodes, powered by dual-processors AMD Opteron with a speed of 2GHz. Nodes are connected via Gigabit Ethernet.

Search strategies evaluation: Figure 4.10 shows benchmark results of applying different search strategies (described in Section 4.2.4) to flow-shop. The selected instance of flow-shop is 16 jobs / 20 machines. Results show that FIFO is for all experiments always the fastest, the speedup between 20 CPUs and 60 CPUs is 4.63. This is a super linear speedup (> 3) owing to increase the total of CPUs allows a larger generation of the search tree in parallel and thereby, improving the GUB faster to prune more none promising branches. Breadth-first search scales with a very good speedup, the speedup between 20 CPUs and 60 CPUs is 5.44, also super linear. The high speedup is normal because more breadth-first search is deployed on nodes the more the tree is explored in parallel. Depth-first search speedup is linear, 3, and for priority search the speedup is 1.73. The speedup is particularly high with all these experiments, because with 60 CPUs the chosen flow-shop instance can be widely explored in parallel whatever the search strategy. The built search tree rapidly provides the best solution as upper bound, thus each process can delete many branches.

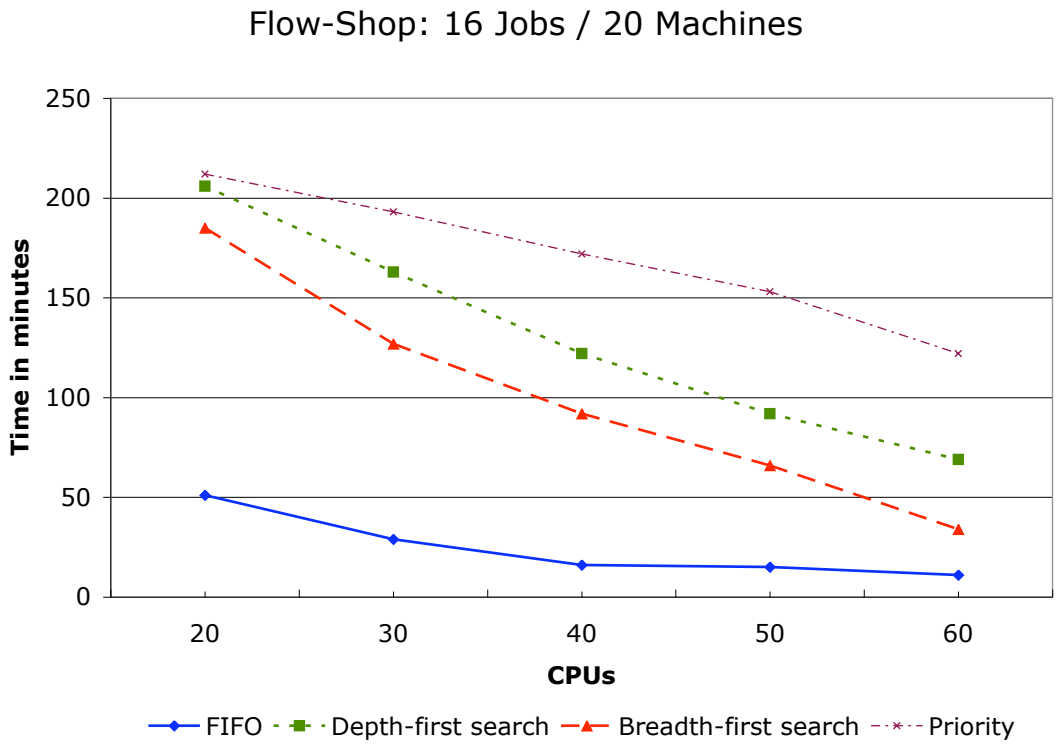

Figure 4.10: *Single cluster experiments: benchmarking search tree strategies*

Experiments report very good results with super linear speedup or with speedup under linear. Previous work have already reported these observations [LAI 84, LI 86], and these phenomena are known as *speedup anomalies*.

They are mainly due to the fact that the time needed to explore the whole search tree depends on the tree size. The branching, bounding, and the tree node processing priority (*i.e.* the search strategy) impacts on the size tree.

Evaluation of dynamically sharing the global upper bound with communication: With the same instance of flow-shop and with the FIFO strategy, we benchmarked the impact of dynamically sharing GUB with communications; experiments are deployed on the same cluster as previously. We first benchmark flow-shop with communications between workers for sharing GUB and then benchmark flow-shop without dynamically sharing GUB between workers (no communication). In the case of no communication, the master keeps the GUB up-to-date with all results from computed tasks; and when a task is allocated to a worker by the master, it sets the current GUB value to the task. Figure 4.11 shows results of solving flow-shop with and without dynamically sharing GUB. Using communications to share GUB is always better. But the speedup, $\frac{T\ No\ Communications}{T\ Communications}$, is lower for 50 CPUs than 40 CPUs, this decrease comes from the fact that more than 40 CPUs this flow-shop instance has enough CPUs to explore the whole tree in parallel, *i.e.* it is the optimal deployment.

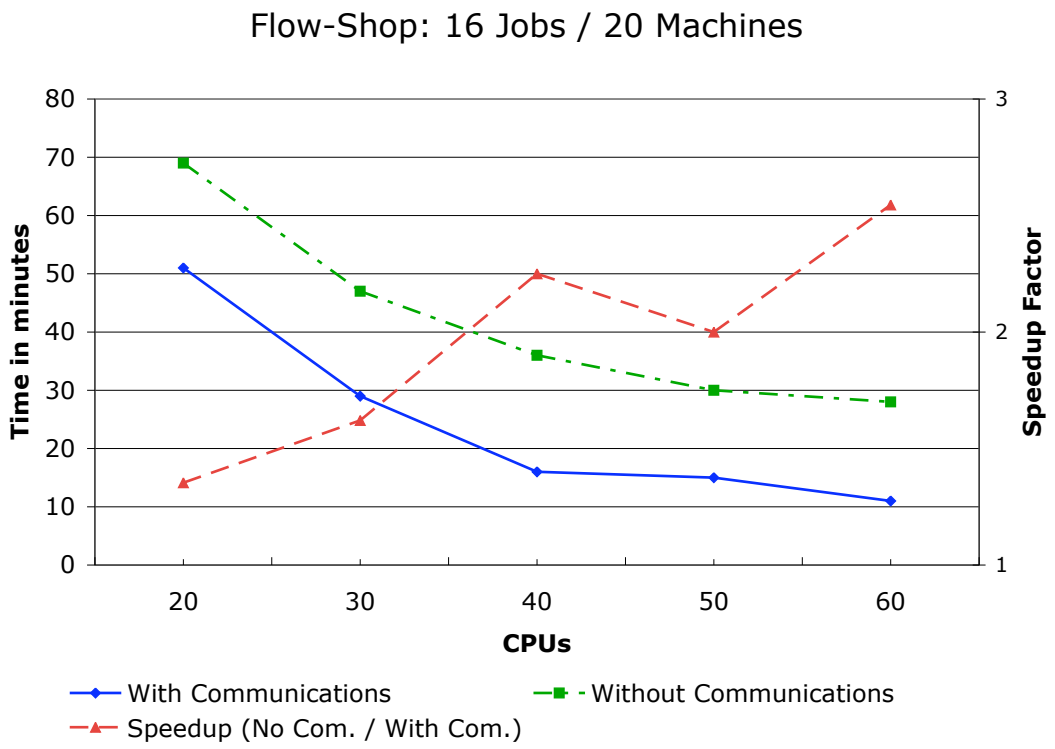

Figure 4.11: *Single cluster experiments: sharing GUB with communications versus no dynamic GUB sharing*

These experiments on a single cluster show that dynamically sharing GUB with communications between workers improve execution time, and that choosing the right search strategy considerably affects performances.

4.5.2 Large-scale experiments

In order to experiment *Grid'BnB* on Grids, we have access to a large-scale nationwide infrastructure for Grid research, *Grid'5000*.

Grid'5000 project (G5K) [CAP 05] The G5K project aims at building a highly reconfigurable, controllable, and monitorable experimental Grid platform gathering 9 sites geographically distributed in France currently featuring a total of 3586 CPUs. The main purpose of this platform is to serve as an experimental testbed for research in Grid Computing. For the moment, 17 laboratories are involved, nation wide, in the objective of providing the community of Grid researchers a testbed allowing experiments in all the software layers between the network protocols up to the applications.

The current plans are to assemble a physical platform featuring 9 local platform (at least one cluster per site), each with 100 to a thousand PCs, inter-connected by RENATER the French national education and research network. G5K is composed of a large number of machines, which have different kinds of CPUs (dual-core architecture, AMD Opteron 64 bits, PowerPC G5 64 bits, Intel Itanium 2 64 bits, Intel Xeon 64 bits), of operating systems (Debian, Fedora Core 3 & 4, MacOs X, *etc.*), of supported JVMs (Sun 1.5 64 bits and 32 bits, and Apple 1.4.2), and of network connection (Gigabit Ethernet and Myrinet). All clusters will be connected to RENATER with a 10Gb/s link (or at least 1 Gb/s, when 10Gb/s is not available yet).

Figure 4.12 shows the global topology of the platform.

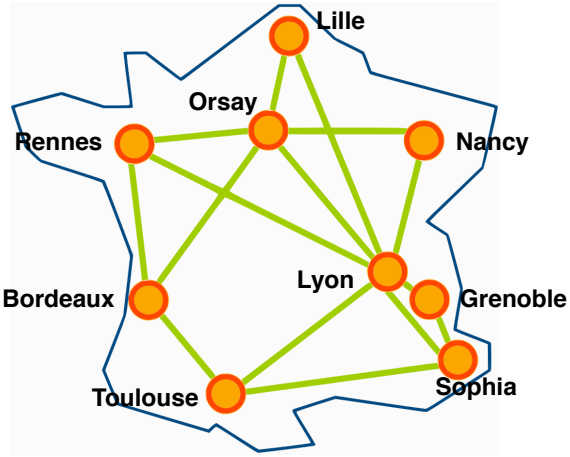

Figure 4.12: *Grid'5000 topology*

Table 4.1 describes architectures and cores (for us, a core is a CPU) distribution site by site.

Experiments Grid experiments run with the same implementation of flow-shop, as previous single cluster experiments. The selected instance of flow-shop is now 17 jobs / 17 machines, which is more difficult to solve compared to the previous one. The search tree strategy is FIFO and communications are used to dynamically share GUB. Results of experiments with G5K are summarized in Figure 4.13 and Table 4.2.

Table 4.2 shows the total number of tasks generated for each experiment. The flow-shop implementation has a ratio $\frac{Task_{272}/Task_{96}}{272/96} = 0.98$, the generation of tasks is linear with CPUs. But, between 272 and 621 CPUs, this ratio up to 1.24, the generation of tasks does not scale with CPUs, *i.e.* too much tasks are generated in regard of available CPUs.

Table 4.3 reports the distribution of CPUs by sites for each experimentation.

The dashed line in Figure 4.13 shows that the execution time strongly decrease until 272 CPUs, the speedup between 96 CPUs and 272 CPUs is 2.32. From 272 to 621 CPUs

Cores / Sites	Orsay	Grenoble	Lyon	Rennes	Sophia	Bordeaux	Lille	Nancy	Toulouse	Cores total
AMD Opteron 2214			16							**16**
AMD Opteron 2218					200					**200**
AMD Opteron 246	432		112	198	148			94		**984**
AMD Opteron 248				128		96	106		116	**446**
AMD Opteron 250	252		140							**392**
AMD Opteron 252							40			**40**
AMD Opteron 275					224					**224**
AMD Opteron 285							104			**104**
Intel Itanium 2		206								**206**
Intel Xeon 5110								480		**480**
Intel Xeon 5148 LV				264						**264**
Intel Xeon EM64T 3GHz						102				**102**
Intel Xeon IA32 2.4GHz		64								**64**
PowerPC				64						**64**
Sites total	**684**	**270**	**268**	**654**	**572**	**198**	**250**	**574**	**116**	**3586**

Table 4.1: *Grid'5000 sites / cluster descriptions*

Table 4.2: *Large-scale experiments results*

# CPUs	# Sites	Time	Tasks	% of tested permutations	Gathered time
96	1	107 m	1425	0.152%	165 h
100	1	104 m	1567	0.152%	167 h
200	1	60 m	2515	0.165%	181 h
255	1	51 m	3078	0.174%	195 h
272	2	46 m	2802	0.176%	171 h
296	2	45 m	3366	0.187%	184 h
300	2	44 m	3729	0.189%	196 h
422	3	45 m	4731	0.22%	227 h
452	3	41 m	5701	0.232%	255 h
488	3	41 m	5919	0.241%	255 h
492	4	40 m	5447	0.239%	251 h
521	4	37 m	6005	0.248%	258 h
621	5	35 m	6968	0.261%	267 h

the execution time is almost constant, the speedup between 272 and 621 CPUs is 1.31. Then, the global speedup, between 96 and 621 CPUs, is 3. Our *Grid'BnB* flow-shop scales well up to 272 (close to linear speedup). However, for more than 272 CPUs, the execution time decreases slowly. Nevertheless, the solid line shows the percentage of branches explored in the search tree, *i.e.* total number of tested permutations, this line increases with the number of CPUs. This line is indeed the total work done by the computation.

Efficiency Figure 4.14 shows the efficiency E, this value estimates how CPUs are utilized for the computation. Values of E are between 0 and 1, a single-processor computation and linear speedup have $E = 1$. Here, we consider the execution time (T) efficiency corrected with the work (W: total number of tested permutations) because *Grid'BnB* computes more work with increasing CPUs. Thus, the efficiency for n CPUs: $E_n = \frac{T_n/T_{96} * W_{96}/W_n}{96/n}$. The figure shows that between 96 and 300 CPUs, E is close to 1

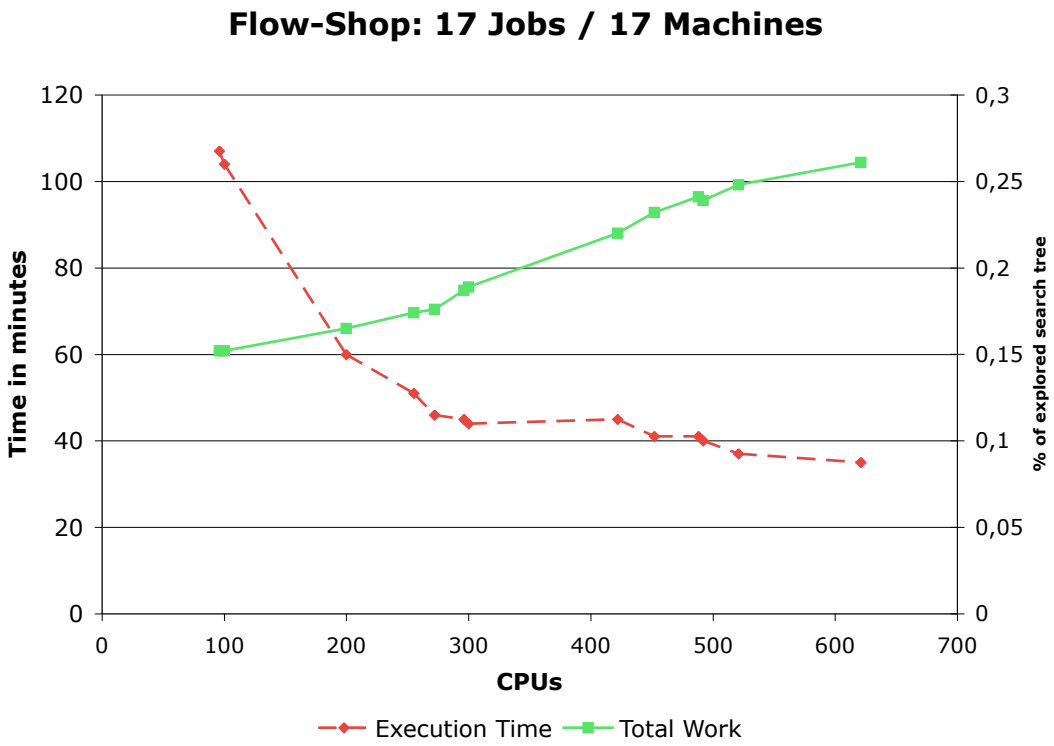

Figure 4.13: *Large-scale experiments results*

(0.9), which is very good. But, for 422 and more, E decreases to 0.8, it is still a good value. This decrease can be explain by the fact that for experiments with less than 422 CPUs are done on 1 or 2 Grid sites and for 422 and more 3 up to 5 sites nationally-distributed. In addition, Grid sites are heterogeneous in regards of CPUs power and inter-site network connections.

Experiments on single cluster and large-scale Grid show that it is better to use communications to dynamically share GUB, and that it is important for users to choose the adapted search tree strategy to their problems to solve. Large experiments also show that *Grid'BnB* can be used on Grid environments, we deploy flow-shop on a nationwide Grid of 5 clusters gathering a total of 621 CPUs.

4.6 Conclusion

In this chapter we described *Grid'BnB* a parallel B&B framework for Grids. *Grid'BnB* provides a framework to help users solve optimization problems. The framework hides all Grids, parallelism, and distribution related issues. The framework is based on hierarchical master-worker architecture with communications between processes. Communications are used to share the best global upper bound to explore less parts of the search tree and to decrease the execution time for solving optimization problems. Because Grids provide a large-scale parallel environment, we propose to organize workers

Table 4.3: *Large-scale experiments: site distribution*

# CPUs	Site distribution
96	sophia.helios
100	orsay
200	orsay
255	orsay
272	sophia.sol - 176 sophia.helios - 96
296	sophia.sol - 200 sophia.helios - 96
300	rennes.paraquad - 200 rennes.paravent - 100
422	sophia.sol - 200 sophia.azur - 126 sophia.helios - 96
452	sophia.helios - 184 sophia.azur - 142 lyon.sagittaire - 126
488	rennes.paraquad - 220 sophia.azur - 142 lyon.sagittaire - 126
492	rennes.paraquad - 200 lyon.sagittaire - 122 sophia.azur - 100 lille - 70
521	rennes.paraquad -202 lyon.sagittaire - 126 sophia.azur - 100 lille - 93
621	rennes.paraquad - 202 lyon.sagittaire - 126 sophia.sol - 100 sophia.azur - 100 lille - 93

in groups of communications. Groups reflects Grids topology. This feature aims to optimize inter-cluster communications and to update more rapidly the global upper bound on all processes. *Grid'BnB* also proposes different search tree algorithms to help users choose the most adapted for the problem to solve. Finally, the framework allows fault-tolerance for long-running executions.

Experiments have shown that *Grid'BnB* scales on a real nationwide Grid, such as Grid'5000. We are able to deploy a permutation optimization problem, flow-shop, on up to 621 CPUs distributed on five sites.

Finally, we believe that *Grid'BnB* can used for more than B&B without modification of the framework, it may be used to do divide-and-conquer or used as farm skeleton. More generally, *Grid'BnB* is framework for parallel programming that targets all em-

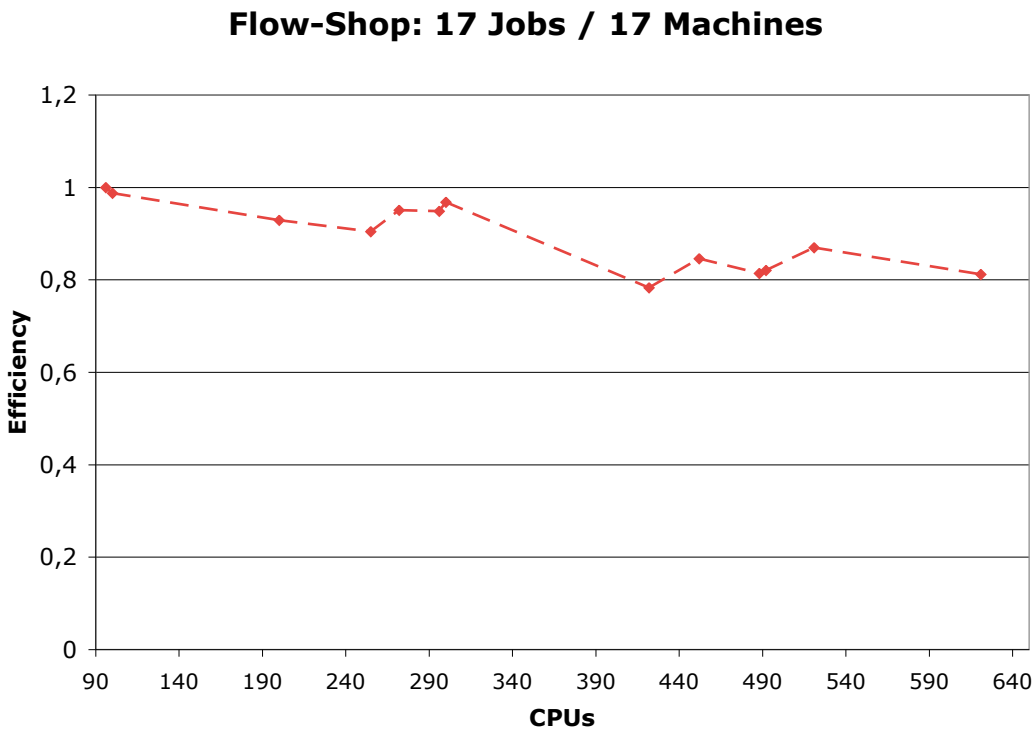

Figure 4.14: *Large-scale experiments: the efficiency*

barrassingly parallel problems.

In the next chapter, we report large-scale experiments of *Grid'BnB* with the P2P infrastructure.

Chapter 5

Mixing Clusters and Desktop Grid

This chapter relates our large-scale experiments with the *Grid'BnB* framework and the peer-to-peer infrastructure, with which we build a Grid composed of both desktop machines and clusters.

This chapter is organized as follows: first, we remember the context and point out our motivations and objectives of these large-scale experiments; second, we report results from experiments on a Grid that includes machines from the *INRIA Sophia P2P Desktop Grid* and from the French nation-wide Grid, named *Grid'5000*; third, we present some recent results from experiments that aims at dynamically acquiring numerous resources from *Grid'5000* with the peer-to-peer infrastructure.

5.1 Motivations and objectives

In previous chapters, we presented a peer-to-peer infrastructure and we experimented it as a desktop Grid. We also described *Grid'BnB*, our framework for branch-and-bound, and we experimented it on a nation-wide Grid.

This chapter aims to demonstrate that our framework and our infrastructure used together allow large-scale utilization of Grids, which are composed of desktop machines and clusters. In the next part of this chapter, we describe the modifications and the improvements of both in order to reach this goal. In Chapter 2, we identified several requirements that the infrastructure has to fulfill in order to address all Grid challenges. Some of those requirements have been not treated in Chapter 3, which are:

- *sharing clusters*, this point has been not previously considered by the infrastructure, it has to propose solutions to include clusters in the Grid; and
- *cross firewalls*, in order to mix desktop machines and clusters, which are usually in different local networks allowing identified and secure connections.

Furthermore, the framework was only considered with a static deployment on clusters. Now, the framework has to handle a dynamic deployment, *i.e* resource acquisition at runtime, and it has to dynamically keep communication groups organized because workers may be added during the problem solving. In summary, the framework has to:

- provide *dynamic resource acquisition*; and
- *dynamically organize workers* in group to optimize communication.

In addition to experiment the framework plus the infrastructure, we noticed during experiments that it is hard to obtain a large number of resources on Grid'5000, even with long-time before reservations.

The problem is that the platform is used by many users, we counted 567 user accounts. Figure 5.1 shows the global usage of the platform. Grid'5000 currently totalizes a total of 3586 cores (as reported by Table 4.1), which provides a total of $\approx 2.6e+6$ hours of computations a month. The figure shows a pick close to $\approx 1.4e+6$. However, the figure does not show the platform up-time. In other words, maintenance periods and failure down times are not considered in the figure.

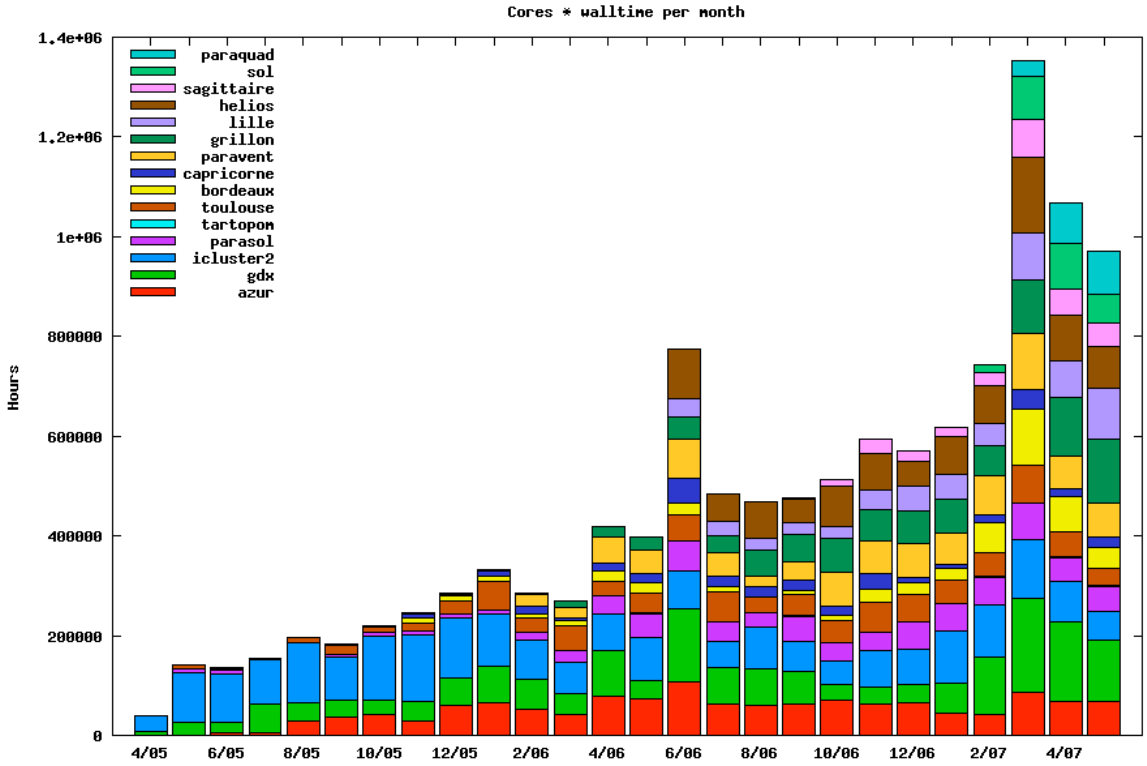

Figure 5.1: *Grid'5000 platform usage per month and per cluster*

This observation shows that it is very hard to obtain a large number of resources for experiments. In contrast, several nodes by clusters are free for sometime few minutes and summed they may totalize hundred of CPUs always available.

Hence, we believe that using this few-minutes-available resources may profit to CPU intensive application, such as the n-queens. We thus propose to use the peer-to-peer infrastructure to dynamically acquire available resources of Grid'5000 in order to run CPU intensive application.

In summary, the objectives of this chapter are double:

- showing that *Grid'BnB* can take benefit from the peer-to-peer infrastructure; and
- showing that using the peer-to-peer infrastructure over a Grid allows efficient dynamic resource acquisition for CPU intensive applications.

5.2 Design and architecture

This section describes the improvements of the infrastructure and the framework in order to support large-scale Grids. Thanks to the ProActive middleware on which we rely, many of these issues are easily solved.

5.2.1 Sharing clusters with the peer-to-peer infrastructure

In Chapter 3, we described the implementation of the infrastructure within the Grid middleware, named ProActive. ProActive provides a complete deployment framework for Grids, which is described in Chapter 6.1. This deployment framework allows to deploy Java Virtual Machines (JVMs) on clusters, via the job submission scheduler (such as PBS, LSF, *etc.*).

With ProActive, applications do not access directly to remote JVMs but manipulate nodes, which are execution environments for active objects (more details in the next chapter). Thus, as we previously explained, peers of the P2P infrastructure share these nodes.

In the infrastructure, peers use the deployment framework of ProActive to select which resources to share. In other words, when a peer shares its local machine, the peer uses the deployment mechanism to instantiate a node inside its local JVM. Furthermore, a peer is able to share and to manage resources that are not locals. This is simply done by using a deployment descriptor that allows to create JVMs on remote machines, such as a cluster. The choice of which resources are shared by peers is done by the administrator of the infrastructure, *i.e* the installer of the infrastructure. Figure 5.2 shows connections between peers and their shared nodes.

5.2.2 Crossing firewalls

The problem of crossing firewalls is indeed solved by the ProActive middleware. ProActive provides a simple message forwarding mechanism [MAT 06], a JVM where a ProActive runtime is running can be used to route communications between two others active objects. This feature combined to RMI/SSH as transport layer is useful to pass through firewalls or NAT.

Concretely, this mechanism is enabled at deployment time by the deployment framework. Figure 5.3 shows a peer sharing nodes on clusters located behind a firewall.

5.2.3 Managing dynamic group of workers with Grid'BnB

We already presented the implementation of *Grid'BnB*, and explained that groups of workers are indeed ProActive hierarchical groups. These groups are group of groups, a ProActive may see as a collection of active objects. Groups allow broadcast communication with all group members.

Because ProActive groups are implemented like collection, it is relatively easy to add new elements (actives objects or groups) in such groups. We also shown, in Chapter 3.3, that a ProActive application can use the event/listener mechanism to dynamically acquire resources from the P2P infrastructure. *Grid'BnB* is implemented within the active object programming model. In other words, our branch-and-bound framework can be

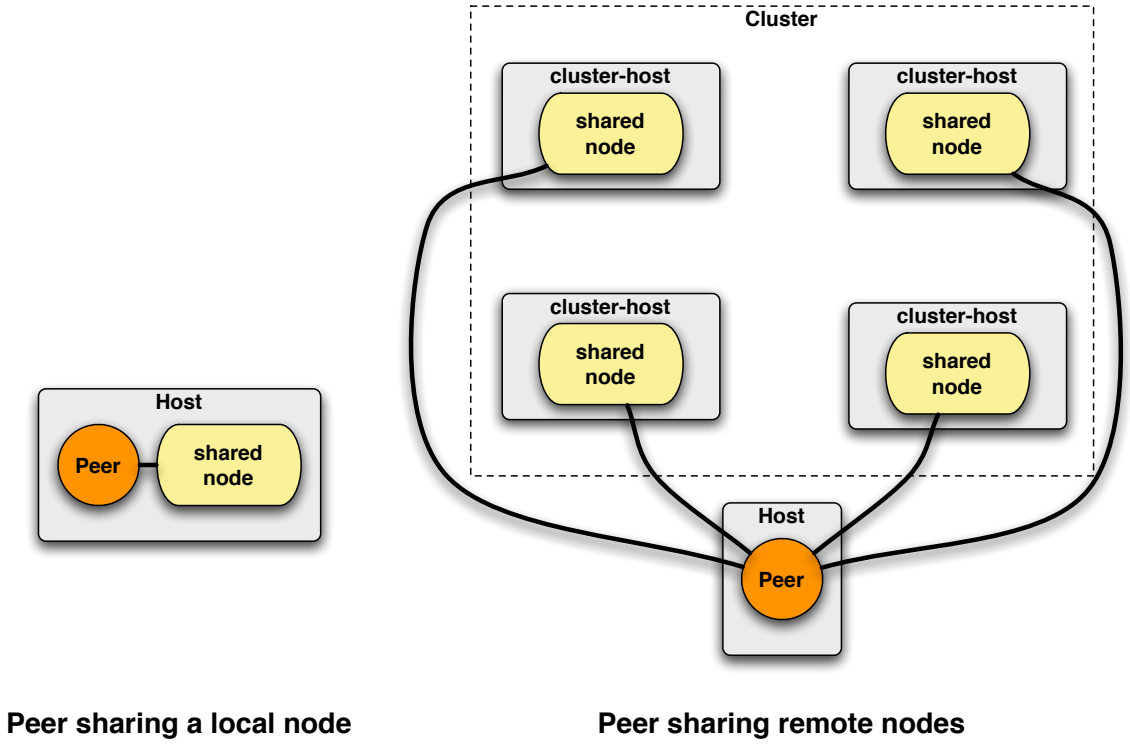

Figure 5.2: *Peers can share local resource or remote resources*

considered as a common ProActive application. Hence, acquiring and adding resources is solved by few implementation improvements. The issue is how to organize workers in groups.

In this chapter, we target to build a Grid composed of clusters and machines from a desktop Grid. We argued that *Grid'BnB* rely on a feature provided by the deployment framework of ProActive to organize workers. This mechanism consists in tagging deployed nodes with a tag to identify if nodes have been deployed by the same chain of deployment processes. This mechanism is fully described in the next chapter.

Hence, nodes on the same cluster have the same tag. Thus it is relatively easy to create a group with nodes of same tag. The problem is with nodes from the desktop Grid because each node is deployed by a different process and has an unique tag. In order to solve this issue, we decide to gather all nodes, which have a single tag, in the same group. In other words, the main group contains all workers that are running on a node, which has a unique tag, and other group are created if at least two nodes have the same tag.

5.3 Large-scale experiments: mixing desktops and clusters

This section presents the results of large-scale experiments on a Grid composed of desktops machines and clusters. First, we experiment this Grid with the n-queen application, which has no communication between workers. Second, we experiment the flow-

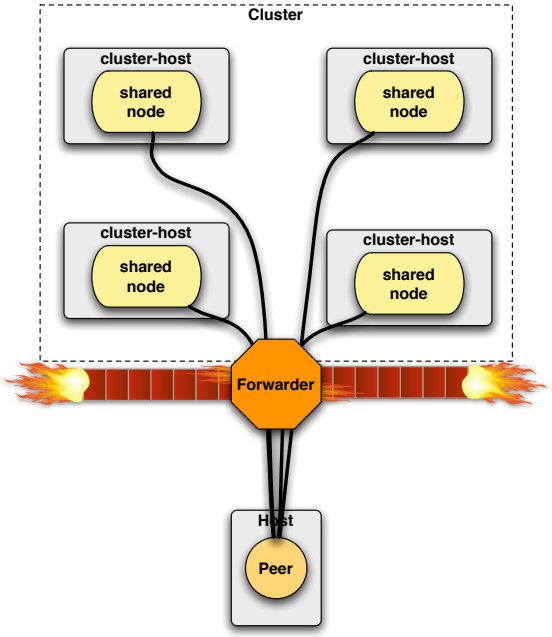

Figure 5.3: *Peers that shares resources behind a firewall*

shop application, which has communication between workers.

5.3.1 Environment of experiments

Figure 5.4 shows the Grid used for our experiments. This Grid is a mix of the *INRIA Sophia P2P Desktop Grid* (InriaP2P), described in Chapter 3.4.2, and clusters from Grid'5000 (G5K), described in Chapter 4.5.2. The left of the figure shows the INRIA Sophia P2P Desktop Grid wherein INRIA-2424 peers are used as registries (all registries use themselves as registries); and at fixed moments the rest of INRIA-ALL machines join the P2P infrastructure by contacting those registries.

In addition, the right of the figure shows the G5K platform, clusters of G5K are connected to the P2P infrastructure by a few INRIA-2424 peers, and each of these peers handles a G5K site. These peers do not share their local JVMs but share JVMs deployed on clusters. G5K and INRIA networks are both closed network with only few SSH access points, and G5K is a NAT, thus communications between INRIA and G5K are tunneled via SSH.

5.3.2 Experiments with n-queens

We took the same application as previously, the n-queens (see Chapter 3.4.1), and ran it on a Grid that is a mix of machines from *INRIA Sophia P2P Desktop Grid* and from clusters of G5K.

Experiments run with $n = 22$. Figure 5.5 shows all results. Counting up all the machines, we reached 1007 CPUs. The execution time decreases with a large number of CPUs on a heterogeneous Grid, mix of desktop and clusters.

We show that an embarrassingly parallel application, such as n-queens, benefits of the large number of CPUs provided by our P2P infrastructure, even if this Grid is highly

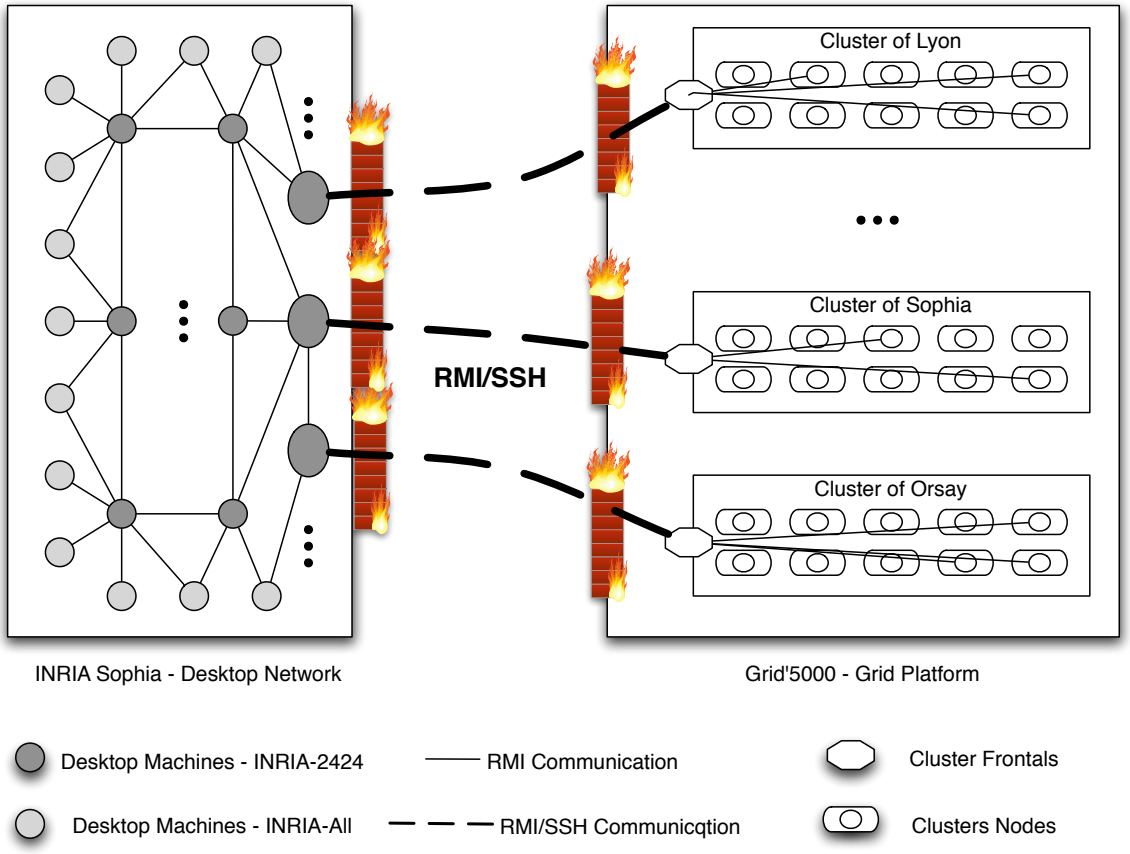

Figure 5.4: *Mixing Desktop and Clusters: environment of experiment structure*

heterogeneous. For that kind of application, mixing desktop and cluster machines is beneficial.

5.3.3 Experiments with flow-shop

To illustrate that our P2P infrastructure can be used to deploy communicating applications, we consider an application for solving flow-shop problems, see Chapter 4.4.1.

The implementation of flow-shop with *Grid'BnB* has a low communication/computation ratio, even if at the beginning of the computation the application has to acquire nodes and to deploy workers on all CPUs. Workers are intensively broadcasting new better solutions. The intensive phase takes 20 minutes for the run of one hour on 628 CPUs. With this run we were able to measure the communication size on 116 CPUs of the G5K Sophia cluster. We measured 143 MB of network traffic inside the cluster for the first 20 minutes. The bandwidth used is about 120 KB/s. After, there are only sporadic communications until the best solution is found.

Figure 5.6 shows all results of flow-shop computations with an instance of *17 jobs / 17 machines*. An analysis of Figure 5.6 shows that computation time decreases with the number of used CPUs. However, the increase in execution time between 201 and 220 comes from a communication bottleneck between workers of INRIA-2424, which are

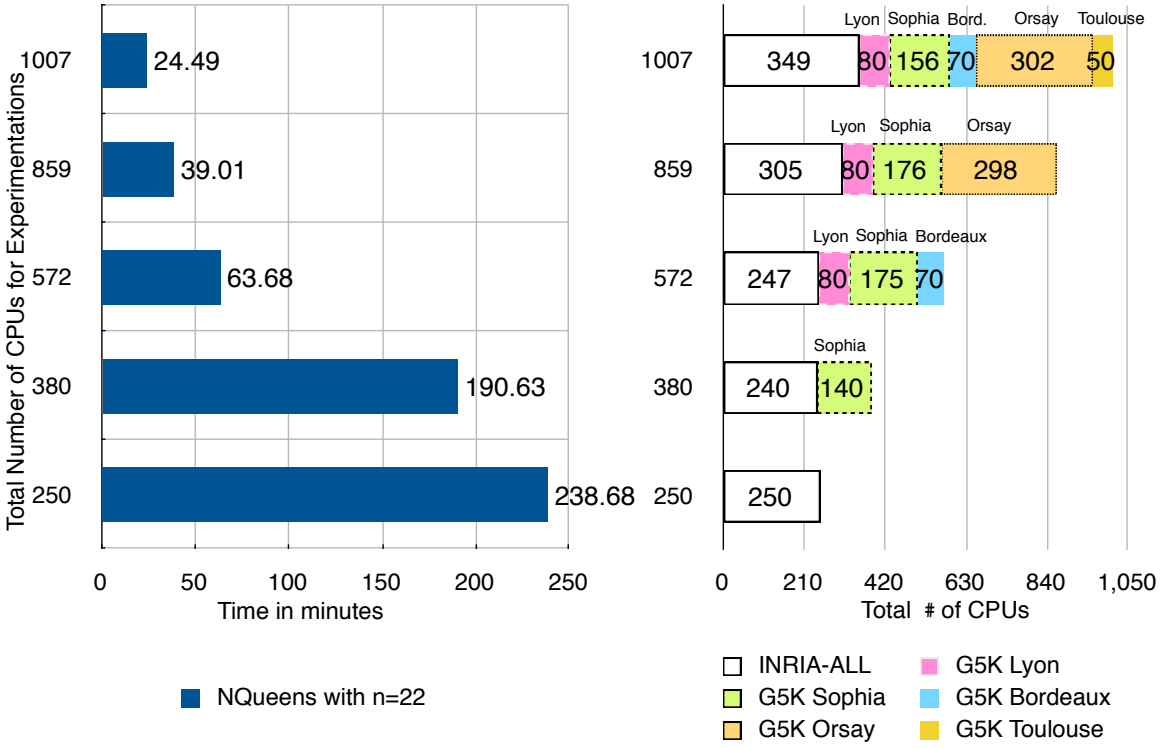

Figure 5.5: *Mixing Clusters and Desktops: n-queens with $n = 22$ benchmark results*

desktop machines, and workers of G5K Sophia, which are clusters. Communications between *INRIA P2P Desktop Grid* and G5K are tunneled in *SSH*. This bottleneck can also be observed on the run with 321 CPUs on three sites. The lower impact of bottleneck with 313 CPUs can be explain by the distribution of the tasks and by the fact that there is only one cluster. Then the last experimentation with 628 CPUs has an execution time close to the bench with 346 CPUs, we explain that by a long phase of acquiring resources and the creation of workers, 11 minutes, opposed to only 6 minutes for 346 CPUs.

In addition, all these benchmarks were realized during working days, so usual users have a higher priority to execute their processes.

To conclude, we have been able to deploy a communicating application using a P2P infrastructure on different sites which provided 628 CPUs. Those CPUs were composed of heterogeneous architectures, and came from desktop machines and clusters.

5.4 Peer-to-Peer Grid experiments

This section reports results of experimenting the P2P infrastructure over Grid'5000. We aim to demonstrate that a CPU intensive application, such as n-queens, can take benefit of using Grid resources, which are available for few minutes.

We reported that a Grid platform as Grid'5000 is over-loaded in terms of users and available nodes. However, we observed that many nodes are free, *i.e* not reserved or

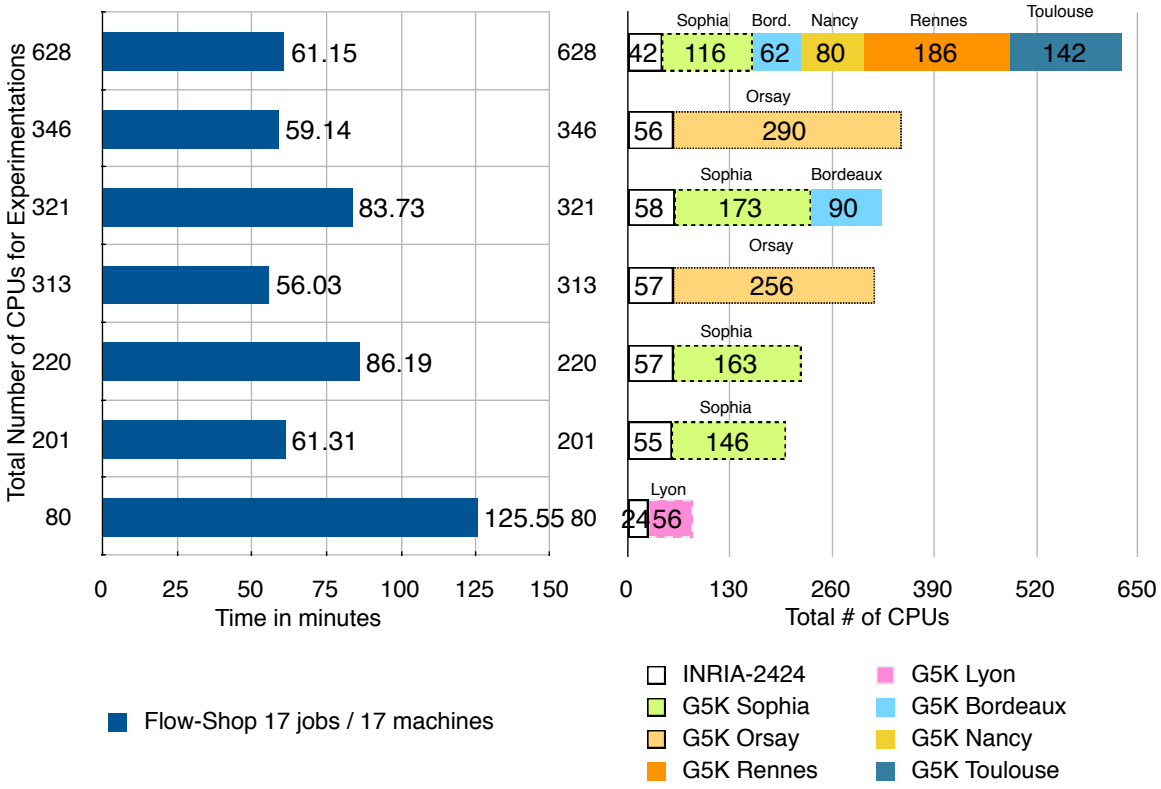

Figure 5.6: *Mixing Clusters and Desktops: Flow-Shop benchmark results*

used, for few minutes. Thus, we believe that a dynamic infrastructure, such as our peer-to-peer infrastructure, can gather these resources and can provide them to applications. We also think an application of type master-worker with tasks that require few minutes to execute, can be deployed on this kind of highly dynamic Grids.

5.4.1 Experiment design

In order to deploy our infrastructure, we need to have some Grid nodes reserved for all the time of the experiments. Hence, one or two nodes by sites are reserved before each experiments and for at least 1 hour (the expected experiment time). These nodes are used to host our infrastructure registries, and like with the *INRIA Sophia P2P Desktop Grid* registries are themselves registry for registries.

Finally, a script is started for Grid sites. This script continually submits reservations to obtain free nodes. For all obtained nodes, a peer is started and added to the infrastructure.

5.4.2 Experiments results

Table 5.1 reports all results of our 48 experiments. These experiments have used from 2 to 11 clusters of Grid'5000, at the same time. Also, the n-queens used from 312 to 1384 CPUs at the same time. The average of CPUs for all experiments is between 301 and

917.

Because, all benchmarks are done each time with a different infrastructure, in terms of number of clusters used, different clusters, and the time of CPU availabilities, it does not make sense to compare all experiments on the same graph. However, Figure 5.7 compares experiments, the top graph shows the experiment time as function of the maximum CPUs working concurrently during the benchmark, the bottom graph shows the experiment time as function of the average of CPUs working concurrently during the benchmark.

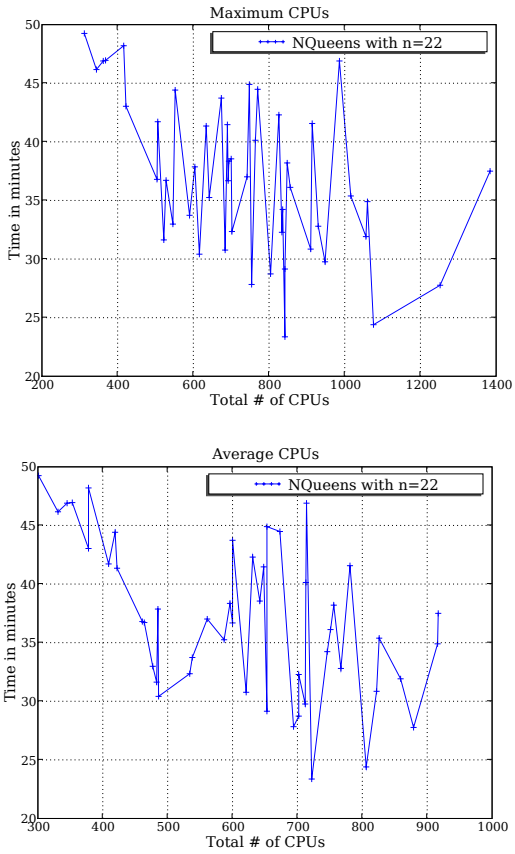

Figure 5.7: *Summary of all experiments*

Figure 5.8 compares for each benchmark the maximum number of CPUs to the average. The gap between the top of CPUs working concurrently and the average is for the majority of benchmarks more than 100 CPUs. In contrast, for experiments with less CPUs, about 350, this gap is almost null.

This bar chart helps us to determine what is a "perfect" benchmark. A perfect benchmark for us is a benchmark with the same (or almost the same) number of maximum CPUs and average CPUs. Figure 5.9 reports all results of one of these perfect experiments. On the top graph, the diamond plot represents the number of tasks computed by minutes, and the triangle plot represents the number of CPUs working by minutes. The left pie chart shows the CPUs distribution by sites. The right pie chart then shows the number of tasks computed by sites.

This experiment shows that the number of CPUs stayed almost constant for the

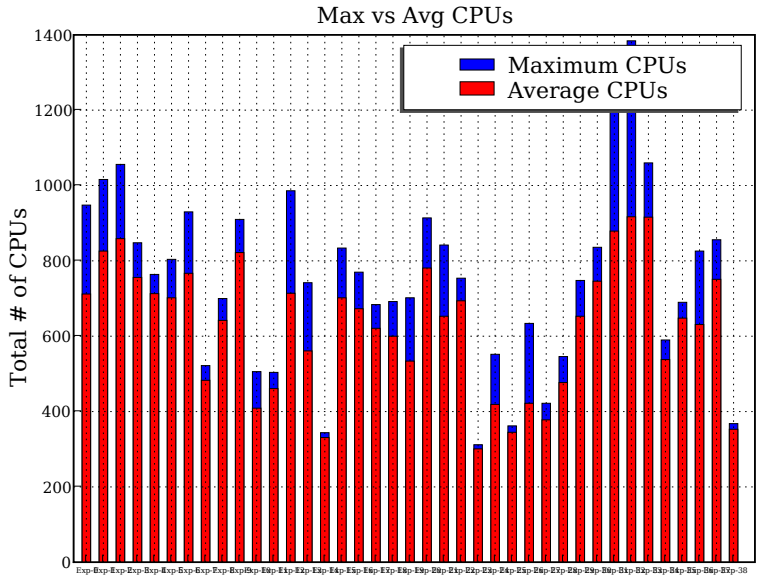

Figure 5.8: *For each bench, comparing maximum and average number of CPUs*

whole benchmark, and the peak of tasks is due to the fact that the first n-queens tasks are smaller to solve.

In contrast, some benchmarks are considered as "bad". For instance, Figure 5.10 reports a bad experiment. Between the 30 and 35 minute of the experiments, we notice a drop of the number of CPUs and computed tasks, this interval measures almost 4 minutes. After an investigation, we determined that drop is due to the LDAP servers of the Grid'5000 platform, which do not support the load of requests from our experiments. This problem has been reported to the platform staff and will be fixed in the next update of the platform, which is scheduled for mid-September.

Figure 5.11 and Figure 5.12 report large-scale experiments involving more than 1000 CPUs.

5.5 Conclusion

In this chapter, we demonstrated with experiments that *Grid'BnB* on top of the peer-to-peer infrastructure allows large-scale deployments on Grids, which are composed of desktop machines and clusters.

We also showed that the peer-to-peer infrastructure can be used as a Grid infrastructure for gathering available resource from a real Grid platform. Hence, a CPU intensive application, such as n-queens, can dynamically acquire them for computing, even if resources are available for few minutes.

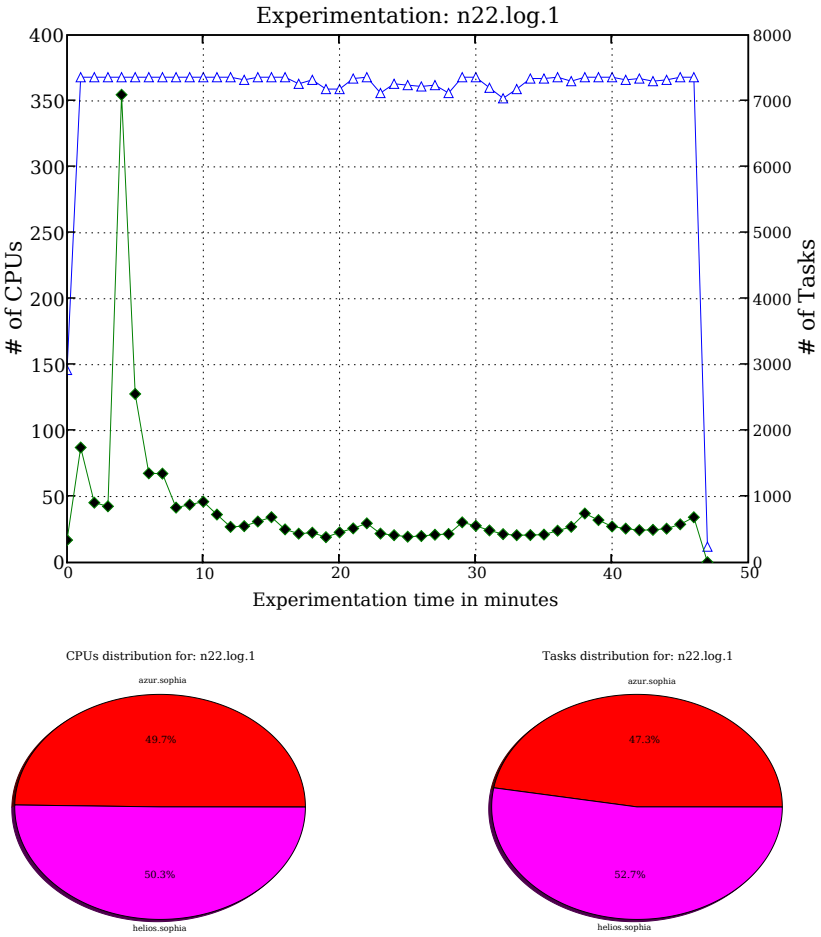

Figure 5.9: *"Perfect" experimentation n22.log.1*

Table 5.1: *The P2P infrastructure over Grid'5000 experiments with 22-queens*

Name	# of Sites	Max # CPUs	Avg # CPUs	Total CPU Time	Time
n22.log.1	2	368	353	278h 50m 47s	0h 46m 57s
n22.log.3	6	506	409	287h 49m 33s	0h 41m 43s
n22.log.4	7	552	419	305h 30m 19s	0h 44m 25s
n22.log.5	4	842	653	281h 59m 19s	0h 29m 9s
n22.log.6	6	1056	859	277h 29m 39s	0h 31m 55s
n22.log.7	7	634	422	294h 18m 44s	0h 41m 21s
n22.log.8	8	910	822	252h 8m 44s	0h 30m 51s
n22.log.9	10	986	714	271h 26m 2s	0h 46m 54s
n22.log.10	9	702	534	260h 58m 53s	0h 32m 21s
n22.log.11	9	764	713	257h 55m 54s	0h 40m 8s
n22.log.12	5	362	345	247h 17m 19s	0h 46m 54s
n22.log.13	11	848	756	258h 39m 36s	0h 38m 12s
n22.log.14	10	856	751	263h 14m 35s	0h 36m 7s
n22.log.15	8	700	642	250h 1m 2s	0h 38m 33s
n22.log.16	5	312	301	241h 26m 57s	0h 49m 15s
n22.log.18	8	836	746	259h 35m 22s	0h 34m 14s
n22.log.19	8	826	631	280h 6m 13s	0h 42m 18s
n22.log.20	8	422	378	249h 17m 49s	0h 43m 2s
n22.log.21	7	522	483	242h 18m 12s	0h 31m 38s
n22.log.22	7	590	538	248h 59m 55s	0h 33m 44s
n22.log.23	6	690	648	258h 43m 51s	0h 41m 28s
n22.log.24	7	748	653	255h 12m 47s	0h 44m 54s
n22.log.25	10	754	694	253h 59m 22s	0h 27m 50s
n22.log.26	9	770	673	257h 12m 7s	0h 44m 29s
n22.log.27	7	504	461	251h 27m 4s	0h 36m 48s
n22.log.28	10	546	477	251h 17m 16s	0h 32m 59s
n22.log.29	11	1060	916	253h 34m 31s	0h 34m 54s
n22.log.30	6	344	331	246h 37m 1s	0h 46m 10s
n22.log.31	7	1016	826	244h 59m 32s	0h 35m 23s
n22.log.32	7	684	621	246h 50m 20s	0h 30m 46s
n22.log.33	10	834	702	250h 53m 42s	0h 32m 17s
n22.log.34	9	930	767	250h 34m 53s	0h 32m 48s
n22.log.35	8	804	702	245h 47m 55s	0h 28m 44s
n22.log.38	11	1252	879	263h 30m 20s	0h 27m 46s
n22.log.39	7	692	600	268h 56m 59s	0h 36m 41s
n22.log.40	9	1384	917	257h 5m 43s	0h 37m 30s
n22.log.41	9	948	712	252h 5m 8s	0h 29m 46s
n22.log.42	8	914	781	247h 27m 52s	0h 41m 34s
n22.log.45	11	742	561	282h 8m 49s	0h 37m 1s
n22.log.46	5	416	378	285h 23m 59s	0h 48m 12s
n22.log.48	11	674	600	254h 22m 4s	0h 43m 44s
n22.log.49	8	528	464	262h 5m 54s	0h 36m 43s
n22.log.50	9	604	485	283h 55m 29s	0h 37m 52s
n22.log.51	9	642	587	252h 15m 43s	0h 35m 15s
n22.log.52	9	694	596	254h 20m 13s	0h 38m 21s
n22.log.53	10	1076	806	235h 46m 12s	0h 24m 24s
n22.log.54	9	616	486	252h 21m 6s	0h 30m 25s
n22.log.56	8	842	722	243h 58m 4s	0h 23m 22s

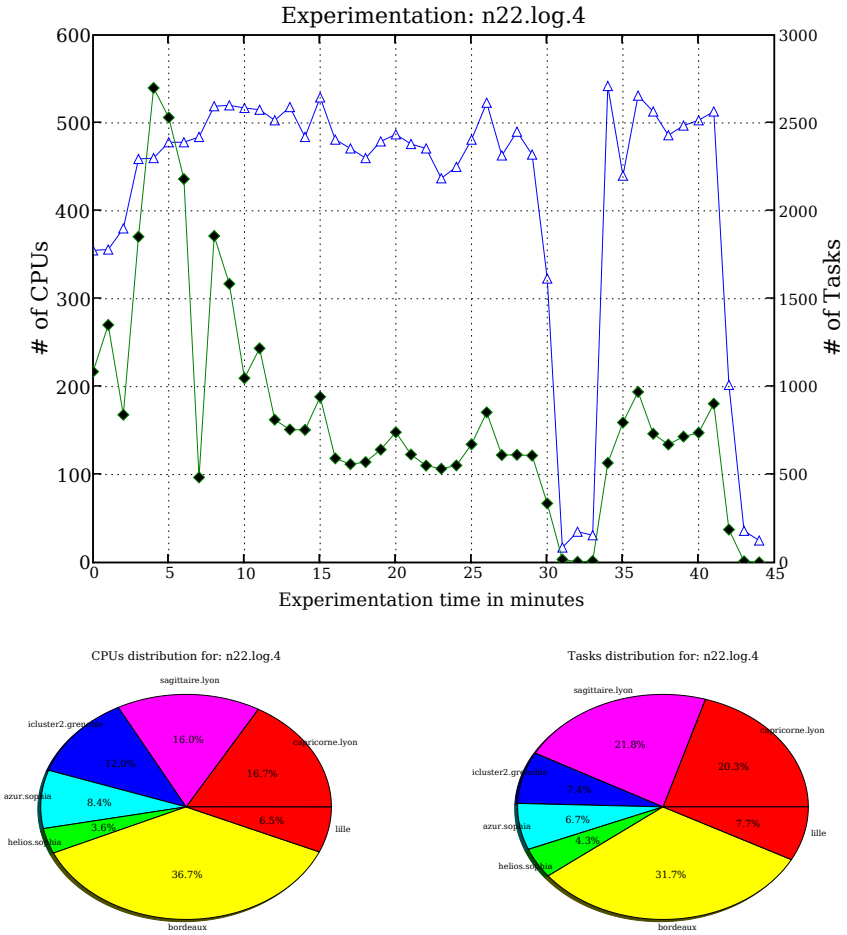

Figure 5.10: *"Bad" experimentation n22.log.4*

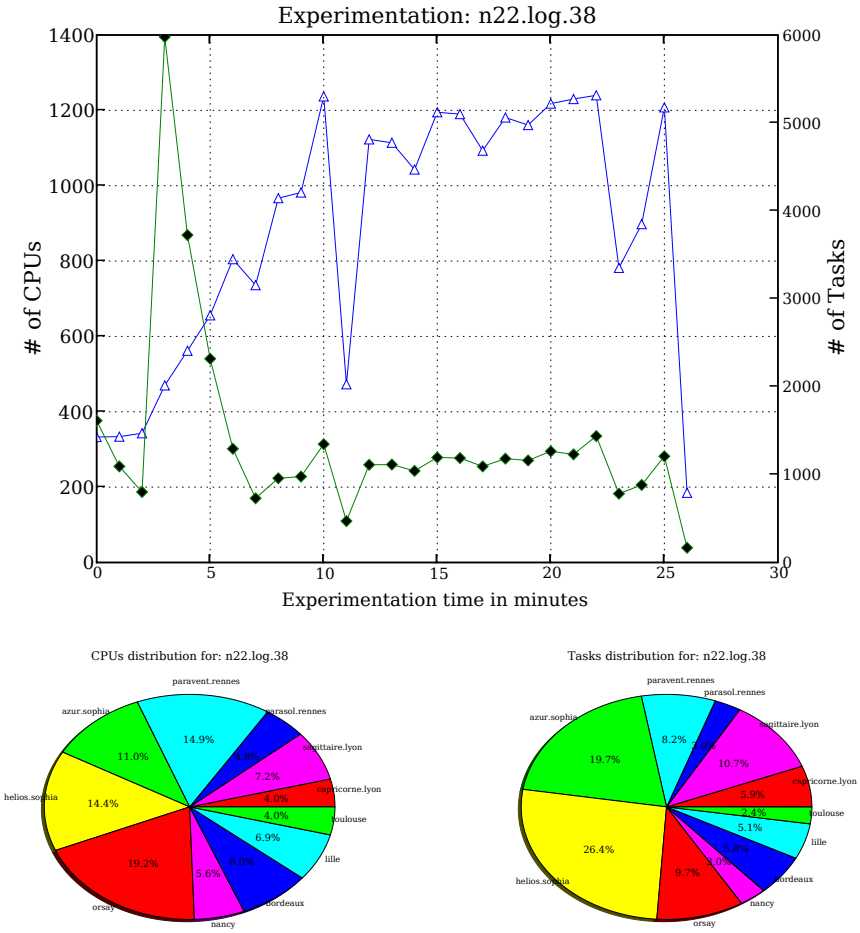

Figure 5.11: *Large-scale experiments: 1252 CPUs concurrently working from 11 clusters*

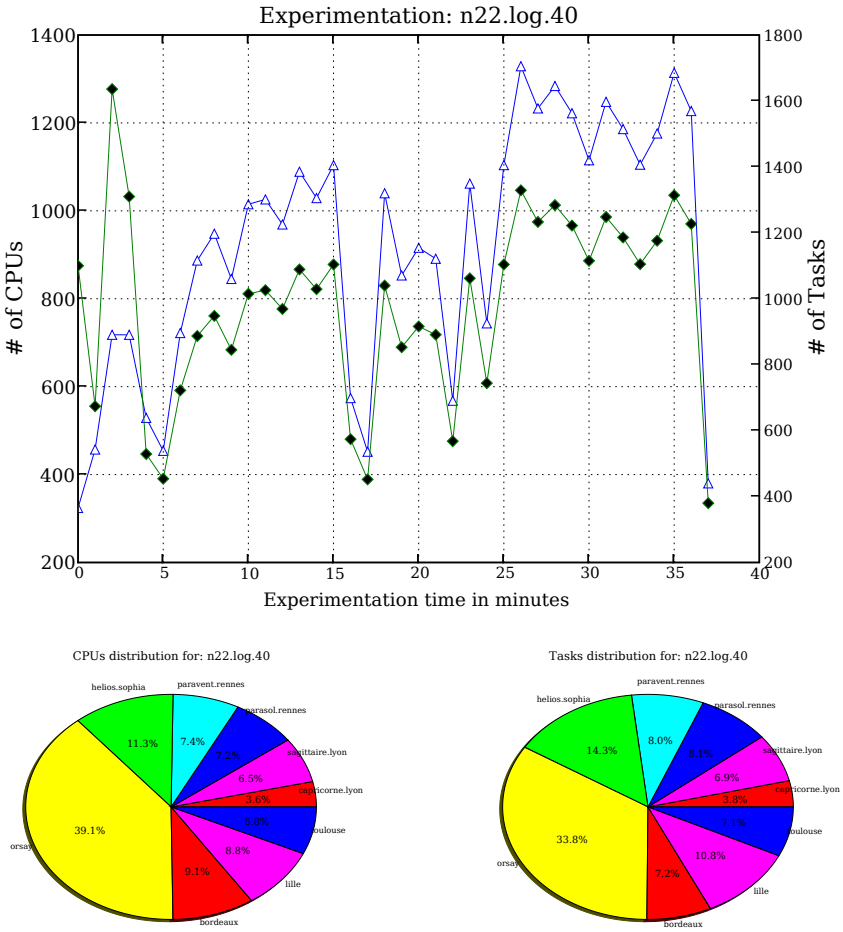

Figure 5.12: *Large-scale experiments: 1384 CPUs concurrently working from 9 clusters*

Chapter 6

Advanced Features and Deployment

The first objective of this thesis is to provide a dynamic infrastructure for Grid computing. This infrastructure is based on an unstructured peer-to-peer architecture that allows to mix desktop machines and clusters.

The second objective is a parallel branch-and-bound framework for Grids, which permits to solve combinatorial optimization problems. This framework relies on the master-worker paradigm with communications between workers to increase the computation speedup.

The link between our framework and our infrastructure is the *deployment*. This thesis work has for context the Java ProActive Grid middleware. Thereby in this chapter we introduce the deployment framework of ProActive. Next, we present our first improvement that allows localization of nodes on Grids. This mechanism is used by the branch-and-bound framework to organize workers in groups for reducing communication cost.

After that, we describe our second improvement of the deployment in ProActive with the activation of non-functional services, named *technical services*. These services are fault-tolerance or load-balancing for instance.

Then, we propose the *virtual nodes descriptors*, which is a kind of application descriptor. This application descriptor allows developers to specify deployment constraints to their applications. Constraints are for instance technical services to deploy, minimum number of nodes, or even computing architectures.

Finally, we describe in details a technical service for load-balancing. This load-balancing mechanism depends on the P2P infrastructure presented in this thesis.

6.1 Context: ProActive deployment framework

We described the ProActive Grid middleware in Section 2.4. Especially, we presented the programming model and some other features but we have not detailed the deployment framework.

In this section, we fully describe the deployment framework of ProActive.

6.1.1 Deployment model

The deployment of Grid applications is commonly done manually through the use of remote shells for launching the various virtual machines or daemons on remote computers and clusters. The commoditization of resources through Grids and the increasing complexity of applications are making the task of deploying central and harder to perform.

ProActive succeeded in completely avoiding scripting for configuration, getting computing resources, etc. ProActive provides, as a key approach to the deployment problem, an abstraction from the source code in order to gain in flexibility [BAU 02]. A first principle is to *fully* eliminate from the source code the following elements:

- machine names,
- creation protocols,
- registry and lookup protocols.

The goal being to deploy any application anywhere without changing the source code. The deployment sites are called *Nodes*, and correspond for ProActive to JVMs which contain active objects.

To answer these requirements, the deployment framework in ProActive relies on XML descriptors. These descriptors use a specific notion, *Virtual Nodes* (VNs):

- a VN is identified by a name (a simple string),
- a VN is used in a program source,
- a VN, after activation, is mapped to one or to a set of *actual ProActive Nodes*, following the mapping defined in an XML descriptor file.

A VN 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 VNs and nodes: the function created by the deployment, the mapping. This mapping is specified in the deployment descriptor. By definition, the following operations can be configured in the deployment descriptor:

- the mapping of VNs to nodes and to JVMs,
- the way to create or to acquire JVMs,
- the way to register or to lookup VNs.

Figure 6.1 summarizes the deployment framework provided by the ProActive middleware. Deployment descriptor can be separated in two parts: mapping and infrastructure. The VN, which is the deployment abstraction for applications, is mapped to nodes in the deployment descriptor and nodes are mapped to physical resources, *i.e.* to the infrastructure.

6.1.2 Retrieval of resources

In the context of the ProActive middleware, nodes designate physical resources from a physical infrastructure. They can be created or acquired. The deployment framework is responsible for providing the nodes mapped to the virtual nodes used by the application. Nodes may be created using remote connection and creation protocols. Nodes may also be acquired through lookup protocols, which notably enable access to the P2P infrastructure as explained below.

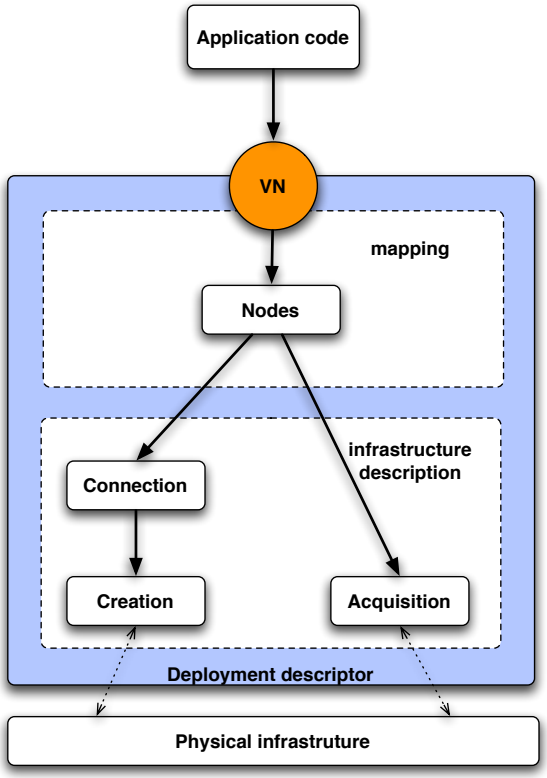

Figure 6.1: *Descriptor-based deployment*

6.1.2.1 Creation-based deployment

Machine names, connection and creation protocols are strictly separated from application code, and ProActive deployment descriptors provide the ability to create remote nodes (remote JVMs). For instance, deployment descriptors can use various protocols:

- local
- ssh, gsissh, rsh, rlogin
- lsf, pbs, sun grid engine, oar, prun
- globus (GT2, GT3, and GT4), unicore, glite, arc (nordugrid)

Deployment descriptors allow to combine these protocols in order to create seamlessly remote JVMs, *e.g.* log on a remote cluster frontend with ssh, and then use pbs to book cluster nodes to create JVMs on them. All processes are defined in the *infrastructure* part of the descriptor.

In addition, the JVM creation is handled by a special process, *localJVM*, which starts a JVM. It is possible to specify the classpath, the Java install path, and all JVM arguments. In addition, it is in this process that the deployer specifies which transport layer the ProActive node uses. For the moment, ProActive supports as transport layer: RMI, HTTP, RMIssh, and Ibis [NIE 05].

6.1.2.2 Acquisition-based deployment

The deployment framework also provides the opportunity to acquire nodes, instead of creating them. In the context of this thesis, we presented the P2P infrastructure (see Chapter 3), which allows the deployment framework to acquire previously created nodes. The infrastructure also provides the capability of dynamic deployment, *i.e.* acquiring more nodes at runtime.

6.1.3 Large-scale usages

It would be difficult to qualify a middleware as a Grid middleware without demonstrating its Grid capabilities: deployment on a large number of hosts from various organizations, on heterogeneous environments and using different communication and connection protocols.

In the context of this thesis, we shown in Section 3.4 that ProActive with the P2P infrastructure can be used for long running experiments. Likewise in Section 5.3, we illustrated the capabilities of the P2P infrastructure for large-scale deployments. In addition to these large-scale experiments, we outline the Grid PlugTests events, which demonstrate the abilities of ProActive for large-scale usages.

The Grid PlugTests events held at ETSI in 2004, 2005, and 2006 demonstrated the capacity of the ProActive library to create virtual organizations and to deploy applications on various clusters from various locations. The first Grid PlugTests event [ETS 05b] gathered competing teams, which had to solve the n-queens problems using the Grid built by coordinating universities and laboratories of 20 different sites in 12 different countries, resulting in 800 processors and a computing power of 100 Gigaflops (SciMark 2.0 benchmark for Java). In the second Grid PlugTests [ETS 05a] event, a second computational problem was added: the permutation flow-shop. Heterogeneous resources from 40 sites in 13 countries were federated, resulting in a 450 Gigaflops grid, with a total of 2700 processors. In the third Grid PlugTests [ETS 06] event, a 1700 Gigaflops Grid from 22 sites in 8 countries was provided to competing teams in order to deploy n-queens and flow-shop on a total Grid of 4130 processors.

6.2 Grid node localization

ProActive succeeds in completely avoiding scripts for configuration, getting computing resources, and launching the computation. ProActive provides, as a key approach to the deployment problem, an abstraction of the physical infrastructure from the source code to gain in flexibility (see Section 6.1).

In the context of this thesis, we proposed to organize workers in groups for optimizing the communications in our B&B framework (see Chapter 4). The selection criterion for group acceptance for a worker is its physical localization on a cluster. Therefore, the node localization on the Grid is important for an efficient implementation of our *Grid'BnB* framework. Hence we propose a mechanism to localize nodes on Grids. This mechanism can also be used by all kinds of applications to optimize the communication between active objects distributed on Grids and/or to do data localization.

The ProActive deployment framework provides a high-level abstraction of the underlying physical infrastructure. Once deployed, the application cannot find out the topology of the physical infrastructure on which it is deployed. In other words, the

deployment is abstracted into VNs. Within ProActive, virtual nodes represent the application topology, for instance in our B&B framework the logical topology should be two VNs, one for master/sub-masters and a second for workers.

Because applications usually need to organize active objects in groups for optimizing communications between clusters, the abstraction of the physical infrastructure laid by the ProActive deployment framework prevents easy resource localization. For instance, programmers have to compare node addresses to determine if two nodes are deployed on the same cluster. But, two nodes may have the same sub-net address on different clusters, with network of NATs. Hence, programmers need to have a strong knowledge in network programming for using metric functions, such as latency, to determine if nodes are close. Consequently, organizing workers in group by clusters and optimizing communication between clusters is a very difficult task. Because of the strong abstraction of the physical infrastructure, we introduce a new mechanism in the ProActive deployment framework to identify nodes, which are deployed on the same cluster or even on the same machine.

The creation of a node is the result of a deployment graph (a directed acyclic graph: DAG) combined with connection protocols. This deployment graph is specified within the deployment descriptor. The deployment node tagging mechanism proposed here aims to tag nodes in regard of the deployment graph on which they are mapped in the deployment descriptor. This tag will allow the application to organize groups in regard to the deployment process that created the node.

The virtual node is indeed the root node of the deployment graph. Then the deployment process explores the DAG in breadth-first order. A node of this DAG is a connection protocol to instantiate and a leaf node is the JVM creation.

Figure 6.2 shows the process of tagging nodes. The tag is built by concatenating identifiers added at each level of the building of the deployment graph. At the beginning of the deployment, a new tag is instantiated for each virtual node. For leaf nodes of the DAG, which are JVM creations, no identifier is added. Therefore, all nodes deployed with the same path in the DAG have the same tag.

With this mechanism, all deployed nodes are tagged with an identifier at deployment time. The purpose of this tag is that if several nodes have the same tag value they have been deployed by the same deployment process. As a result, nodes with the same tag have a high probability to be located in the same cluster and/or the same local network.

The tag is an abstraction of the physical infrastructure; it provides more information about how nodes have been deployed. It is now possible to know at the application level that the same deployment graph has deployed two nodes.

The deployment tag can be used for instance by applications to optimize communication between nodes or to do data localization. More especially *Grid'BnB* uses the deployment tag to dynamically organize worker communications between clusters. In this context, Figure 6.3 shows the deployment result of a single virtual node on three clusters: *clusterA*, *clusterB*, and *clusterC*. The deployment has returned nine nodes: four nodes on *clusterA*, two on *clusterB*, and three on *clusterC*. The node tag mechanism has tagged nodes on *clusterA* with tag *0-0*, tag *0-1* on *clusterC*, and tag *0-2* on *clusterB*. Tags are finally used to organize workers in groups of communication to optimize communication between clusters.

Follow, the code of this example for deploying the virtual node Workers described by the deployment descriptor DeploymentDescriptorFile.xml:

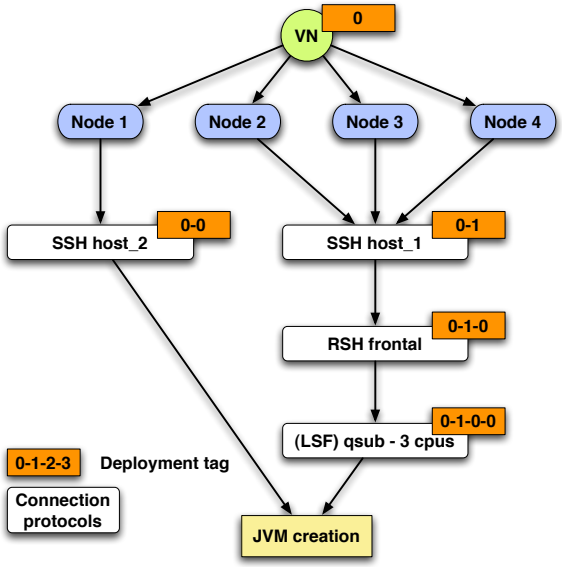

Figure 6.2: *Deployment tag mechanism*

```
// Activating deployment of virtual node Workers
ProActiveDescriptor pad = ProActive.getProactiveDescriptor("DeploymentDescriptorFile.xml");
pad.activateMapping("Workers");
VirtualNode vnWorkers = pad.getVirtualNode("Workers");

// Getting ProActive deployed nodes
Node[] workerNodes = vnWorkers.getNodes();
```

With deployed nodes workerNodes, it is now possible to know if nodes are deployed by the same list of processes. For instance, an application can group active objects with respect to the deployment tag:

```
// For each node: get the deployment tag, instantiate a new worker, and add the worker in the
    good group
for (Node node: workerNodes) {
        // Getting deployment tag for the node
        DeploymentTag tag = node.getNodeInformation().getDeploymentTag();

        // Creating a new active object, Worker, on the remote node
        Worker worker = (Worker) ProActive.newActive(Worker.class.getName(), null, node);

        // Adding the new worker in its group with respect to its deployment tag
        groupsOfWorkers.add(worker, tag);
}
```

The deployment framework featured by ProActive succeeded at deploying application on large-scale Grids, as shown by Section 6.1.3. But, the abstraction of the physical infrastructure is harder to allow easy localization of nodes. Therefore, we extend the deployment framework with the presented node tagging mechanism, which allows ap-

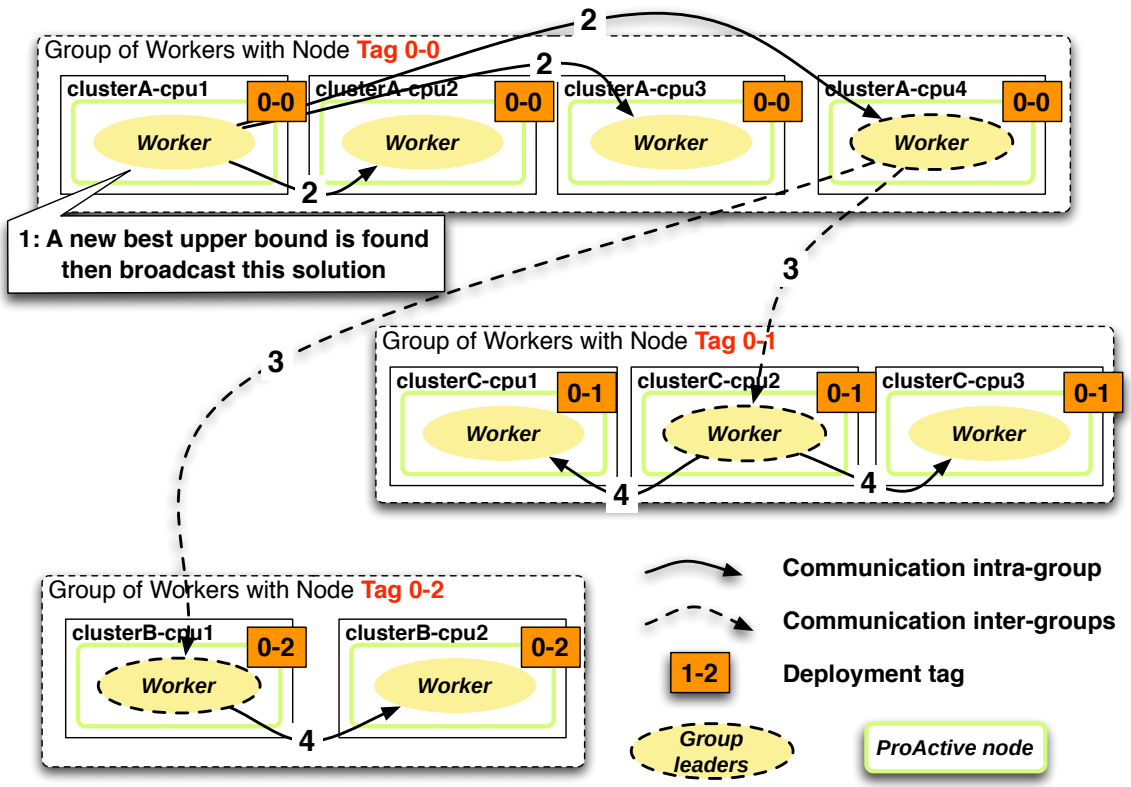

Figure 6.3: *Application of node tagging to Grid'BnB for organizing workers in groups*

plications to determine if two nodes has been deployed by the same deployment graph.

In addition, we use this mechanism in the implementation of *Grid'BnB* (see Section 4.3) for optimizing communications between clusters.

However, the node tagging cannot be optimal with the P2P infrastructure, presented in this thesis (see Chapter 3). Nodes are firstly deployed and tagged by peers; and then when the application acquires nodes from the infrastructure, its deployment process overwrite the previous tag with a new one, which is identical for all returned nodes. In Section 7.1.2.1 we present the concept of *node family* to solve this issue.

6.3 Technical services for Grids

The last decade has seen a clear identification of the so called *Non-Functional* aspects for building flexible and adaptable software. In the framework of middlewares, *e.g.* business frameworks such as the Enterprise JavaBeans (EJB) [MIC 01], architects have been making a strong point at separating the application operations, the *functional aspects*, from services that are rather orthogonal to it: transaction, persistence, security, distribution, *etc.*

The frameworks, such as EJB containers (JBoss, JOnAs, *etc.*), can further be configured for enabling and configuring such non-functional aspects. Hence, the application logic is subject to various settings and configuration, depending of the context. Overall, it opens the way to effective component codes usable in various contexts, with the

crucial feature of parameterization: choosing at deployment time various *Technical Services* to be added to the application code. In the framework of Grids, current platforms are falling short to provide such flexibility. One cannot really add and configure fault-tolerance, security, load-balancing, *etc.* without intensive modification of the source code. Moreover, there are no coupling with the deployment infrastructures.

In an attempt to solve this shortcoming of current Grid middlewares, we propose to improve ProActive deployment framework with a mechanism for defining such Technical Services dynamically, based on the application needs, potentially taking into account the underlying characteristics of the infrastructure.

6.3.1 Technical services principles

The concept of non-functional requirements, *i.e.* technical services, was first introduced in the field of component models. Such models allow a clear separation between the functional code written by the developer and the non-functional services provided by the framework. For J. Kienzle and R. Guerraoui [KIE 02] a technical service must be developed by an expert in the field, such as an expert in load-balancing for implementing a load-balancing service, because a field expert can provide a good quality-of-service for a large-scale of applications.

In the EJB [MIC 01] framework, technical services are specified by Sun Microsystems; in the Corba Component Model (CCM) [020 99], they are provided by CORBA. These services are at the component container level, *i.e.* they are parts of the framework. For all these frameworks, users specify and configure technical services at the deployment time. Consequently, users have choice between few technical services imposed by the models, thus a fault-tolerance expert cannot propose her own solution. Then, users are limited in their choices, they cannot choose between different versions or implementations of the same service.

In the field of Grids, the Open Grid Services Architecture (OGSA) [FOS 02] defines a mechanism for creating, managing, and discovering Grid services, which are network-enabled entities that provide some capability through the exchange of messages. The OGSA specifies a uniform service semantic that allows users to build their Grid applications by assembling some services from enterprises, service providers, and themselves.

Some parts of an application may require specific non-functional services, such as security, load-balancing, or fault-tolerance. These constraints can only be expressed by the deployer of the application because she is the only one that can configure them for the physical infrastructure.

Because the deployment infrastructure is abstracted into virtual nodes, we propose to express these non-functional requirements as contracts [FRØ 98] in deployment descriptors (Figure 6.4). This allows a clear separation between the conceptual architecture using virtual nodes and the physical infrastructure where nodes exist or are created; it maintains a clear separation of the roles: the developer implements the application without taking into account the non-functional requirements; and the deployer, considering the available physical infrastructure, enforces the requirements when writing the mapping of virtual nodes in the deployment descriptor. Then, the expert implements and provides non-functional services as technical services. Moreover, we propose to leverage the definition of deployment non-functional services with the introduction of dynamically applicable technical service.

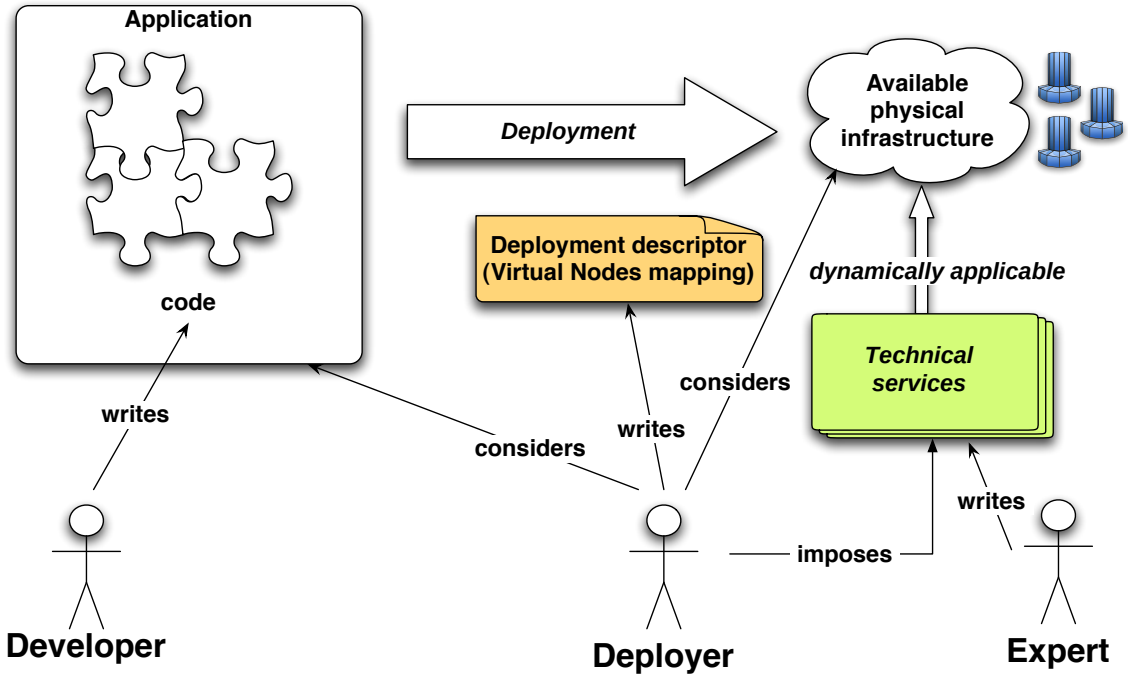

Figure 6.4: *Deployment roles and artifacts*

Unlike component frameworks, our technical services mechanism is an extensible model that allows programmer experts to propose their own implementation of non-functional services. Like EJB and CCM, users specify and configure the technical service at the deployment time.

Unlike our approach, the OGSA does not limit services to be only non-functional; for example a Grid service can be a cluster for data storage. On the other hand, non-functional services are parts of the architecture model and users cannot configure security or fault-tolerance for their own applications.

6.3.2 Technical services framework

A technical service is a non-functional requirement that may be dynamically fulfilled at runtime by adapting the configuration of selected resources.

From the expert programmer point of view, a technical service is a class that implements the TechnicalService interface. This class defines how to configure a node. From the deployer point of view, a technical service is a set of "key-value" tuples, each of them configuring a given aspect of the application environment.

For instance, for configuring fault-tolerance, a FaultToleranceService class is provided; it defines how the configuration is applied from a node to all the active objects hosted by this node. The deployer of the application can then configure in the deployment descriptor the fault-tolerance using the technical service XML interface.

A technical service is defined as a stand-alone block in the deployment descriptor. It is attached to a virtual node (it belongs to the virtual node container tag); the configu-

ration defined by the technical service is applied to all the nodes mapped to this virtual node. A technical service is defined as follows:

```
<technicalServiceDefinition id = "myService" class="services.Service1">
   <arg name="name1" value="value1"/>
   <arg name="name2" value="value2"/>
</technicalServiceDefinition>
```

The class attribute defines the implementation of the service, a class which must implement the TechnicalService interface:

```
public interface TechnicalService {
   public void init(Map argValues);
   public void apply(Node node);
}
```

The configuration parameters of the service are specified by arg tags in the deployment descriptor. Those parameters are passed to the init method as a map associating the name of a parameter as a key and its value. The apply method takes as parameter the node on which the service must be applied. This method is called after the creation or acquisition of a node, and before the node is used by the application.

A technical service is attached to a virtual node as following:

```
<virtualNodesDefinition>
 <virtualNode name="virtualNode1" serviceRefid="myService"/>
</virtualNodesDefinition>
```

Figure 6.5 summarizes the deployment framework with the added part for non-functional aspects.

ProActive provides a mechanism for load-balancing active objects (see Section 6.5), the drawback of this work was that the activation of the mechanism is at the code level. This service is now ported as a technical service, in order to start and configure load-balancing at deployment time. The implementation of the load-balancing technical service is:

```
public class LoadBalancingTS implements TechnicalService {

        public void init(Map argValues) {
                String metricFactoryName = (String) argValues.get("MetricFactory");
                MetricFactory mf = (MetricFactory) Class.forName(metricFactoryName).
                    newInstance();
                LoadBalancing.activate(mf);
        }

        public void apply(Node node) {
                LoadBalancing.addNode(node);
        }
}
```

The implementation does the same thing of what the developer does for activating load-balancing at source code level.

Two or several technical services could be combined if they touch separate aspects.

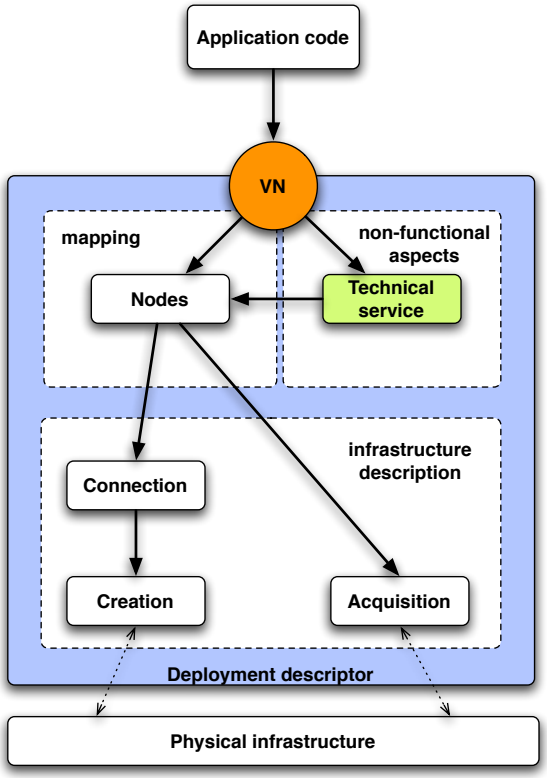

Figure 6.5: *Deployment descriptor model with the technical service part*

Indeed, two different technical services, which are conceptually orthogonal, could be incompatible *at source code level*.

In practice, there is an incompatibility in the implementation of fault-tolerance and load-balancing services, developed by two different programmers. That is why a virtual node can be configured by only *one* technical service. However, combining two technical services can be done at source code level, by providing a class extending TechnicalService that defines the correct merging of two concurrent technical services.

6.3.3 A complete example: fault-tolerant flow-shop on peer-to-peer

This section illustrates the concept of dynamically deploying and configuring technical services: it presents a use case involving the deployment of an application with some fault-tolerance requirements on the P2P infrastructure (presented in Chapter 3); it demonstrates how technical services help resolving deployment in the most suitable way. Beforehand, an explanation of the fault-tolerance mechanism and configuration in ProActive is provided, which is essential to the understanding of this use case.

6.3.3.1 Fault-tolerance in ProActive

As the use of desktop Grids goes mainstream, the need for adapted fault-tolerance mechanisms increases. Indeed, the probability of failure is dramatically high for such systems: a large number of resources imply a high probability of failure of one of those resources. Moreover, public Internet resources are by nature unreliable.

Rollback-recovery [ELN 96] is one solution to achieve fault-tolerance: the state of the application is regularly saved and stored on a stable storage. If a failure occurs, a previously recorded state is used to recover the application. Two main approaches can be distinguished: the *checkpoint-based* [MAN 99] approach, relying on recording the state of the processes, and the *log-based* [ALV 98] approach, relying on logging and replaying inter-process messages.

Fault-tolerance in ProActive is achieved by rollback-recovery [DEL 07]; two different mechanisms are available. The first one is a Communication-Induced Checkpointing protocol (CIC): each active object has to checkpoint at least every *TTC* (Time To Checkpoint) seconds. Those checkpoints are synchronized using the application messages to create a *consistent* global state of the application [CHA 85]. If a failure occurs, *every active object*, even the non faulty one, must restart from its latest checkpoint. The second mechanism is a Pessimistic Message Logging protocol (PML): the difference with the CIC approach is that there is no need for global synchronization, because all the messages delivered to an active object are logged on a stable storage. Each checkpoint is independent: if a failure occurs, only the faulty process has to recover from its latest checkpoint.

Basically, those two approaches can be compared on two metrics: the failure-free overhead, *i.e.* the additional execution time induced by the fault-tolerance mechanism without failure, and the recovery time, *i.e.* the additional execution time induced by a failure during the execution. The failure-free overhead induced by the CIC protocol is usually low [BAU 05], as the synchronization between active objects relies only on the messages sent by the application. Of course, this overhead depends on the TTC value, set by the programmer; the TTC value depends mainly on the assessed frequency of failures. A small TTC value leads to very frequent global state creation and thus to a small rollback in the execution in case of failure. But a small TTC value leads also to a higher failure free overhead. The counterpart is that the recovery time could be high since all the application must restart after the failure of one or more active object.

As for CIC protocol, the TTC value impacts on the global failure-free overhead, but the overhead is more linked to the communication rate of the application. Regarding the CIC protocol, the PML protocol induces a higher overhead on failure-free execution. But the recovery time is lower as a single failure does not involve all the system: only the faulty has to recover.

Fault-tolerance Configuration: Choosing the best protocol depends on the characteristics of the application, and of the underlying hardware that are known at deployment time; the fault-tolerance mechanism is designed such that making a ProActive application fault-tolerant is automatic and transparent to the developer; there is no need to consider fault-tolerance concerns in the source code of the application. The fault-tolerance settings are actually contained in the nodes: an active object deployed on a node is configured by the settings contained in this node.

Fault-tolerance is a technical service as defined in Section 6.3. The designer can specify in the virtual nodes descriptor the needed reliability of the different parts of the application, and the deployer can choose the adapted mechanism to obtain this reliability by configuring the technical service in the deployment descriptor. The deployer can then select the best mechanism and configuration:

- the protocol to be used (CIC or PML), or no protocol if software fault-tolerance is not needed on the used hardware,
- the Time To Checkpoint value (TTC),
- the URLs of the servers.

6.3.3.2 Technical service code example

To illustrate the mechanism of technical services, we consider a master-worker application for solving flow-shop problems (see Section 4.4.1). An implementation of this problem with *Grid'BnB* has already been presented in Chapter 4. In this previous version the fault-tolerance is handled by the B&B framework. This example is a simple implementation of the flow-shop without the previous presented framework, it aims to show the application of technical services.

General architecture: The solution tree of the problem is divided by a master in a set of sub-tasks, these sub-tasks are allocated to a number of sub-managers, which can also be at the top of a hierarchy of sub-managers. Sub-managers manage sub-task allocation to the workers and also perform communications between them to synchronize the best current solution. Sub-managers handle dynamic acquisition of new workers as well as worker failures by reallocating failed tasks. As a consequence, there is no need for applying an automatic fault-tolerance mechanism (then to pay an execution-time overhead) on the workers. On the contrary, the manager and the sub-managers must be protected against failures by the middleware since there is no failure-handling at application level for them.

Deployment descriptor: A complete example of a deployment descriptor based on the P2P infrastructure:

```
<ProActiveDescriptor>
  <componentDefinition>
    <virtualNodesDefinition>
      <virtualNode name="masters" serviceRefid="ft-master" />
      <virtualNode name="workers" />
    </virtualNodesDefinition>
  </componentDefinition>
  <deployment>
    <mapping>
      <map virtualNode="masters">
        <vmName value="localJVM" />
        <vmName value="p2plookup_sub-masters" />
      </map>
      <map virtualNode="workers">
        <vmName value="p2plookup_workers" />
      </map>
    </mapping>
  </deployment>
  <infrastructure>
    <processes>
      <processDefinition id="localJVM">
        <jvmProcess class="org.objectweb.proactive.core.process.JVMNodeProcess" />
      </processDefinition>
    </processes>
```

```
    <aquisition>
      <P2PService id="p2plookup_sub-masters" nodesAsked="10">
        <peer>rmi://registry1:3000</peer>
      </P2PService>
      <P2PService id="p2plookup_workers" nodesAsked="1000">
        <peer>rmi://registry1:3000</peer>
      </P2PService>
    </aquisition>
  </infrastructure>
  <technicalServiceDefinitions>
    <service id="ft-master" class="services.FaultTolerance">
      <arg name="protocol" value="pml" />
      <arg name="server" value="rmi://host/FTServer1" />
      <arg name="TTC" value="60" />
    </service>
  </technicalServiceDefinitions>
</ProActiveDescriptor>
```

This descriptor defines two virtual nodes: one for hosting the masters and one for hosting the workers. Only the master virtual node is configured by a technical service defining the most adapted fault-tolerance configuration regarding the underlying hardware; here, the protocol used is PML, set with a short TTC value as we are in P2P with volatile nodes.

Fault-tolerance technical service code: The functional code of the fault-tolerance is in fact implemented in the ProActive core code for logging messages. Thus the technical service just sets some global properties for starting the logging of messages by the fault-tolerance. The full implementation of our fault-tolerance technical service is:

```
public class FaultToleranceTS implements TechnicalService {
    private String SERVER;
    private String TTC;
    private String PROTOCOL;

    public FaultToleranceTS() {
    }

    public void init(Map argValues) {
        this.SERVER = (String) argValues.get("proactive.ft.server.global");
        this.TTC = (String) argValues.get("proactive.ft.ttc");
        this.PROTOCOL = (String) argValues.get("proactive.ft.protocol");
    }

    public void apply(Node node) {
        node.setProperty("proactive.ft.server.global", this.SERVER);
        node.setProperty("proactive.ft.ttc", this.TTC);
        node.ssetProperty("proactive.ft.protocol", this.PROTOCOL);
    }
}
```

Experimentation: The proposed technical service mechanism has been implemented and experiments have been done with the sub-set of the desktop infrastructure, which

is described in Section 3.4. This infrastructure provides a pool of desktops, which have heterogeneous configuration and hardware.

Figure 6.6 shows the Flow-Shop application deployed with the technical service for fault-tolerance on the P2P infrastructure. The instance of the flow-shop problem is 15 jobs / 20 machines. The fault-tolerance protocol used is PML with a TTC of 60 seconds. We observe that the computation time decreases with the number of CPUs.

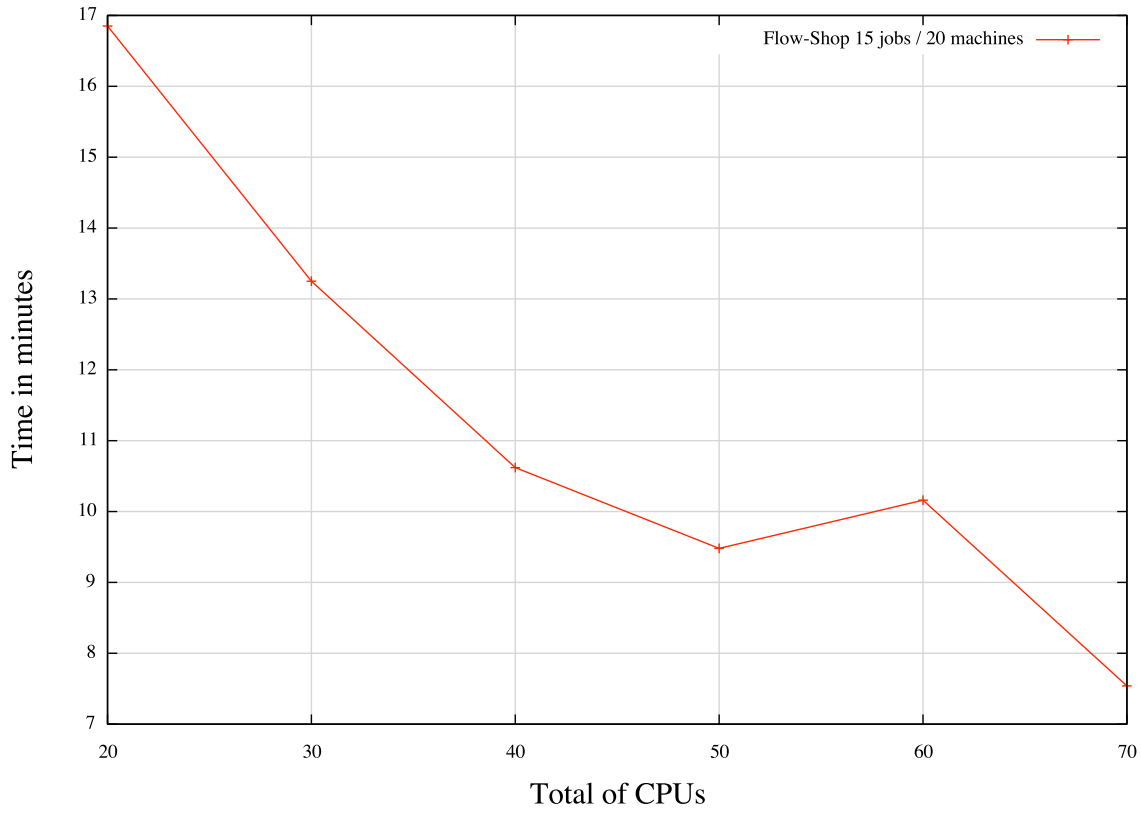

Figure 6.6: *Experiments with flow-shop and the technical fault-tolerance service*

The increase from 50 to 60 CPUs is due to the fact that some tasks of the problem run on more slower machines than for 50 or 70 CPUs. Benchmarks run on a desktop Grid and use a different set of machines at each run. It is hard to control expected machines with the peer-to-peer aspects of our benchmarks.

For more benchmarks on the fault-tolerance itself, we invite you to look up this article [BAU 05].

6.3.4 Technical services future work

In this thesis, we propose a way to attach Technical Services to Virtual Nodes, mapping non-functional aspects to the containers, dynamically at deployment and execution time. More investigations are needed to look into the fit of the architecture with respect to the complexity of Grid platforms, and to the large number of technical services to be composed and deployed. In the short term, we are planning to explore the combination of two technical services: fault-tolerance and load-balancing.

Technical services allow deployers to apply and configure applications non-functional requirements. However, deployers have to be aware of which technical services applications need. To solve this lack for connections between developers and deployers, we propose in the next section a mechanism to fix this issue.

6.4 Virtual node descriptor

In the context of this thesis, we have presented (in the previous section) a mechanism to dynamically apply non-functional requirements to applications. This mechanism, which is technical services, extends the ProActive deployment framework. Technical services are applied and configured within the deployment descriptor, which is separated from the application. In other words, the developers write the application codes and the deployers write the deployment descriptors.

Thus, the deployment framework, including technical services mechanism, lacks for a tool that helps developers to specify their applications requirements for deployers. In this section, we propose a mechanism to complete the deployment framework with Virtual Nodes descriptor [CAR 06c] to solve this issue. This mechanism is also designed to integrate the paradigm of component for Grids [MOR 06].

Problematic: Software engineering defines clear separations between the roles of actors during the development and usage of a software component. In particular, a designer is expected to specify not only the functional services offered or required by a component, but also the conditions on the environment that are required for a correct deployment – so that the deployer can fulfill her task. The designer must therefore have a way to specify environmental requirements that must be respected by targeted deployment resources. The deployer, from her knowledge of the target infrastructure, must be able to specify optimized and adequate adaptations or creations of the resources. Programmers of applicative components, who should mostly concentrate on the business logic, may be provided with abstractions for the distribution of the components; deployment requirements can be specified on these abstractions.

In the context of Grid computing, current platforms are falling short to express these deployment requirements, especially dynamically fulfillable ones, *i.e.* requirements that can be fulfilled in several manners at deployment time. Adding and configuring fault-tolerance, security, load-balancing, *etc.* usually implies intensive modification of the source code.

Thus we propose a mechanism for Grid computing frameworks, for specifying environmental requirements that may be optimized by deployers.

These requirements are specified by designers by parameterizing deployment abstractions, and are fulfilled dynamically by the deployers. Practically, we propose a mechanism for specifying deployment constraints and dynamically applying them on deployment infrastructures. Deployment constraints are specified on Virtual Nodes, which are deployment abstractions for the ProActive Grid middleware.

Context component based programming: In addition to the standard object oriented programming paradigm, ProActive also proposes a component-based programming paradigm, by providing an implementation [BAU 03] of the Fractal component model [BRU 02] geared at Grid computing.

Fractal is a modular and extensible component model, which enforces separation of concerns, and provides a hierarchical structure of component systems. Because it is a

simple though extensible model with clear specifications, Fractal has been chosen as a base for the Grid Component Model, currently under specification in the CoreGrid European NoE.

In the implementation of the Fractal model with ProActive, components are implemented as active objects, therefore all underlying features of the library are applicable to components.

The deployment of components is addressed in two ways: with a dedicated and standardized Architecture Description Language (ADL) [TUT] for describing Fractal components, and with the ProActive deployment framework described in the Section 6.1.

6.4.1 Constrained deployment

6.4.1.1 Rationale

Some components may require specific non-functional services, such as availability, reliability, security, real-time, persistence, or fault-tolerance. Some constraints may also express deployment requirements, for example the expected number of resources (minimum, maximum, exact), or a timeout for retrieving these resources. These constraints can only be expressed by the designers of the components or by the application developers.

Because the deployment infrastructure is abstracted into virtual nodes, we propose to express these non-functional requirements as contracts [FRØ 98] in a dedicated descriptor of virtual nodes (Fig. 6.7). This allows a clear separation between the conceptual architecture using virtual nodes and the physical infrastructure where nodes exist or are created; it enforces designer-defined constraints; it maintains a clear separation of the roles: the designer/developer specifies deployment constraints, and the deployer, considering the available physical infrastructure, enforces the requirements when writing the mapping of virtual nodes in the deployment descriptor. Moreover, we propose to leverage the definition of deployment constraints with the introduction of dynamically fulfillable constraints.

6.4.1.2 Constraints

Expressing deployment constraints at the level of virtual nodes enforces a strict separation of non-functional requirements from the code. By using a dedicated descriptor of virtual nodes, the constraints may easily be modified or adapted by the designer to express new requirements. Virtual nodes descriptors also allow a strict separation between the non-functional requirements and the description of the application. Because virtual nodes are abstractions that may be used in component ADLs or in application codes, constrained deployment through virtual nodes descriptors is applicable for both component-based and object-based applications.

We distinguish *statically* fulfilled requirements, which may not usually change in selected nodes (for instance the operating system), from *dynamically* fulfilled requirements, which may be applied at runtime by configuring the nodes (for instance the need for fault-tolerance or load-balancing). There are many ways to specify static constraints, as proposed in OLAN [BEL 97], in Corba Software Descriptor files [Obj 02], or more specifically in the context of Grid computing, as recently proposed by the Global Grid Forum in the Job Submission Description Language (JSDL) [ANJ 05]. The JSDL could be extended for defining constraints that may be dynamically fulfilled.

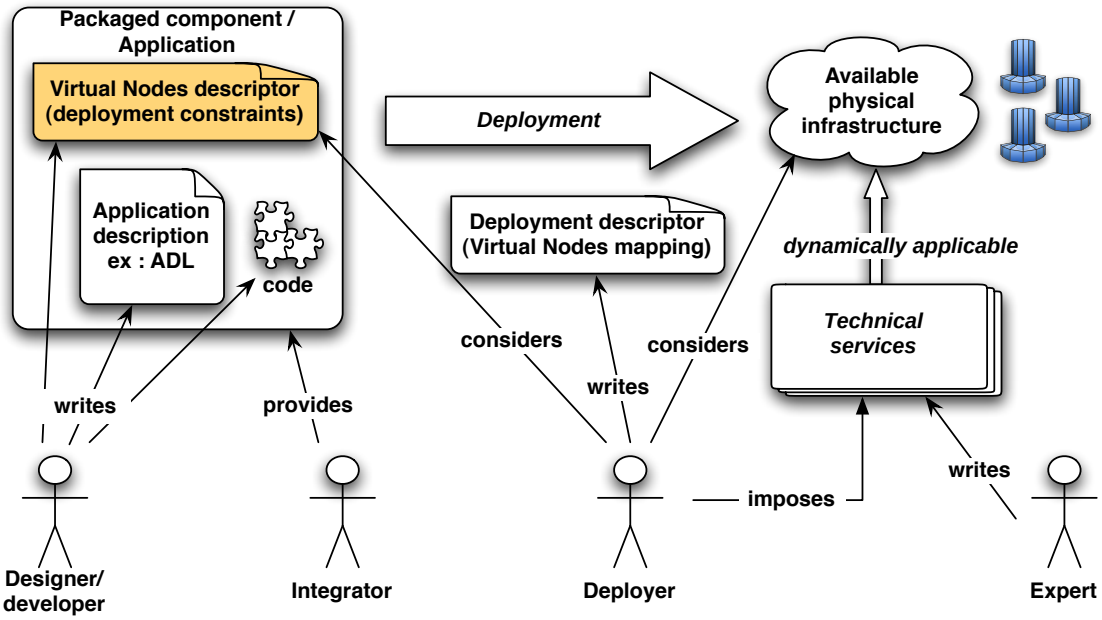

Figure 6.7: *Deployment roles and artifacts*

6.4.1.3 Dynamically fulfilled constraints

The deployer may decide to use an acquisition-based deployment, which means retrieving existing nodes from a given infrastructure (for instance a P2P infrastructure). In that case, available nodes exhibit static configurations as chosen by the administrator when deploying the P2P infrastructure. The deployer or deployment tool filters available nodes based on these requirements and should only propose matching nodes. In general, the deployment of a given application currently takes place on pre-configured infrastructures.

This selection process is unfortunately restrictive, as when using an existing node infrastructure, one may not find any matching resource. Deployment on this existing infrastructure is therefore impossible in such a case.

Moreover, some requirements that are usually considered as static, may actually be dynamically fulfilled. An example being the operating system: for instance, when deploying on the french Grid'5000 infrastructure, the operating system can be installed at deployment time [CAP 05].

Lastly, different strategies may be applied to fulfill non-functional requirements and the most adequate strategies may depend on the characteristics of the infrastructure at runtime, for example the topology.

The concept of technical services, which is described in Section 6.3, can be used to allow such dynamic adaptation.

6.4.1.4 Problem with composition of components

If two separate and packaged components define incompatible constraints on homonymous virtual nodes, they *cannot* be deployed on the same target nodes. Fortunately, this problem can be solved by creating a composite component containing these components

and performing renaming of the virtual nodes: deployment can then be performed on a disjoint set of nodes, which eliminates the incompatibility issue. Figure 6.8 provides an illustration of this method: component 1 and component 2 are packaged components that both define constraints on a virtual node named VN1, but these constraints are incompatible. By wrapping these components into the composite component named component 3, it is possible to remap the deployment of components 1 and 2 onto the separate virtual nodes VNA and VNB. The remapping takes place in the ADL of the composite component; we provide an extension of the ADL for this purpose.

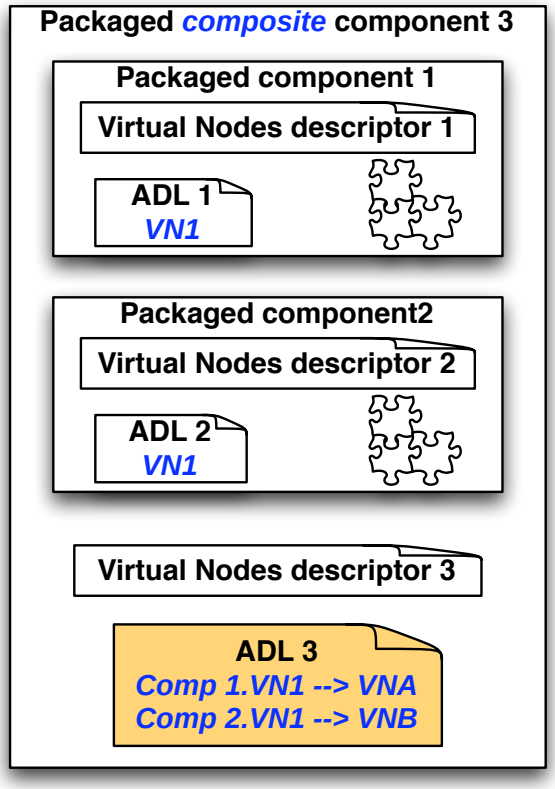

Figure 6.8: *Composition of components with renaming of virtual nodes*

6.4.2 Virtual Nodes Descriptors

The description of virtual nodes is expressed in a dedicated virtual nodes descriptor, in XML format :

```
<virtual-nodes>
   <virtual-node name="VN1">
     <technical-service type="services.Service1"/>
     <processor architecture="x86"/>
     <os name="linux" release="2.6.15"/>
   </virtual-node>
</virtual-nodes>
```

Non-functional contracts are here expressed in a simple way. The technical-service tag specifies the technical service (more precisely the type, class or idealy interface, defining the technical service), which has to be applied on the virtual node VN1 at deployment

time. Regarding static constraints, we are also considering adopting the JSDL naming conventions for defining static constraints on virtual nodes.

The deployer in charge of writing the deployment descriptor is aware of the requirements by looking at the virtual nodes descriptor, and must ensure the infrastructure matches the requirements. There is no contract management module, such as in [LOQ 04], nor deployment planner such as in [LAC 05]. Indeed, contracts are verified when retrieving nodes from the physical infrastructure, resulting in runtime errors if contracts are not respected. This ensures a simple framework in terms of specification and verification, eludes resource planning issues, and could still be plugged to a resource allocator framework such as Globus' GARA [FOS 00].

6.4.3 Deployment process: summary

The specification of non-functional constraints as well as the dynamic fulfilling of the constraints expressed as technical services imply a new deployment process, which is summed up in a standard case in Fig. 6.7. In this figure, roles and artifacts are explicitly expressed :

- the designer and developer provide the code and the description of the component system (ADL), as well as the non-functional constraints in a virtual node descriptor;
- the integrator gathers compiled sources, ADL and virtual nodes descriptor into a deliverable package;
- the deployer writes or adapts a deployment descriptor so that the deployment of the component system verifies the virtual nodes descriptor with respect to the available infrastructure. Technical services are applied at runtime if needed.

Two other scenarios are possible:

- a designer wants to use a given set of Grid resources, and the interface to these resources is a deployment descriptor provided by the system administrator/provider of the resources;
- a deployer wants to use available packaged components, and deploy them on a given infrastructure for which a deployment descriptor is available.

In all scenarios, the specification of non-functional constraints in the virtual nodes descriptor ensures that the application requirements and the physical infrastructure are compatible, and this compatibility is possibly attained by dynamically updating the configuration of the physical resources.

6.4.4 Use case: deployment on a P2P infrastructure with fault-tolerance requirements

This section shows the concept of dynamically fulfilled deployment constraints through technical services: it presents a use case involving the deployment of a component system with some fault-tolerance requirements on a P2P infrastructure; it demonstrates how the proposed approach helps resolving deployment and QoS requirements in the most suitable way.

6.4.4.1 Virtual nodes descriptor example

To illustrate our mechanism of constrained deployment, we consider the same *master-worker* application for solving flow-shop problems as described in Section 6.3.3.2. As we already explained, there is *no* need for applying an automatic fault-tolerance mechanism (then to pay an execution-time overhead) on the workers. On the contrary, the manager and the sub-managers *must* be protected against failures by the middleware since there is no failure-handling at application level for them.

In this case, the designer of the application specifies in the virtual nodes descriptor that a fault-tolerance technical service must be applied on the virtual node that hosts manager components, while there is no such constraint on a worker component:

```
<virtual-nodes>
   <virtual-node name="managers">
      <technical-service type="services.FaultTolerance"/>
      <processor architecture="x86"/>
   </virtual-node>
<virtual-nodes>
```

When deploying the application, the deployer can choose the most adapted fault-tolerance mechanism depending on the environment by configuring the technical service. This technical service must fit, *i.e.* extends or implements, the type specified in the virtual node descriptor. For instance, suppose that the application is deployed on a desktop Grid provided by the P2P infrastructure. Such resources being strongly prone to failure, the chosen fault-tolerance mechanism must deal with very frequent failures, and thus provide a reactive and fast recovery, even at the expense of a high overhead on execution time. Using a lighter but weaker mechanism in this case could lead the system to continuously recover. Finally, the deployer chooses the PML approach, with a small TTC value (60 sec) as in the following deployment descriptor:

```
<ProActiveDescriptor>
  <componentDefinition>
    <virtualNodesDefinition>
      <virtualNode name="managers" property="multiple" serviceRefid="ft-master"/>
      <virtualNode name="workers" property="multiple"/>
    </virtualNodesDefinition>
  </componentDefinition>
 ...
  <aquisition>
    <aquisitionDefinition id="p2pservice">
      <P2PService nodesAsked="1000">
        <peerSet>
          <peer>rmi://peer.registry:3000</peer>
        </peerSet>
      </P2PService>
  </acquisitionDefinition>
 ...
  <technicalServiceDefinitions>
    <service id="ft-master" class="services.FaultTolerance">
      <arg name="proto" value="pml"/>
      <arg name="server" value="rmi://host/FTServer"/>
      <arg name="TTC" value="60"/>
    </service>
  </technicalServiceDefinitions>
```

</ProActiveDescriptor>

6.4.5 Virtual nodes descriptors analysis

In the previous example, the concept of technical service has allowed to apply the *necessary and sufficient* fault-tolerance mechanism when deploying the application:

- *necessary* thanks to the virtual nodes descriptor; the designer has specified the minimum fault-tolerance requirements for its application. Without this specification, the deployer could have unnecessarily applied fault-tolerance on *all* the application;
- *sufficient* thanks to the possibility to choose *at deployment time* how the constraint specified by the designer should be fulfilled to take into account the characteristics of the available resources. If the fault-tolerance aspect had been fully specified by the designer at development time, the chosen fault-tolerance mechanism could have been too weak to be able to deploy on a desktop Grid.

The previous example also illustrates the pertinency of the virtual nodes descriptor mechanism in a concrete use-case: deploying an component-based application with fault-tolerance on an heterogeneous Grid provided by the P2P infrastructure proposed in this thesis.

This mechanism aims at specifying environmental requirements that may be defined by developers, and specified by deployers by parameterizing deployment abstractions. This mechanism is integrated in the ProActive middleware, and allows flexible component deployments thanks to easily configurable technical services. Application designers can specify minimum deployment requirements, and deployers are able to apply the optimal configuration that fulfills those requirements.

6.5 Balancing active objects on the peer-to-peer infrastructure

One of the main features of a distributed system is the ability to redistribute tasks among its processors. This requires a redistribution policy to gain in productivity by dispatching the tasks in such a way that the resources are used efficiently, *i.e.* minimizing the idle time average of the processors and improving applications performance. This technique is know as *load-balancing*. Moreover, when the redistribution decisions are taken at runtime, it is called *dynamic load-balancing*.

ProActive proposes a dynamic load-balancing mechanism [BUS 05] based on the algorithm of Shivaratri and *al.* [SHI 92]. In addition of adapting this algorithm to active objects, the load-balancing mechanism is designed to take advantage of dynamic resources acquisition provided the P2P infrastructure proposed in this thesis (see the Chapter 3).

This section first explains the fundamentals of our active objects load-balancing algorithm, this work is a collaboration with Javier Bustos from OASIS project.Then, we show implementation issues and benchmarks of our algorithm with a Jacobi parallel application.

6.5.1 Active object balancing algorithm

Dynamic load-balancing on distributed systems is a well studied issue. Most of the available algorithms [SHI 92] focus on fully dedicated processors with homogeneous networks, using a threshold monitoring strategy and reacting to load imbalances. On P2P networks, heterogeneity and resource sharing (like processor time) are key aspects and most of these algorithms become inapplicable, producing poor balance decisions to low capacity processors and compensating with extra migrations.

Moreover, due to the fact that processors connected to a P2P network share their resources not only with the network but also with the machine owner, new constraints like reaction time against overloading and bandwidth usage become relevant.

In this section, the relation between active object service and processing time is presented. Then, the adaptation of the Shivaratri and *al.* load-balancing algorithm [SHI 92] for P2P active object networks is presented.

6.5.1.1 Active objects and processing time

When an active object waits idly (without processing), it can be on a *wait-for-request* or a *wait-by-necessity* state (see Figure 6.9). While the former represents an under utilization of the active object, the latter means 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 lesser application performance. Therefore, we focus on a reduction in the *wait-by-necessity* time delay.

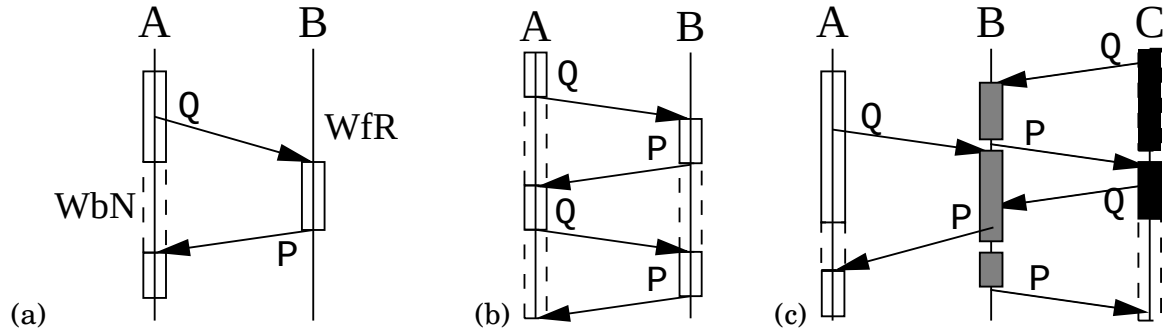

Figure 6.9: *Different behaviors for active objects request (Q) and reply (P): (a) B starts in wait-for-request (WfR) and A made a wait-by-necessity (WfN). (b) Bad utilization of the active object pattern: asynchronous calls become almost synchronous. (c) C has a long waiting time because B delayed the answer*

Even though the balance algorithms will speed up applications like in Figure 6.9 (b), we will not consider this kind of behavior, because the time spent by message services is so long that the usage of *futures* is pointless. In this sort of application design, asynchronism provided by *futures* will unavoidably become synchronous. This is the same behavior experienced when using an active object as a central server. Migrating the active object to a faster machine will reduce the application response time but will not correct the application design problem.

Therefore, we focus on the behavior presented by Figure 6.9 (c), where the active object on C is delayed because the active object on B has not 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. This happens, because C *wait-by-necessity* time will shorten, and B will decrease its load.

6.5.1.2 Active object balance algorithm on a central server approach

Suppose a function called load(A,t) exists, which gives the usage percentage of processor A since t units of time. Defining two threshold: OT and UT (OT > UT), we say that a processor A is *overloaded* (resp. *underloaded*) if load(A,t) > OT (resp. load(A,t) < UT).

The load balancing 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 choose 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 minimizing the reaction time against overloadings and, as explained in Section 6.5.1.1, 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.

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. This would produce the same mechanism, with the drawback of adding the search time to reaction time against overloading. In general, all search mechanisms 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[1] 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. Otherwise it unregisters from the central server.
3. Overloaded processor A migrates an active object to the underloaded one.

[1]Choosing the correct processor B requires further research, but for now the median has proved reasonable approach.

6.5.1.3 Active object balancing using P2P infrastructure

Looking for a better underloaded processor selection, we adapted the previous algorithm, using a subset of peer acquaintances from the P2P infrastructure (fully described in Chapter 3) to coordinate the balance.

Suppose the number of computers on the P2P network is N, large enough to suppose them independents on their load. If p is the probability of having a computer on an underloaded state, and the acquaintances subset size is $n << N$, then the probability of having at least k responses is

$$\sum_{i=k}^{n} (\tbinom{n}{k}) p^i (1-p)^{n-i}$$

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 $\mathrm{rank(A)}$ to a subset n of its acquaintances (using group communication).
2. When a process B receives a balance request, it checks if $\mathrm{load(B,T)} < \mathrm{UT}$ and $\mathrm{rank(B)} \geq \mathrm{rank(A)}$-$\epsilon$ (where $\epsilon > 0$ is to avoid discarding similar, but unequal processors), 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.

6.5.1.4 Migration

A main load-balancing algorithm problem is the *migration time*, defined as the time interval since the processor requests an object migration, until the objects arrives at the new processor [2]. Migration time is undesirable because the active object is halted while migrating. Therefore, minimizing this time is an important aspect on load-balancing.

While several schemes try minimizing the *migration time* using distributed memory [FRI 98] (not yet implemented in Java), or migrating idle objects [GRI 97] (almost inexistent on intensive-communicated parallel applications), we exploit our P2P architecture to reduce the migration time. Using a group call, the first reply will come from the nearest acquaintance, and thus the active object will spend the minimum time traveling to the closest unloaded processor known by the peer.

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

[2] In ProActive, an object abandons the original processor upon confirmation of arrival at the new processor.

object generally has a longer queue than similar objects on the new processor, thus a low priority for moving.

By experimentation (see the next Section), we have observed that these migration problems are not present when using this approach.

6.5.2 Experiments

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 3.4 GHz and 6 Pentium XEON 2.0 GHz, 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 P2P infrastructure to share JVMs. Functions $\mathrm{load}()$ (*resp.* $\mathrm{rank}()$) of section 6.5.1.2 and 6.5.1.3 were implemented with information available on $/\mathrm{proc}/\mathrm{stat}$ (resp. $/\mathrm{proc}/\mathrm{cpuinfo}$). Load-balancing algorithms were developed using *ProActive* on Java 2 Platform (Standard Edition) version 1.4.2.

In our experience, using our knowledge of the lab networks, we experimentally defined the algorithm parameters as OT = 0.8 (to avoid swapping on migration time), and UT = 0.3; in order to have, in normal conditions, 80% of desktop computers on underloaded state.

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: $rank(P) = \log_{10} speed(P)$, with $\epsilon = 0.5$.

When we implemented the algorithm, a new constraint came to light: 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 affects 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:

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

This formula has a constant component (migration time) and a dynamic component which decrease the update time while the load increase, minimizing 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 neighbors in their last iteration. We divided a 3600x3600 matrix in 36 workers all equivalents, and each worker communicates with its direct matrix neighbors.

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 in Section 6.5.1.2 (having a cpu clock of $3GHz$ as reference) and then using the P2P version defined in Section 6.5.1.3 (having $n = 3$). Measured values of these experiences can be found in Figure 6.10.

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 in Figure 6.10 are the values of this experience. Applying the information from the *non-balanced* experience, we tested the *number of actives objects* as a load index, defining $\mathrm{UT}{=}3,\mathrm{OT}{=}4$ to have around 3 active objects per machine. Measured values for this experience are represented by the x symbol in Figure 6.10.

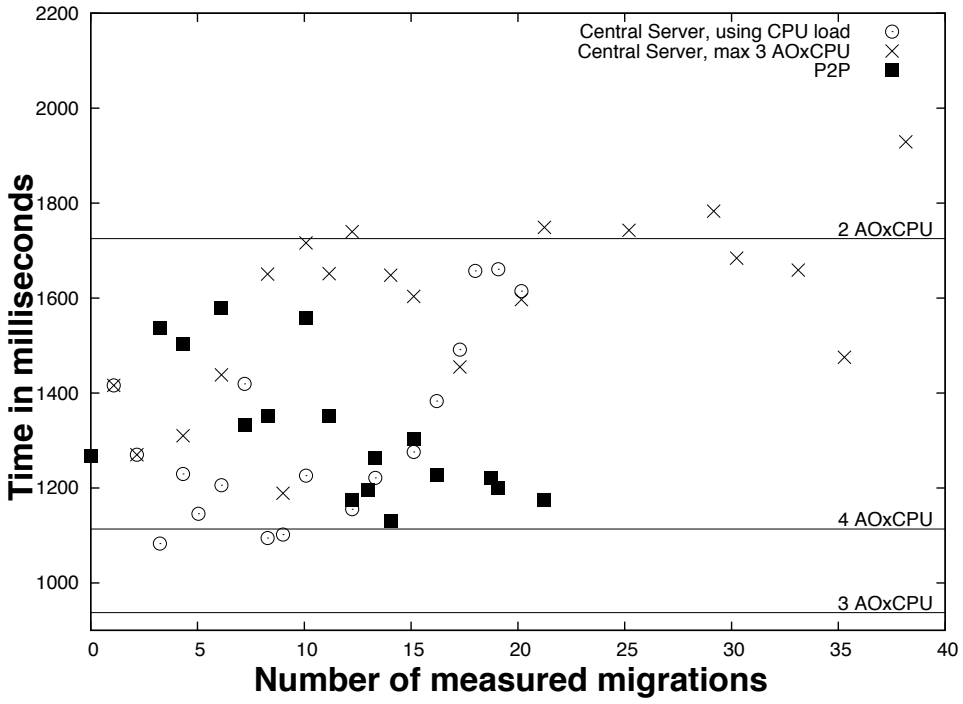

Figure 6.10: *Impact of load-balancing algorithms over Jacobi calculus*

While using the information from the *non-balanced* experience seemed to be a good idea, the heterogeneity of the P2P network produced the worse scenario: large number of migrations and bad migration decisions, therefore poor performance on Jacobi calculus. Using CPU usage as load index had better performance than the previous case: while the central server oriented algorithm produced low mean times for low rate of migrations (an initial distribution near to the optimal), the P2P oriented algorithm presents a better performance while the number of migrations increases. Moreover, considering the addition of migration time on Jacobi calculus performance, the P2P balance algorithm produces the best migration decisions only using a minimal subset of its neighbors. The use of this minimal subset produces also a minimization in number of messages for balance coordination. This fact and the neighbor approach of our P2P network provide automatically scalability conditions for large networks.

6.5.3 Load-balancing analysis

We have introduced a P2P dynamic load-balancing for active objects, focusing on intensively communicating parallel applications. We started introducing the relation between active objects and CPU load. Then, an order relation to improve the balance was defined. The case study showed that, if the number of migrations increase (this can occurs due to a non-optimal distribution or due to the dynamic behavior of the P2P network), the performance (on reaction time and migration decisions) increases for the P2P algorithm and decreases for the central server approach. Also, the load-balancing algorithm exploits the P2P architecture to provide scalability conditions for large networks.

As we have shown with experiments this load-balancing is implemented and it can be used as technical service (as presented before in this chapter).

6.6 Conclusion

In this chapter, we have described the deployment framework of the ProActive Grid middleware. This deployment framework is based on the concept of virtual nodes, which is an abstraction of the physical infrastructure for the application point of view.

Virtual nodes do not allow easier resource localization on Grids, therefore we have improved the deployment with a mechanism to localize nodes on Grids. This new mechanism is used by the implementation of our branch-and-bound framework (*Grid'BnB*), see Section 4.3. The node localization permits to organize workers in groups, where workers are located on the same cluster, in order to reduce the cost of inter-worker communications.

The second contribution presented in this chapter is the concept of technical services. Technical services are non-functional requirements that may be dynamically fulfilled at runtime by adapting the configuration of selected resources. We have implemented a technical service for fault-tolerance and a second for load-balancing.

The load-balancing technical service is specially adapted to take advantage of resources acquisition provided the P2P infrastructure proposed in this thesis (see the Chapter 3).

The last contribution of this chapter is the virtual nodes descriptor. Application developers can specify minimum deployment requirements, such as technical services, with the virtual nodes descriptor, and deployers are able to apply the optimal configuration that fulfills those requirements.

Finally, the ProActive deployment framework and virtual nodes descriptors are used in the Grid Component Model (GCM) proposal from the European Coregrid network, a Europen academic Network of Excellence (NoE). A draft of the standard [GRI b] is in preparation by the GridComp project, which aims to define and implement the GCM.

Chapter 7

Ongoing Work and Perspectives

This chapter introduces ongoing and perspective work based on this thesis contributions. We are currently working on a job scheduler that allows INRIA Sophia people to freely use the *INRIA Sophia P2P Desktop Grid*. In addition, we are working to improve the resource discovery mechanisms of the P2P infrastructure. For the moment, we are exploring two different ways to achieve this goal: first, by modifying the message protocols; second, by a concept of tagging peers.

The branch-and-bound framework is also subject to ongoing work, especially with the improvement of our flow-shop implementation. Furthermore, we plan to do larger-scale experiments by including clusters located in Netherlands and in Japan.

Finally, we introduce our current work on Grid application deployments. This work aims at providing a contract between developers, infrastructure managers, and application users.

The chapter is organized as follow. First, we introduce all current peer-to-peer related work. Second, we present branch-and-bound ongoing and perspective work. Last, we present a contract mechanism for Grid deployment.

7.1 Peer-to-Peer

7.1.1 Job scheduler for the peer-to-peer infrastructure

With the n-queens computation success, the usability and the viability of the *INRIA Sophia P2P Desktop Grid* has been proved. This experiment especially shows that the infrastructure is well suited to a mono-application usage, which requires all available resources for a long-running computation. We now would like to open the infrastructure to all INRIA Sophia researchers.

After several discussions with our colleagues, we pointed out that the infrastructure may involve some people of the lab for running very long network simulations, or image rendering, or even simulating the human heart. However, all these scenarios require a small utilization of the infrastructure, in terms of number of used resources. Also, we noticed that all these cases may be described by a set of independents or dependent tasks. In other words, we can specify several kinds of job application: single task, parametric sweep tasks, and work-flow tasks.

Hence, we decided to start the development of a job scheduler for the P2P infrastructure. This development is in collaboration with Johann Fradj, Jonathan Martin, and Jean-Luc Scheefer. In addition, we decided to provide a graphical tool to help user submit and manage their jobs, Figure 7.1 shows a screen-shot of the current GUI.

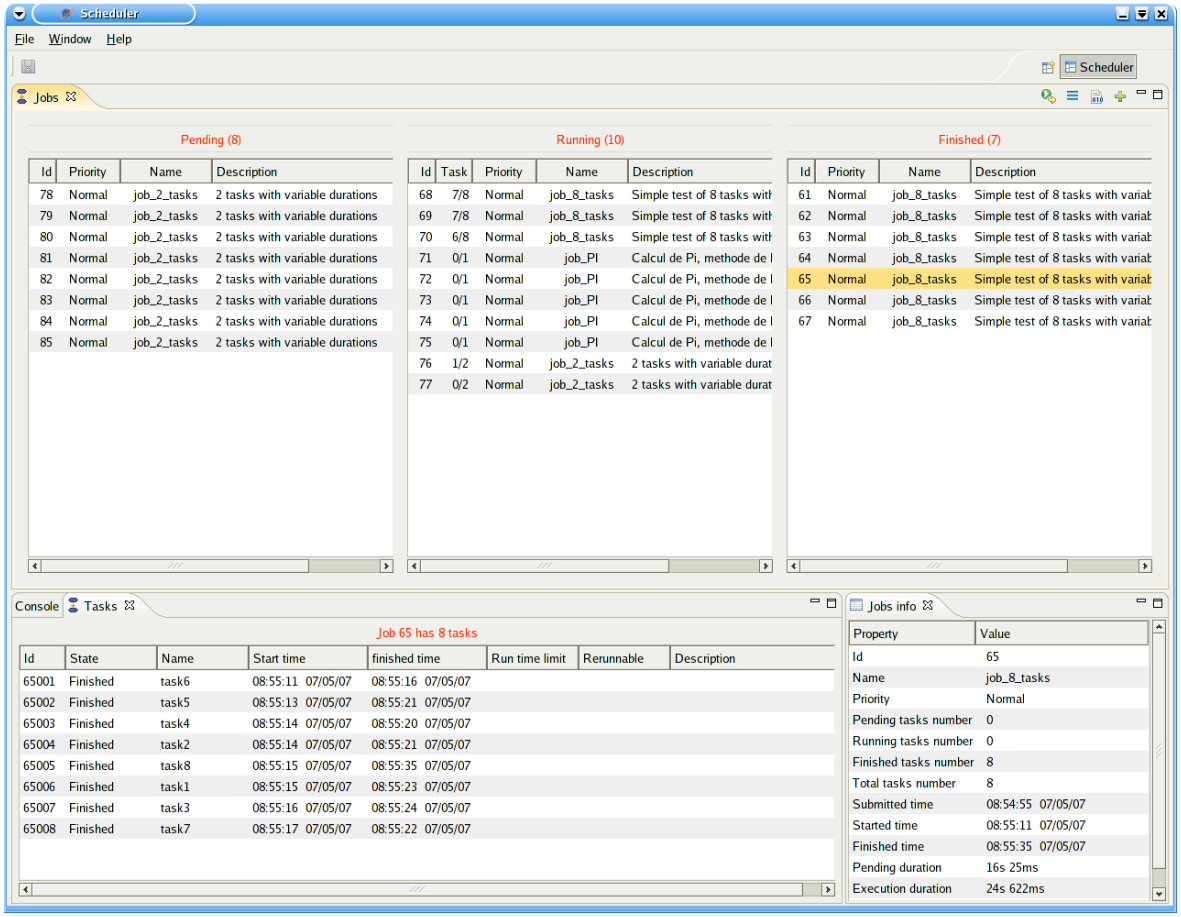

Figure 7.1: *Screen-shot of the job scheduler*

Because, we plan to have several users of the infrastructure, we started to develop another graphical tool to manage the infrastructure, this tool is for infrastructure administrators. Figure 7.2 shows a screen-shot of the current infrastructure manager GUI.

7.1.2 Peer-to-Peer resource localization

The P2P infrastructure proposed in this thesis, see Chapter 3, considers all shared resources as similar. From the infrastructure point of view, resources can be free or busy, it is the only one difference between them. In Chapter 4, we presented *Grid'BnB*, a framework for Grids, which has for main particularity to organize workers in groups with respect to their Grid localization. A mechanism has been presented, in the previous chapter, to solve this issue. This mechanism of Grid node localization is based on a tagging mechanism. Unfortunately, this mechanism is not optimal with the P2P infrastructure, even if we used it for large-scale experiments (see Chapter 5). Furthermore,

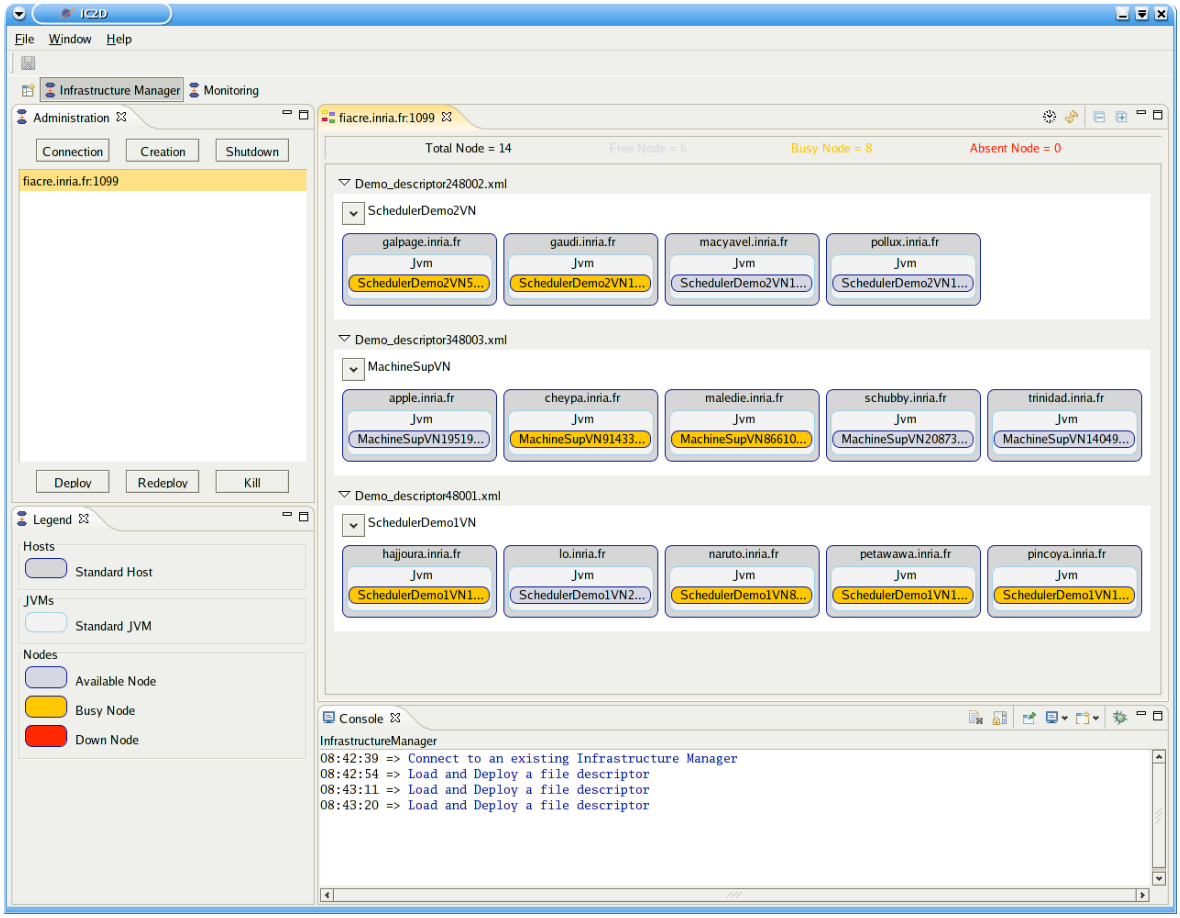

Figure 7.2: *Screen-shot of the infrastructure manager*

the first users of the scheduler reported the need of requesting particular resources, which have a specific system library, enough memory, *etc.*

In addition, we presented in this thesis some work using the transparent fault-tolerance provided by ProActive. The fault-tolerance mechanism proposes two configurable protocols, these protocols are applied in regards of the infrastructure on which the application is deployed. In other words, it is not the same fault-tolerance protocol for a desktop machine and for a cluster one.

All these advanced uses of the infrastructure point out the need of solutions for identifying resources on Grids. In this section, we introduce two different strategies that aim to solve the localization of specific computational resource on Grids managed by the P2P infrastructure.

First, we introduce the *node family* mechanism that is a simple way to identify resources especially for fault-tolerance. We then describe a new message protocol for the infrastructure that allows complex resource requests.

7.1.2.1 Node family

We are currently working with Christian Delbé from OASIS research group on the concept of *node family*. In the previous chapter, we pointed out the lack of resource localiza-

tion in the P2P infrastructure. Especially, for the ProActive fault-tolerance configuration, which has to be differently configured depending of desktop machines or clusters.

This work aims to help applications to have a finer use of resources acquired from the P2P infrastructure. The proposed mechanism has for main particularity to avoid modifying the current message protocol of the infrastructure.

In this work, we assume that an administrator installs and manages the deployed P2P infrastructure. For instance, the *INRIA Sophia P2P Desktop Grid* has three people who are in charge of maintaining it up. The key idea of the *node family* concept is that the administrator groups peers in *families*. A family gathers peers with the same set of characteristics. Characteristics may be whatever the administrator thinks that is important and usable, such as creating a family with Windows OS peers or with 1 GB RAM or even both.

When a new peer is installed by the administrator, she sets the family names in which the future peer would be member. The peer has now a table with its family names. The request messages have a new parameter: *a list of family names*. Hence, P2P infrastructure protocols are still the same, except that protocols now match the list of requested families with the local table before returning free nodes. In addition, when nodes are returned to applications, they can check the family membership of nodes.

The list of available families with their description must be accessible by the infrastructure users. Thus, users are aware of the resource selection that they may be able to do in their applications. For instance, users may choose to dynamically apply fault-tolerance to acquired nodes. Users know that there are two families: *desktop*, which means the node runs on a desktop; and *cluster*, which means the node runs on a cluster. In regards of these families, users may choose the most adapted fault-tolerance protocols for the acquired machines.

7.1.2.2 Resource discovery for unstructured peer-to-peer

With Fabrice Huet and Imen Filali of the OASIS research group, we are currently working on a new message protocol for the P2P infrastructure [FIL 07]. Our goal is to design a new unstructured P2P resource discovery protocol in a distributed Grid environment in order to enhance scalability, robustness, and efficiency.

The P2P infrastructure presented in this thesis relies on an unstructured P2P model where connection between peers are arbitrarily established. Such model makes resource discovery more challenging because of a lack of global routing guaranties offered by the overlay.

The resource discovery mechanism actually adopted in this infrastructure is limited to asking for free computational nodes without any explicit description of application requirements. In addition, it relies on Breadth-First Search algorithm (BFS): by using a flooding approach, a peer needing computational resources broadcasts to all its neighbors a message containing the number of asked nodes. An available peer receiving the query sends its reference to the requester. Otherwise, a busy peer forwards the query according to the flooding mechanism. The procedure is executed until the number of the requested nodes is satisfied or TTL (Time to Live) of the message reaches zero. However, the flooding-based query algorithm generates a large amount of traffic.

Aiming to enhance the actual resource discovery deployed in the the *INRIA Sophia P2P Desktop Gird*, we present in this ongoing work a new peer-to-peer resource discovery mechanism in a distributed Grid environments. Reducing network traffic, message duplication, and search time are the main challenges of this work. Our objective is then to come up with a new sophisticated distributed protocol for resource discovery and reservation in peer-to-peer Grid environments.

The proposal of this work is called *Resource Discovery Protocol* (RDP). In RDP, we assume that each peer in the network has to announce periodically its available resources so that other nodes in the network will be aware of which peer can better fulfill their future requirements. A resource can have several formats such CPU, disk storage, and a block of a specified information. We design RDP in such a way to reduce the control messages exchanged in the network by introducing the use of cache systems in the peers to store an up-to-date view of the location of the available resource in the network. For this end we define four messages and their formats which are distributively exchanged between peers either to offer *resources* (Grant message), to *ask for resources* (Request message), to *reserve resources* (OK request), or to *confirm a reservation* (OK reply).

One of the main contributions of this work is the design and implementation of a flexible simulation framework for resource discovery and reservation protocols used to simulate both previous approaches, namely RDP and the flooding technique [FIL 07]. With this simulation framework, we show that using the RDP scheme can achieve an important success rate. Compared with Flooding, we demonstrate that RDP can reduce the traffic load on the peers by limiting the number of received messages and duplicated messages. We have also studied the impact of some RDP parameters on the success rate value such as message distribution parameters, message cache lifetime.

There are several issues that still need to be investigated and they are left for a future work. Because of the important number of parameters on which depend this proposal, running other simulations with other topologies and more different scenarios is very important in order to investigate RDP behavior under more different conditions.

Another interesting research area is the investigation of the performance of RDP, and especially the cache system in a dynamic network where the arrival and the departure of peers is frequent.

Another potential area of future work is the management of more complex queries where peers can ask for many resources such as CPU, memory, storage space. Therefore, the design of flexible formalism for resource description is planned.

7.2 Branch-and-Bound

In this section, we present some perspective work around our branch-and-bound framework and the flow-shop problem.

With Laurent Baduel from the Tokyo Institute of Technology, we recently started a collaboration that aims to run large-scale experiments with *Grid'BnB*. For the moment, we plan to improve our flow-shop implementation with a better objective function, such as the lower bound technique proposed by Lageweg [LAG 78].

Likewise, we want to run larger scale experiments on a worldwide Grid, by mixing

Grid'5000 (France), DAS (Netherlands), and clusters located in Japan. For these experiments, we have first to improve the forwarder mechanism that allows us to pass through firewalls, and then to test the B&B framework to fix eventual bottleneck issues. We aim to dynamically build this Grid with the P2P infrastructure. Furthermore, we would also include in the Grid desktop machines of the INRIA lab.

7.3 Deployment contracts in Grids

In this thesis, we introduced the concept of *technical services* (see the Section 6.3), which are non-functional requirements of the application that are configured and applied at deployment time. In addition to technical services, we presented the *virtual nodes descriptor* (see the Section 6.4), which allows programmers to describe their application requirements.

In this section, we introduce our current work with Mario Leyton of the OASIS research group, this work aim at extending technical services and virtual nodes descriptor to allow contract based deployment [BAU 07].

7.3.1 Design goals

Traditionally the programming and execution of a distributed application has been handled by a single individual. The same individual programs the application, configures the resources, and performs the execution of the application on the resources. Nevertheless, the increasing sophistication and complexity of distributed applications and resource infrastructures has led to the specialization of expert roles.

On one side we find the *developers* of distributed applications, and on the other side the *infrastructure managers* who maintain resources such as Desktop machines, Servers, Cluster and Grids. Between both of these expert roles we can identify the *users* who take the applications and execute them on a distributed infrastructure to solve their needs.

The separation of these roles raises the issue of how programmers and infrastructure experts relate to solve the needs of the users. The complexity of this issue is emphasized when considering that the programmers and infrastructure managers are unacquainted. That is to say, a user has to deploy and execute an unfamiliar application on unfamiliar resources without knowing the requirements of either.

In this work we address the issue of reaching contractual agreement between distributed applications and resource infrastructures during deployment. We propose the deployment time as the key moment to reach an agreement between the infrastructure and the application. Using the contracts, users are able to perform the deployment and execution of an unfamiliar application on unfamiliar resources effortlessly.

7.3.2 Contracts and agreements

The three roles that we have identified: *developers*, *infrastructure managers*, and *users* are related with the *applications*, *descriptors*, and *deployment/execution* respectively. The developer writes the application, the infrastructure manager writes the deployment descriptor, and the user performs the deployment and execution of the application on the infrastructure using the deployment descriptor.

To begin with, the application and descriptor must agree on the name of the virtual-node. Nevertheless, the virtual-node name is not the only agreement problem that the application and descriptor have. More importantly, the application and descriptor must agree on the required and provided technical services such as: fault-tolerance, load-balancing, *etc.*

Modifying the application or the descriptor can be a painfull task, specially if we consider that the user 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. Figure 7.3 illustrates the issue. The user is not aware of the application or descriptor requirements.

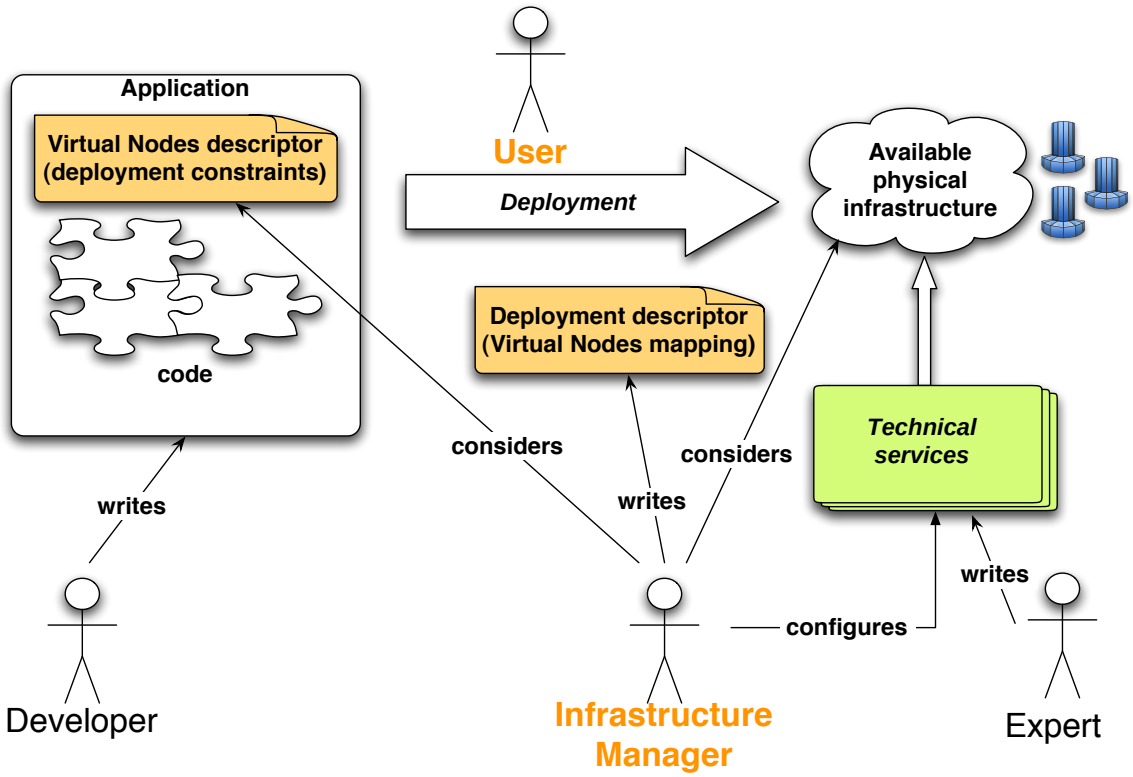

Figure 7.3: *Contract deployment roles and artifacts*

In the rest of this section we analyze different scenarios where the roles of developers, users, and infrastructure managers are combined or separated into different people, and explain different approaches that are able to solve these scenarios.

7.3.2.1 Application virtual node descriptor

Virtual node descriptor is a mechanism for specifying the environmental requirements of the applications, fully described in the previous chapter.

Using virtual node descriptors, the user does not have to be aware of the application design and implementation. By simple inspection of the virtual node descriptor, the user can know the requirements of the application.

Table 7.1: *Types*

Type Name	Provides Value	Requires Value	Set constraints	Priority
Application	App	Desc	Desc	App
Descriptor	Desc	App	App	Desc
Application-Priority	App,Desc	Desc	App,Desc	App,Desc
Descriptor-Priority	Desc,App	App	Desc,App	Desc,App
Environment	Env	Desc,App	Desc,App	Env

By default there is no contract management module, such as in [LOQ 04], nor deployment planner such as in [LAC 05]. Indeed, virtual node descriptors are verified when retrieving nodes from the physical infrastructure, resulting in runtime errors if the requirements are not satisfied. This ensures a simple framework in terms of specification and verification, eludes resource planning issues, and could still be plugged in a resource allocator framework such as Globus's GARA [FOS 00].

Nevertheless, developers do not know on which infrastructure the applications will be deployed, and the infrastructure may not support some specific requirement of the application. Therefore, in Section 7.3.3 we propose to describe the infrastructure with a mechanism based on coupling contracts, which is described in the next section.

7.3.2.2 Coupling contracts

Coupling Contracts proposes to capture the properties of *how* information agreement takes place between parties, specifically between applications and descriptors [BUS 06]. To achieve this, each party provides an interface holding a set of typed clauses. The clauses specify *what* information is required and provided by each party, and the type specifies how an agreement on this information is reached. If the interfaces are compatible, the coupling of the interfaces yields a contract with the agreed values for the clauses.

7.3.2.3 Concepts: contracts, interfaces, and clauses

Typed Clauses correspond to the information that both parties must agree on. A clause is defined by a type, a name and a value. The clauses are typed with one of the alternatives shown in Table 7.1. As an example, the *Application* type specifies that the value of the clause can only be set by the application. The descriptor specifies a value as required, forcing the application to provide a value. Another example corresponds to the *Descriptor-Priority* type which specifies that a default value can be provided by the application, and that the value can be overridden by the descriptor. Additionally, parties can enforce constraints on the value of the clauses such as maximal and minimal value, choices, *etc.* The default constraint corresponds to non-emptiness.

Interfaces represent a grouping of clauses that are exposed by each party. An interface is defined by a name and a set of clauses.

Coupling Contracts are the results of coupling two interface. The contract holds the clauses and their values. The values of the clauses are resolved using the specific type for each clause. If there is a conflict of types, or the value does not satisfy the

constraints, then the contract is invalid and the coupling is not allowed. When a contract is valid, then both parties can query the contract to get the value of the agreed clauses.

Typed clauses can also be used to perform advertisement and matchmaking in the Condor style [RAM 98]. Both parties can expose their interfaces (advertisements) to a matchmaker or broker. To determine if the two parties are a suitable match, the coupling contract can be generated and validated.

The clauses belonging to the interfaces will specify *what* information is shared (provided or required) for the matchmaking. And the type of the clauses will specify *how* the information is shared for the coupling.

7.3.3 Deployment contracts example

In this section we show how the concepts introduced in Section 7.3.2 can be merged and applied to provide full separation of roles: developer, infrastructure manager, and user. Specifically, we aim at creating deployment contracts between the applications and the deployment descriptors using the Grid middleware ProActive. We will show how the deployment framework can benefit from the use of: technical services, virtual node descriptors, and coupling contracts to deploy unfamiliar applications with unfamiliar infrastructures.

The example presented in this section uses the fault-tolerance mechanism provided by ProActive, presented in Chapter 6.3.3.1. The fault-tolerance is based on rollback recovery. Several parametrized protocols can be used, with regard to the application requirements and the characteristics of the infrastructure.

The application specifies its provisions and requirements in the virtual node descriptor. Figure 7.4 shows an example for a master-worker application. Symmetrically, Figure 7.6 shows the provisions and requirements of the descriptor. The coupling contract is composed of the clauses specified in both, and the values of this contract will be used in the virtual node descriptor (Figure 7.4), application (Figure 7.5), and in the deployment descriptor (Figure 7.7).

VN_MASTER & VN_WORKERS are of Descriptor type. These clauses will hold the required and provided names of the virtual nodes.

NUM_NODES is of type Application-Priority. The virtual-node-descriptor specifies that the application requires 16 nodes. The descriptor-interface specifies that this value must be grater than zero, and smaller than the maximum number of allowed nodes.

FT_PROTOCOL is of type Descriptor-Priority. The virtual node descriptor specifies that the application requires the fault-tolerance protocol to be either cic or pml, suggesting cic as the default value. On the other hand, the descriptor-interface specifies that the protocol must be one of: pml, cic.

ARCH is of type Application-Priority. The virtual-node-descriptor specifies that the architecture must be configured to x86 because it provides specific binary code for this architecture. The descriptor-interface provides the following architectures: x86, sparc, ppc, and any.

OS is of type Application-Priority. The virtual-node-descriptor specifies that the operating system must be configured to Linux because it provides specific binary code for this operating system. The descriptor-interface provides the following operating systems: Linux, MacOS, Windows, and any.

In the virtual node descriptor, the developer activates the fault-tolerance technical service for the master virtual-node, since it represents a single point of failure in the application. The protocol used for fault-tolerance will correspond to the agreed value of the coupling contract, which in the example corresponds to pml. The developer also specifies the required number of nodes, which is validated using the virtual node descriptor against the allowed minimum. On the other hand, the infrastructure manager specifies in the descriptor the optimistic maximum number of nodes that the infrastructure can provide, and validates the application's required number of nodes using the clause constraints.

The architectures and operating systems that are supported by the infrastructure are specified in the descriptor using typed clauses. The application requirements are also specified as clauses, but in the virtual node descriptor. This is useful for applications that have binary code which runs only on a specific operating system with a specific infrastructure. When the coupling contract is generated, both descriptor and application have reached an agreement on the characteristic of the resources. In the example the agreement corresponds to: Linux, x86.

7.3.4 Contract: related work

The problem of finding suitable resources for a given application have already been addressed by techniques such as matchmaking in Condor [RAM 98, RAM 03], collections in Legion [CHA 97], or using resource management architectures like Globus [CZA 98].

However, the approaches presented in this work not only focus on acquiring resources, but also on generating contractual agreements during the deployment process.

Therefore, our approach pertains more to Service Level Agreement, and more specifically, *how* to manage the negotiation, in order to end up with an agreement between what is usually called customers and providers: *e.g.* with the help of software agents to coordinate the negotiation, as in [GRE 06], or orchestrated along a specific algorithm in the Meta Scheduling Service described in [WIE 06].

Another related approach corresponds to the Web Services Agreement Specification [AND 05b] (WS-Agreement), which is about to become a draft recommendation of the Global Grid Forum [FOR 04]. The WS-Agreement is a two layer model: Agreement Layer and Service Layer. Many of the concepts introduced in our work find their reflection in the Agreement Layer. According to the specification "an *agreement* defines a dynamically-established and dynamically-managed relationship between parties", much like the proposed coupling contracts. Also, the proposed coupling interfaces can be seen as *agreement templates* in WS-Agreement, since they are both used to perform advertisement. Additionally, in the same way that interfaces and contracts are composed of clauses, in WS-Agreement templates and agreements are composed of *terms*. Finally, the concept of constraints is present in both approaches.

The similarity of our proposed approach and WS-Agreement Specification is encouraging when we consider that both were conceived independently. On the other hand,

```
<virtual-nodes>
  <clauses>
    <interface name="application-master-worker-interface">

      <Descriptor name="VN_MASTER" />
      <Descriptor name="VN_WORKERS" />

      <ApplicationPri name="NUM_NODES" value="16"/>

      <DescriptorPri name="${FT_PROTOCOL}" value="cic">
        <or>
          <equals>cic</equals>
          <equals>pml</equals>
        </or>
      </DescriptorPri>

      <ApplicationPri name="ARCH" value="x86"/>
      <ApplicationPri name="OS" value="Linux"/>
    </interface>
  </clauses>

  <virtual-node name="${VN_MASTER}">
    <technical-service type="service.FaultTolerance"/>
  </virtual-node>

  <virtual-node name="${VN_WORKERS}">
    <processor architecture="${ARCH}"/>
    <os name="${OS}"/>
    <nodes required="${NUM_NODES}" minimum="10"/>
  </virtual-node>
</virtual-nodes>
```

Figure 7.4: *Application: VN Descriptor*

```
//If the application and descriptor can not be coupled an exception will be thrown
ProActiveDescriptor pad = ProActive.getProactiveDescriptor("descriptor.xml", "vn-descriptor.xml"
    );

//Retrieving Clauses from the Contract
CouplingContract cc = pad.getCouplingContract();
String vnMasterName = cc.getValue("VN_MASTER");
String vnWorkersName = cc.getValue("VN_WORKERS");

VirtualNode vnMaster=pad.getVirtualNode(vnMasterName);
VirtualNode vnWorkers=pad.getVirtualNode(vnWorkersName);
...
```

Figure 7.5: *Application Code*

```
<clauses>
  <interface name="descriptor-master-worker-interface">
    <Descriptor name="VN_MASTER" value="vn-master"/>
    <Descriptor name="VN_WORKERS" value="vn-workers"/>

    <Descriptor name="MAX_NODES" value="100"/>
    <ApplicationPri name="NUM_NODES" value="1">
      <and>
        <biggerThan>0</biggerThan>
        <smallerThan>${MAX_NODES}</smallerThan>
      </and>
    </ApplicationPri>

    <DescriptorPri name="${FT_PROTOCOL}" value="pml">
      <or>
        <equals>pml</equals>
        <equals>cic</equals>
      </or>
    </DescriptorPri>

    <ApplicationPri name="ARCH" value="any">
      <or>
        <equals>x86</equals>
        <equals>ppc</equals>
        <equals>sparc</equals>
        <equals>any</equals>
      </or>
    </ApplicationPri>
    <ApplicationPri name="OS" value="any">
      <or>
        <equals>Linux</equals>
        <equals>MacOS</equals>
        <equals>Sun</equals>
        <equals>any</equals>
      </or>
    </ApplicationPri>
  </interface>
</clauses>
...
```

Figure 7.6: *Deployment Descriptor Interface*

```
...
  <virtualNodesDefinition>
   <virtualNode name="${VN_MASTER}" serviceId="ft-serv"/>
   <virtualNode name="${VN_WORKERS}"/>
  </virtualNodesDefinition>
...
<technicalServiceDefintions>
  <service id="ft-serv" class="services.FaultTolerance">
    <arg name="proto" value="${FT_PROTOCOL}"/>
    <arg name="server" value="rmi://host/FTServer"/>
    <arg name="TTC" value="60"/>
  </service>
</technicalServiceDefinitions>
...
```

Figure 7.7: *Deployment Descriptor*

the main difference in the approaches is that the definition of a protocol for negotiating agreements is outside of the WS-Agreement Specification scope.

From the WS-Agreement perspective, typed clauses can be seen as an automated negotiation approach because they provide an automated mechanism for accepting or rejecting an agreement.

7.3.5 Conclusion and perspectives

In this section we have addressed the separation of roles: application developer, infrastructure manager, and user. We have identified that agreements must be made between these different roles in order to execute the application on a distributed infrastructure: desktop machines, clusters, and Grids.

We have argued that the key moment to perform an agreement corresponds to the deployment time. During the deployment, the application and infrastructure must reach a contractual agreement. The contract will allow the execution of the application on distributed resources by specifying, among others, the technical services.

To generate the deployment contract we have described the application's provisions and requirements using virtual-node-descriptors, and symmetrically, we have specified the infrastructure provisions and requirements in deployment descriptor interfaces.

In the future we would like to simplify the coupling contracts to allow negotiation with typeless clauses, using constraint satisfaction instead. We would also like to investigate dynamic renegotiation of contracts after the deployment.

Furthermore, this work will be standardized as an European Grid standard [GRI b] by the GridComp European project.

Chapter 8

Conclusion

Grid computing provides a large amount of gathered computational resources. This power attracts new users to Grids. Thanks to this huge number of resources, Grids seem to be well adapted for solving very large combinatorial optimization problems. Nevertheless, Grids introduce new challenges that emphasis the need of new infrastructures and frameworks to hide all Grids related issues.

We proposed a parallel branch-and-bound framework for solving combinatorial optimization problems, the framework addresses the specificity of Grid computing, particularly the heterogeneity, the deployment, the communication, the fault-tolerance, and the scalability. Our framework relies on a peer-to-peer infrastructure that allows to dynamically acquire resources from desktop machines and clusters.

Our contributions may be listed as follows:

- Thanks to an analysis of existing Grid infrastructures and of existing branch-and-bound frameworks, we identified the requirements that our contributions have to fulfill:
  - the *infrastructure* has to allow building Grids by mixing desktop machines and clusters, to enable deployment of communicating applications, and to be able of achieving computations that take months on clusters; and
  - the *framework* has to allow communication between processes, the implementation of different search tree algorithms, sharing the best current bound, and to be fault-tolerant.
- A *desktop Grid infrastructure* based on an unstructured peer-to-peer architecture, fully integrated to the ProActive Grid middleware. With which we achieved:
  - deployed a permanent Grid at INRIA Sophia lab, named *INRIA Sophia P2P Desktop Grid*, gathering a total of 260 desktops; and
  - a world computation record by solving the n-queens problem with 25 queens, the computation took 6 months to complete.
- *Grid'BnB* a parallel branch-and-bound framework for Grids, which focuses on hiding all Grid issues to users. Its main features are:
  - hierarchical master-worker architecture with communication between workers;
  - dynamic task splitting;

  - organizing workers in groups to optimize inter-cluster communications; and
  - fault-tolerance.
- Large-scale experiments by mixing *INRIA Sophia P2P Desktop Grid* machines to clusters from a French-wide Grid, *Grid'5000*, with which we deployed n-queens on 1007 CPUs.
- Improvements of Grid application deployment mechanisms:
  - a node localization mechanism that allows to determine if two nodes are deployed on the same cluster;
  - a framework for deploying and configuring non-functional application requirements, such as fault-tolerance or load-balancing;
  - a mechanism to describe application requirements; and
  - a load-balancing mechanism based on the peer-to-peer infrastructure.

A large part of this thesis work on desktop Grid are now subject to industrialization efforts to make it as a production desktop Grid, especially with the development of a specific job scheduler and tools to ease the infrastructure administration.

Some parts of our application Grid deployment research is now considered as a base for Grid standardization at European level, particularly with the CoreGrid and GridComp projects.

Part II

Résumé étendu en français

(Extended french abstract)

Chapter 9

Introduction

L'objectif principal de cette thèse est de proposer une bibliothèque adaptée aux environnements de Grilles de calcul pour la résolution de problèmes d'optimisation combinatoire basée sur l'algorithme « Élagage et Branchement » (de l'anglais, Branch-and-Bound). Comme nous visons les Grilles à grande échelle, cette thèse propose aussi une infrastructure basée sur une architecture de type pair-à-pair. Cette infrastructure permet en outre de créer des Grilles en combinant aussi bien des ordinateurs de bureau que des grappes de calcul.

9.1 Problématique

Ces dernières années les Grilles de calcul ont été très largement déployées autour du monde afin de fournir des outils de calcul très performants pour la recherche et l'industrie. Les Grilles rassemblent un très grand nombre de ressources hétérogènes qui sont géographiquement distribuées en une seule organisation virtuelle. Les ressources sont le plus souvent organisées en grappes de calcul, qui sont administrées par différents domaines (laboratoires, universités, *etc.*).

L'algorithme de « branch-and-bound » est une technique pour la résolution de problèmes, tels que ceux de la classe optimisation combinatoire. Cette technique a pour but de trouver la solution optimale d'un problème donné et de prouver qu'aucune autre solution n'est meilleure. L'algorithme procède en divisant le problème original en sous-problèmes de tailles inférieures pour lesquels la fonction de résolution ou *« objective function »* (en anglais) calcule les bornes inférieures/supérieures.

A cause de la grande taille des problèmes à résoudre (nombre d'énumérations et/ou NP-difficile), trouver la solution optimale pour un problème donné peut s'avérer impossible sur un simple ordinateur. Toutefois, il est relativement aisé de paralléliser l'algorithme de « branch-and-bound » et, grâce au très grand nombre de ressources que les Grilles fournissent, elles semblent bien adaptées à la résolution de problèmes très importants en taille avec le « branch-and-bound ».

En parallèle du développement des Grilles de calcul, une nouvelle approche, dite pair-à-pair, pour le partage et l'utilisation de ressources a été aussi développée. Le pair-à-pair se concentre sur le partage de ressources, la décentralisation, la dynamicité et la tolérance aux pannes.

Les utilisateurs des Grilles n'ont généralement accès qu'à une ou deux grappes de calcul et ils doivent, de plus, partager leur temps d'utilisation avec les autres utilisateurs ; le plus souvent il ne leur est d'ailleurs pas permis d'exécuter des calculs qui prendraient des mois à terminer. Plus généralement l'utilisation exclusive de la grappe pour leurs expérimentations ne leur est pas autorisée. Ensuite, ces chercheurs travaillent dans des laboratoires ou des institutions qui sont très bien équipés en ordinateurs de bureau ; ces derniers étant le plus souvent sous-utilisés et utilisables par une utilisatrice unique. En outre, ces ordinateurs sont hautement volatiles (par exemple : éteints, redémarrage, pannes). Ainsi organiser ces ordinateurs de bureau, comme un réseau pair-à-pair pour le calcul ou plus généralement le partage de ressources, est devenu de plus en plus populaire.

Toutefois les modèles et les infrastructures existants pour le calcul pair-à-pair sont assez limités avec principalement l'exécution de tâches indépendantes le plus souvent sans communication entre elles. Cependant, le pair-à-pair semble bien adapté aux applications avec un faible rapport communications/calculs, comme par exemple les algorithmes de recherche parallèle. Nous proposons donc dans cette thèse une infrastructure pair-à-pair composée de nœuds de calculs pour les applications distribuées communicantes, comme une bibliothèque de « branch-and-bound » pour les Grilles.

De plus, la Grille introduit de nouveaux défis qui doivent être pris en comptent par l'infrastructure et la bibliothèque. Ces défis peuvent être listés comme suit :

- *Hétérogénéité* : les Grilles rassemblent des ressources provenant de différents sites institutionnels (laboratoires, universités, *etc.*). Cet ensemble de ressources implique des ressources de différents constructeurs informatique, différents systèmes d'exploitation et l'utilisation de différents protocoles réseau. Au contraire, chacun des sites est le plus souvent composé d'une seule grappe de calcul qui est un environment informatique très homogène (même matériel, même système d'exploitation, même architecture réseau pour l'ensemble des ordinateurs de la grappe).
- *Déploiement* : le grand nombre de ressources hétérogènes complique la tâche du déploiement en termes de configurations et de connections aux ressources distantes. Les sites doivent être spécifiés avant le déploiement ou automatiquement découverts à l'exécution.
- *Communication* : la résolution de problèmes d'optimisation combinatoire, même en parallèle avec un très grand nombre de ressources, peut s'avérer très difficile. Néanmoins l'utilisation de communications entre les processus distribués peut améliorer le temps d'exécution. Cependant, les Grilles ne sont pas le meilleur environnement pour les communications et ce, à cause des problèmes d'hétérogénéité, de latence importante entre les sites et de passage à l'échelle.
- *Tolérance aux pannes* : les Grilles sont composées de ressources hétérogènes qui sont administrées par des domaines différents, ainsi la probabilité d'avoir des nœuds fautifs ne sont pas négligeables.
- *Passage à l'échelle* : c'est l'un des défis les plus importants. C'est aussi l'un des plus difficiles à traiter. D'une part, pour le grand nombre de ressources que les Grilles fournissent ; d'autre part, pour la large distribution des ressources qui implique une latence importante et une réduction de la bande-passante.

9.2 Objectifs et contributions

Ce travail se situe dans les domaines de recherches des infrastructures de Grilles et des bibliothèques de « branch-and-bound » et notre principal objectif est de *définir une infrastructure et une bibliothèque pour la résolution de problèmes d'optimisation combinatoire spécialement adaptées des Grilles de calcul*.

Les Grilles fédèrent un grand nombre de ressources hétérogènes à travers des sites géographiquement distribués en une seule organisation virtuelle. Grâce à ce nombre important de ressources qu'elles fournissent, les Grilles peuvent être utilisées pour la résolution de gros problèmes avec l'algorithme de « branch-and-bound ». Néanmoins, les Grilles introduisent de nouveaux challenges comme le déploiement, l'hétérogénéité, la tolérance aux pannes et le passage à l'échelle.

Dans cette thèse, nous considérons que certains de ces challenges doivent être traités par l'infrastructure de Grilles sous-jacentes, tout particulièrement le déploiement, le passage à l'échelle et l'hétérogénéité ; et que les autres challenges ainsi que le passage à l'échelle doivent être pris en compte par la bibliothèque. Dans l'infrastructure de Grilles proposée cette thèse, nous voulons faciliter l'accès à un grand ensemble de ressources. De plus, avec cet ensemble de ressources nous voulons aussi fournir une bibliothèque qui soit capable de tirer partie de cette importante puissance de calcul. En outre, l'infrastructure et la bibliothèque doivent cacher toutes les difficultés liées aux Grilles.

Les principales contributions de cette thèse sont :

- une analyse des architectures pair-à-pair existantes, et plus particulièrement celles pour les Grilles de calcul ;
- une analyse des bibliothèques de « branch-and-bound » pour les Grilles ;
- une infrastructure pair-à-pair pour les Grilles de calcul, qui permet de combiner les ordinateurs de bureau ainsi que les grappes de calcul ; l'infrastructure est décentralisée, auto-organisée et configurable ;
- cette infrastructure a été déployée de manière opérationnelle comme une Grille de bureau permanente au laboratoire INRIA de Sophia Antipolis, avec laquelle nous avons pu réaliser un record mondial de calcul en résolvant le problème des n-reines avec 25 reines ; et
- une bibliothèque de « branch-and-bound » pour les Grilles qui est basée sur une architecture maître-travailleur hiérarchique et qui fournit de manière transparente un système de communication entre les tâches.

9.3 Plan

Ce document est organisé comme suit :

- Le Chapitre 2 positionne notre travail dans le contexte des Grilles de calcul. Tout d'abord, nous donnons une vue d'ensemble des systèmes existants de Grilles ; ce qui nous permet de faire ressortir ce qui doit être amélioré. Ensuite, nous définissons la notion de systèmes pair-à-pair et nous montrons que ces systèmes peuvent fournir des Grilles plus dynamiques et flexibles. Nous positionnons aussi notre travail avec les systèmes pair-à-pair existants pour les Grilles. En plus, nous présentons les modèles existants de « branch-and-bound » pour les Grilles et nous relions

notre travail aux autres bibliothèques pour les Grilles. Pour finir, nous décrivons le modèle à objets actifs et le « middleware » ProActive sur lesquels notre travail repose.

- En Chapitre 3, nous proposons une infrastructure de Grille de bureau basée sur une architecture pair-à-pair avec laquelle nous avons été les premiers à résoudre le problème des n-reines avec 25 reines : ce calcul a pris 6 mois pour terminer.
- Dans le Chapitre 4, nous décrivons notre bibliothèque pour les Grilles de « branch-and-bound » pour la résolution de problèmes d'optimisation combinatoire. Nous y reportons aussi nos expérimentations avec le problème du « flow-shop » sur la Grille française de taille nationale, *Grid'5000*.
- Le Chapitre 5 décrit une extension de notre infrastructure pair-à-pair pour permettre de combiner les ordinateurs de bureau et les grappes de calcul. Avec cette Grille à grande échelle, nous rapportons nos expérimentations avec les problèmes des n-reines et du « flow-shop ».
- Le Chapitre 6 présente le système de déploiement fournit par ProActive. Nous y présentons aussi quelques améliorations : la localisation des nœuds sur une Grille (ce mécanisme est utilisé par l'implémentation de notre bibliothèque de « branch-and-bound » afin d'optimiser les communications), le déploiement de services non-fonctionnels (comme la tolérance aux pannes ou l'équilibrage de charges), et un système qui permet de décrire les besoins nécessaires d'une application à déployer. Ensuite, nous présentons un mécanisme d'équilibrage de charges reposant sur notre infrastructure pair-à-pair.
- En Chapitre 7, nous donnons une vue d'ensemble de nos travaux courants et des améliorations que nous envisageons dans un futur proche.
- Pour finir, le Chapitre 8 conclut et résume les contributions majeures de cette thèse.

Chapter 10

Résumé

10.1 État de l'art et Positionnement

Dans ce chapitre, nous justifions notre choix d'un système pair-à-pair pour l'acquisition de ressources de calcul sur les Grilles. Pour se faire, nous évaluons les systèmes de Grilles et pair-à-pair existants ainsi que les bibliothèques existantes de « branch-and-bound » pour les Grilles. Nous identifions les problèmes non traités par ces bibliothèques dans le but de justifier notre modèle pour notre bibliothèque de « branch-and-bound » pour les Grilles.

10.1.1 Grilles de calcul

L'un des objectifs fondamentaux des Grilles est de fournir un accès aisé voire transparent à des ressources de calcul hétérogènes et réparties sur des domaines administratifs différents. Ceci est également dénommé « virtualisation des ressources ». Notre objectif est d'offrir à la fois un environnement d'exécution réalisant la virtualisation et une bibliothèque de programmation permettant de résoudre de manière optimale des problèmes d'optimisation combinatoire dans un environnement d'exécution de Grilles.

La plupart des Grilles sont des infrastructures statiques et utilisent des grappes de calcul dédiées. L'inclusion d'ordinateurs supplémentaires à ces organisations virtuelles doit être le plus souvent prévue et budgetée longtemps à l'avance. Il n'est donc pas possible de rajouter dynamiquement des ressources libres d'un site pour temporairement augmenter la puissance de calcul de la Grille. Pratiquement toutes les Grilles déployées actuellement sont des plates-formes expérimentales pour aider les chercheurs à développer la prochaine génération.

Bien que la plupart des projets de Grilles soient définies comme des Grilles, ces infrastructures manquent de dynamicité pour répondre à la définition de Grille donnée dans cette thèse. C'est pourquoi nous proposons d'utiliser une infrastructure pair-à-pair pour inclure dynamiquement des ressources dans une Grille.

10.1.2 Pair-à-Pair

Il existe différentes sortes de réseaux pair-à-pair telles que les architectures maître-travailleur, les réseaux pur pair-à-pair, les réseaux hybrides, ou encore les tables de « hashage » distribué. Dans cette section nous montrons que les réseaux dit pur pair-à-pair sont les plus adpatés pour la Grille. Un réseau pair-à-pair dit « pur » peut être défini comme suit :

*Un système distribué est appelé Pair-à-Pair (P-to-P, P2P, etc.), si les nœuds de ce réseau partagent une partie de leurs ressources (temps de calcul, espace disque, interface réseau, imprimante, etc.). Ces ressources sont nécessaires pour fournir les services et les contenus offerts par le système (e.g. le partage de fichiers ou les espaces de travail collaboratifs). Ils sont accessibles directement par les autres pairs sans passer par des entités intermédiaires. Les participants de ce genre de réseau sont à la fois des fournisseurs de ressources (services et contenus) et des utilisateurs de ces ressources. Ils sont ainsi clients et serveurs. Ces réseaux sont considérés comme « **Pur Pair-à-Pair** » si l'on peut enlever de façon arbitraire un nœud de ce réseau sans qu'il y ait aucune dégradation ou perte du service offert par ce réseau.*

10.1.3 « Branch-and-Bound »

L'algorithme de « branch-and-bound » est une technique pour résoudre les problèmes de recherche, tel que le voyageur de commerce ou bien les problèmes d'ordonnancement. Cette technique permet de trouver la meilleure solution pour une instance de problèmes donnés et de prouver qu'aucune autre solution est meilleure.

Dans ce chapitre, nous avons positionné le travail de cette thèse dans le contexte des Grilles de calcul. Nous avons aussi montré que les communautés des Grilles et du pair-à-pair partagent le même objectif : la coordination et le partage de grands ensembles de ressources distribuées. De plus, nous avons démontré la validité d'utiliser une approche pair-à-pair comme infrastructure de Grille. Nous avons aussi justifié les besoins et les caractéristiques de notre bibliothèque de « branch-and-bound » pour les Grilles.

10.1.4 Conclusion

Cette thèse se concentre sur la couche « Grid middleware » avec l'infrastructure pair-à-pair et sur la couche « Grid programming » avec la bibliothèque de « branch-and-bound ». Cette thèse traite tous les défis des Grilles que nous avons introduits : la distribution, le déploiement, les multiples domaines d'administration, le passage à l'échelle, l'hétérogénéité, la haute-performance, la dynamicité et les modèles de programmation.

Nous affirmons ainsi que les infrastructure de Grilles doivent être dynamiques pour permettre l'inclusion de nouveaux sites. Nous proposons donc une infrastructure pair-à-pair pour le partage de ressources de calcul. Dans cette thèse, nous proposons aussi une bibliothèque de « branch-and-bound » adaptée aux Grilles.

10.2 Pair-à-Pair de bureau

Dans ce chapitre, nous présentons la première partie de la contribution de cette thèse, qui est une infrastructure de Grilles. L'infrastructure proposée est en fait basée sur une architecture pair-à-pair. Le principal objectif de cette infrastructure est de gérer un grand ensemble de ressources et grâce à l'infrastructure, les applications ont un accès facilité à ces ressources.

Dans le Chapitre 2, nous avons identifié les spécificités des Grilles de calcul. En outre, nous avons montré que les Grilles et les réseaux pair-à-pair partagent le même

but et, ainsi, que les architectures pair-à-pair peuvent être utilisées comme infrastructures pour les Grilles.

Notre infrastructure pair-à-pair pour la construction de Grilles est un réseau pair-à-pair non-structuré qui permet de partager des ressources de calcul. Elle est aussi capable de déployer et de terminer des calculs qui prendraient des mois à terminer sur des grappes de calcul.

Dans le but de valider notre approche, nous avons déployé une Grille de bureau permanente gérée par notre infrastructure pair-à-pair dans notre laboratoire. Cette Grille fédère les ordinateurs de bureau sous-exploités de l'INRIA Sophia. Avec cette infrastructure expérimentale, nous sommes les premiers au monde à avoir résolu le problème des n-reines pour 25 reines. Ce calcul a pris six mois et a permis de valider notre infrastructure pour l'exécution de longs calculs.

De plus, nous avons aussi montré les possibilités de l'infrastructure à gérer et à déployer des applications non-Java, tout comme TELEMAC-2D qui est une application de type SPMD-MPI.

10.3 « Branch-and-Bound » : une bibliothèque communicante

Dans le Chapitre 2, nous avons introduit les principes du « branch-and-bound » parallèle et nous avons aussi identifié les besoins auxquels une bibliothèque de « branch-and-bound » pour les Grilles doit répondre. Avec ces besoins, nous présentons maintenant la seconde partie de la contribution de cette thèse, une bibliothèque de « branch-and-bound » parallèle pour les Grilles de calcul. Cette bibliothèque est dénommée *Grid'BnB*.

Grid'BnB est une bibliothèque qui aide les utilisateurs à résoudre des problèmes d'optimisation combinatoire. La bibliothèque cache tous les problèmes liés aux Grilles,au parallèlisme et à la distribution. Elle est basée sur une architecture maître-travailleur hiérarchique avec des communications entre les travailleurs. Les communications sont en fait utilisées pour partager la meilleure borne afin d'explorer moins de parties de l'arbre de recherche et ainsi diminuer le temps d'exécution pour résoudre le problème donné. Comme les Grilles sont des environnements parallèles à grande-échelle, nous proposons d'organiser les travailleurs en groupes de communications. Ces groupes reflètent la topologie de la Grille sous-jacente. Cette propriété a pour but d'optimiser les communications inter-grappes de calcul et de mettre à jour le plus rapidement la borne globale sur tous les travailleurs. *Grid'BnB* propose aussi différents algorithmes de générations d'arbres de recherche pour que les utilisateurs puissent choisir le plus adapté au problème à résoudre. Pour finir, la bibliothèque est tolérante aux pannes.

Les expérimentations ont montré que *Grid'BnB* passe à l'échelle sur une vraie Grille de taille nationale, *Grid'5000*. Ainsi, nous avons pu déployer un problème d'optimisation combinatoire, le « flow-shop », sur une Grille composée de 621 processeurs répartis sur cinq sites.

Finalement, nous pensons que *Grid'BnB* peut être utilisée sans modification pour d'autres algorithmes que celui de « branch-and-bound », comme par exemple l'algorithme de « diviser pour régner » ou encore le skeleton de ferme. De manière plus générale,

Grid'BnB est une bibliothèque pour le calcul parallèle qui vise la résolution de problèmes parallèle.

10.4 Grappes de calcul et Grille de bureau

Ce chapitre rapporte nos expérimentations à grand-échelle avec la bibliothèque *Grid'BnB* et l'infrastructure pair-à-pair, avec lesquelles nous avons construit une Grille composée d'ordinateurs de bureau (*INRIA Sophia P2P Desktop Grid*) et de grappes de calcul (*Grid'5000*).

Nous montrons ainsi que *Grid'BnB*, utilisé avec l'infrastructure pair-à-pair, permet des déploiements à grande échelle sur des Grilles composées d'ordinateurs de bureau et de grappes de calcul.

Nous avons aussi montré que l'infrastructure pair-à-pair peut être utilisée comme infrastructure de Grille pour rassembler les ressources disponibles d'une Grille. Ainsi, une application processeur intensif, comme les n-reines, peut dynamiquement acquérir les ressources disponibles, même si elles ne le sont que pour quelques minutes.

10.5 Déploiement et améliorations

Le premier objectif traité par cette thèse était de fournir une infrastructure dynamique pour les Grilles de calcul. Cette infrastructure est basée sur une architecture pair-à-pair non-structurée qui permet de combiner des ordinateurs de bureau et des grappes de calcul.

Le deuxième objectif était une bibliothèque de « branch-and-bound » parallèle pour les Grilles afin de résoudre des problèmes d'optimisation combinatoire. Cette bibliothèque repose sur le paradigme maître-travailleur avec des communications entre les travailleurs dans le but d'optimiser la résolution des problèmes.

Le lien entre notre bibliothèque et notre infrastructure est le *déploiement*. Dans ce chapitre nous présentons une amélioration du système de déploiement fournit par ProActive afin de permettre la localisation de nœuds sur les Grilles. Ce mécanisme est utilisé par notre bibliothèque de « branch-and-bound » dans le but d'organiser les travailleurs en groupes pour réduire le coût des communications.

Ensuite, nous décrivons une seconde amélioration avec un mécanisme pour la configuration et le déploiement de services non-fonctionnels, nommés *technical services*. Ces services sont la tolérance-aux-pannes et l'équilibrage de charges, par exemple.

Enfin, nous proposons un mécanisme nommé *virtual nodes descriptors* qui permet de décrire les besoins d'une application. Les programmeurs peuvent ainsi spécifier les contraintes de déploiement de leurs applications. Les contraintes sont par exemple les « technical services » à déployer, le nombre minimal de nœuds nécessaires, ou bien encore les architectures machines.

Finalement, nous présentons le « technical service » d'équilibrage de charges qui est basé sur l'infrastructure pair-à-pair.

10.5.1 Localisation de nœuds sur Grilles

ProActive a réussit dans son système de déploiement à complètement enlever les scripts pour la configuration, l'acquisition des ressources et le démarrage du calcul. ProActive fournit, comme clef au problème du déploiement, une abstraction dans le code source de l'infrastructure physique dans un but de flexibilité.

Dans le contexte de cette thèse, nous avons proposé d'organiser les travailleurs en groupes pour optimiser les communications dans la bibliothèque de « branch-and-bound ». Le critère de sélection pour l'appartenance à un groupe est la localisation physique d'un travailleur sur une grappe de calcul. Ainsi, la localisation des nœuds sur une Grille est un problème important pour une implémentation efficace de notre bibliothèque *Grid'BnB*. Nous proposons donc un mécanisme de localisation des nœuds sur une Grille. Ce mécanisme peut aussi être utilisé par toutes sortes d'applications pour optimiser les communications entre les objets actifs qui sont distribués sur une Grille, ou bien à faire de la localisation de données.

Nous avons ainsi étendu le système de déploiement fournit par ProActive avec un mécanisme de marqueurs, permettant aux applications de déterminer si deux nœuds ont été déployés par le même graphe de déploiement.

De plus, ce mécanisme de marqueurs est utilisé par l'implémentation de notre bibliothèque, *Grid'BnB*, pour optimiser les communications entre les grappes de calcul.

10.5.2 Services Techniques pour les Grilles

Cette dernière décennie est apparue une identification des aspects *non-fonctionnels* pour le développement de logiciels flexibles et adaptatifs. Nous proposons d'étendre le système de déploiement de ProActive avec un mécanisme qui permet de définir dynamiquement des services non-fonctionnels, *technical services*, pour la Grille et, prenant en compte aussi bien les besoins de l'application que les contraintes de l'infrastructure.

Les « technical services » permettent aux déployeurs d'appliquer et de configurer les besoins non-fonctionnels des applications. Cependant, les déployeurs doivent connaître à l'avance les services non-fonctionnels dont les applications ont besoin. Afin de résoudre ce manque de connexion entre les programmeurs et les déployeurs, nous proposons dans le chapitre suivant une solution.

10.5.3 Descripteur de nœud virtuels

Dans ce chapitre nous proposons un mécanisme aux programmeurs afin qu'ils puissent spécifier les besoins en services non-fonctionnels de leurs applications, et ainsi permettre aux déployeurs de configurer ces services en fonction de l'infrastructure.

Ce mécanisme est descripteur au niveau de l'application où la programmeuse spécifie les besoins environnementaux, comme les « technical services ». Ainsi le déployeur peut appliquer la configuration optimale afin de remplir ces contraintes.

10.5.4 Équilibrage de charges sur une infrastructure pair-à-pair

Nous avons, avec la collaboration du Dr. Javier Bustos, développé un « technical service » pour l'équilibrage de charges. Ce service est spécialement fait pour tirer partie

de l'infrastructure pair-à-pair et permet de répartir dynamiquement les activités d'une application sur une Grille de bureau.

10.6 Travaux en cours et perspectives

Ce chapitre introduit les travaux en cours et futurs basés sur les contributions de cette thèse. Nous travaillons actuellement sur un ordonnanceur de tâches qui permet aux personnes de l'INRIA Sophia d'utiliser en toute liberté la Grille *INRIA Sophia P2P Desktop Grid*. De plus, nous travaillons à améliorer le mécanisme de découverte des ressources de l'infrastructure pair-à-pair. Pour le moment, nous envisageons deux manières différentes pour atteindre cet objectif : d'une part, en modifiant le protocole de messages ; d'autre part en introduisant un marqueur de pairs.

La bibliothèque de « branch-and-bound » est aussi sujette à des amélioration en cours, notamment avec l'amélioration de notre implémentation du problème du « flow-shop ». Nous envisageons aussi de faire plus d'expérimentations à grandes échelles en utilisant des grappes de calcul localisées aux Pays-Bas et au Japon.

Pour finir, nous travaillons aussi sur le déploiement d'applications sur les Grilles. Nous sommes en train de fournir un mécanisme de contrats entre les développeurs, les administrateurs d'infrastructures et les utilisateurs d'applications.

Chapter 11

Conclusion

Les Grilles de calcul rassemblent et fournissent un grand nombre de ressources de calcul. Cette puissance de calcul attire de plus en plus d'utilisateurs vers les Grilles. Grâce à ce grand nombre de ressources, les Grilles sont bien adaptées à la résolution de problèmes d'optimisation combinatoire de très grande taille. Cependant, les Grilles introduisent de nouveaux défis qui mettent en avant le besoin de nouvelles infrastructures et de nouveaux outils afin de masquer les problèmes liés aux Grilles.

Dans cette thèse, nous avons proposé une bibliothèque basée sur une implémentation parallèle de l'algorithme de « branch-and-bound » pour la résolution de problèmes d'optimisation combinatoire. Cette bibliothèque répond précisément aux spécificités liées aux Grilles, en particulier l'hétérogénéité, le déploiement, les communications, la tolérance aux pannes et le passage à l'échelle. Elle repose sur une infrastructure de type pair-à-pair qui permet l'acquisition dynamique de ressources, aussi bien des ordinateurs de bureau que de grappes de calcul.

Les contributions de cette thèse peuvent être énumérées comme suit :

- Grâce à une analyse de l'existant, aussi bien des infrastructures de Grilles que des bibliothèques de « branch-and-bound », nous avons pu identifier les besoins nécessaires que nos contributions doivent satisfaire :
  - l'*infrastructure* doit permettre de créer des Grilles qui sont composées aussi bien de ordinateurs de bureau que de grappes de calcul, mais également permettre le déploiement d'applications communicantes et être capable de supporter des calculs pouvant mettre des mois à terminer sur des grappes de calcul ;
  - la *bibliothèque* doit permettre aux différents processus de communiquer entre-eux et l'implémentation de différents algorithmes de parcours d'arbres. Elle doit aussi avoir un mécanisme performant pour le partage de la meilleure borne courante et être tolérante aux pannes.
- Une *infrastructure de Grille de bureau* basée sur une architecture pair-à-pair non-structurée et complètement intégrée au « middleware » de Grille ProActive. Cette infrastructure nous a permis de :
  - déployer une Grille de bureau permanente au laboratoire INRIA Sophia, nommée *INRIA Sophia P2P Desktop Grid*, rassemblant jusqu'à 260 ordinateurs de bureau ; et

  - d'établir un record du monde en résolvant le problème des n-reines pour 25 reines, ce calcul a pris 6 mois pour terminer.
- *Grid'BnB*, une bibliothèque de « branch-and-bound » parallèle pour la Grille, qui s'efforce de cacher aux utilisateurs toutes les difficultés liées à la Grille. Ses principales caractéristiques sont :
  - une architecture maître-travailleur hiérarchique avec des communications entre les travailleurs ;
  - la division dynamique des tâches ;
  - l'organisation des travailleurs en groupe afin d'optimiser les communications inter-grappes de calcul ; et
  - la tolérance aux pannes.
- Expérimentations à grande-échelle en combinant notre Grille de bureau, *INRIA Sophia P2P Desktop Grid*, aux grappes de calcul de la Grille française, *Grid'5000*. Avec cet environment nous avons déployé les n-reines sur 1007 processeurs.
- Amélioration des mécanismes pour le déploiement d'applications sur les Grilles :
  - proposition d'un système pour la localisation de nœuds qui permet de déterminer si deux nœuds sont déployés sur la même grappe de calcul ;
  - une bibliothèque pour le déploiement et la configuration de services non-fonctionnels, comme la tolérance aux pannes ou le l'équilibrage de charges ;
  - un mécanisme pour spécifier les besoins des applications à déployer sur les Grilles ; et
  - un mécanisme d'équilibrage de charges utilisant l'infrastructure pair-à-pair.

Une grande partie du travail de cette thèse sur les Grilles de bureau est maintenant en cours d'industrialisation afin de fournir des Grilles de production, avec particulièrement le développement d'un ordonnanceur de « jobs » et d'outils pour l'administration de l'infrastructure.

Quelques parties de notre travail sur le déploiement d'applications sur la Grille sont considérées comme base de travail pour la normalisation de la Grille au niveau européen, et plus particulièrement avec les projets CoreGrid et GridComp.

Bibliography

[020 99] OMG TC DOCUMENT ORBOS/99 02-05. "Corba components specifiaction". Technical report, Object Management Group, 1999. Technical Report.

[AID 03] K. AIDA, W. NATSUME, and Y. FUTAKATA. Distributed computing with hierarchical master-worker paradigm for parallel branch and bound algorithm. *Cluster Computing and the Grid, 2003. Proceedings. CCGrid 2003. 3rd IEEE/ACM International Symposium on*, pages 156–163, 2003.

[AID 05] K. AIDA, and T. OSUMI. A Case Study in Running a Parallel Branch and Bound Application on the Grid. *Proc. IEEE/IPSJ The 2005 Symposium on Applications & the Internet (SAINT2005)*, pages 164–173, 2005.

[ALB 02] E. ALBA, F. ALMEIDA, M. BLESA, J. CABEZA, C. COTTA, M. DIAZ, I. DORTA, J. GABARRO, C. LEON, J. LUNA, and al. MALLBA: A library of skeletons for combinatorial optimisation. *Proceedings of the International Euro-Par Conference, Paderborn, Germany, LNCS*, 2400:927–932, 2002.

[ALL 05] GABRIELLE ALLEN, KELLY DAVIS, TOM GOODALE, ANDREI HUTANU, HARTMUT KAISER, THILO KIELMANN, ANDRE MERZKY, ROB V. VAN NIEUWPOORT, ALEXANDER REINEFELD, FLORIAN SCHINTKE, THORSTEN SCHOTT, ED SEIDEL, and BRYGG ULLMER. "The Grid Application Toolkit: Towards Generic and Easy Application Programming Interfaces for the Grid". In *Proceedings of the IEEE*, volume 93, pages 534–550, March 2005.

[ALV 98] L. ALVISI, and K. MARZULLO. Message logging: Pessimistic, optimistic, causal, and optimal. *Software Engineering*, 24(2):149–159, 1998.

[AND 02] DAVID P. ANDERSON, JEFF COBB, ERIC KORPELA, MATT LEBOFSKY, and DAN WERTHIMER. Seti@home: an experiment in public-resource computing. *Commun. ACM*, 45(11), 2002.

[AND 04] DAVID P. ANDERSON. "Boinc: A system for public-resource computing and storage.". In *GRID*, pages 4–10, 2004.

[AND 05a] NAZARENO ANDRADE, LAURO COSTA, GUILHERME GERMOGLIO, and WALFREDO CIRNE. "Peer-to-peer grid computing with the ourgrid community". In *Proceedings of the SBRC 2005 - IV Salão de Ferramentas*, May 2005.

[AND 05b] ALAIN ANDRIEUX, KARL CZAJKOWSKI, ASIT DAN, KATE KEAHEY, HEIKO LUDWIG, TOSHIYUKI NAKATA, JIM PRUYNE, JOHN ROFRANO, STEVE TUECKE, and MING XU. Web services agreement specification (ws-agreement), 2005. Draft Version 2005/09. http://forge.gridforum.org/projects/graap-wg.

[ANJ 05] A. ANJOMSHOAA, F. BRISARD, M. DRESCHER, D. FELLOWS, A. LY, A. S.MCGOUGH, D. PULSIPHER, and A. SAVVA. Job submission description language (jsdl) specification, version 1.0. http://forge.gridforum.org/projects/jsdl-wg, 2005.

[APP 91] D. APPLEGATE, and W. COOK. A computational study of the job-shop scheduling problem. *ORSA Journal on Computing*, 3(2):149–156, 1991.

[ARM 99] ROB ARMSTRONG, DENNIS GANNON, AL GEIST, KATARZYNA KEAHEY, SCOTT KOHN, LOIS MCINNES, STEVE PARKER, and BRENT SMOLINSKI. "Toward a Common Component Architecture for High-Performance Scientific Computing". In *Proceedings of the 1999 Conference on High Performance Distributed Computing*, 1999. http://www-unix.mcs.anl.gov/ curfman/cca/web/cca_paper.html.

[ATA 99] M. ATALLAH. "*Algorithms and theory of computation handbook*". CRC Press, 1999.

[ATT 05] ISABELLE ATTALI, DENIS CAROMEL, and ARNAUD CONTES. "Deployment-based security for grid applications". In *The International Conference on Computational Science (ICCS 2005), Atlanta, USA, May 22-25*, LNCS. Springer Verlag, 2005.

[AUT 95] G. AUTHIÉ, JEAN-MARIE GARCIA, A. FERREIRA, J.L. ROCH, G. VILLARD, J. ROMAN, C. ROUCAIROL, and B. VIROT, teurs. "*Algorithmique parallèle et applications irrégulières*". Hermes, Paris (F), 1995.

[BAD 02] LAURENT BADUEL, FRANÇOISE BAUDE, and DENIS CAROMEL. "Efficient, Flexible, and Typed Group Communications in Java". In *Joint ACM Java Grande - ISCOPE 2002 Conference*, pages 28–36, Seattle, 2002. ACM Press. ISBN 1-58113-559-8.

[BAD 05] LAURENT BADUEL, FRANÇOISE BAUDE, and DENIS CAROMEL. "Object-Oriented SPMD". In *Proceedings of Cluster Computing and Grid*, Cardiff, United Kingdom, May 2005.

[BAD 06] LAURENT BADUEL, FRANÇOISE BAUDE, DENIS CAROMEL, ARNAUD CONTES, FABRICE HUET, MATTHIEU MOREL, and ROMAIN QUILICI. "*Grid Computing: Software Environments and Tools*", chapter Programming, Deploying, Composing, for the Grid. Springer-Verlag, January 2006.

[BAH 06] JM BAHI, R. COUTURIER, and P. VUILLEMIN. JaceP2P: an Environment for Asynchronous Computations on Peer-to-Peer Networks. *Cluster Computing, 2006 IEEE International Conference on*, pages 1–10, 2006.

[BAL 00] HENRI BAL, RAOUL BHOEDJANG, RUTGER HOFMAN, CERIEL JACOBS, THILO KIELMANN, JASON MAASSEN, ROB VAN NIEUWPOORT, JOHN ROMEIN, LUC RENAMBOT, TIM RüHL, RONALD VELDEMA, KEES VERSTOEP, ALINE BAGGIO, GERCO BALLINTIJN, IHOR KUZ, GUILLAUME PIERRE, MAARTEN VAN STEEN, ANDY TANENBAUM, GERBEN DOORNBOS, DESMOND GERMANS, HANS SPOELDER, EVERT-JAN BAERENDS, STAN VAN GISBERGEN, HAMIDEH AFSERMANESH, DICK VAN ALBADA, ADAM BELLOUM, DAVID DUBBELDAM, ZEGER HENDRIKSE, BOB HERTZBERGER, ALFONS HOEKSTRA, KAMIL ISKRA, DRONA KANDHAI, DENNIS KOELMA, FRANK VAN DER LINDEN, BENNO OVEREINDER, PETER SLOOT, PIERO

SPINNATO, DICK EPEMA, ARJAN VAN GEMUND, PIETER JONKER, ANDREI RADULESCU, CEES VAN REEUWIJK, HENK SIPS, PETER KNIJNENBURG, MICHAEL LEW, FLORIS SLUITER, LEX WOLTERS, HANS BLOM, CEES DE LAAT, and AAD VAN DER STEEN. The distributed asci supercomputer project. *SIGOPS Oper. Syst. Rev.*, 34(4):76–96, 2000.

[BAU 00] FRANÇOISE BAUDE, DENIS CAROMEL, FABRICE HUET, and JULIEN VAYSSIÈRE. "Communicating Mobile Active Objects in Java". In *Proceedings of HPCN Europe 2000*, volume 1823 de *LNCS*, pages 633–643. Springer, May 2000.

[BAU 02] FRANÇOISE BAUDE, DENIS CAROMEL, LIONEL MESTRE, FABRICE HUET, and JULIEN VAYSSIÈRE. "Interactive and descriptor-based deployment of object-oriented grid applications". In *Proceedings of the 11th IEEE International Symposium on High Performance Distributed Computing*, pages 93–102, Edinburgh, Scotland, July 2002. IEEE Computer Society.

[BAU 03] FRANCOISE BAUDE, DENIS CAROMEL, and MATTHIEU MOREL. "From distributed objects to hierarchical grid components". In *International Symposium on Distributed Objects and Applications (DOA), Catania, Sicily, Italy, 3-7 November*, volume 2888, pages 1226–1242, Springer Verlag, 2003. Lecture Notes in Computer Science, LNCS.

[BAU 05] FRANÇOISE BAUDE, DENIS CAROMEL, CHRISTIAN DELBÉ, and LUDOVIC HENRIO. "A Hybrid Message Logging-CIC Protocol for Constrained Checkpointability". In *Proceedings of EuroPar2005*, LNCS, pages 644–653, Lisbon, Portugal, August-September 2005. Springer.

[BAU 06] FRANÇOISE BAUDE, DENIS CAROMEL, MARIO LEYTON, and ROMAIN QUILICI. "Grid File Transfer during Deployment, Execution, and Retrieval". In *Proceeedings of On the Move to Meaningful Internet Systems 2006*, lncs, Montpellier, France, November 2006. Springer.

[BAU 07] FRANÇOISE BAUDE, DENIS CAROMEL, ALEXANDRE DI COSTANZO, CHRISTIAN DELBÉ, and MARIO LEYTON. "Towards deployments contracts in large scale clusters & desktop grids". In *Workshop on Large-Scale and Volatile Desktop Grids (PCGrid 2007)*. IEEE Computer Society, March 2007. Invited paper.

[BEL 97] L. BELLISSARD, M.-C. PELLEGRINI, and M. RIVEILL. Integration and Distribution of Legacy Software with Olan. *Object-Based Parallel and Distributed Computation, France-Japan Workshop*, Toulouse, October 15-17, 1997.

[BEN 07] AHCENE BENDJOUDI, NOUREDINE MELAB, and EL-GHAZALI TALBI. "A parallel p2p branch-and-bound algorithm for computational grids". In *CCGRID*, pages 749–754, 2007.

[BLO 05] STEVENS LE BLOND, ANA-MARIA OPRESCU, and CHEN ZHANG. Early Application Experience with the Grid Application Toolkit. Global Grid Forum Information Document, from the Workshop on Grid Applications: from Early Adopters to Mainstream Users, June 2005.

[BRÜ 99] A. BRÜNGGER, A. MARZETTA, K. FUKUDA, and J. NIEVERGELT. The parallel search bench ZRAM and its applications. *Annals of Operations Research*, 90:45–63, 1999.

[BRU 02] ERIC BRUNETON, THIERRY COUPAYE, and JEAN-BERNARD STEFANI. "Recursive and dynamic software composition with sharing". In *Proceedings of the 7th ECOOP International Workshop on Component-Oriented Programming (WCOP'02)*, 2002.

[BUS 05] JAVIER BUSTOS-JIMENEZ, DENIS CAROMEL, ALEXANDRE DI COSTANZO, and MARIO LEYTON ND JOSE M. PIQUER. "Balancing active objects on a peer to peer infrastructure". In *Proceedings of the XXV International Conference of the Chilean Computer Science Society (SCCC 2005)*, Valdivia, Chile, NOV 2005. IEEE.

[BUS 06] JAVIER BUSTOS-JIMENEZ, DENIS CAROMEL, MARIO LEYTON, and JOSE M. PIQUER. "Coupling contracts for deployment on alien grids". In *Proceedings of the International Euro-Par Workshops*, lncs, Dresden, Germany, August 2006. springer.

[C. 05] BRIGNOLLES C., ROYER L., CALVET F., VIALA Y., and DE SAINT SEINE J. "Modélisation en deux dimensions de la dynamique des crues de la durance". In *Proc. Systèmes d'information et risques naturels*, France, 2005.

[CAH 03] SÉBASTIEN CAHON, EL-GHAZALI TALBI, and NORDINE MELAB. "Paradiseo: A framework for parallel and distributed metaheuristics". In *IPDPS '03: Proceedings of the 17th International Symposium on Parallel and Distributed Processing*, page 144.1, Washington, DC, USA, 2003. IEEE Computer Society.

[CAP 05] FRANCK CAPPELLO, EDDY CARON, MICHEL DAYDE, FREDERIC DESPREZ, YVON JEGOU, PASCALE PRIMET, EMMANUEL JEANNOT, STEPHANE LANTERI, JULIEN LEDUC, NOUREDINE MELAB, GUILLAUME MORNET, RAYMOND NAMYST, BENJAMIN QUETIER, and OLIVIER RICHARD. "Grid'5000: a large scale and highly reconfigurable grid experimental testbed". In *6th IEEE/ACM International Workshop on Grid Computing*, 2005.

[CAR] DENIS CAROMEL, CHRISTIAN DELBÉ, LUDOVIC HENRIO, and ROMAIN QUILICI. Dispositifs et procé dé s asynchrones et automatiques de transmission de ré sultats entre objets communicants (asynchronous and automatic continuations of results between communicating objects). French patent FR03 13876 - US Patent Pending.

[CAR 93] D. CAROMEL. Towards a Method of Object-Oriented Concurrent Programming. *Communications of the ACM*, 36(9):90–102, September 1993.

[CAR 04] DENIS CAROMEL, LUDOVIC HENRIO, and BERNARD PAUL SERPETTE. "Asynchronous and deterministic objects". In *POPL '04: Proceedings of the 31st ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 123–134, New York, NY, USA, 2004. ACM Press.

[CAR 05a] D. CAROMEL, and G. CHAZARAIN. "Robust exception handling in an asynchronous environment". In A. ROMANOVSKY, C. DONY, JL. KNUDSEN, and A. TRIPATHI, teurs, *Proceedings of ECOOP 2005 Workshop on Exception Handling in Object Oriented Systems*. Tech. Report No 05-050, Dept. of Computer Science, LIRMM, Montpellier-II Univ. July. France, 2005.

[CAR 05b] DENIS CAROMEL, and LUDOVIC HENRIO. "*A Theory of Distributed Objects*". Springer-Verlag, 2005.

[CAR 06a] DENIS CAROMEL, VINCENT CAVÉ, ALEXANDRE DI COSTANZO, CÉLINE BRIGNOLLES, BRUNO GRAWITZ, and YANN VIALA. "Executing hydrodynamic simulation on desktop grid with objectweb proactive". In *HIC2006: Proceedings of the 7th International Conference on HydroInformatics*, Nice, France, September 2006.

[CAR 06b] DENIS CAROMEL, CHRISTIAN DELBE, and ALEXANDRE DI COSTANZO. "Peer-to-peer and fault-tolerance: Towards deployment based technical services". Unpublished, January 2006.

[CAR 06c] DENIS CAROMEL, ALEXANDRE DI COSTANZO, CHRISTIAN DELBÉ, and MATTHIEU MOREL. "Dynamically-fulfilled application constraints through technical services - towards flexible component deployments". In *Proceedings of HPC-GECO/CompFrame 2006, HPC Grid programming Environments and COmponents - Component and Framework Technology in High-Performance and Scientific Computing*, Paris, France, June 2006. IEEE.

[CAR 07a] D. CAROMEL, A. DI COSTANZO, and C. MATHIEU. Peer-to-Peer for Computational Grids: Mixing Clusters and Desktop Machines. *Parallel Computing*, 33(4-5):275–288, 2007.

[CAR 07b] DENIS CAROMEL, GUILLAUME CHAZARAIN, and LUDOVIC HENRIO. "Garbage collecting the grid: a complete dgc for activities". In *Proceedings of the 8th ACM/IFIP/USENIX International Middleware Conference*, Newport Beach, CA, November 2007.

[CAR 07c] DENIS CAROMEL, ALEXANDRE DI COSTANZO, LAURENT BADUEL, and SATOSHI MATSUOKA. Grid'BnB: A Parallel Branch & Bound Framework for Grids. *HiPC'07*, 2007. To appear.

[CHA 85] K. M. CHANDY, and L. LAMPORT. "Distributed snapshots: Determining global states of distributed systems". In *ACM Transactions on Computer Systems*, pages 63–75, 1985.

[CHA 97] STEVE CHAPIN, DIMITRIOS KATRAMATOS, JOHN KARPOVICH, and ANDREW GRIMSHAW. Resource management in legion, 1997. Legion Winter Workshop.

[CHA 03] YATIN CHAWATHE, SYLVIA RATNASAMY, LEE BRESLAU, NICK LANHAM, and SCOTT SHENKER. "Making gnutella-like p2p systems scalable". In *SIGCOMM '03: Proceedings of the 2003 conference on Applications, technologies, architectures, and protocols for computer communications*, pages 407–418, New York, NY, USA, 2003. ACM Press.

[CHI 03] A. CHIEN, B. CALDER, S. ELBERT, and K. BHATIA. Entropia: architecture and performance of an enterprise desktop grid system. *Journal of Parallel and Distributed Computing*, 63(5):597–610, 2003.

[CLA 99] J. CLAUSEN, and M. PERREGAARD. On the best search strategy in parallel branch-and-bound: Best-First Search versus Lazy Depth-First Search. *Annals of Operations Research*, 90:1–17, 1999.

[CLA 00] I. CLARKE, O. SANDBERG, B. WILEY, and T.W. HONG. Freenet: A distributed anonymous information storage and retrieval system. *Workshop on Design Issues in Anonymity and Unobservability*, 320, 2000.

[COH 03] B. COHEN. "ncentives to build robustness in bittorrent". documentation from BitTorrent: http://www.bittorrent.com, 2003.

[COL 91] MURRAY COLE. "*Algorithmic skeletons: structured management of parallel computation*". MIT Press, Cambridge, MA, USA, 1991.

[CON 93] CONDOR, 1993. http://www.cs.wisc.edu/condor/.

[CRA 06] TEODOR GABRIEL CRAINIC, BERTRAND LE CUN, and CATHERINE ROUCAIROL. "*Parallel Combinatorial Optimization*", chapter Parallel Branch-and-Bound Algorithms, pages pp 1–28. Wiley, 2006.

[CUN 94] B. LE CUN. Bob++ framework: User's guide and api, 1994. Available at http://www.prism.uvsq.fr/blec/Research/BOBO.

[CZA 98] KARL CZAJKOWSKI, IAN T. FOSTER, NICHOLAS T. KARONIS, CARL KESSELMAN, STUART MARTIN, WARREN SMITH, and STEVEN TUECKE. "A resource management architecture for metacomputing systems". In *IPPS/SPDP '98: Proceedings of the Workshop on Job Scheduling Strategies for Parallel Processing*, volume 1459 de *Lecture Notes in Computer Science*, pages 62–82, London, UK, 1998. Springer-Verlag.

[DAK 65] RJ DAKIN. A tree-search algorithm for mixed integer programming problems. *The Computer Journal*, 8(3):250, 1965.

[DAN 05] MARCO DANELUTTO. "Qos in parallel programming through application managers". In *PDP '05: Proceedings of the 13th Euromicro Conference on Parallel, Distributed and Network-Based Processing (PDP'05)*, pages 282–289, Washington, DC, USA, 2005. IEEE Computer Society.

[Dav 95] DAVID ABRAMSON AND ROK SOSIC AND J. GIDDY AND B. HALL. "Nimrod: a tool for performing parametrised simulations using distributed workstations". In *Proceedings of the Fourth IEEE International Symposium on High Performance Distributed Computing*, 1995.

[DEL 07] CHRISTIAN DELBÉ. "*Tolérance aux pannes pour objets actifs asynchrones - protocole, modèle et expérimentations*". PhD thesis, Ecole Doctorale STIC, 2007.

[dis 97] DISTRIBUTED.NET. http://www.distributed.net, 1997.

[Dyn] "*Dynamic Query Protocol*". http://www.the-gdf.org/index.php?title=Dynamic_Querying.

[ECK 00] J. ECKSTEIN, C.A. PHILLIPS, and W.E. HART. PICO: An Object-Oriented Framework for Parallel Branch and Bound. *RUTCOR Research Report*, pages 40–2000, 2000.

[EGE 04] EGEE. http://public.eu-egee.org/, 2004.

[ELN 96] M. ELNOZAHY, L. ALVISI, Y.M. WANG, and D.B. JOHNSON. "A survey of rollback-recovery protocols in message passing systems". Technical report CMU-CS-96-181, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, USA, oct 1996.

[ETS 05a] ETSI PLUGTESTS. 2nd Grid Plugtests. http://www.etsi.org/plugtests/History/2005GRID.htm, 2005.

[ETS 05b] ETSI PLUGTESTS. Grid Plugtest: Interoperability on the Grid. Grid Today online, January 2005.

[ETS 06] ETSI PLUGTESTS. 3rd Grid Plugtests. http://www.etsi.org/plugtests/History/2006GRID.htm, 2006.

[FIL 07] IMEN FILALI. Peer-to-peer resource discovery in a grid environment. Master's thesis, Nice University France, June 2007.

[FOR 04] GLOBAL GRID FORUM, 2004. http://www.ggf.org/.

[FOS 98] I. FOSTER, and C. KESSELMAN. "*The grid: blueprint for a new computing infrastructure*". Morgan Kaufmann Publishers Inc. San Francisco, CA, USA, 1998.

[FOS 00] IAN FOSTER, ALAIN ROY, and VOLKER SANDER. "A quality of service architecture that combines resource reservation and application adaptation". In *Proceedings of the Eight International Workshop on Quality of Service (IWQOS 2000)*, pages 181–188, June 2000.

[FOS 01] I. FOSTER, C. KESSELMAN, and S. TUECKE. The Anatomy of the Grid: Enabling Scalable Virtual Organizations. *International Journal of High Performance Computing Applications*, 15(3):200, 2001.

[FOS 02] IAN FOSTER, CARL KESSELMAN, JEFFREY M. NICK, and STEVEN TUECKE. Grid services for distributed system integration. *Computer*, 35(6):37–46, 2002.

[FOS 03] I. FOSTER, and A. IAMNITCHI. On Death, Taxes, and the Convergence of Peer-to-Peer and Grid Computing. *2nd International Workshop on Peer-to-Peer Systems (IPTPS03)*, pages 118–128, 2003.

[FOS 05] IAN FOSTER. "Globus Toolkit Version 4: Software for Service-Oriented Systems". In *IFIP International Conference on Network and Parallel Computing*, number 3779 in LNCS. Springer-Verlag, 2005.

[FOX 03] GEOFFREY FOX, DENNIS GANNON, SUNG-HOON KO, SANGMI LEE, SHRIDEEP PALLICKARA, MARLON PIERCE, XIAOHONG QIU, XI RAO, AHMET UYAR, and WENJUN WU MINJUN WANG. "*Grid Computing: Making the Global Infrastructure a Reality*", chapter 18. Peer-to-Peer Grids, pages 471–490. Wiley, March 2003.

[FRE 01] JAMES FREY, TODD TANNENBAUM, MIRON LIVNY, IAN FOSTER, and STEVEN TUECKE. "Condor-g: A computation management agent for multi-institutional grids". In *HPDC '01: Proceedings of the 10th IEEE International Symposium on High Performance Distributed Computing*, Washington, DC, USA, 2001. IEEE Computer Society.

[FRI 98] M. FRIGO, C.E. LEISERSON, and K.H. RANDALL. The implementation of the Cilk-5 multithreaded language. *Proceedings of the ACM SIGPLAN 1998 conference on Programming language design and implementation*, pages 212–223, 1998.

[FRØ 98] SVEND FRØLUND, and JARI KOISTINEN. Quality-of-service specification in distributed object systems. *Distributed Systems Engineering*, 5(4):179–202, 1998.

[GAN 02] D. GANNON, R. BRAMLEY, G. FOX, S. SMALLEN, A. ROSSI, R. ANANTHAKRISHNAN, F. BERTRAND, K. CHIU, M. FARRELLEE, M. GOVINDARAJU, and al. Programming the Grid: Distributed Software Components, P2P and

Grid Web Services for Scientific Applications. *Cluster Computing*, 5(3):325–336, 2002.

[GAR 76] M.R. GAREY, D.S. JOHNSON, and R. SETHI. The complexity of flowshop and jobshop scheduling. *Mathematics of Operations Research*, 1(2):117–129, 1976.

[GEI 94] A. GEIST. "*Pvm: Parallel Virtual Machine-A Users' Guide and Tutorial for Networked Parallel Computing*". MIT Press, 1994.

[GEN 94] B. GENDRON, and T.G. CRAINIC. Parallel Branch-And-Bound Algorithms: Survey and Synthesis. *Operations Research*, 42(6):1042–1066, 1994.

[GGF04] Global Grid Forum, Available: http://forge.gridforum.org/projects/saga-rg/. "*Simple API for Grid Applications Research Group*", 2004.

[GIL 01] VINCENT NÉRI GILLES FEDAK CÉCILE GERMAIN, and FRANCK CAPPELLO. "Xtremweb : A generic global computing system". In IEEE PRESS, teur, *CCGRID2001, workshop on Global Computing on Personal Devices*, May 2001.

[GNU 00] GNUTELLA, 2000. http://www.gnutella.com.

[GOO 05] TOM GOODALE, SHANTENU JHA, HARTMUT KAISER, THILO KIELMANN, PASCAL KLEIJER, GREGOR VON LASZEWSKI, CRAIG LEE, ANDRE MERZKY, HRABRI RAJIC, and JOHN SHALF. SAGA: A Simple API for Grid Applications, High-Level Application Programming on the Grid. http://www.cs.vu.nl/ kielmann/papers/saga-sc05.pdf, 2005. submitted for publication.

[GOU 00a] J.P. GOUX, S. KULKARNI, J. LINDEROTH, and M. YODER. An Enabling Framework for Master-Worker Applications on the Computational Grid. *Proc. 9th IEEE Symp. on High Performance Distributed Computing*, 2000.

[GOU 00b] JP GOUX, J. LINDEROTH, and M. YODER. Metacomputing and the master-worker paradigm. *Preprint ANL/MCS-P792-0200, Mathematics and Computer Science Division, Argonne National Laboratory*, 2000.

[GRE 06] D. GREENWOOD, G. VITAGLIONE, L. KELLER, and M. CALISTI. "Service Level Agreement Management with Adaptive Coordination". In *Int. conference on Networking and Services (ICNS'06)*, 2006.

[GRI a] OPEN SCIENCE GRID. http://www.opensciencegrid.org.

[GRI b] GRIDCOMP. Grid; gcm; grid component model part 1: Gcm interoperability deployment (dts/grid-0004-1). http://webapp.etsi.org/WorkProgram/Report_WorkItem.asp?WKI_ID=26362.

[Gri c] GRIDLAB. the GridSphere portal. http://www.gridsphere.org.

[GRI 97] A.S. GRIMSHAW, and W.A. WULF. The Legion vision of a worldwide virtual computer. *Communications of the ACM*, 40(1):39–45, 1997.

[GRO 96] W. GROPP, E.L. LUSK, N. DOSS, and A. SKJELLUM. A High-Performance, Portable Implementation of the MPI Message Passing Interface Standard. *Parallel Computing*, 22(6):789–828, 1996.

[HAL 03] EMIR HALEPOVIC, and RALPH DETERS. "Jxta performance study". In *PACRIM. 2003 IEEE Pacific Rim Conference on Communications, Computers and signal*, volume 1, pages .pp. 149– 154, August 2003.

[HEY 00] ELISA HEYMANN, MIQUEL A. SENAR, EMILIO LUQUE, and MIRON LIVNY. "Adaptive scheduling for master-worker applications on the computational grid". In *GRID '00: Proceedings of the First IEEE/ACM International Workshop on Grid Computing*, pages 214–227, London, UK, 2000. Springer-Verlag.

[IAM 00] A. IAMNITCHI, and I. FOSTER. A Problem-Specific Fault-Tolerance Mechanism for Asynchronous, Distributed Systems. *29th International Conference on Parallel Processing (ICPP), Toronto, Canada, August*, pages 21–24, 2000.

[JIN 04] HAI JIN. "ChinaGrid: Making Grid Computing a Reality". In *7th International Conference on Asian Digital Libraries, ICADL 2004, Shanghai, China, December 13-17, 2004*, number 3334 in LNCS. Springer, 2004.

[JM. 03] HERVOUET J-M. "*Hydrodynamique des écoulements à surface libre*". Presse de l'école nationale des Ponts et chaussées, 2003.

[KAR 03] N.T. KARONIS, B. TOONEN, and I. FOSTER. MPICH-G2: A Grid-enabled implementation of the Message Passing Interface. *Journal of Parallel and Distributed Computing*, 63(5):551–563, 2003.

[KAZ 00] KAZAA, 2000. http://www.kazaa.com.

[KIE 02] JÖRG KIENZLE, and RACHID GUERRAOUI. "Aop: Does it make sense? the case of concurrency and failures". In *ECOOP '02: Proceedings of the 16th European Conference on Object-Oriented Programming*, pages 37–61, London, UK, 2002. Springer-Verlag.

[LAC 05] SÉBASTIEN LACOUR, CHRISTIAN PÉREZ, and THIERRY PRIOL. "Generic application description model: Toward automatic deployment of applications on computational grids". In *6th IEEE/ACM International Workshop on Grid Computing (Grid2005), Seattle, WA, USA*. Springer-Verlag, november 2005.

[LAG 78] BJ LAGEWEG, JK LENSTRA, and A.H.G.R. KAN. A General Bounding Scheme for the Permutation Flow-Shop Problem. *Operations Research*, 26(1):53–67, 1978.

[LAI 84] T.H. LAI, and S. SAHNI. Anomalies in parallel branch-and-bound algorithms. *Communications of the ACM*, 27(6):594–602, 1984.

[LAN 60] AH LAND, and AG DOIG. An Automatic Method of Solving Discrete Programming Problems. *Econometrica*, 28(3):497–520, 1960.

[LAS 01] GREGOR VON LASZEWSKI, IAN FOSTER, JAREK GAWOR, and PETER LANE. A Java Commodity Grid Kit. *Concurrency and Computation: Practice and Experience*, 13:643–662, 2001.

[LAS 05] GREGOR VON LASZEWSKI. The grid-idea and its evolution. *Journal of Information Technology*, 47:319–329, 2005.

[LAU 04] E. LAURE, F. HEMMER, F. PRELZ, S. BECO, S. FISHER, M. LIVNY, L. GUY, M. BARROSO, P. BUNCIC, P. KUNSZT, and al. Middleware for the next generation Grid infrastructure. *proceedings of CHEP, Interlaken, Switzerland*, 2004.

[LAW 85] EL LAWLER, JK LENSTRA, AHG RINNOOY KAN, and DB SHMOYS. "*The traveling salesman problem: A guided tour of combinatorial optimization.*". John Wiley & Sons. X, 1985.

[LI 86] G.J. LI, and B.W. WAH. Coping with anomalies in parallel branch-and-bound algorithms. *IEEE Transactions on Computers*, 35(6):568–573, 1986.

[LIT 88] MICHAEL LITZKOW, MIRON LIVNY, and MATTHEW MUTKA. "Condor - a hunter of idle workstations". In *Proceedings of the 8th International Conference of Distributed Computing Systems*, June 1988.

[LOQ 04] ORLANDO LOQUES, and ALEXANDRE SZTAJNBERG. "Customizing component-based architectures by contract". In *Second International Working Conference on Component Deployment (CD 2004)*, volume 3083 de *Lecture Notes in Computer Science*, pages 18–34, Edinburgh, UK, May 2004. Springer-Verlag.

[MAN 99] D. MANIVANNAN, and M. SINGHAL. "Quasi-synchronous checkpointing: Models, characterization, and classification". In *IEEE Transactions on Parallel and Distributed Systems*, volume 10, pages 703–713, 1999.

[MAT 06] CLÉMENT MATHIEU. Déploiement hiérarchique pour la grille. Master's thesis, Nice University France, 2006.

[MEZ 07a] M. MEZMAZ, N. MELAB, and E-G. TALBI. A grid-enabled branch and bound algorithm for solving challenging combinatorial optimization problems. *In Proc. of 21th IEEE Intl. Parallel and Distributed Processing Symp.*, March 2007.

[MEZ 07b] M. MEZMAZ, NOUREDINE MELAB, and EL-GHAZALI TALBI. An efficient load balancing strategy for grid-based branch and bound algorithm. *Parallel Computing*, 33(4-5):302–313, 2007.

[MIC 01] SUN MICROSYSTEMS. "Enterprise javabeans specication. version 2.1". Technical report, Sun MicroSystems, 2001. Technical Report.

[MOR 06] MATTHIEU MOREL. "*Components for Grid Computing*". PhD thesis, Ecole Doctorale STIC, November 2006.

[NAP 99] NAPSTER, 1999. http://www.napster.com.

[NAR] "*The Naregi project*". http://www.naregi.org/index_e.html.

[NIE 04] ROB V. VAN NIEUWPOORT, JASON MAASSEN, GOSIA WRZESINSKA A ND THILO KIELMANN, and HENRI E. BAL. Satin: Simple and efficient java-based grid programming. *Accepted for publication in Journal of Parallel and Distribute d Computing Practices*, 2004.

[NIE 05] ROB VAN NIEUWPOORT, JASON MAASSEN, GOSIA WRZESINSKA, RUTGER F. H. HOFMAN, CERIEL J. H. JACOBS, THILO KIELMANN, and HENRI E. BAL. Ibis: a flexible and efficient java-based grid programming environment. *Concurrency - Practice and Experience*, 17(7-8):1079–1107, 2005.

[OAS 06] OASIS TEAM AND OTHER PARTNERS IN THE COREGRID PROGRAMMING MODEL VIRTUAL INSTITUTE. "Proposals for a grid component model". Technical report D.PM.02, CoreGRID, Programming Model Virtual Institute, Feb 2006. Responsible for the delivrable.

[Obj 02] OBJECT MANAGEMENT GROUP. Corba component model, v3.0. specification, 2002.

[OGC] Open grid computing environment (ogce). http://www.ogce.org.

[ORA 01] ANDY ORAM. "*Peer-to-Peer : Harnessing the Power of Disruptive Technologies*". O'Reilly & Associates, Sebastopol, CA, 2001.

[ORE 60] O. ORE. Note on Hamiltonian circuits. *Amer. Math. Monthly*, 67:55, 1960.

[PAP 77] C.H. PAPADIMITRIOU. The Euclidean traveling salesman problem is NP-complete. *Theoretical Computer Science*, 4(3):237–244, 1977.

[PAP 98] C.H. PAPADIMITRIOU, and K. STEIGLITZ. "*Combinatorial Optimization: Algorithms and Complexity*". Dover Publications, 1998.

[PRO 07] TOP500 PROJECT. List of supercomputers, June 2007. http://www.top500.org/lists/2007/06.

[RAL 04] T. RALPHS, and M. GUZELSOY. The SYMPHONY callable library for mixed integer programming. *Proceedings of the Ninth Conference of the INFORMS Computing Society*, 2004.

[RAM 98] R. RAMAN, M. LIVNY, and M. SOLOMON. "Matchmaking: Distributed resource management for high throughput computing". In *In Proceedings of the Seventh IEEE International Symposium on High Performance Distributed Computing*, 1998.

[RAM 03] R. RAMAN, M. LIVNY, and M. SOLOMON. "Policy driven heterogeneous resource co-allocation with gangmatching". In *Proc. of the 12th IEEE Int'l Symp. on High Performance Distributed Computing (HPDC-12)*, 2003.

[RAT 01] S. RATNASAMY, P. FRANCIS, M. HANDLEY, R. KARP, and S. SCHENKER. "*A scalable content-addressable network*". ACM Press New York, NY, USA, 2001.

[RIT] JORDAN RITTER. Why Gnutella can't scale. No, really. http://www.darkridge.com/ jpr5/doc/gnutella.html.

[ROW 01] ANTONY ROWSTRON, and PETER DRUSCHEL. "Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems". In *IFIP/ACM International Conference on Distributed Systems Platforms (Middleware)*, pages 329–350, November 2001.

[SAT 97] M. SATO, H. NAKADA, S. SEKIGUCHI, S. MATSUOKA, U. NAGASHIMA, and H. TAKAGI. Ninf: A network based information library for global world-wide computing infrastructure. *HPCN Europe*, pages 491–502, 1997.

[SCH 01] RÜDIGER SCHOLLMEIER. "A definition of peer-to-peer networking for the classification of peer-to-peer architectures and applications.". In *Peer-to-Peer Computing*, 2001.

[SEK 96] S. SEKIGUCHI, M. SATO, H. NAKADA, S. MATSUOKA, and U. NAGASHIMA. "Ninf: Network-based information library for globally high performance computing". In *Parallel Object-Oriented Methods and Applications (POOMA)*, pages 39–48, 1996. http://ninf.etl.go.jp.

[SEY 02] K. SEYMOUR, H. NAKADA, S. MATSUOKA, J. DONGARRA, C. LEE, and H. CASANOVA. Overview of GridRPC: A Remote Procedure Call API for Grid Computing. *3rd International Workshop on Grid Computing, November*, 2002.

[SEY 05] K. SEYMOUR, A. YARKHAN, S. AGRAWAL, and J. DONGARRA. NetSolve: Grid Enabling Scientific Computing Environments. *Grid Computing and New Frontiers of High Performance Processing*, 2005.

[SHI 92] NG SHIVARATRI, P. KRUEGER, and M. SINGHAL. Load distributing for locally distributed systems. *Computer*, 25(12):33–44, 1992.

[SHI 95] Y. SHINANO, M. HIGAKI, and R. HIRABAYASHI. A generalized utility for parallel branch and bound algorithms. *Parallel and Distributed Processing, 1995. Proceedings. Seventh IEEE Symposium on*, pages 392–401, 1995.

[SIM 91] S. SIMON. Peer-to-peer network management in an IBM SNA network. *Network, IEEE*, 5(2):30–34, 1991.

[SIT] TOP500 PROJECT SUPERCOMPUTER SITES. http://www.top500.org.

[SLO 05] NEIL J. SLOANE. Sloane a000170, 2005. http://www.research.att.com/.

[SMI 03] O. SMIRNOVA, P. EEROLA, T. EKELOF, M. ELLERT, JR HANSEN, A. KONSTANTINOV, B. KONYA, JL NIELSEN, F. OULD-SAADA, and A. WAANANEN. The NorduGrid Architecture and Middleware for Scientific Applications. *International Conference on Computational Science*, pages 264–273, 2003.

[STO 03] ION STOICA, ROBERT MORRIS, DAVID LIBEN-NOWELL, DAVID R. KARGER, M. FRANS KAASHOEK, FRANK DABEK, and HARI BALAKRISHNAN. Chord: a scalable peer-to-peer lookup protocol for internet applications. *IEEE/ACM Trans. Netw.*, 11(1):17–32, 2003.

[SUN 01] INC. SUN MICROSYSTEMS. Project jxta: An open, innovative collaboration, April 2001. http:// www.jxta.org/project/www/docs/.

[TAI 93] E. TAILLARD. Benchmarks for basic scheduling problems. *European Journal of Operations Research*, 64:278–285, 1993.

[TAN 03] Y. TANAKA, H. NAKADA, S. SEKIGUCHI, T. SUZUMURA, and S. MATSUOKA. Ninf-G: A Reference Implementation of RPC-based Programming Middleware for Grid Computing. *Journal of Grid Computing*, 1(1):41–51, 2003.

[TER] TeraGrid project. http://www.teragrid.org.

[THA 05] DOUGLAS THAIN, TODD TANNENBAUM, and MIRON LIVNY. Distributed computing in practice: the Condor experience. *Concurrency - Practice and Experience*, 17(2-4):323–356, 2005.

[TSC 96] S. TSCHOKE, and T. POLZER. Portable Parallel Branch and Bound Library User Manual, Library Version 2.0. *Department of Computer Science, University of Paderborn*, 1996.

[TUT] FRACTAL ADL TUTORIAL. http://fractal.objectweb.org/tutorials/adl/index.html.

[UNI] Unicore Forum. http://www.unicore.org.

[WAL 00] JIM WALDO, and KEN ARNOLD. "*The Jini Specifications*". Addison-Wesley, 2000.

[WIE 06] P. WIEDER, R. YAHYAPOUR, O. WÄLDRICH, and W. ZIEGLER. "Improving workflow execution through sla-based advance reservation". Technical report CoreGRID TR-053, CoreGRID Network of Excellence, 2006.

[XU 05] Y. XU, TK RALPHS, L. LADANYI, and MJ SALTZMAN. ALPS: A framework for implementing parallel search algorithms. *The Proceedings of the Ninth Conference of the INFORMS Computing Society*, 2005.

[YAN 03] BEVERLY YANG, and HECTOR GARCIA-MOLINA. Designing a super-peer network. *icde*, 00:49, 2003.

[YOU 93] K. YOUNG. Look no server (peer-to-peer networks). *Network*, pages 21–2, 1993.

[ZUT] WILLY DE ZUTTER. Boincstats.com. Retrieved on 2007-01-30.

Branch-and-Bound with Peer-to-Peer for Large-Scale Grids

Abstract

This thesis aims at facilitating the deployment of distributed applications on large-scale Grids, using a peer-to-peer (P2P) infrastructure for computational Grids. Furthermore, this thesis also propose a framework for solving optimization problem with branch-and-bound (B&B) technique.

Existing models and infrastructures for P2P computing are rather disappointing: only independent worker tasks with in general no communications between tasks, and very low level API. This thesis proposes to define a P2P infrastructure of computational nodes and to provide large-scale Grids. The infrastructure is an unstructured P2P network self-organized and configurable, also allowing deployment of communicant applications.

P2P environment seems well adapted to applications with low communication/computation ratio, such as parallel search algorithms and more particularly B&B algorithm. In addition, this thesis defines a parallel B&B framework for Grids. This framework helps programmers to distribute their problems over Grids by hiding all distribution issues. The framework is built over a hierarchical master-worker approach and provides a transparent communication system among tasks to improve the computation speedup.

First, we realized an implementation of this P2P infrastructure on top of the ProActive Java Grid middleware, therefore benefiting from underlying features of ProActive. The P2P infrastructure was deployed as a permanent desktop Grid, with which we have achieved a computation world record by solving the n-queens problem for 25 queens. Second, we achieved an implementation of this B&B framework, also on top of ProActive. We demonstrate the scalability and efficiency of the framework by deploying an application for solving the flow-shop problem on a nationwide Grid (Grid'5000). Finally, we mixed this Grid with our permanent desktop Grid to experiment large-scale deployment of both n-queens and flow-shop.

Keywords: Peer-to-Peer, Branch-and-Bound, Grid Computing

Branchement et Élagage sur Grilles Pair-à-Pair à Grande-Échelle

Résumé

Cette thèse a pour objectif de faciliter le déploiement d'applications distribuées sur des grilles de calcul à grande échelle, en utilisant une infrastructure pair-à-pair (P2P) pour les grilles. De plus, cette thèse propose aussi une bibliothèque basée sur la technique « Élagage et Branchement » (de l'anglais, Branch-and-Bound – B&B) pour résoudre les problèmes d'optimisation combinatoire.

Les modèles et infrastructures pour le P2P existant sont plutôt décevants : seulement des tâches indépendantes généralement sans communication entre les tâches, et des API de bas niveau. Cette thèse propose une infrastructure P2P qui partage des noeuds de calcul, afin de fournir des grilles à grande échelle. L'infrastructure est un réseau P2P non-structuré, auto-organisé, configurable et qui permet le déploiement d'applications communicantes.

Les environnements P2P semblent être bien adaptés aux applications avec un faible ratio communication/computation, comme les algorithmes de recherche parallèle et plus particulièrement les algorithmes de B&B. En plus d'une infrastructure P2P, cette thèse propose une bibliothèque de B&B parallèle pour la grille. Cette bibliothèque aide les utilisateurs, en masquant toutes les difficultés liées à la distribution, à paralléliser leurs problèmes sur des grilles. La bibliothèque repose sur un modèle maître-travailleur hiérarchique et offre un système transparent de communication afin d'améliorer la vitesse de résolution.

Nous avons tout d'abord implémenté notre infrastructure P2P au-dessus de l'intergicielle Java pour la grille, ProActive. Cette infrastructure P2P a été déployée comme grille de bureau de manière permanente, avec laquelle nous avons pu réaliser un record mondial de calcul en résolvant le problème des n-reines avec 25 reines. Ensuite, nous avons aussi implémenté avec ProActive notre bibliothèque pour le B&B. Nous montrons le passage à l'échelle et l'efficacité de la bibliothèque en déployant sur une grille de taille nationale (Grid'5000) une application qui résout le problème du « flow-shop ». Pour finir, nous avons mixé Grid'5000 et notre grille de bureau pour expérimenter le déploiement à grande échelle des n-reines et du flow-shop.

Mots-clefs : Pair-à-Pair, Élagage et Branchement, Grilles de calcul