Faculté des Sciences Appliquées
Département d'Electricité, Electronique et Informatique
Institut Montefiore

Systematic Cooperation in P2P Grids

Année Académique
2008-2009

Thèse présentée par
Cyril Briquet
en vue de l'obtention du grade de
Docteur en Sciences (orientation Informatique)

Abstract

P2P Grid computing seeks the convergence of Grid and P2P technologies. Deploying a P2P Grid middleware on a set of computers enables an organization to automatically barter computing time with other Internet-connected organizations. Such P2P exchanges of computing time enable individual Peers, i.e. organizations, to transparently aggregate large amounts of computational power with minimal infrastructure requirements or administrative cost.

Challenges arise from the requirement for scalability and robustness. Individual worker nodes are unreliable, as P2P Grids operate on unmanaged desktop computers. A specificity of P2P Grids is that each Peer can reclaim at any time the computational power of worker nodes supplied to other Peers, leading to bursts of execution preemption. These are the major contributions of our dissertation:

- Firstly, we propose a new P2P Grid architecture, the Lightweight Bartering Grid (LBG). Through systematic cooperation between Grid nodes, the reliability of execution of computational requests is greater than the sum of the reliabilities of worker nodes.
- Secondly, we propose a highly scalable data transfer architecture. It is based both on the BitTorrent P2P file sharing protocol and on the removal of the temporal cost of downloading redundant copies of input data files.
- Thirdly, besides a middleware implementation of LBG, we also provide an implementation of a discrete-event simulator. Its originality resides in the weaving of the simulator code into the bartering code of the middleware, which is made possible through the virtualization of Grid nodes. This enables reproducible testing and accurate performance evaluation of the bartering policies because the Peers of a simulated Grid make the same bartering decisions as Peers deployed on real computers.

The LBG architecture exhibits the following remarkable features:

- The scheduling model supports the queueing of external requests and the architecture enables a flexible study of bartering policies.
- The architecture is open, flexible, lightweight and facilitates software engineering. It enables the easy development, testing, evaluation and deployment of combinations of scheduling policies.
- The architecture is fully P2P.

Résumé

Le domaine du calcul en grille pair-à-pair (P2P) a pour but la convergence entre les technologies de calcul en grille et d'organisation P2P. Déployer un middleware de grille P2P sur un réseau d'ordinateurs permet à une organisation d'échanger automatiquement du temps de calcul avec d'autres organisations basées sur Internet. De tels échanges P2P de temps de calcul offrent à chaque pair, i.e. organisation, la possibilité d'agréger de manière transparente une grande puissance de calcul, sans lourdeur administrative ou d'infrastructure.

Les impératifs de robustesse et de résistance aux facteurs d'échelle posent plusieurs défis. Les noeuds de calcul sont peu fiables, étant donné que les grilles P2P sont déployées sur des PC de bureau ou domestiques. Une spécificité des grilles P2P est que chaque pair peut récupérer à tout instant l'usage de ses noeuds de calcul qui sont prêtés à d'autres pairs, ce qui conduit à des rafales de préemption d'exécution. Les contributions les plus significatives de notre thèse sont les suivantes:

- Premièrement, nous proposons une nouvelle architecture de grille P2P, la grille de troc légère, Lightweight Bartering Grid (LBG). Grâce à une coopération systématique entre les noeuds de la grille, la fiabilité d'exécution des requêtes de calcul est plus grande que la somme des fiabilités des noeuds de calcul.
- Deuxièmement, nous proposons une architecture de transferts de données extrêmement résistante aux facteurs d'échelle. Elle est basée sur le protocole de transfert de fichiers BitTorrent et sur la suppression du coût temporel des transferts de copies redondantes de fichiers de données.
- Troisièmement, en plus de fournir une implémentation de LBG sous forme de middleware, nous fournissons également une implémentation sous forme d'un simulateur à événements discrets. L'originalité de celui-ci provient d'un fin tissage du code de simulation sur celui du middleware, qui est rendu possible par la virtualisation des noeuds de la grille. Cela permet de réaliser des tests reproductibles ainsi que des évaluations de performance précises des politiques de troc, étant donné que les pairs d'une grille simulée prennent les mêmes décisions de troc que des pairs déployés sur des ordinateurs réels.

L'architecture LBG a les caractéristiques remarquables suivantes:

- Le modèle d'ordonnancement supporte la mise en file d'attente des requêtes externes et l'architecture permet une étude flexible des politiques de troc.
- L'architecture est ouverte, flexible, légère et facilite l'ingénierie logicielle. Elle permet un développement, une évaluation et un déploiement aisés de combinaisons de politiques d'ordonnancement.
- L'architecture est complètement P2P.

Acknowledgements

I would like to thank:

Professor P.A. de Marneffe, my advisor, for his trust and support, for sharing the art of teaching and - most importantly - for creating an enjoyable workplace.

Members of the jury, Professors B. Boigelot, G. Leduc, P. Manneback, L. Moreau, M. Quinson for kindly contributing time to read this dissertation.

Three colleagues who greatly shaped my years at the Algorithmics lab: Sébastien Jodogne, one long-time friend, for our joint research work and for his enthusiasm and continuous support for all those years; Gérard Dethier for our joint research work, for reading this dissertation so systematically and for being a trusted pillar; Xavier Dalem first for being a great student to co-advise, then for our joint, geeky research work and the excellent Java code contributed to the LBG middleware.

Valérie Leroy, Thomas Leuther and Cédric Thiernesse for all the great moments. Simon Balon, Hassan Bougrine, Raff and Amandine Brancaleoni, Jorge Cham, Sébastien Charlier, Delphine Daxhelet, David Detry, Anne Falier, Jean-Marc François, Pierre and Lisiane Holzemer, Djenaba Kante, Delphine Kirkove, Danielle Lange, Axel Legay, Jean Lepropre, Sylvain Martin, Claire Monti, Christophe Noël, Marie-Thérèse Ratz, Sébastien Reginster, Pascale Renders, Philippe Rigo, Krzysztof Rzadca, Fabien Scalzo, and Rosette Vandenbroucke for the positive impact they had on my work. The EECS (Professors B. Boigelot, P.A. de Marneffe and F.X. Litt, sysadmins A. Bodart and M. Frédéric), Math (Professors P. Lecomte and M. Rigo, sysadmin S. Delsemme) and Chemical Engineering (Prof. P. Marchot) departments for letting me enlist up to 125 CPU cores in my HPC experiments, at no extra cost to our alma mater. The OurGrid team for opening the way in P2P Grids.

Claire Kopacz and Elisabeth Spilman for their thorough linguistic review - that I'm sure may have caused some headaches - of my publications.

My parents, close friends and family, as well as my swordmaster for their continuous support, presence and encouragements through the years.

Neupré, July 14, 2008.

Contents

List of Figures

List of Tables

Typographic Conventions

- terms that are defined are *emphasized*
- quoted text chunks are both *emphasized* and surrounded by a pair of quotes
- important text chunks are typeset with a **bold typeface**, or underlined
- paragraphs typeset with a Sans Serif typeface describe related work only
- paragraphs typeset with a `Monospace typeface` present code-level examples

Chapter 1

Introduction

Trust is a peculiar resource which is increased, rather than depleted, through use.

- suggested by Diego Gambetta

1.1 A Brief History of Grid Computing

This dissertation takes its root in the distributed computing domain. The concept of distributed computing exists since the deployment of high speed networks in the 1970s [276]. Large-scale distributed computing has emerged with the Information Wide-Area Year [110] (I-Way). Temporary links between 11 research centers were established just before and during the Supercomputing '95 conference, which was an occasion to demonstrate an experimental software prototype used to harness new amounts of computing power. In the continuity of metacomputing [152] research, this successful early experiment in large-scale distributed computing has led to funding from the Defense Advanced Research Projects Agency (DARPA). This research work produced the Globus Toolkit software [159], the first Grid middleware.[1]

With large amounts of computational power increasingly available from large cluster farms and, increasingly, from a myriad of computers at the edge of the Internet, aggregating computational resources at large-scale both within and between administrative domains has become a possibility within reach.

Globus was first released in 1997 and the term *Grid* appeared at a seminar at the Argonne National Laboratory. The term *Grid* was selected as an analogy to the electrical power grid because the vision of *Grid computing* [161] is that of a computing utility service that transparently provides computing power everywhere. In 1998, Foster and Kesselman published the first edition of what is considered as the

[1] Middleware = Software *"communication layer that allows applications to interact across hardware and network environments."* [122]

reference book of the domain of Grid computing: *The Grid: Blueprint for a New Computing Infrastructure*, followed by a second edition in 2004 [149].

1.1.1 What is the Grid?

Grid computing is a form of distributed processing. Grid computing is about *"coordinated resource sharing and problem solving in dynamic, multi-institutional collaborations."* [221] An important motivation for Grid computing is to run large-scale applications: *"Grid computing typically involves using many resources [...] to solve a single, large problem that could not be performed on any one resource."* [221]

A Grid may be defined as *"a number of computers linked together via the Internet so that their combined power may be harnessed to work on difficult problems"* [236], but this is not sufficient. Foster has informally proposed the following list of criteria [151] to determine if a distributed system is a *Grid*:

1. Decentralized coordination of resources from multiple administrative domains;
2. Use of standard, open, general-purpose protocols and interfaces;
3. Delivery of nontrivial quality of service, e.g. the reliability offered by a Grid is greater than the sum of the reliabilities of Grid resources.

The Grid is the (currently nonexistent) global interconnection of all the world Grids running on the Internet with standard, open and general-purpose protocols and interfaces like, for example, those proposed by the Open Grid Forum [231]. It can be compared to the Internet, that interconnects all the world networks.

The main goal of distributed computing is to help Grid participants to cope with peaks in their request environment by enabling them to dynamically consume (e.g. through bartering, borrowing, buying, ...) the resources of other participants. The removal of a centralized point of control is essentially what distinguishes Grid computing from classic distributed computing.

An additional goal of Grid computing is to help Grid participants to cope with the instability of the resource environment. A Grid is expected to provide higher levels of availability, reliability, autonomicity [27],... than the sum of its resources. It should be able to tolerate individual resource failures by dynamically negotiating resource sharing arrangements with other participants.

1.1.2 Expectations on Resource Sharing

Resource Sharing as a Performance Enabler

Resource sharing allows Peers to cope with variability in the submitted computational requests and in the availability [119] of Resources from other Peers. The *response time* of a computational request is the total time elapsed between the submission of a computational request to a Peer and its completion. Resource sharing can potentially speed up the response time of computational requests or solve large problems that cannot be solved with one's own Resources only.

Synchronized consumption by a given Peer of multiple Resources belonging to other Peers (i.e. nontrivial QoS in Foster's well known 3-point checklist [151]) may dramatically accelerate computations by temporally aggregating a great number of external Resources, leading to substantial reductions in response times of computational requests.

Resource sharing can enable to **shorten the computing time required** to process requests by **aligning the available computing power with the peaks of computational requests**.

Resource Sharing as a Reliability Enabler

The request environment of a Peer is set by User Agents, but its Resource environment can be stabilized: Resource sharing can provide reliability in the completion of execution either through consumption of computing time from Peers estimated to be reliable, or through redundant[2] consumption of computing time.

Automated Resource Sharing in a Non-dedicated Environment

In the recent years, in the wake of the early concept of Network of Workstations [37], so-called Volunteer Grid computing [11, 72] and Desktop Grid computing [230, 139] have emerged as possible, and relevant, answers to the need to gather computational power in large amounts without having to support the prohibitively high costs typically associated with supercomputers or clusters.

This need has become, in the current computing environment, a need to run arbitrary computations on a set of inexpensive, not-necessarily-reliable, Internet-connected computers... beyond the boundaries of organizations (such as illustrated on Figure 1.1). With Grid Volunteer computing and Desktop Grid computing, it

[2]Redundant consumption, e.g. through task replication [86, 308], requires extra work time. It is not considered in this dissertation.

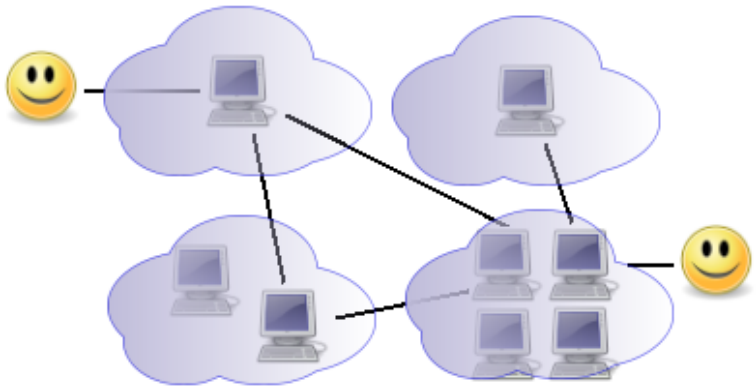


Figure 1.1: Transparent resource sharing beyond the boundaries of organizations.

can be argued that the benefits of the gathering of computational power go to a small set of selected organizations. However, this does not fulfill the requirement to enable Resource sharing between organizations.

The flocking mechanism of the Condor middleware [91] meets, to a certain extent, this new need. However, to the best of our current knowledge, there is no enforcement of reciprocity between organizations (i.e. *pools* in Condor lingo). Furthermore, a Condor deployment requires some involvement of the human system administrators and users: Not only the peering relationships must be manually configured, but some knowledge of the configuration of one another's deployment is required. Even though advances brought distributed management of resource sharing to the Condor middleware, it is far from being fully automated.

This lack of automation calls for the use of a scalable resource sharing mechanism to automate large-scale deployments (say, thousands of organizations and even more worker nodes) from both the human system administrators and users' perspective. Indeed, although it an be useful in some specific contexts to be able to detail very specific requirements when deploying a system or submitting computational requests to this system, most human users typically want to use a simple submission interface that do not require any special knowledge of the underlying computing system.

1.1.3 Towards Scalable Resource Sharing Mechanisms

Historically, resource sharing arrangements were centrally coordinated[3], with a central component playing an arbitration role. However, a centralized organization

[3]Unicore [299], UC Grid [298], XtremWeb [318], Koala [190], Condor [91] and MOAB [219] are examples of Grid middlewares that support centrally coordinated resource sharing.

introduces a single point of failure and is not intrinsically scalable for a very large system size.

Resource sharing mechanisms have thus evolved towards decentralized coordination[4] where software agents autonomously (from one another, and from human administrators) act on behalf of Grid participants, e.g. evaluate the reliability of the supplying behavior of one another [148, 58] to make scheduling decisions. A decentralized organization offers high fault-tolerance, scalability and autonomicity but maintaining fairness in resource sharing is more challenging.

With either a totally stable resource environment or instantaneous and perfect information about it, centralized coordination could generate better overall utility. On the other hand, it is unlikely to achieve the same level of scalability and fault-tolerance as fully decentralized coordination.

Two distributed and fully automated mechanisms for the sharing of Grid resources between multiple organizations have been proposed, with practical relevance, in the recent years:

- The so-called *Grid economy* [67, 41, 68, 66, 297] and market-based methods are semi-centralized as they assume the presence of a central banking component (needed to enforce the control of the monetary mass).
- *Bartering* [13, 266, 265, 82, 59] consists of non-monetary exchanges of computing time. It can be organized in a fully decentralized fashion.

1.1.4 P2P Grid Computing

The very recent Peer-to-Peer (P2P) Grid computing [233, 13, 84, 286] subdomain is essentially founded on a fully decentralized organization of resource sharing that is based on bartering, i.e. pure exchange, of computing time.

Grids and P2P systems have *"emerged in the past few years, both claiming to address the problem of organizing large-scale computational societies."* [147] P2P has been defined by Foster and Iamnitchi as a *"class of applications that takes advantage of resources [...] available at the edge of the Internet."* A *P2P network* can be defined as a network of *edge computers* (called *Peers*) exchanging resources (files, computing time, ...) without central control or management. P2P networks and applications may be conceptually as old as the Internet itself [289] but they have become well-known with the widespread use of popular file sharing applications.

[4] gLite [158], the Globus Toolkit [159], GridBus [164], SimGrid [278], Condor [91] and MOAB [219] are examples of Grid middlewares that support individually coordinated resource sharing.

The first P2P network rose to existence in June 1999 following the initial public release of Napster [222], which can retrospectively be characterized as a P2P file sharing application (of MP3-encoded music files) with centralized Peer discovery.

Combining Grid and P2P technologies has the potential to lead to the development of large-scale computational Grids based on edge resources (i.e. common desktop PC) with built-in fault-tolerance. **In a P2P Grid, each Peer:**

- **Can act both as *consumer* of computing time supplied by other Peers, and as *supplier* of computing time to other Peers;**
- **Autonomously makes its own resource sharing decisions;**
- **Cooperates[5] with other Peers by exchanging computing time but, as an autonomous entity, seeks to achieve its own objectives.**

From this perspective, ***P2P Grid computing***, which seeks the convergence of Grid and Peer-to-Peer technologies, could be defined as **computational resource sharing in Grids organized into P2P networks**. A *P2P Grid* is defined as a transient Virtual Organization [150] (see Appendix D.1.2) that emerges in a bottom-up fashion, out of exchanges of computing time between participants (i.e. Peers) connected together through a P2P network. In this dissertation, 2-levels P2P Grids are considered, i.e. each Peer (first level) manages a set of worker nodes (second level). Moreover, the considered worker nodes are *edge computers*, i.e. heterogeneous, low cost, possibly unreliable, standard computers.

The **partial and intermittent nature of resources** is also a defining feature of a P2P Grid. Indeed, because of the fully decentralized nature of a P2P Grid, the resources involved are typically *"partial and intermittent"* [119], i.e. often exhibiting degraded performance and intermittent availability.

The **informational opacity** between Peers is another defining feature of a P2P Grid. Indeed, the amount and contents of control messages emitted and received by Peers should be minimal for two reasons:

- The full decentralization precludes Peers to exchange a lot of metadata, because the architecture should remain scalable.
- The full decentralization also precludes Peers to consider the metadata received from other Peers as trustworthy: Indeed, we hypothesize that there is enough trust between Peers to enable their cooperation, but not enough to fully trust one another's behavior.

[5] Cooperation in P2P Grids should be understood as *emergent cooperation*, i.e. there is no explicit shared goal, in Doran et al.'s typology [124].

P2P Grid computing is very recent. If a milestone had to be selected, a good choice would be a paper by Foster and Iamnitchi [147] published in 2003 that discussed and called for the convergence between the domains of Grid computing and Peer-to-Peer technologies.

1.2 Dissertation Statement

Human users of computing systems are increasingly interested in running an ever larger number of computational tasks. A challenge of high practical interest consists of composing a **user-friendly Grid that automates every operation** required to **complete sets of independent**[6] **computational tasks** (so-called Bags of Tasks). This implies the **opportunistic use of additional resources** as they become available, the **graceful recovery** of operation **as some resources become unavailable** and the **automatic placement and transfer of data files**.

The goal is to **hide the complexity of the Grid behind a simple submission interface** so that human users only have to wait for the results of their computational tasks to be automatically retrieved from the Grid. Furthermore, **from the point of view of a typical human user, the wait time should be as short as possible**, even if there is no QoS guarantee; a useful metric that should be minimized is thus the mean Bag of Tasks response time (MBRT). An additional requirement is to **respect the natural tendency of human activities to be structured into separate organizations**. This calls for a set of distributed mechanisms to enable the sharing of computational time between separate organizations.

Our dissertation addresses this challenge.

- We claim that bartering is a mechanism that allows software agents, called Peers, to share in an autonomous and scalable way the resources of the human organizations they represent. Such mechanism promotes the opportunistic use of resources supplied by other Peers but comes at the price of a lack of control over these resources and of transparency of other Peer's operations.

- Task execution failures are common events due to preemption, as Peers can reclaim for their own use the computational power they have supplied to other Peers. To mask failures, we claim that Peers can be organized around queues of tasks, so that preempted tasks do not necessarily have to be cancelled. We propose a scheduling model that structures the scheduler of each Peer into policy decision points, enabling to control the scheduling process.

[6] Examples of intended use cases are provided in Section 2.4.

- Another issue is the massive size and amount of data files to be transferred across the Grid (so-called data deluge). The massive amount of data can appear even for one small file because the same file processed by multiple tasks of a Bag of Tasks may have to be replicated to several worker nodes. In practice, indeed, the data files often present repetitive patterns. For example, some human users will want to repeatedly process the same set of files over time, while other human users will want to process the same set of files in a variety of ways. We claim that a fully decentralized data transfer architecture, based on BitTorrent and designed to take advantage of the repetitive patterns in sets of data files can address these requirements. It is also designed so that worker nodes can collaborate beyond Peer boundaries to enable scalable data transfers.

These proposals are realized through the Lightweight Bartering Grid (LBG) architecture [55], a fully decentralized P2P Grid architecture to build **2-level Grids organized in a Peer-to-Peer fashion, where each Peer controls a set of worker nodes** (so-called Resources) and is totally independent from - and opaque to - other Peers. A middleware implementation of LBG that can run arbitrary applications coded in Java has been produced. Furthermore, it has been shown that it can run tightly-coupled sets of tasks that require co-allocation, which are not typically run on a P2P Grid. Finally, we also claim that **accurate and reproducible simulation can and should be used as a software engineering tool to build P2P Grids**. Finding efficient combinations of scheduling policies is challenging because of the massive number of possible combinations that are made possible by the open nature of the scheduling model. To debug and evaluate in a reproducible way a massive number of combinations of scheduling policies, we propose to **weave a discrete-event simulator into the code of a virtualized version of the middleware**.

1.3 Our Contributions to the P2P Grids Domain

Our work is mainly focused on the collective (top) layer of the Grid architecture [150]. The state of the-art P2P Grid middleware, OurGrid [233, 84, 286], has solved several important issues, notably protection against free-riding and accurate accounting of exchanged computing time. However, given the recency of the domain of P2P Grids, many research issues are still open.

An initial observation is that the supplying side of bartering has been well studied, while the consumption side has received less attention. With OurGrid, a Peer splits between its users the computing time consumed from other Peers (see Appendix 2.5.3).

Therefore, OurGrid's scheduling model does not allow a Peer to adaptively select the Peers from which to consume computing time; it also does not support queueing of computational requests from other Peers.

The benefits of queueing support for computational requests of other Peers are:

- Scheduling can be either online or batch-mode;
- A preempted computational request of another Peer (e.g. following the crash of a worker node) can be requeued and subsequently rescheduled a few moments later, instead of being cancelled and sent back to the other Peer;
- The input data files of the preempted and subsequently rescheduled computational request may still be available, while they would probably have been unavailable had it been cancelled and scheduled to another supplier Peer.

Peers in any P2P Grid act in their own interest, which leads them to cooperate through the bartering of computing time. We hypothesize that cooperation is beneficial at several other levels in a P2P Grid architecture. We have identified several areas of a P2P Grid architecture where our contributions would be original, timely and valuable: scheduling policies, software engineering and data transfers. Moreover, although the selected Grid application model targets sets of independent computational requests, we have been able to demonstrate how heavily-communicating applications can be run on a P2P Grid.

1.3.1 Cooperation between Robust Scheduling Policies

Guaranteeing the robust execution of computational requests is absolutely critical in the unreliable computing environment of P2P Grids. Beyond execution failures due to worker failures, preemption of the execution of external computational requests is a major and frequent cause of failure in P2P Grids. Indeed, some Peers may choose to preempt the execution of external computational requests in order to reclaim for their own, immediate use of the computational power of their worker nodes.

Regarding the robustness of execution, we would rather use the more general term *fault-management* rather than *fault-tolerance* to cover fault-tolerance itself, as well as fault-avoidance and fault-prevention. We combine several original as well as existing fault-management mechanisms.

We first propose to support two middleware-level fault-tolerance mechanisms that enable a P2P Grid to continue operating in presence of faults: reexecution, and control of computational requests. The former is a widespread mechanism that

consists of reexecuting failed requests until they successfully complete. Control of computational requests consists of preempting computational requests that take an unusually long time to complete. We also propose a variant of existing middleware-level *fault-avoidance* mechanisms: ranking and blacklisting of suppliers of computing time. We then introduce a new middleware-level *fault-prevention* mechanism that is complementary to the proposed fault-avoidance mechanism: adaptive preemption. It consists of adaptively deciding whether to requeue or not a preempted computational request (in order to prevent it from experiencing long queueing delays), and whether or not to give the computational request a short grace period during which it can try to complete its execution before being preempted.

Finally, we introduce a variant of an existing application-level fault-tolerance mechanism that enables a robust and scalable execution of heavily-communicating - thus tightly interdependent - computational requests. This is of high practical interest because this type of application is not expected to run on P2P Grids, as it requires the continuous, i.e. without interruption, co-allocation of a large number of resources.

These fault-management mechanisms operate independently from one another and it is their cooperation that provides a robust execution of computational requests. An important benefit of our approach is that runtime estimates of the computational requests are not required; this has however the side effect of limiting the information available to the scheduling policies.

1.3.2 Cooperation between Software Implementations

There are two consequences to the distributed and unreliable nature of a P2P environment: Implementing P2P Grid middleware is difficult and evaluating the performance of new scheduling policies is difficult.

As P2P Grid software is distributed and asynchronous in nature, as well as intended to run in unstable environments that are hard to reproduce, it is challenging to test and debug. Apart from a few rare experience reports [52], the software engineering of P2P Grids has received little attention so far.

Performance evaluation is typically done with discrete-event system simulation, in order to reduce the time span of the evaluation. The LBG architecture, that is implemented as a middleware, might thus also be implemented as a simulator separate from the middleware. However, a drawback is that the differences between the middleware implementation and the simulator implementation inevitably decrease the accuracy of the simulation measured as a the distance between simulated and real values of performance metrics, e.g. completion time of computational requests.

Instead, we propose to virtualize Grid nodes, in the sense of virtualizing their implementation, not in the sense of running them in a Virtual Machine. The code of a discrete-event simulator is weaved into the code of the virtualized middleware at boundaries between Grid nodes and their environment. Most of the code of the middleware is reused in the simulator implementation, which we refer to as *code once, deploy twice*. This enables the execution of a whole P2P Grid in a controllable and reproducible way on a single computer, thus allowing full observation and facilitating the testing and debugging of the common code, notably scheduling policies. Moreover, Peers of a simulated P2P Grid make the same bartering decisions as when deployed on real computers, leading to high simulation accuracy.

Only the contemporarily proposed GRAS component [253] of the SimGrid [78, 278] middleware introduces a mechanism of similar nature. Both approaches are intended as tools to facilitate software engineering but differ in some of their use cases and features. GRAS/SimGrid is designed to be generic (bottom-up approach) while the LBG simulator is specifically tailored to the LBG middleware (top-down approach). The LBG simulator offers a simulation description language in which the configuration of Peer policies can be easily expressed. This tight integration enables to rapidly evaluate new combinations of scheduling policies [59, 63]. The simulation of multithreaded code (of the LBG middleware) constituted a requirement of the LBG simulator, while it is only recently [78] that support for it was added to GRAS/SimGrid through the SimIX component. Had we started our research in 2008, GRAS/SimGrid would have constituted a good starting point to enable the virtualization and simulation of LBG.

1.3.3 Cooperation Between P2P Technologies

Data transfers of large input data files can constitute a major bottleneck [19], delaying the completion of computational requests. The data transfer mechanisms commonly used are centralized, thus not scalable and leading to an unbalanced network load over Grid nodes. Existing mechanisms to reduce data transfers are based on data reuse, and do not simultaneously enable data reuse and parallelism of the execution of multiple computational requests.

We introduce a scalable data transfer architecture based on a transparent increase of the number of download sources. The architecture is intended to spread the load caused by data transfers, across the P2P Grid. Grid nodes are equipped with data caches and data transfer (download) as well as data sharing (upload) softwares.

To enable execution parallelism for computational requests with some redundancy in their required input data files, we propose to use a P2P data transfer mechanism

to transfer data, specifically the BitTorrent P2P file sharing protocol. BitTorrent is intrinsically more scalable than direct data transfer mechanisms, e.g. FTP.

We propose to use a combination of novel and existing data-aware scheduling policies to support the data transfer architecture. The combination of BitTorrent and these scheduling policies ensures an efficient download of uncached data: Every input data file must obviously be downloaded at least once into the P2P Grid, but our algorithms are designed to reduce the cost of downloading identical - i.e. redundant - copies of input data files, be they needed simultaneously or over time.

1.4 Overview of the Dissertation

Figure 1.2 illustrates the structure of this dissertation as well as the dependencies between the chapters. The review of the state of the art and the related work devoted to specific areas of P2P Grids is distributed among the relevant chapters.

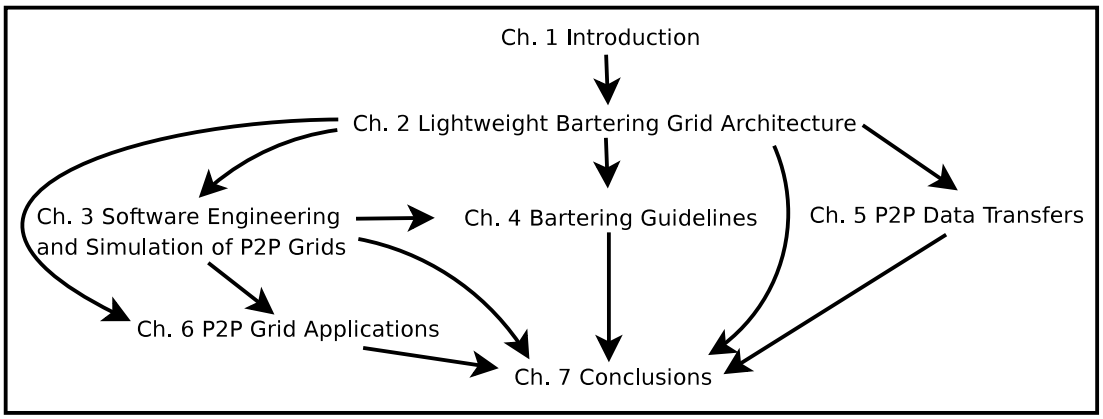


Figure 1.2: Structure of the dissertation.

Chapter 2 introduces our proposed P2P Grid architecture, the Lightweight Bartering Grid, the middleware implementation of which is also discussed. Basic concepts in the domain of P2P Grids are defined and the state of the art in resource sharing methods and P2P Grid software is reviewed.

Chapter 3 discusses tools to facilitate the software engineering of P2P Grid software. It explains how to virtualize Grid nodes and implement a discrete-event P2P Grid simulator that also instantiates our proposed P2P Grid architecture.

Chapter 4 discusses guidelines for bartering policies, notably taking into account the reliability of supplier Peers.

Chapter 5 introduces a scalable data transfer architecture, and proposes to combine P2P Grid and P2P file sharing technologies.

Chapter 6 reviews applications that, non-expectedly, can be structured as sets of independent computational Tasks to be run on P2P Grids.

Chapter 7 puts the contributions of this dissertation into perspective.

Beyond these main chapters, several appendices complete this dissertation.

Appendix A describes the application and service interfaces of the Lightweight Bartering Grid architecture.

Appendix B defines the job description language that is used to submit Bags of Tasks to Peers. It also defines the simulation description language.

Appendix C provides complementary notes on Peer middleware internals and implementation notes.

Appendix D provides complementary notes on Virtual Organizations.

Appendix E provides complementary notes on resource negotiation and standard Grid protocols.

Appendix F provides a general summary of future work.

Chapter 2

Lightweight Bartering Grid Architecture

Take what you need, give what you don't need.

The middleware implementation of the Lightweight Bartering Grid architecture provides a software that can autonomously share computing time between separate organizations in a totally decentralized way, respecting the hypothesis of informational opacity between Peers. The middleware relies on a bartering mechanism based on the existing Network of Favors (NoF) model to do so. The reclaiming of computational power by a Peer for its own use, i.e. preemption, considerably increases the unreliability of an already unreliable environment. The flow of computational requests through a Peer is organized around the management of queues of computational requests, as well as several fault-management mechanisms. To mitigate the impact of preemption, an original feature of the associated scheduling model is to support the queueing of external computational requests. This enables to completely mask Task execution failures to the other Peers that submitted them.

Basic concepts of P2P Grids are provided in Sections 2.1-2.2. Each type of Grid nodes (Peer, Resource, User Agent, Search Engine) is introduced in Section 2.1. Basic mechanisms of a P2P Grid are introduced in Section 2.2. The state of the art in Resource sharing that is relevant to P2P Grids is then presented in Section 2.3. A classification of Grid application types is given in Section 2.4. In Section 2.5, a classification of lightweight Grids is proposed and the state-of-the-art P2P Grid middlewares are subsequently reviewed.

Our proposed P2P Grid architecture, *Lightweight Bartering Grid* [59] (LBG), enables to build 2-levels[1] P2P Grids based on bartering[2] where Peers autonomously

[1] 2-levels = each Peer (1st level) manages a set of worker nodes (2nd level).

[2] Bartering in LBG = non-monetary Resource sharing mechanism that can be organized in a

act and model their environment. The LBG architecture is described in Sections 2.6-2.9; its middleware implementation is also commented.

Software gridification, i.e. how to enable a software application to run on the Grid, is first briefly considered. The Grid application model of LBG is introduced in Section 2.6, The LBG middleware is fully implemented [55] in Java and targets Grid applications that are also implemented in Java.

Computer gridification is then considered. Making the computational power of a (worker node) computer available to the Grid is the purpose of the LBG Resource middleware, described in Section 2.7. Completing computational requests submitted by users and bartering computing time through the sharing of the computational power of worker nodes are the purposes of the LBG Peer middleware, described in Section 2.9. The communication protocol used to transmit control messages between Grid nodes is also described, in Section 2.8.

2.1 Grid Nodes

The term *Grid node* is introduced to name any networked computer that is inside or at the edge of a Grid: User Agents, Peers, Resources and Search Engines. This term is introduced to easily designate all the computers connected to a Grid and to facilitate the discussion of network communications within, and at the edge of, a Grid. Figure 2.1 introduces the symbols that are used to visually represent Grid nodes and their storage capacity in the remainder of this dissertation.

The definitions of Resource, Peer and User Agent that are given in this section do not exclude the situation where one given computer simultaneously runs more than one type and/or instance of Grid middleware[3]. For instance, a User Agent, a Peer and a pair of Resources could be deployed on the same multi-core computer, while additional Resources are deployed on other computers. Figure 2.2 represents a typical P2P Grid composed of 4 sites, with multiple Grid nodes interacting together: 2 User Agents, 4 Peers and 8 Resources.

2.1.1 Resources

There exist many types of resources: computational devices (desktop PC, supercomputer, laptop), storage devices, networking devices, mobile devices (e.g. smart phone, PDA, ...), sensors, ... The scope of our research encompasses typical office

fully decentralized fashion, as introduced in Section 1.1.3, and as discussed in Section 2.3.2.

[3] These cases are supported, but are not further covered in order to keep the text readable.

Figure 2.1: Symbols representing Grid nodes, and their storage capacity.

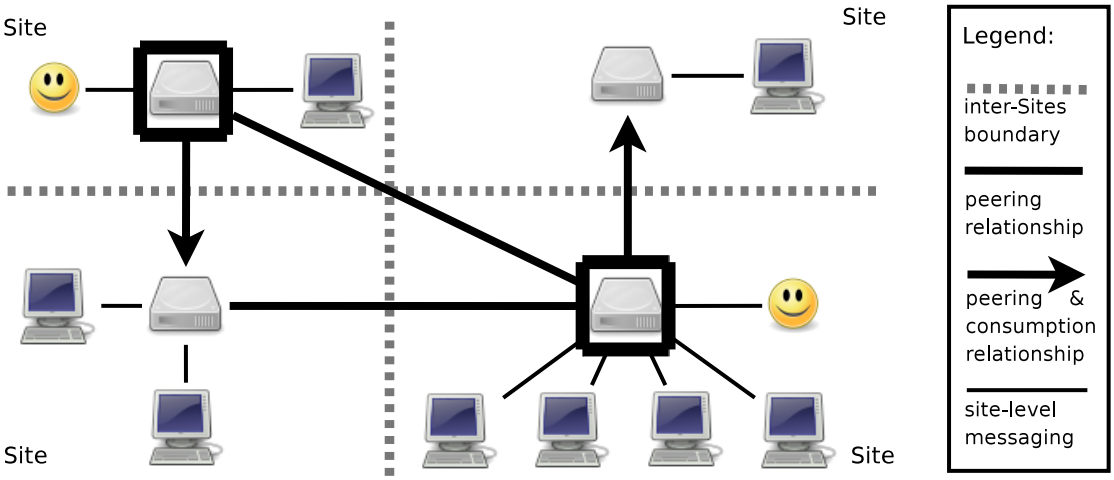


Figure 2.2: P2P Grid.

or home computers connected to the Internet - so-called *edge computers*. Therefore, the term *Resource* designates heterogeneous, low cost, possibly unreliable, standard computers with standard Internet connectivity, with some storage capacity and a Java Virtual Machine (J2SE 5.0 [178]) available.

The Resource middleware is the software that makes a computer part of a Grid. In the following, the context will make it clear whether the term Resource refers to the Resource middleware or to the computer itself.

2.1.2 Peers

A *site* is a set of heterogeneous (in terms of hardware and software) Resources (possibly of small size, i.e. consisting of 1 or even 0 Resource [4]) of an administratively independent organization. Several sites are independent from one another. The administrators of a site group the Resources under their control to aggregate computational power for the human users of the site. The Resources of a site are all within the same administrative control. Local management of the Resources of a site is controlled by the human site administrators.

[4]Special case of a Peer with 0 Resource: see Section D.3.5.

The following concept of the Peer-to-Peer domain is introduced to refer to the computer and associated Grid middleware controlling a given site on behalf of its human administrators. A *Peer* is a standard networked computer (i.e. similar in capacity to a Resource) of a site, designated to control the Resources of a site.

There is one and only one Peer associated to each site. It acts on behalf of the site administrators, i.e. it autonomously uses and shares the Resources of a site. The purpose of a Peer is to complete as fast as possible the computational requests submitted by the human users of a site.

The Peer middleware is the software - organized as a software agent - that enables a computer to control the Resources of the site. In the following, the context will make it clear whether the term Peer refers to the Peer middleware that runs on the computer, or only to the computer itself.

We now point out a difference in our proposed definitions from commonly available definitions. A Grid *Resource Management System* (RMS) has sometimes been defined as *"the subsystem of a Grid that identifies requirements, matches resources to applications, allocates, schedules, monitors Grid resources over time in order to run Grid applications as efficiently as possible."* [221] Our definition of a Peer matches this definition. But, in our view, the term *RMS* refers only to the specialized software component of a Peer that manages Resources (i.e. control and handling of every communication with the Resources) along other Peer software components that handle other operations (e.g. scheduling, queueing, ...).

2.1.3 User Agents

The term *User Agent* refers to the standard networked computer that is used by human users of a site to interact with the Peer of the site, e.g. submitting computational requests, retrieving computed results.

The User Agent middleware is the software that enables a human user to interact with a Peer. It is essentially an interface to submit computational requests to the P2P Grid. In the following, the context will make it clear whether the term User Agent refers to the User Agent middleware that runs on the computer, or only to the computer itself. Human users are referred to as such (prefixed with "human", with lower case *u*).

2.1.4 Search Engine

Peer discovery is the mechanism enabling the Peers of a P2P network, in particular of a P2P Grid, to become aware of other Peers. The LBG architecture relies on existing results for the Peer discovery mechanism, as it has been widely studied in the P2P domain.

A *Search Engine* keeps track of the Peers that are online. Each Peer proactively registers itself with a Search Engine, typically as it comes online, to eventually become visible to others. Peers can connect to a Search Engine to download the location data of other Peers. To ensure the freshness of its database, the Search Engine periodically checks that registered Peers are still online. In response to a Peer discovery request, a Search Engine provides location data of an arbitrary number of randomly selected Peers, as well as a timestamp. This timestamp can be used in the next Peer discovery request to ask only for Peers that were registered afterwards; Peer clocks thus do not need to be synchronized as only the time of the Search Engine is involved [99]. This mechanism based on Search Engines is similar to the Rendezvous Point [327] mechanism, with out-of-Grid deployment. Rendezvous Point has been shown to be more efficient under high load than pure propagation-based mechanisms.

Peer discovery is discussed essentially in Sections 2.8 and 2.9.5, with complementary information provided in Appendix D.2. One can therefore safely assume that, in this dissertation, the term Grid node does not encompass Search Engines.

2.1.5 Data Caches

The term *data cache* refers to the storage component of an internal Grid node. In the following, the context will make it clear whether the term data cache refers to the hardware capacity[5] considered alone or together with the middleware-level software components that make it controllable by a Grid node.

2.2 LBG Architecture Overview

The goal of LBG is to **enable Peers to autonomously and reliably run sets of independent computational tasks** submitted by human users. The User Agent is introduced as a **submission interface** to the Peer; it completely hides the complexity of the P2P Grid to the human users.

[5] OS-level file system and physical storage device.

To process computational requests submitted by User Agents, a Peer has 2 options (see Figure 2.3):

- to **schedule and run computational requests on its own Resources**;
- to **submit computational requests to other Peers**, that accept to supply some computing time of some of their Resources.

The second option of **consuming computing time from other Peers is particularly interesting at peak**, when a given Peer has no available Resources to run concurrently all the computational requests submitted by User Agents.

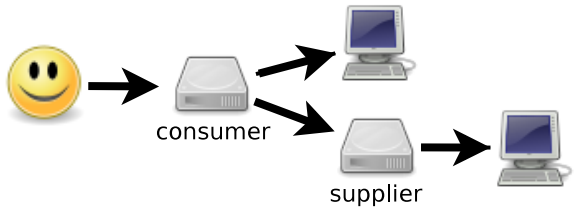


Figure 2.3: Scheduling to one's own Resource vs. scheduling to another Peer.

The terms *consumer* and *supplier* (also illustrated on Figure 2.3) refer to the roles (dynamically) assumed by Peers sharing Resources. A *consumer Peer* is a Peer **consuming computing time** of a Resource of another Peer. A *supplier Peer* is a Peer **supplying computing time** of one of its Resources to another Peer. Every Peer can act both as a consumer and as a supplier of computing time. Periods of consumption and supplying of a given Peer are typically alternating but may overlap under certain circumstances.

The responsibility of a Resource is to execute computational tasks. A Peer can schedule (steps 3 and 7 of Figure 2.6) and preempt (step 5 of Figure 2.6) the execution of a computational task to any of the Resources under its control. The execution of a computational task on a Resource is **sandboxed** (as will be illustrated on Figure 2.12 in Section 2.7.4) so that the computational task cannot compromise or interfere with the computer host.

If computational tasks remain to be scheduled and none of its Resources is available, a Peer can request the supplying of computing time from other Peers. Potential supplier Peers are located using a **Search Engine**.

The role of the Search Engine is to provide location information of other Peers (as illustrated on Figure 2.4). Using this information, the Peer can contact the located Peers to negotiate the supplying of computing time from their Resources (as illustrated on Figure 2.5, which occurs between step 1 and step 2 of Figure 2.6).

Figure 2.4: Example of two Search Engines, thus two P2P Grids.

The purpose of the negotiation process is to prevent consumer Peers to submit computational tasks where they would face long waiting times. The contacted Peers can reply with consumption grants. The negotiation needs not be complex and does not involve counter-bids. The decision of the contacted Peers to accept to supply computational power is based on the availability of their own Resources.

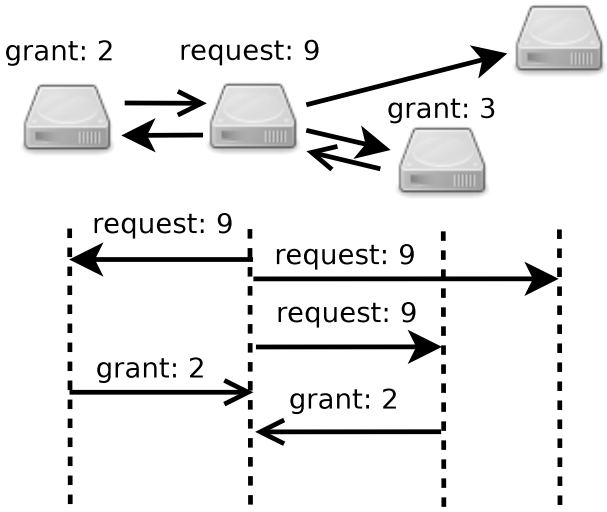


Figure 2.5: Example of negotiation.

Upon successful negotiation, the Peer can then schedule computational tasks to the supplier Peers that have sent the consumption grants (step 2 of Figure 2.6). **The supplier Peers will first filter the submitted computational tasks** before actually accepting their submission. Negotiation (i.e. the initial request) and filtering (of the actual submission of computational tasks) are kept separate. Thanks to this separation, **negotiation is non-binding, which precludes any requirement for stateful negotiations**. As negotiation is non-binding, computational tasks submitted to a supplier Peer (which thus has previously sent consumption grants) may be rejected. This is not an issue as the Peer repeatedly negotiates with other Peers as long as computational tasks remain to be completed.

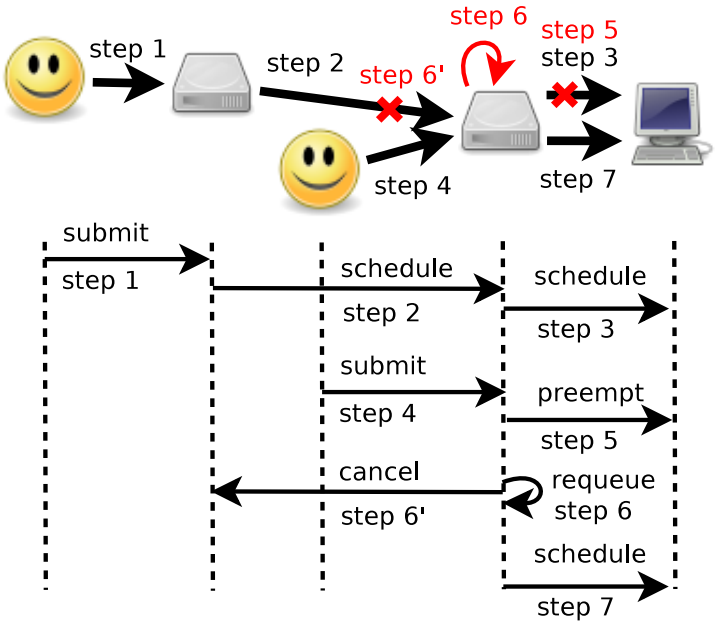


Figure 2.6: Example of preemption/cancellation of Task execution.

When Resources are available and all of its computational tasks are scheduled (either to Resources under its control or to other Peers), **a Peer schedules to its Resources the computational tasks submitted by other Peers** (step 3 of Figure 2.6).

If new computational tasks are submitted by its User Agents (step 4 of Figure 2.6), **a Peer can consider to reclaim the computational power of those of its Resources that are supplied to other Peers.** In practice, this translates into the preemption of the execution of computational tasks submitted by other Peers (as illustrated on Figure 2.6). **Execution failures of computational tasks are thus a common occurrence** in a P2P Grid, as they can originate from preemption of execution, in addition to Resource failure.

After the preemption of a computational task submitted by a consumer Peer, **a Peer can decide to cancel the computational task**, i.e. to notify the consumer Peer that the execution will not be completed (step 6' of Figure 2.6). **A Peer could also decide to mask the execution failure of a computational task** to the consumer Peers by requeueing the preempted execution failure (step 6 of Figure 2.6 is not performed). Doing so removes the need for the consumer Peer to locate another willing supplier Peer. **In order to mask execution failures of computational tasks, a supplier Peer maintains a queue of the computational tasks submitted by other Peers**, similarly to the queue of computational tasks that is maintained to mask execution failures to User Agents.

The bartering interactions presented in Figure 2.6 can be summarized as follows:

1. a User Agent submits a BoT to a Peer
2. this Peer has no available Resource; it submits a Task to a supplier Peer
3. supplier Peer schedules the accepted Task to one of its available Resources
4. a User Agent of the supplier Peer submits a BoT
5. the supplier Peer has no available Resource; it preempts the execution of the Task from the consumer Peer
6. the supplier either requeues the preempted Task or cancels it and send it back, uncompleted, to the consumer Peer
7. the supplier Peer schedules its own Task to the freed Resource

Grid nodes (Peers, Resources, User Agents) communicate with one another at the Grid-level, systematically through **message-passing**. Grid nodes use a simple, middleware-specific[6] messaging protocol, the **Grid Node Messaging Protocol** (GNMP). Incoming messages are processed on each Grid node by a component called service.

The behavior of Grid nodes is as asynchronous as possible. In particular, the service operations are designed to be non-blocking, i.e. in the few instances where a return value to an incoming GNMP message is expected, this value is computed asynchronously[7] from the ongoing operations of the Grid node.

Nonetheless, any Grid node that expects to receive a specific message from another Grid node (such as successful completion of execution of a computational task, or such as a negotiation message), should not block for an infinite amount of time - even if there is no deadlock or no impact on other operations, and if only because it wastes O.S.-level resources. As an example, network partitions may occur: The cancellation message (step 6) on Figure 2.6 may not reach the consumer Peer (even though the supplier Peer will attempt repeatedly to resend the message). As another example, computational tasks scheduled to local Resources or to other Peers may never complete because the Grid application has entered an infinite loop. As another example, a computational task scheduled to another Peer may remain queued forever, without being ever scheduled. Therefore, **a timer mechanism should enforce some (possibly arbitrarily large) limit on the time that certain operations may take**. The timer mechanism generates internal events, which are processed by a dedicated component, the internal event processor.

[6]This restriction is planned to be removed in the near future.
[7]This require great care in the implementation.

2.3 Related Work - How to Share Resources?

Peers exchange computing time when they have reached a mutual agreement by which a supplied capability can be used to perform some work on behalf of the consumers [149]. *"The key concept is the ability to negotiate resource-sharing arrangements among a set of participating parties (providers and consumers) and then to use the resulting resource pool for some purpose."* [151]
Resource negotiation is the process by which participants reach an agreement to share Resources. This dissertation focuses mainly on Resource sharing mechanisms and also negotiation objectives, at the expense of negotiation objects and protocols (see Appendix E). As a consequence, the negotiation protocol used to share Resources (introduced in Section 2.9.5) needs not be complex.
In this section, the state of the art of Resource sharing mechanisms that could be used to design P2P Grids is reviewed, while other aspects of Resource negotiation are briefly discussed in Appendix E.

Market-based methods are first reviewed, then bartering is proposed as a simplified form of Grid economy, and reviewed. An existing bartering model, Network of Favors, is subsequently presented. Multi-Agent Systems are then briefly reviewed and it is explained why they are not relevant to the specific requirements of this dissertation, despite their intrinsic quality.

2.3.1 The Grid Economy and Market-Based Methods

Resource sharing mechanisms describe how to select trading partners and agreeing to, refusing or proposing sharing agreements. An important body research devoted to Resource sharing mechanisms for large-scale systems is centered around market-based methods. Among them, the so-called computational economy or Grid economy [67, 41, 68, 66, 297] has been shown to adequately fulfill Grid computing requirements.

The idea of a Grid economy stems from the observation that some incentive must be offered to all suppliers to sustain the interest of the Peer to engage into Resource sharing on a regular basis [67, 65]. It is then only natural to propose a *"Grid economy as a model for managing and handling requirements of both Grid providers and consumers."* [65]

In a Grid economy, access to resources, e.g. computing time, is considered as a commodity that can be bought and sold on a Resource market. A market mechanism can be defined as a kind of competitive equilibrium protocol that adjusts the price of a valuable Resource given the demand for it, until demand matches supply [67, 297] in the considered Resource market. GridBus [164] is a Grid mid-

dleware supporting the Grid economy and implementing many advanced concepts and market-based methods [324, 66]. The interest of market-based Resource sharing is mainly twofold: to enable Peers to find the desired Resources at the lowest cost possible and to stabilize the price of traded Resources.

With few exceptions [311], most research about the Grid economy has considered a centralized organization, because some central banking component is required to enforce the control of the monetary mass of the so-called Grid currency [154, 67]. Therefore, a common issue in market-based Resource sharing is that most research making the assumption that pricing is available has *" assumed it could be supplied by some oracle agent."* [271] The requirement of centralized component, acting as a single point of failure, may be characterized as a weakness, and is one of the main arguments motivating the development and use of P2P Grids, which have a fully decentralized organization.

Grid Currency

A Grid currency is a virtual currency that may or may not be directly tied to a real currency. Some organizations would be willing to earn real money with the supplying of computing time [290, 9, 249], and spend real money as well in order to handle spikes of computational requests with extra Resources supplied by a Grid. But some organizations cannot or do not want to trade Resources for money, especially small and underfunded organizations, like small research labs [286]. So the Grid economy is probably never going to be a comprehensive approach fitting every need.

Resource Valuation

Another common issue is that market-based mechanisms, while efficient at stabilizing a Resource market, do not yet sufficiently explain how to take into account the valuation of the Resources by the Grid policies: Market-based mechanisms simply suppose that if a Resource is important for a Peer, its Peer will demand much of it. This problem can certainly be related to the definition of negotiation objectives presented in this section. Indeed, each Peer may seek to maximize its own utility or its benefits in a short-term perspective only, or to minimize risks associated with the considered agreement [177].

While this approach is certainly worthy in totally unstable Resource markets, without structure or repeating patterns, it does not promote the building of trust and lasting trading relationships which could bring more benefits in the long-term. Many Grid stakeholders would certainly benefit from advances in the study of long-term trading relationships, including departments of the same university or

subsidiaries of the same global corporation, as their Resource markets will, over time, exhibit trading behavior patterns that can be taken advantage of.

Most market-based methods seek to reach market equilibrium. The decisions are guided mostly by price, but typically not by other variables that are difficult to embed directly into a price. For instance, in a market with perfect atomicity, quality of service (e.g. reliability, predictability) could potentially help to differentiate multiple suppliers all offering comparable resources at, or very near, some optimal price. This issue is much more than a subtle refinement. It is a core issue: Schopf et al. pointed out that human users of a Grid *"want not only fast execution times from their applications, but predictable behavior, and would be willing to sacrifice some performance in order to have reliable run times."* [272] Taking into account factors that cannot easily be embedded into a price is, in our opinion as well, a promising idea when negotiating access to Resources.

2.3.2 Bartering as a Simplified Form of Grid Economy

As seen in the previous paragraphs, not all Peers seek immediate financial profit from the supplying of computing time. The main objective of many Peers is to offer good service to *their own* User Agents, i.e. computational requests, under classic constraints (temporal deadlines, response time, utilization). To reach this objective, they are eager to consume Resources from other Peers when their own Resources cannot handle their computational load. But they may not be willing or able to pay real money for this consumption. They may however be eager to supply their own resources when they don't use them. In this sense, there is no absolute requirement for currency-based transactions, be it real money or a virtual currency.

Orthogonally, the dependence of a system on a central component, in this case a central bank, introduces a single point of failure, as well as some degradation of performance when the system has to deployed at a large scale.

Both these arguments (the observation that a currency-based organization is not mandatory, and the drawbacks of a centralized organization) are compatible with a lightweight form of commerce that consists of distributed, non-monetary sharing: *bartering* [67]. It can be defined as *"exchanging (goods or services) for other goods or services without using money."* [236] In the context of Grid computing, it can be defined as a form of decentralized, non-monetary exchange of computing time [13, 82, 59].

For the purpose of this dissertation, *bartering* is thus defined as a non-monetary Resource sharing mechanism that can be organized in a fully decentralized fashion. In practice, bartering has been shown in related

works [86, 266, 265] to enable highly scalable, fully accounted and fair sharing of computing time between Peers.

Delayed vs. Immediate Reciprocity

Requiring immediate reciprocity might lead to issues in the bootstrapping of Resource sharing among Peers [82]. Moreover, **there is no cost [13] for idle Peers** (i.e. online Peers that have no waiting requests of their own) **to supply computing time in excess** - to make a gift [263] - of the quantity of Resources these have supplied in the past (i.e. take what you need, give what you don't need). As computing time both not needed and not supplied is wasted, it is actually in the long-term interest of idle Peers to increase their reputation by supplying computing time to any Peer.

The definition of bartering is sometimes restricted to a consumption of computing time immediately followed by a reciprocal supplying. Our understanding of bartering is closer to Obreiter et al.'s concept of *community pattern*[8] [228] except that, in their view, it relates to a local - rather than global - computing environment.

With delayed bartering, free riding is allowed so as to enable Peers to establish their reputation. A priority mechanism can however prevent any impact from free riders by serving local requests first, and only then external requests, in order of decreasing reputation of the requester. Such a priority mechanism is described in Section 2.3.4.

Supposing the availability and use of a priority mechanism to mitigate free riding (which will be discussed in the following subsections), **Peers that barter computing time should not limit their supplying to reciprocal amounts. Therefore, Peers can and should spend as much time they can into supplying computing time to other Peers.** This generates as much consumption potential as possible for the suppliers, as well as this promotes high system utilization.

Bootstrapping the Grid Economy

Bartering has also been proposed as a mechanism to bootstrap the Grid economy, in order to break the initial lack of trust between Peers. Such a perspective implies that market-based methods always depend on bartering to start the system.

[8] *"A community is a group of entities whose incentives for cooperation are based on the trust gained by providing services to other entities of the community."*

An interesting P2P Grid middleware [82], built on the SHARP (Secure Highly Available Resource Peering) system, envisions a *"bartering economy as providing the basis for decentralized growth."* [154] Resource discovery and creation of a secure P2P overlay are thoroughly considered. A simple Tit-for-Tat policy incites Resource sharing. It must be noted that the establishment of trust is explicit and requires the deployment of the SHARP system, which cannot be considered lightweight at all.

An interesting observation that can be made about SHARP is that bartering is presented as a mandatory first step in the evolution of a Grid economy towards currency-based commerce involving the most reliable Peers only.

Service Level Agreements

With a fully decentralized organization of bartering, it would prove difficult to enforce penalties resulting from the breach of contracts, beyond refusing to supply Resources to consumers that do not reciprocate past supplying. Studying the use of so-called Service Level Agreements [96, 323] (SLA) is therefore not a priority in this dissertation.

Centralized Bartering

Not all proposed bartering mechanisms are fully decentralized, thus not all are relevant to our research work.

The Faucets middleware [138, 186] is a centralized, zero-sum cluster bartering architecture which relies upon a database located on a central server. Credit is granted to the computing Peers to consume Resources. A bidding system allows the Peers to compete for consumption of computing time.

As another example of bartering (not related to Grid computing, though), a bartering mechanism proposed for health care management [1] involves a centralized component and, furthermore, targets synchronized Resource sharing (i.e. there are system-wide coordination events of the sharing of the considered Resources). Both of these attributes (central component, system-wide coordination) make this bartering mechanism not relevant to our research work.

Existing Bartering-based Grids

OurGrid [233, 84, 286], a free-to-join P2P Grid with a focus on a global computing environment, matches pretty well our understanding of bartering. It is based on the so-called Network of Favors model [13]. Peers exchange computing time with one

another: They supply access to computational Resources (i.e. to *make favors*, in OurGrid terms) in the hope of reciprocal behavior on behalf of the consumers. Each Peer keeps its own accounting [266, 265] of consumption and supplying of computing time, enabling a fully distributed architecture. The Network of Favors model and accounting of bartering are thoroughly described in the next section.

GridIS [317] is yet another incentive-based P2P Grid middleware which seeks to optimize the scheduling of computational requests. It provides incentives to the Grid participants to continue consuming and supplying computing time. Similarly to OurGrid, the focus is mainly on supplier policies, without much attention given to consumer policies. Suppliers may be configured to adopt an aggressive (potentially bigger compensation) or conservative (less risk) request filtering policy.

The request filtering policy of GridIS is somewhat related to Irwin et al.'s work [177] on Grid scheduling in a Grid where Resource sharing is market-based. This system proposes a Resource sharing market with a focus on request filtering policy. The idea is to balance risk and reward by considering the opportunity cost of accepting new jobs.

2.3.3 Expectations and Free Riding in Bartering Systems

If a Peer does not own enough Resources to complete the computational requests of its User Agents without having an infinitely growing requests queue, it systematically consumes more Resources than it can supply. **Such a behavior where there is little to no reciprocity in the exchanges is called *free riding***, as illustrated on Figure 2.7. It is a widespread issue in P2P networks to which bartering is not intrinsically immune.

We first discuss what can be expected of a bartering system, then examine the free riding issue.

Work Time Invariance

The *work time* of a Peer for a request is the total amount of time spent to process the request. What is often forgotten is the fact that consumer Peers should compensate the work time of supplier Peers, so that these remain interested in supplying computing time.

For instance, a Peer may actually spend only a small fraction of its work time on its own requests, a large amount of work time on requests from other Peers, and still be better-off, i.e. achieving faster response times and higher utilization. **Resource sharing can thus be seen as a way of shortening response time, but certainly not as a way of decreasing work time** (see Figure 2.8).

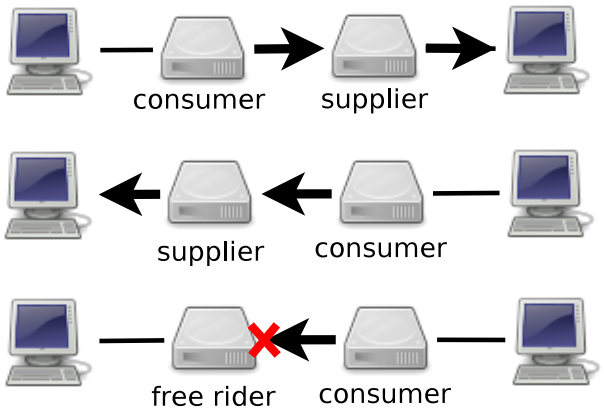
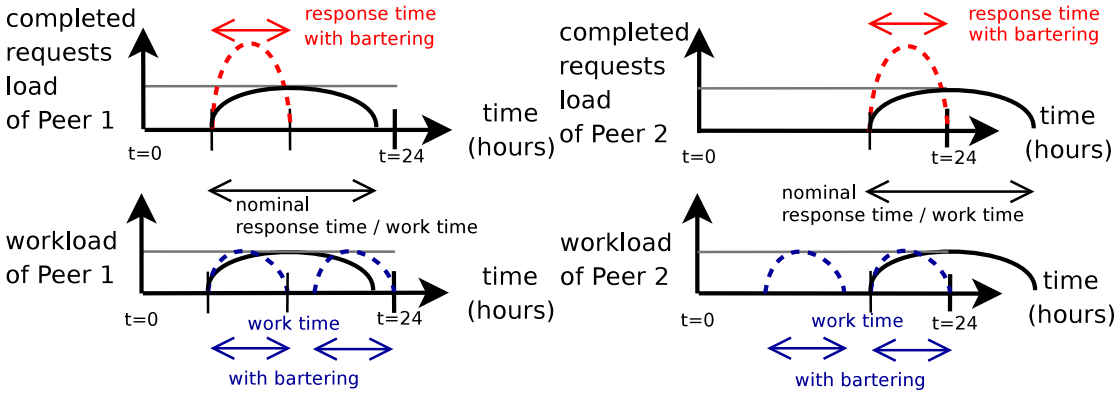


Figure 2.7: Reciprocating vs. free riding.



Over a period of 24 hours, 2 Peers with a fixed nominal power have a (temporally complementary) peak of local requests, i.e. expected work time using their nominal power only, of 8 hours. With bartering, both Peers achieve a response time of 4 hours only; their work time remaining 8 hours.

Figure 2.8: Bartering can reduce the response times; work times are invariant.

Computational Power Invariance

When sharing Resources, **a Peer should make consumption and supplying decisions so that its total computational power remains invariant over time**. The total computational power decreases if the Peer loses access to some Resources (it cannot do anything about it) or discards some Resources (but this is not typical). The total computational power increases if the Peer consumes computing time but does not supply it back. For example, a finite time span can be assumed. When a Peer consumes computing time from another Peer of similar power, it should not try to process twice as many requests in this time span. Rather, it should process the same amount of computational requests - that are

thus completed twice as fast (under optimal conditions) - and then supply back the consumed computing time.

Prevalence of Free Riding

Honesty of other Peers cannot be simply expected, despite what has sometimes been suggested (*"service acquisition implies guarantee of service"* [149]). On the contrary, free riding tends to become prevalent in P2P networks whenever no mechanism enforces fairness; incentive mechanisms encouraging cooperation between Peers thus need to be built directly into the P2P networks [141].

The nature of P2P leads to bottom-up formation of Virtual Organizations (see Appendix D). As a side effect, Peers often exchange computing time with other Peers they had never dealt with before. P2P thus exhibits the dark side of allowing dishonest Peers to exchange computing time with honest Peers. Peers with naive or simplistic exchange behaviors would exhibit supplying patterns that could be taken advantage of by dishonest Peers.

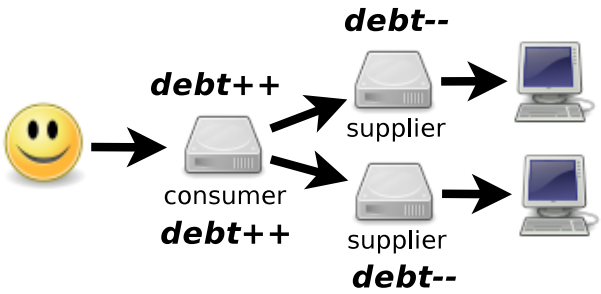
Unintentional Free Riding

A Peer consuming computing time from other Peers spends less work time on its own computational requests. It gains idle time that would have been spent as work time for its own computational requests if it had not consumed time from other Peers. This gained idle time should be invested as work time for other Peers; not supplying it is actually free riding. Acquiring more hardware is the only way to increase the nominal computing power of a Peer.

2.3.4 Network of Favors Model

The Network of Favors [13] (NoF) bartering model is the mechanism used in the OurGrid [233, 84, 286] middleware. It has been proposed by Andrade et al. to enable Peers to maintain, in a fully decentralized fashion, their own private bookkeeping of computing time exchanged with other Peers. It is helpful to Peers as it provides the basis to make supplying decisions.

Additionally, **the NoF bartering model can mitigate the impact of free riders**. It is thus a solid basis for the building of a P2P Grid.




```
consumer:  debts towards top supplier += consumed computing time
consumer:  debts towards bottom supplier += consumed computing time
top supplier:  debts towards consumer -= supplied computing time
bottom supplier:  debts towards consumer -= supplied computing time
```

Figure 2.9: Network of Favors accounting.

Decentralized Measurement of Resource Sharing

Following the NoF model, a consumer Peer can autonomously estimate its debt towards a supplier Peer. This debt consists of the computational power that the supplier has spent[9] to compute computational requests of the consumer Peer. A supplier Peer can also autonomously estimate the amount of computational power supplied to a consumer Peer.

The NoF model is based on the concept of favor. A *favor* is defined as an estimation of the computational power exchanged between two Peers. A consumer Peer and a supplier Peer may have a different estimation of the Favor the supplier has supplied to the consumer.

Each Peer involved in a Network of Favors maintains a separate favor balance for each Peer it has exchanged computing time with, as illustrated on Figure 2.9. A favor balance maintained by a given Peer P_1 about another Peer P_2 is a nonnegative favors count representing the current amount of the *Resource (or computing time) debt* the given Peer P_1 should compensate to the other Peer P_2.

Each time a Resource has been consumed by a given Peer from a supplier Peer, the given Peer increases the favor balance maintained for this supplier Peer. In other words, this given Peer acknowledges the beneficial behavior of the supplier Peer by taking note of an increase in its debt toward the supplier. Reciprocally, each time a Resource has been supplied by a given Peer to a consumer Peer, the given Peer decreases the favor balance maintained for this consumer Peer. In other words, this supplier Peer decreases its debt towards the consumer Peer.

[9]It is the Resources of the Peer that actually spend time completing computational requests.

Following these two simple rules, each Peer is able to measure the debts to/of other Peers by relying only on metadata it has directly collected and not at all on metadata communicated by a third-party Peer or a Grid-level monitoring service. Effective measurement of Resource sharing, i.e. estimation of debts of computing time, can thus be made in a totally decentralized fashion, with an efficiency that has been demonstrated to increase [14] with the size of the P2P Grid.

Fair Supplying

The purpose of maintaining favor balances is to rank Peers, in order to determine to which Peer to supply an idle Resource, whenever there are no waiting local requests. Priority of supplying by a given Peer is given to Peers with the highest Favors balance, i.e. towards which the supplier has accumulated the highest amount of debt.

A very important design choice is that consumers are ranked only when a supplier Peer cannot fulfill both all its own computational needs and all the Supplying Requests of the consumers. It means that, instead of remaining idle, a Peer increases the debts of other Peers by supplying computing time. This supplying-by-default policy allows to bootstrap the Grid by automatically establishing an initial amount of trust, allowing new Peers which have never supplied Resources to use the idle Resources of the other Peers.

The penalty associated to this policy is however low. Free riders have a very low priority and obtain access to Resources from a given Peer only when no better ranked Peer has asked to consume Resources from the given Peer. Furthermore, as every Peer always use its Resources for its own needs, no free rider could directly disrupt the operations of a given Peer.

As mentioned, a favor balance is always nonnegative so as to avoid ID-changing attacks. By attributing the same debt, i.e. 0 favor, both to Peers that have yet to supply any Resource (which could therefore be free riders), and to those Peers that have already been completely compensated for Resources they have previously supplied, a consumer Peer discourages free riders as it denies any advantage for a malicious Peer to enter into a free ride/change-ID cycle.

Each Peer can preempt or cancel the execution of computational requests on supplied Resources to avoid that unusually long executions of computational requests from other Peers cause delay to its own computational requests. The consumer Peers that submitted the preempted or cancelled computational requests perceive the given supplier Peer as slow or unreliable. This may reduce their willingness to exchange computing time in the future.

Robustness of Decentralized Measurement of Resource Sharing

An important issue is how to accurately account the amount of consumption and supplying of computing time. In other words, how to accurately estimate favors.

Several accounting policies have been proposed by Santos et al. [266, 265]. The policy that has been found to be the most accurate is called Relative Power. The main advantage of the Relative Power policy is that it is robust to free riders that may be tempted to Supply artificially slow Resources.

Following the Relative Power policy, a supplied Favor is the amount of time the given Peer has supplied a Resource to a consumer Peer. A consumed Favor is the amount of time the given Peer has benefited from a Resource of a supplier Peer, weighted by the relative performance of the supplier Peer. To compute the relative performance of the supplier Peer, several policies are possible. In OurGrid, each Peer uses its own performance as baseline: The mean response time of the supplier Peer (over all computational requests computed by the supplier Peer) is divided by the mean response time of the given Peer (over all computational requests computed by the given Peer). This estimation may be improved using Task benchmarking, but at a cost.

Working Assumptions

As described in the previous subsections, **the existing Network of Favors (NoF) bartering model essentially solves free riding**. As NoF is used in the LBG architecture, free riding is not studied as such in this dissertation. It is important to note that this simplifies the presentation of other important issues; it does not avoid to address an important issue, because free riding in a P2P Grid has already been addressed in related works. Nonetheless, the evaluation of LBG's implementation of NoF and its tolerance to free riding should be evaluated as part of future work.

In the remainder of the text, Peers are thus supposed to be controlling enough Resources to serve their User Agents, except if noted otherwise. In practice, it means that they are supposed, over time, to be able to reciprocate their consumption of computing time. More formally, Peers are assumed to be stable systems [36]: The arrival rate of jobs submitted to a given Peer is on average less than or equal to its job completion rate.

A consequence is that Peers not controlling any Resource - thus unable to compensate suppliers - are not considered; such Peers using external, i.e. out-of-Grid, compensation to consume Resources are briefly discussed in Appendix D.3.5.

2.3.5 Multi-Agent Systems

The Multi-Agent Systems (MAS) paradigm naturally comes to mind when considering how to share Resources. It is now finally examined why we feel this mechanism cannot be used in the context of this dissertation.

A reciprocity mechanism based on the MAS paradigm has been proposed with a focus on mitigating collusion between malicious Peers [34]. Like the OurGrid Network of Favors model, decisions are based on past interactions. However, as opposed to OurGrid, *"newcomers are not helped until their reputation is above a threshold for at least one resource type."* Moreover, an additional mechanism is proposed to mitigate collusion between malicious Peers: Peers that trust one another regularly exchange their own view of the reputation of other Peers.

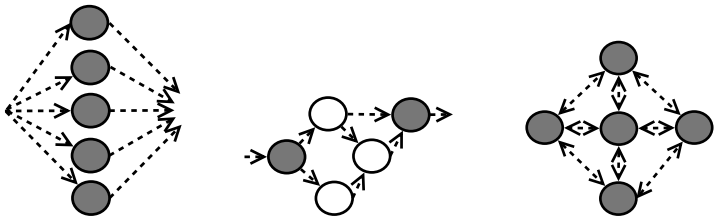
Another reciprocity mechanism based on the MAS paradigm uses reputation referral capabilities [129]. Variants of such a mechanism based on reputation referral can also separate reputation and accuracy [32].

Despite their intrinsic quality and success to tackle very challenging issues, these mechanisms are based on the explicit sharing of information between agents. Consequently, they cannot be used in the context of this dissertation, because of the hypothesis of informational opacity between Peers that is introduced in Section 1.1.4.

2.4 Grid Application Types

The computational requests submitted by User Agents to Peers are Grid applications, each composed of a set of Grid Tasks. In LBG, **a Grid Task** (simply: *a Task*) **is defined as a computational task that processes input data files and that produces output data files**.

To determine what type of sets of Tasks can be run on a P2P Grid, we propose to **classify Grid applications according to their communication patterns, i.e. inter-Task data dependencies**. Figure 2.10 illustrates our proposed classification into Bag of Tasks, Workflow, Iterative Stencil. Nodes on Figure 2.10 represent Tasks (gray = start/finish Tasks, white = regular Tasks) and edges represent data dependencies. Although this classification by no means intends to be exhaustive, it can probably express the inter-Task data dependencies of most computations of practical relevance.



(a) Bag of Tasks (b) Workflow (c) Iterative Stencil

Figure 2.10: Grid application types.

2.4.1 Bag of Tasks Applications

An application that can be structured as a set of independent computational Tasks is called a *Bag of Tasks* [13, 84] (BoT). Each Task usually requires some input data and produces some output data. No data is exchanged between Tasks, but multiple Tasks of a Bag of Tasks may depend on - i.e. process - identical input data files. Bags of Tasks constitute typical P2P Grid applications.

BoTs constitute an important class of applications, covering domains such as: bioinformatics [113], cellular microphysiology [76], computer vision [184], data mining [267, 184], discrete optimization [293], geographical information systems (GIS) [54, 171, 22], medical image processing (tomography) [280], parameter sweeps[10] [2], pattern matching [273], protein folding [143] and docking [123], Search Engine crawling and indexing [107, 108].

Data-Intensive Bags of Tasks [267, 113, 76, 184, 257] constitute a subclass of the Bags of Tasks applications, where the amount of processed data is large, leading to data transfer times that are long compared to computing times.

These applications have also been called PHD (Processors of Huge Data [267]). With increasing capacities of data acquisition, most of Grid applications from typical BoT domains are often Data-Intensive BoTs.

2.4.2 Workflow Applications

An application that can be structured as a set of dependent computational Tasks is called a *Workflow application* [326]. Workflow Tasks are often, but not always, connected with a so-called Directed Acyclic Graph (DAG).

[10] A *parameter sweep* designates a BoT where the Grid application and input data files of all Tasks are identical, with only the parameters being different. It has a high relevance in practice.

Typical domains of Workflow applications include [149]: astronomy (e.g. pulsar search), bioinformatics (e.g. basic local alignment search tool - BLAST), high energy physics (HEP), medical image processing (functional magnetic resonance imaging - fMRI).

Workflow applications are harder to run than Bags of Tasks because the required data management is more complex as output data files of some Tasks are also used as input data files of other Tasks. Workflow applications are also harder to run on P2P Grids [125] than on classic, top-down Grids because the Resources are not as reliable. Consequently, P2P Grid research has been mainly dedicated to Bag of Tasks applications.

2.4.3 Iterative Stencil Applications

An application that can be structured as a set of periodically recomputed dependent computational Tasks is called an *Iterative Stencil application* [187]: *"A stencil [application] updates every point in a regular grid with a weighted subset of its neighbors"*, as if a stencil were figuratively applied [187].

In an Iterative Stencil application, there are communications between running Tasks. In a Workflow application, there are communications between completed and unstarted Tasks. Workflows are typically modelled using a directed acyclic graph: As an Iterative Stencil typically involves cycles in the communication graph, it cannot be modelled as a Workflow.

Typical domains of Iterative Stencil applications include [187, 114]: computational fluid dynamics, electromagnetics, geometric modeling, image processing, partial differential equations solving.

Iterative Stencil applications are inherently hard to compute, even in stable environments. Computing them on a P2P Grid is a real challenge, although this specific combination has been examined in very recent research [125, 115, 116].

2.5 Related Work - Lightweight Grids

The concept of Lightweight Grid is first discussed. A review of state-of-the-art P2P Grid middlewares is then proposed. Finally, a comparison of LBG and OurGrid (the most closely related P2P Grid middleware) is provided.

Expected requirements and features	Actual support in the Lightweight Bartering Grid architecture
1. Lightweight and generic	YES (lightweight) YES (generic: any Java application) NO (generic: nonstandard protocol)
2. Static and dynamic metadata	YES
3. Dynamic deployment of components	PARTIALLY (some support implemented but not integrated)
4. Reconfiguration and adaptivity	YES
5. Support for both client/server and P2P Resource sharing	YES (as the concepts of Peer/Resource are separated, a P2P Grid can be deployed as a Desktop Grid)
6. On-demand, provider-centric service provision	YES (Peers provide service for User Agents, other Peers)
7. Minimal but sufficient security model	PARTIALLY (protection against application-level attacks but lacks authentication, encryption)
8. Binding and coordination	YES (provision, utilization and coordination are fully decentralized in a P2P Grid)
9. Additional services	NO
10. Distributed management	PARTIALLY (centralized implementation of Peer discovery)

Table 2.1: Lightweight Grid - expected requirements, features vs. actual support.

2.5.1 Classification of Lightweight Grids

A *lightweight Grid* [286] is a Grid that either does not expose standard interfaces or does not completely support standardized features. Grid standards (see Appendix E.2) are only beginning to emerge. Their proposed features are numerous and primarily intended for more controlled environments that P2P networks. It is thus a reasonable choice to build a lightweight Grid. Our proposed architecture meets most requirements and supports most features that have been identified by Badia et al. [33] as worthwhile for a lightweight Grid (see Table 2.1).

Several specialized lightweight Grid architectures - so-called Desktop Grids, Volunteer Grids and P2P Grids - have emerged in the last few years. These lightweight Grid architectures are now discussed so as to contrast P2P Grids with Desktop Grids and Volunteer Grids. Finally, existing P2P Grid middlewares are reviewed.

Desktop Grids

A Grid architecture sharing undedicated Resources, that has generated increasing interest in corporate environments, is *Desktop Grid Computing* [230, 139]. There are differences in the existing definitions of Desktop Grid, but also some common patterns:

- the sites belong to the same real-world organization rather than to individual users scattered on the Internet;
- there is a small number of sites, mainly composed of desktop PC.

It can certainly be argued that Desktop Grids are not really Grids because they are unmanaged clusters spanning only one site. However, usage of the term *Desktop Grid* has been steadily increasing in the recent years.

BOINC [51], DG-ADAJ [230], ProActive [251] are middlewares that support the Desktop Grid architecture. So-called Cycle Stealing projects like Condor [255] might also be considered as Desktop Grids. XtremWeb [318] also supports the Desktop Grid architecture.

Volunteer Grids

A Grid architecture sharing absolutely unreliable Resources that has attracted considerable attention in the last few years has been named *Internet Computing* [72] or, more figuratively, *Volunteer Computing* [11]. Several sites (usually a huge number) supply Resources, to a limited number of sites (usually one). Resources are managed by a centralized RMS.

SETI@home [273], Folding@home [143] and Docking@home [123] are middlewares that support the Volunteer Grid architecture. Cycle Stealing projects like Condor [255] can also be considered as Volunteer Grids.

Interestingly, BOINC [51] and XtremWeb [318] both support the Desktop Grid and Volunteer Grid architectures, and, as indicated in a previous paragraph, XtremWeb [318] also supports a lightweight form of the centralized Resource sharing (see Section 1.1.3).

P2P Grids

P2P Grid Computing is a very recent Grid architecture. We have defined it as computational Resource sharing in Grids organized into P2P networks (see Section 1.1.4). The shared Resources are edge computers, which are at best moderately reliable. Two key concepts are:

- each site is considered as both a supplier and a consumer of computing time;
- sites are organized into P2P networks, which requires that automatic Peer discovery is available.

Peer discovery is independent from the Peers themselves: It may be either fully distributed or centralized, even though it can be argued that full decentralization is required to have a true P2P network. Automatic Peer discovery is a very important issue, but is beyond the scope of this dissertation. Complementary notes are available in Appendix D.2.

An important embodiment of P2P Grid computing has been originally proposed in Brazil and has resulted in the development of the OurGrid middleware [233, 13, 84, 286] to serve the needs of small research labs lacking the funding to fulfill ever-increasing computing needs. The OurGrid middleware has been successfully put into large-scale production since December 2004 [52] to run a large variety of applications. The more recent Zorilla middleware [125], developed in the Netherlands, and the Lightweight Bartering Grid middleware [59], presented in this dissertation, are also middlewares that support the P2P Grid architecture.

2.5.2 P2P Grid Middlewares

Several existing P2P Grid middlewares, as well as one relevant Grid middleware, are now reviewed (the list is sorted in alphabetical order of middleware name). LBG is also included for comparison purposes.

Most of the reviewed middlewares are initial implementations and only one has been deployed to large-scale production: OurGrid [233, 84, 286], which has already achieved great success [52].

AssessGrid

AssessGrid [24, 25, 307] is a Grid middleware. It cannot be considered as a P2P Grid middleware because consumption of computing time is not reciprocated with supplying of computing time, but instead with out-of-Grid compensations (see Appendix D.3.5), i.e. real money. Moreover, it is not clear whether Peer discovery is fully automated.

We selected AssessGrid for inclusion in this list of P2P Grid middlewares because some of its concepts are very relevant to the development of P2P Grids. In particular, AssessGrid proposes that Resource brokers, acting as intermediaries between the User Agents and supplier-only Peers, learn the reliability of Peers. This is similar to what we

have proposed in previous research [60] (see Section 2.3). In this view, lessons learned from the AssessGrid project are likely to be relevant to future P2P Grid research.

The focus is on fault-tolerance, reliability, risk management and user-level decision making tools. AssessGrid leverages lots of source code from existing projects. CCS [90], programmed in a mix of C, C++ and Perl, is used as a basis for Peers. GridSphere [166], programmed in Java, is deployed to fulfill the role of User Agents. The programming of new code is in Java. Writing Grid applications, in any language, is easy. Although most negotiation and scheduling work is performed automatically, the scheduling and sharing decisions must ultimately be done manually by human users through the Grid User middleware.

AssessGrid is under development at a number of European universities and firms, with a strong participation from University of Paderborn, Germany.

DGET

DGET [172, 173, 174] is a P2P Grid middleware where each Peer acts as a Resource. The Peer and Resource concepts are merged.

The focus is on extreme scalability, fault-tolerance, ease of use and rich API. Computational requests written in Java can be submitted to the P2P Grid. A Java GUI enables to control a Peer. Writing Grid applications, in Java, is easy. No bartering mechanism is proposed.

DGET is developed at University College Dublin, Ireland.

JNGI

JNGI [300, 183] is a P2P Grid middleware with 2 levels. The concepts of Peer and Resources are clearly separated, although both Peers and Resources are Peers in JNGI, with so-called Task Dispatcher Peers corresponding to Peers and Worker Peers corresponding to Resources.

The focus is on Peer discovery service (using JXTA), fault-tolerance, self-organization (bottom-up VO formation). Writing Grid applications, in Java, requires to use low-level API calls. No bartering mechanism is proposed.

JNGI is a Free and Open Source software mainly developed at Sun Microsystems.

LBG

Lightweight Bartering Grid [55, 57, 63, 62, 56, 59] (LBG) is the P2P Grid middleware developed for the research presented in this dissertation. The concepts of Peer and Resources are clearly separated and correspond to those presented in the previous sections.

The focus is on fault-tolerance, autonomy, data transfers and software engineering. Computational requests written in Java can be submitted to the P2P Grid. Writing Grid applications, in any language, is easy. The deployment of the Grid by a human administrator is easy. The advanced bartering model proposed by OurGrid (see below) is used. Everything is fully automated in LBG, enabling human users to *submit and forget* computational requests to the User Agent middleware, and later retrieve completed results.

LBG is a Free and Open Source software developed at University of Liège.

OurGrid

OurGrid [233, 84, 286, 13, 52, 53, 266, 265, 267, 98] is a *free-to-join peer-to-peer grid that has been in production since December 2004* [233]. The concepts of Peer and Resources are clearly separated and correspond to those presented in the previous sections.

The focus is on fault-tolerance, autonomy and ease of deployment and programming. Computational requests written in Java or in any other language can be submitted to the P2P Grid. Writing Grid applications, in any language, is easy. The deployment of the Grid by a human administrator is easy. A deployment downside is that User Agents require O.S.-level access to Resources. An advanced bartering model is proposed. Everything is fully automated in OurGrid, enabling human users to *submit and forget* computational requests to the User Agent middleware, and later retrieve completed results.

OurGrid is a Free and Open Source software developed mainly at Federal University of Campina Grande, Brazil and HP.

P2P Disco

P2P Disco [196, 28] is a P2P Grid middleware with 1 level.

The focus is security and the design is driven by the specific requirements of a specific neural network application based on the NeuroSearch algorithm. It is currently used more as a Desktop Grid than a P2P Grid. Computational request written in Java can be submitted. Writing Grid applications is easy, but deploying them is difficult. There

is no support for dynamic code uploading, so the Grid applications must be deployed out-of-Grid (see Appendix D.3.5) before they can be submitted. This is done to augment security.

Moreover, middleware-level IP filtering is performed, which is not yet widespread in Grid middlewares in general. There is no support for data transfers and only application parameters are transmitted to Resources.

P2P Disco is developed at University of Jyväskylä, Finland.

P2P-MPI

P2P-MPI [258] is a P2P Grid middleware with 1 level.

The focus is the execution of MPI applications in an unreliable environment, i.e. edge Resources. A fault-tolerant scheduler mitigates the unreliability of P2P Grid Resources. Another important feature is portability, as P2P-MPI supports MPJ, is developed in Java and enables to run Java applications. Peer discovery relies on JXTA.

P2P-MPI is developed at University of Strasbourg, France.

smartGRID

smartGRID [291] is a P2P Grid middleware with 2 levels, i.e. a Peer manages several Resources. Its variant smartGRID2 is a P2P Grid with 1 level, i.e. the Peer and Resource concepts are merged.

The focus is on use of web services, self-organization (bottom-up VO formation), adaptability to heterogeneous networks, flood scheduling. Writing Grid applications, in Java, requires to use low-level API calls. The flood scheduling process is inspired by neural networks and modelled with Colored Petri Nets. No bartering mechanism is offered, though.

smartGRID was developed at University of Wollongong, Australia.

Zorilla

Zorilla [125] is also a P2P Grid middleware where each Peer acts as a Resource, and the Peer and Resource concepts are merged.

The focus is on the scheduling of workflow applications requiring co-allocation [275, 6, 168], extreme scalability, flood scheduling, fault-tolerance. Writing Grid applications, in any language, is easy. The negotiation process initiated by a Peer that wants to consume Resources is performed by flooding/gossiping with Peers in the neighborhood.

Zorilla is developed at VU University of Amsterdam, the Netherlands.

Other Grid Middlewares

Other Grid Middlewares might have been included in this review.

SimGrid [278, 209, 77, 75] is a very advanced and flexible Grid middleware and simulator. It is possible [175], that it may be used as a basis to build a P2P Grid.

MOAB [219] is a commercial Grid middleware, self-described as a P2P Grid middleware. However, it does not offer a Peer discovery service.

Summary of Features in Existing P2P Grid Middlewares

The main features of the reviewed P2P Grid middlewares are now summarized. Table 2.2 lists architectural and scheduling features. Table 2.3 gives negotiation-related features. Table 2.4 gives security features.

The middlewares can be classified according to the number of control levels: 2-levels P2P Grids where the concepts of Peer/Resource are separated, and 1-level P2P Grids where each Peer controls only one Resource (itself). We hypothesize that, in the long term, 2-levels P2P Grid will become the norm because they greatly facilitate the work of human site administrators.

The bartering mechanism of most reviewed P2P Grid middlewares is actually very basic, even implicit (i.e. all incoming requests are accepted). Nearly all negotiation mechanisms are very basic, i.e. either a request/response scheme or a request flooding mechanism. Only AssessGrid, LBG and OurGrid can mitigate free riding (Table 2.4).

AssessGrid supports SLAs and reservations. However, AssessGrid is not fully automated (it relies on human users to define acceptable levels of risks) and is based on the Grid economy.

Despite not offering SLA support, LBG and OurGrid are the only reviewed P2P Grid middlewares that (similarly AssessGrid, which is not a P2P Grid middleware) can enable decentralized large-scale Resource

sharing between separate organizations. Only LBG and AssessGrid support the queueing of external requests.

Besides AssessGrid, LBG and OurGrid, only P2P-MPI is capable of matchmaking; the other middlewares only support eager scheduling or flooding. Task replication is supported by JNGI, OurGrid and P2P-MPI. Results validation mechanisms typically rely on Task replication. Results validation, a feature typically found in Volunteer Grids, is not supported by any of the reviewed middlewares. Resource protection and secure communications are each supported by about half the reviewed middlewares.

Table 2.5 gives data features. It appears that data-awareness and decentralized data caching, despite their huge impact on performance, are not systematically supported. LBG is the only reviewed middleware with support for P2P data transfers.

Table 2.6 gives technology features. The supported application model is mostly the Bag of Tasks. Interestingly, P2P-MPI supports both Workflows and Iterative Stencils.

Nearly all reviewed middlewares support only Java as the programming language to develop Grid applications.

All reviewed P2P Grid middlewares have been developed in Java (AssessGrid, although partially relying on legacy code, is also developed in Java). We hypothesize that this unanimous choice of Java originates in Java's excellent support for multithreaded and network programming, portability, permissive license and built-in sandboxing mechanism (the virtualization of code execution may provide - if properly configured - a supplementary layer of protection very useful to any Internet-facing software).

Future Grid standards will propose a language-neutral set of interfaces, maybe based on web services, making less important the issue of selecting a technology to develop middlewares.

To conclude this review, it is worth noting that half of the reviewed P2P Grid middlewares are publicly available, with their source code released under a Free and Open Source license.

	Levels	Scheduling	Queueing support*	Task replication
AssessGrid	2	matchmaking	yes	no
DGET	1	flooding	no	no
JNGI	2	eager scheduling	no	yes
LBG	2	matchmaking	yes	not yet
OurGrid	2	matchmaking	no	yes
P2P Disco	1	flooding	no	no
P2P-MPI	1	matchmaking	no	yes
smartGRID	1 or 2	eager scheduling	no	no
Zorilla	1	flooding	no	no

* for external Tasks

Table 2.2: Architectural and scheduling features of P2P Grid middlewares.

	Bartering	Negotiation	Reservations	Checkpointing
AssessGrid	no	SLA	yes	yes
DGET	no	flooding	no	yes
JNGI	no	no	no	no
LBG	NoF	request	no	no
OurGrid	NoF	request	no	no
P2P Disco	no	flooding	no	no
P2P-MPI	no	request	no	no
smartGRID	no	request	no	yes
Zorilla	no	flooding	no	no

Table 2.3: Negotiation features of P2P Grid middlewares.

	Mitigation of free riding	Resource protection	Results validation	Secure communications
AssessGrid	yes	no	no	yes
DGET	no	yes	no	yes
JNGI	no	no	no	optional
LBG	yes	yes	not yet	not yet
OurGrid	yes	yes	no	optional
P2P Disco	no	no	no	no
P2P-MPI	no	no	no	no
smartGRID	no	no	no	no
Zorilla	no	yes	no	no

Table 2.4: Security features of P2P Grid middlewares.

	Data transfers	Data caching	Data-awareness	Code transfer
AssessGrid	centralized	decentralized	yes	yes
DGET	centralized	no	no	yes
JNGI	centralized	centralized	no	yes
LBG	P2P	decentralized	yes	yes
OurGrid	centralized	(de)centralized	optional	yes
P2P Disco	centralized	no	no	no
P2P-MPI	centralized	no	no	yes
smartGRID	centralized	decentralized	yes	yes
Zorilla	centralized	no	no	yes

Table 2.5: Data features of P2P Grid middlewares.

	App model	App language	Technology	In production	License
AssessGrid	Workflow	any	Java, CCS	no	n/a
DGET	BoT	Java	Java	no	n/a
JNGI	BoT	Java	Java, JXTA	possibly	JXTA
LBG	BoT	Java	Java	for LaBoGrid	GPL
OurGrid	BoT	Java, any	Java, RMI	Brazilian Grid	GPL
P2P Disco	BoT	Java	Java, Chedar	for U. Jyväskylä	n/a
P2P-MPI	Workflow Iter. Stencil	Java	Java, JXTA	possibly	GPL
smartGRID	BoT	Java, .Net	Java, web svc	no	n/a
Zorilla	IS	Java, any	Java, Bamboo	no	BSD

Table 2.6: Technology features of P2P Grid middlewares.

2.5.3 Comparison of LBG and OurGrid

Following this review of P2P Grid middlewares, it appears that LBG and OurGrid are the only reviewed P2P Grid middlewares that (like AssessGrid, which is not a P2P Grid) can enable decentralized large-scale Resource sharing between separate organizations Indeed, they are 2-level P2P Grids based on the Network of Favors model. As they use bartering, they do not have to rely on a centralized banking component (like AssessGrid) but they cannot (currently?) support SLAs (unlike AssessGrid).

Scheduling The scheduling model of LBG is more advanced that that of OurGrid, as it is structured around Policy Decision Points (PDPs) and supports queueing of external Tasks. The PDP-based model enables to easily plug new policies and thus constitutes a mechanism to systematically explore a large number of scheduling strategies. With the support for queueing of external Tasks, failures of execution of external Tasks can be hidden to consumer Peers. The support for queueing of external Tasks, opens the possibility to support batch scheduling and possibly, in the long term, planning and reservations as well.

Architecture In LBG, each Peer includes components for queueing, scheduling and negotiation. The User Agent middleware submits BoTs to the Peer at its site, then waits for the Peer to send completed results of each Task. The separation between the User Agents and the Peer to which they submit BoTs effectively separates the submission interface from the core Grid software. In OurGrid [233, 84, 286], negotiation is performed by each Peer, but scheduling is performed by each User Agent and there is no queue for external Tasks. This is different from LBG, where each User Agent essentially submits BoTs and is not involved in scheduling.

Software Engineering Furthermore, LBG and OurGrid are the only reviewed P2P Grid middlewares for which a discrete-event system simulator has been built. The simulator code of LBG, as discussed in Chapter 3, is deeply integrated with the middleware code, which enables reproducible testing.

Data Transfers Finally, only LBG supports P2P decentralized data transfers. At the implementation level, it is worth noting that OurGrid requires Users Agents to have O.S.-level access to Resources in order to transfer files across the P2P Grid. LBG relies only on its embedded data servers to transfer files.

2.6 Grid Application Model

Now that the state of the art in Resource sharing and existing P2P Grid middlewares has been reviewed, we describe the Lightweight Bartering Grid architecture, starting with its Grid application model in this section, then pursuing with the Resource middleware, Grid nodes messaging protocol and Peer middleware in the following sections.

The *Grid Application model* of LBG, is the *Bag of Tasks* (BoT). In this section, a taxonomy of Tasks and BoTs is first given to facilitate the reading of the remainder of the dissertation. Grid application programming is then discussed.

2.6.1 Taxonomy of Tasks

The *owner Peer* of a Task is defined as the Peer to which the Task has been submitted by one of its User Agents. The *runner Peer* of a Task is defined as the Peer which actually runs (on one of its Resources) the Task. The owner of a Task may negotiate the use of computational power from another Peer. In this case, it may submit the Task to this other Peer, which becomes the runner of the Task.

Tasks can be classified into Local Tasks, Consumption Tasks and Supplying Tasks according to both their owner Peer and runner Peer:

- A *Local Task* is a Task run[11] on one Resource of its owner Peer, which is therefore also its runner Peer.
- A *Consumption Task* is a Task that has been submitted by its owner to another Peer, as perceived by its owner. A Consumption Task is completed on the behalf of the owner Peer which consumes one Resource from the runner Peer.
- A *Supplying Task* is a Task that has been submitted by its owner to another Peer, as perceived by the runner Peer. A Supplying Task is completed by the runner Peer which supplies one Resource to the owner Peer.

A Consumption Task queued on a given Peer is owned by this Peer and run by a Resource supplied by another Peer. A Supplying Task queued on a given Peer is run by one of the Resources of the given Peer. To summarize, a given Task submitted by its owner to another Peer is labelled a Consumption Task by its owner and is labelled a Supplying Task by its runner Peer.

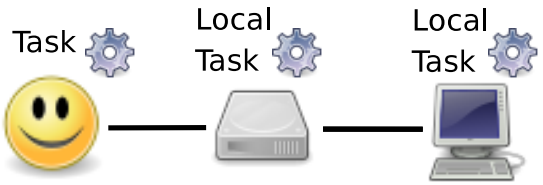
Initially, all the Tasks owned by a Peer are implicitly Local Tasks, until some of them are submitted to other Peers, at which point they are considered as Consumption Tasks by the owner and as Supplying Tasks by the runners. Figure 2.11 illustrates the three Task types.

With the proposed classification, the term designating each Task type totally determines the status (runner, owner) of a Peer that queues a given Task at some point in time. It is useful because the status of a Peer relative to a Task evolves with the movements of this Task around the P2P Grid.

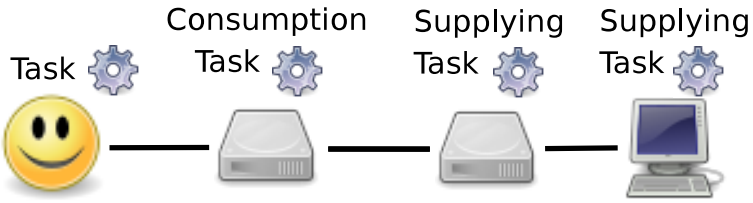
2.6.2 Taxonomy of Bags of Tasks

Bags of Tasks submitted by User Agents, then Bags of Tasks submitted by other Peers, are examined.

[11]The Resource middleware is running at OS-level on a computer. A Grid Task is running at Grid-level on the Resource middleware. At some point, a Grid Task is obviously executed as threads or processes of the OS of the computer where the Grid Resource is running. Explicit distinctions between computational tasks that are local to the computer where they are running (e.g. a word processor, an e-mail client, ...), and Grid tasks that are not local to the computer, could be made. However, such considerations are strictly limited to Section 2.7, as the interactions, on a given computer, between a Grid Task and non-Grid, OS-level "tasks" are outside the scope of this dissertation. All Tasks mentioned in this dissertation can safely be assumed to be Grid Tasks.



(a) without bartering



(b) with bartering

Figure 2.11: Tasks types (a) without bartering (b) with bartering.

Local BoT

Tasks, when submitted to a Peer, are always grouped into a Bag of Tasks (see Section 2.4). Bags of one single Task may still be submitted, though. A *Local BoT* (or *Local job*) is defined as a job composed of one Bag of Tasks that is submitted to a given Peer by one of its User Agents. The goal of a Peer is to achieve for each Local BoT the lowest BoT response time, which is the time between the submission of the BoT and the completion of all of its Tasks.

Each Task of a Local BoT is labelled either as a Local Task or as a Consumption Task, depending on the fact that it is being executed on its owner Peer or on another Peer. By default, all Tasks are Local Tasks, as explained in the previous paragraphs.

In this dissertation, there is only one class of priority among User Agents of a given Peer. Local BoTs submitted by multiple User Agents of a given Peer are queued and processed following a FIFO policy. That does not, however, preclude the concurrent execution of multiple Local BoTs when enough Resources are available (this is similar to FCFS-Share without replication [18] or S-BoT [176]).

Supplying BoT

Tasks are submitted to other Peers one by one. Each time a Task is submitted to a given Peer, this Peer creates an artificial job whose Bag of Tasks is actually a Bag of one Task. Such a job is called a *Supplying BoT* (or *Supplying job*), as its only Task is a Supplying Task (from the perspective of this runner Peer, and this Task is a Consumption Task from the perspective of its owner Peer).

Consequently, a Task submitted by a Peer to another Peer is represented twice in the system: once as a Consumption Task on its owner Peer, and once as a Supplying Task on its runner Peer.

Given the selected Grid application model and definitions of jobs, **the terms BoT and job are used interchangeably in the context of this dissertation**.

BoT Response Time

The *BoT response time* is the total time elapsed between the submission of a BoT to a Peer and the completion of all its Tasks. The *mean BoT response time* is thus the mean of response times of several BoTs.

2.6.3 Grid Application Programming

Tasks that can be submitted to LBG must currently be programmed in the Java language (J2SE 5.0 [178]). Java has excellent support for multithreaded and network programming, provides robust security mechanisms and is highly portable.

The requirement of programming Tasks in Java is a completely arbitrary choice[12]. The requirement for programming in Java depends essentially on one component of the LBG architecture: the Resource's runner VM (Section 2.7.4). With future work, support for Tasks programmed in other languages can definitely be added to LBG.

Any arbitrary Java application can easily be made ready to run on LBG provided that some limited management information is written by the application developer. The *gridification* of an application consists in wrapping it into a Local BoT. In turn, this Local BoT can be submitted to a Peer using an instance of the User Agent middleware.

I/O operations are restricted, essentially in the sense that input data files are made available to the Task and that access to the local file system is severely restricted (for a complete list, see Section 2.7.4).

A Grid application is a Bag of Tasks that can be composed of one or more Tasks. A simple Grid application can thus be a Bag of only 1 Task. The binary code of one Task is a set of Java classes packed into a Java archive (`.jar` file). The input parameters, input data files and binary code (`.jar` file) of each Task of

[12] This choice is motivated in the short discussion given at the end of Section 2.5.2.

a BoT must be declared in a job description file (see Appendix B.1) by the application developer. The job description file is given to the User Agent middleware.

Importantly, no estimation of Task completion time is required of the Grid application developer when writing the job description file. As this management information is notoriously hard to obtain [205] and often very unreliable when provided, the decision was made to not use it.

The Grid application developer must designate (in the job description file) a main class for each Task, where its entry point can be found. The designated main class of each Task must implement a simple interface, the *GridApplication* interface (see Table A.1 in Appendix A), including the `compute()` method which is the entry point of the Task. The requirement to implement the *GridApplication* interface enables the Resource middleware to automatically make available the input data files (that are transferred automatically by the Grid middleware) and input parameters to the Task. Through the *GridApplication* interface, the Resource middleware can then run the Task and, finally, retrieve any output data upon successful completion of Task execution.

2.7 Resource Middleware

The Resource is the basic building block of a Grid. The Resource middleware, required to make the computational power of a given computer available to the Grid, is described in this section.

A Resource has two main responsibilities, Task execution and data management, which are discussed. The life cycle of a Resource is then described.

2.7.1 Task Execution

Resources supply computational power to execute work units at the Task level. When a Task is scheduled to a Resource, the following sequence takes place:

1. The Java archive containing the code of the Grid application (see Section 2.6.3) and the input parameters, if any, are uploaded to the Resource;

2. Input data files - if any and if they are not already stored in the Resource data cache - are downloaded by the Resource, using separate threads so as not to block other operations like the reception of control messages sent by the owner Peer;

3. The Grid application embedded in the Task is run in a separate Java Virtual Machine (Java VM) controlled by a helper thread of the Resource [99];

4. Data results that were computed by the Task are uploaded to the owner Peer (currently inline with the Task itself, although this limitation will be removed in the near future).

Data results are then forwarded from the owner Peer to the consumer Peer if the Resource was supplied to process a Supplying Task.

LBG supports dynamic code uploading . It means that the binary code of a Task is uploaded to the Resource where it is executed. In practice, this precludes the need for out-of-Grid access, which is a huge benefit not available in all existing middlewares. For instance, OurGrid requires that Grid users have an O.S.-level access to worker nodes via ssh (LBG only requires Grid system administrators to have an O.S.-level access).

2.7.2 Task Preemption and Cancellation

A Peer can preempt or cancel, at any time, any Task running on any of its Resources. *Task preemption* from a Resource is defined as halting the execution of the Task currently running, with subsequent requeueing by the Peer. A preempted Task may be rescheduled and completed on another Resource of the owner Peer.

Task cancellation from a Resource is defined as halting the execution of the Task currently running, without subsequent requeueing. A cancelled Task cannot be completed by any Resource of the owner Peer. If a Supplying Task is cancelled, it may still be completed, either by the consumer, by another supplier Peer, or even by the supplier Peer on which it was cancelled, provided that it is submitted again. In case of a Supplying Task, a cancellation decision may come from either the supplier Peer or the consumer Peer, e.g. because Task execution has timed out.

As **there is no middleware-level checkpointing support** in the current specification of LBG, the Task execution state and any intermediate **results are lost upon Task preemption/cancellation**. This causes a loss of computing time as well as other undesirable consequences that are discussed in Chapter 4. However, an application-level P2P-aware P2P checkpointing mechanism will be introduced in Section 6.2.

A Resource can also preempt or cancel the Task which is currently running. In case of an imminent, predictable failure or when an administrator brings it offline, a Resource preempts the currently running Task, if any. It then tries to contact its

owner Peer to communicate that the Task has been preempted, in order to maintain the consistency of Peer state. As will be seen in the following (Section 2.9.6), the Peer cannot rely only upon the Resource-initiated preemption mechanism to maintain consistency, but it is nonetheless helpful in practice.

2.7.3 Resource Life Cycle

Resource Registration Each Peer tracks the state of the Resources it manages (see Section 2.7.6) so that it can identify idle Resources and busy Resources. Resources must thus be registered with their owner Peer before they can be used. Resource registration in Desktop Grids or mainstream Grids is typically Peer-initiated due to a centralized site-level deployment model that is not adapted to a P2P Grid. Resource registration is thus Resource-initiated in our proposed P2P Grid architecture[13], similarly to Volunteer Grids. When a Resource comes online, it registers itself with its owner Peer.

Resource Selection and Matchmaking Resource selection is performed by the scheduler of the Peer middleware. Hardware features, such as a limit on the amount of RAM and a limit on the storage space available, can be described in a Resource configuration file. However, matchmaking based on hardware features (see Appendix D.2) is currently not supported by the LBG middleware. In practice, it can certainly be implemented as a filtering step during Resource selection.

Resource-level Task Queueing As our work focuses mainly on the higher layer of the Grid architecture (i.e. collective layer [150], also called management layer) rather than on lower-level layers which are concerned with communication and execution mechanisms, local, i.e. Resource-level, schedulers are not available. More importantly, Tasks should never be queued on Resources because of the intrinsic unreliability of P2P Grid environments.

Resource-level Task Execution Concurrency A Resource is always either idle or busy: **A Resource can run at most one Task at any time.** A Peer cannot run a Task on a busy Resource, but it can cancel a Task running on a busy Resource. The current scheduling model (that will be introduced in Section 2.9.4) would have to be extended to support the concurrent execution of multiple Tasks on a Resource. Adding support for the concurrent execution of multiple Tasks by a Resource introduces different concerns for multi-core CPU and single-core CPU:

[13]In practice, deployment scripts are also provided with the LBG middleware so that site administrators can deploy multiple Resources of their site.

- For computers based on multi-core CPU, it can be hypothesized that there is enough RAM to accommodate the processing of one computational task on each CPU. A simple, widespread technique to support concurrent Task execution on Resources based on multi-core CPU is to run multiple instances of the Resource middleware, one on each core. This is already possible with the current implementation of LBG, because it does not conflict with the current scheduling model. True multi-core support or support for parallel computers can also be integrated directly into the Resource middleware of a P2P Grid [94, 106], although this would not be trivial to integrate with the Peer middleware.

- For computers based on single-core CPU, the situation is different. It might not be efficient to run multiple Tasks concurrently because Resources of a P2P Grid might not have enough available RAM to process concurrently two or more computational tasks. The effect of two or more Tasks competing for RAM space can quickly lead to thrashing, which would be disastrous for performance. Supplementary research is required in Peer scheduling models that integrate multi-slots Resources, as well as in *out of core* algorithms designed to incrementally load data into RAM. Moreover, support from the operating system should be available to dynamically probe useful management information such as system load, available memory, ... In a nutshell, supporting the concurrent execution of multiple Tasks on Resources based on a single-core CPU is currently precluded by the quantity of RAM available on edge computers.

2.7.4 Resource Protection

The dynamic code uploading mechanism in LBG follows the *on demand code uploading* paradigm, a form of weak code mobility [73, 301]. Weak code mobility means that only the application code is transferred, not its state. Strong code mobility requires both code and state to be transferred. As an example, the Condor middleware [91] and Fortino et al.'s *mobile agents* paradigm [144] both exhibit strong code mobility. Weak code mobility could be substituted with strong code mobility if middleware-level checkpointing support were added to the LBG middleware. However, it must be remarked that the requirements, in terms of storage space, of strong code mobility might not be met by every Resource in a P2P Grid.

With on demand code uploading, there is no need to deploy Grid applications in advance. Grid application deployment is totally independent from human users, Grid administrators and application developers. However, supplementary security measures are required to protect Resources from malicious Tasks.

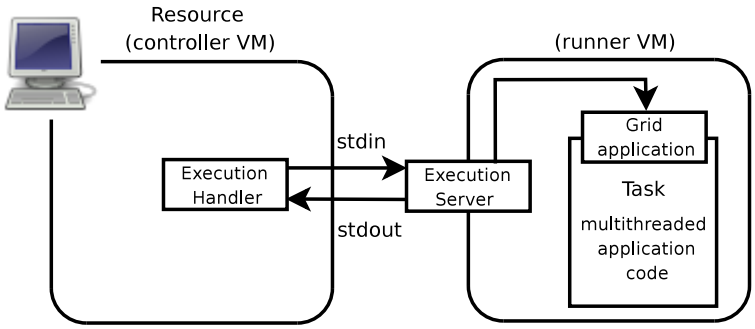


Figure 2.12: The controller VM launches and controls a runner VM for each Task.

Several security measures are taken:

- Each Task is actually executed in a Java Virtual Machine (VM) that is dynamically launched[14] (runner VM) and thus is distinct from the VM running the Resource middleware (controller VM), as illustrated on Figure 2.12;
- The controller VM configures the runner VM to enforce a security policy [111] so that Task execution is sandboxed, e.g. access to local file systems, devices, environment variables, system properties and VM class loader are severely restricted;
- The runner VM is launched with an available amount of RAM that is set by the Resource human administrator when the Resource is deployed;
- As a Task may need to write to and subsequently read some temporary files, a *playpen* [85] directory is provided;
- Task control, i.e. Task preemption/cancellation after a timer has expired, is implemented in the Peer middleware (and will be discussed in Section 2.9.6).

Other security measures should also be taken as the development of other technologies allows:

- As, on one hand, BitTorrent and FTP software components (see Section 2.7.5, Chapter 5) - and maybe also Grid application themselves (see Section 6.2) - heavily depend on Internet access and as, on the other hand, issues related to Grid-aware filtering of network connections constitute a very recent research domain, network connections are currently not filtered by the runner VM.

[14]This design choice offers better security than if the Grid application were run in a thread of the same VM because the runner VM is running in a different context than the controller VM. This is a typical choice, also followed by OurGrid with the SWAN mechanism [79]; SWAN depends on an O.S.-level VM rather than on a Java VM. The temporal overhead of launching a Java VM is negligible as most Grid applications run longer than a few seconds.

- On-the-fly verification of code properties [88, 223] and security of execution [212, 213], i.e. protection of a Task against the Resource that runs it, are not supported but would definitely need to be further investigated.

2.7.5 Resource Data Management

The Resource Data Manager is the component responsible for managing the input data of queued Tasks [56, 57]. Each Resource Data Manager is equipped with a data cache, which is the software component managing data storage. It is also equipped with BitTorrent and FTP software components to manage data transfers.

Basically, the Resource Data Manager purpose is to download input data files from its owner Peer or from other Resources, store and share them with other Resources processing Tasks depending on them. It is discussed in Section 5.2.2, as data-related operations are largely independent of other operations.

2.7.6 Resource Management System (Peer Middleware)

The Resource Management System (RMS) - that is a component residing in the Peer middleware, not in the Resource middleware - is responsible for the control and management of the state of the Resources, from the Peer perspective:

- Resource registration: Resources register themselves with their owner Peer when they come online (see Section 2.7.3);
- Resource selection: selection of an idle Resource following a ranking-based or random policy, so that the Peer scheduler (see Section 2.9.4) can match it with a given Local or Supplying Task;
- Resource preemption/cancellation: selection of a busy Resource in order to preempt or cancel the execution of one running Task (see Section 2.7.3).

On each of these operations[15], the Peer RMS updates the Resource status and data cache contents (see Section 5.2.2) in its internal Resource database.

[15]No lower-level mechanism of continuous verification of the Resource liveness, i.e. so-called heartbeat mechanism, is currently implemented, although it could certainly be done.

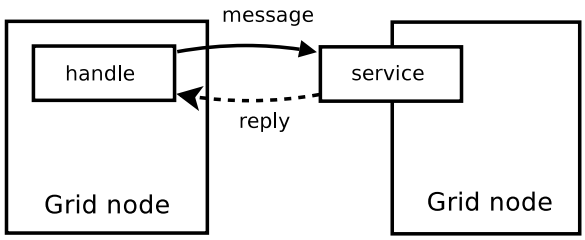


Figure 2.13: Handle/Service pattern.

2.8 Grid Node Messaging Protocol

A simple messaging protocol has been designed to enable message passing between the Grid nodes (Peer ↔ Peer, Peer ↔ Resource, Peer ↔ User Agent). The current implementation of the *Grid Node Messaging Protocol* (GNMP) is based on serialized Java objects transmitted over non-persistent TCP connections.

The requirement of having Grid nodes support a Java-based implementation is a completely arbitrary choice[16]. The requirement for implementing Grid nodes in Java before they can be deployed as part of an LBG-based P2P Grid will be removed in future work that will lead to a technology-neutral definition of GNMP.

Handle/Service Pattern

The protocol follows the handle/service pattern (see Figure 2.13). A *service* of a Grid node is a processor of messages sent by other Grid nodes. A service may or may not transmit back a return value (a reply) to its sender. Each Grid node is equipped with one or more service components, one for each type of Grid nodes to communicate with. To promote a fully asynchronous behavior, communications between Peers should remain unidirectional, i.e. not require replies.

Each service is able to generate handles, that are Java objects encapsulating the necessary logic and data to communicate with the service. To contact a service on the Grid, a Grid node uses a handle. Initially, when coming online, a Resource or a User Agent can download a handle directly from the Peer that it seeks to contact. A Peer can download handles of other Peers from a Search Engine. To do so, the only needed data are IP address and TCP port of the service, that is implemented as a TCP server socket. All the subsequent communications messages sent to a particular target Grid node are initiated through the obtained handle.

[16] This choice is motivated in the short discussion given at the end of Section 2.5.2.

A *message* is a serialized Java object [178, 49] storing the data necessary to invoke a method of a service, i.e. a method name, and method parameters types and values. Upon reception of a message by a service, the Java objects are deserialized and dynamically cast to the appropriate type. The service then calls the processing method, which may send back a Java object as a reply.

Grid Nodes Services

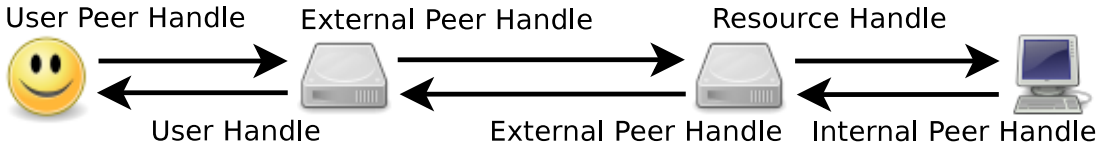


Figure 2.14: Handles/Services types.

There are five implemented service types (see Figure 2.14), each able to process messages from a given type of node to another given type of node:

- each Peer runs three services (see also Figure 2.15), enabling it to be contacted by its User Agents (User Agent Peer Handle/Service), its Resources (Internal Peer Handle/Service) and other Peers (External Peer Handle/Service);
- each User Agent runs a service enabling it to be contacted by the Peer it uses (User Agent Handle/Service), e.g. to upload results of a completed BoT;
- each Resource runs a service enabling it to be contacted by the Peer that owns it (Resource Handle/Service), e.g. to run a Task.

In Appendix A, Tables A.3, A.2 and A.4 (see Appendix A) summarize the operations supported by Resources, User Agents and Peers, respectively. To allow multithreaded processing of incoming GNMP messages, each service thread is augmented with a pool of helper threads.

Related Control Protocols

The proposed control protocol is simple but also sufficient to support the experiments performed for this dissertation. To deploy the LBG middleware across multiple organizations, communications authentication, and thus encryption, should be supported by GNMP (this is purely an implementation issue, as the design of GNMP can perfectly support them). Existing messaging protocols that can be used instead of GNMP include the following.

Java Remote Method Invocation [180] (RMI) could be used, but our proposed lightweight protocol is easier to deploy (it does not need an RMI registry). The recent XMPP-based [136, 137] JIC [211], and ICE [167], support important security features such as communications authentication and encryption. The Web Services Resource Framework (WS-RF) [231, 26], announced as a successor to OGSI [229], could also also probably be used. However, protocols based on web services are only beginning to exhibit decent performance.

2.9 Peer Middleware

In LBG, Peers use their Resources to run Tasks and *Peers barter computing time at the Task level*[17] so that other Peers also run their Tasks. The goal of Peers, as explained, is to complete Bags of Tasks as fast as possible. The challenge in designing the Peer middleware resides in controlling the flow of Tasks across the P2P Grid so that the impact of several disruptive issues - such as asynchronous communications, Peer and Resource unreliability, queueing delays, overly long Task executions, and last but not least Task preemption and cancellation - is mitigated.

Examples of typical interactions between Grid nodes are first provided to help understand the basic operation of a Peer. A bartering model, a scheduling model and a negotiation model are then proposed. Several issues, such as negotiation control, Task control, concurrency management are then discussed. Data management is also briefly discussed.

2.9.1 Typical Interactions

A Peer can be modelled as a system receiving and processing computational requests. As defined in Section 1.1.4, each Peer acts in its own interest, i.e. complete Tasks submitted by its User Agents, but may cooperate with other Peers, i.e. complete Tasks submitted by other Peers or submit its own Tasks to other Peers, by exchanging computing time.

Typical Interactions (without bartering)

A Peer receives computational requests, i.e. Local BoTs, from User Agents. It schedules Tasks of these Local BoTs to the Resources it controls. When a Resource completes a Task, it uploads the output data to its owner Peer, which forwards it to the appropriate User Agent. The Peer then updates its internal state. Figure 2.11a

[17] A computational request sent by a consumer to a supplier is a Task, not a Grid application.

(p. 49) illustrates typical interactions involving no bartering. When the execution of a Task is preempted by the Resource on which it is running or by the Peer, the Peer also updates its internal state. When one of its Resources becomes available, the Peer can schedule queued Tasks.

Typical Interactions (with bartering)

During peaks of computational requests, a Peer can also schedule Tasks to other Peers as Consumption Tasks. A supplier Peer can accept to queue a Supplying Task sent by a consumer Peer. When a Peer can schedule Tasks to its Resources, it first schedules Local Tasks then Supplying Tasks. When a Resource completes a Supplying Task, it uploads the output data to its owner Peer, which forwards it to the consumer Peer. In turn, the consumer Peer forwards it to the appropriate User Agent. Both the supplier and the consumer Peer update their internal state. Figure 2.11b (p. 49) illustrates typical interactions involving bartering. When the execution of a running Task is preempted by the Resource on which it is running, the supplier Peer or the consumer Peer, both Peers also update their internal state.

2.9.2 Peer Service

As explained in Section 2.8, interactions between Grid nodes - e.g. a Peer with another Grid node - are carried through the Grid Node Messaging Protocol. GNMP messages are considered by a Peer as external events to process. There are three classes of external events, one for each possible type of sender Grid node.

External events generated by User Agents:

- submission of a Local BoT,
- query of the status of a Local BoT,
- (User-initiated) cancellation of Local BoT.

External events generated by Resources:

- upload of results of a completed Local or Supplying Task,
- (Resource-initiated) preemption of a Local or Supplying Task,
- Resource registration/deregistration.

External events generated by (other) Peers:

- submission of a Supplying Task,

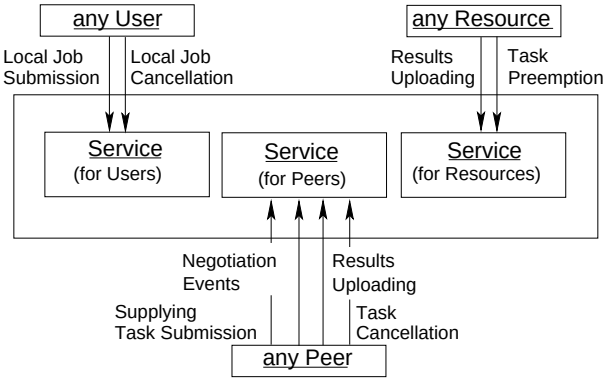


Figure 2.15: External events service model.

- upload of results from a completed Consumption Task,
- (Peer-initiated) cancellation of a Task (any type),
- supplying request (see Section 2.9.5),
- consumption grant (see Section 2.9.5).

The *Peer service* is the component of the Peer middleware processing external events. It is illustrated on Figure 2.15. Three Peer service subcomponents - corresponding to the three external events classes - are represented within a frame symbolizing the Peer service. External event classes are represented as sets of arrows connected to the corresponding Peer service subcomponent. Each individual arrow symbolizes one external event type within the relevant class. A complete listing of the Peer service interface is provided in Appendix A.2, which also provides the interfaces to the User Agent service and to the Resource service.

2.9.3 Bartering Model

Queueing support for Supplying BoTs as well as an explicitly defined bartering model constitute original contributions over existing bartering models.

BoTs Queues

The Queue Manager is the Peer component responsible for the queueing of Local and Supplying BoTs (i.e. Bags of Tasks, see Section 2.6). It maintains two queues from which Tasks can be selected for scheduling: one for Local BoTs and the other for Supplying BoTs. Local BoTs are queued into the *Local BoTs queue*. Accepted Supplying BoTs are queued into the *Supplying BoTs queue*.

The Tasks from each Local BoT can be partitioned into three disjoint sets (see Figure 2.16). There is at least one set nonempty at all times, as every Task of a BoT is always in one of three states: waiting, scheduled or completed.

The Local BoTs queue is composed of three sets: submitted, waiting and scheduled BoTs. All submitted Local BoTs are accepted, as the purpose of a Peer is to serve User Agents. A waiting Local BoT does not have any Task scheduled, while some of Tasks of a scheduled Local BoT are scheduled. A Local BoT is returned to the User Agent that submitted it once all its Tasks are completed or it has been cancelled (by the User Agent). Figure 2.17 illustrates the Local BoTs queue.

Some Tasks of Local BoTs may be scheduled to other Peers as Consumption Tasks. In this case, a Task being scheduled has the meaning that the Consumption Task has indeed been scheduled to a supplier Peer, but may be queued in the supplier's Supplying BoTs queue, without being actually scheduled (yet) to one of the supplier's Resources. Another distinction between Local Tasks and Consumption Tasks is that the former may be either preempted or cancelled, while the latter may only be cancelled (by their consumer Peer).

The Supplying BoTs queue is composed of three sets: submitted BoTs, waiting BoTs, scheduled BoTs. Accepted supplying BoTs are moved to the set of waiting BoTs; those that are rejected are returned to the consumer Peer that submitted them. A waiting Supplying BoT does not have its only Task (see Section 2.6) currently scheduled, while a scheduled Supplying BoT has its only Task currently scheduled. A Supplying BoT is returned to the consumer Peer that submitted it once its Task has been completed or it has been cancelled (either by the supplier Peer or the consumer Peer). Figure 2.18 illustrates the Supplying BoTs queue.

Bartering States

The bartering state of a Peer can be modelled with the states of its two BoTs queues. Each state is symbolized by a couple of integers (l, s) which represent the length of the Local BoTs queue and the Supplying BoTs queue, respectively. To guide the discussion of bartering, a simple 4-states model is illustrated on Figure 2.19. The transitions between states occur following conditions that are controlled (plain lines) or that cannot be controlled (dashed lines) by the Peer.

Initially, a Peer starts in the $(0, 0)$ state. It can accept all incoming Supplying Tasks. If Local Tasks are submitted to the Peer, it switches to the $(many, few)$ state. If Supplying Tasks are submitted to the Peer, it switches to the $(few, many)$ state.

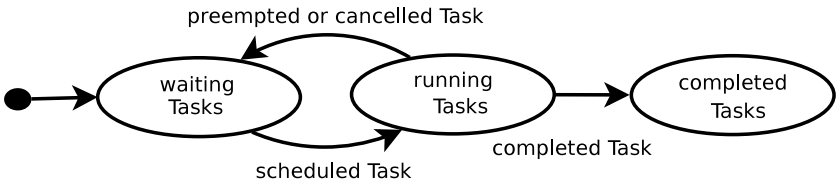


Figure 2.16: States of Tasks of an accepted BoT (i.e. a BoT which is either in the set of waiting BoTs or in set of BoTs with scheduled Tasks).

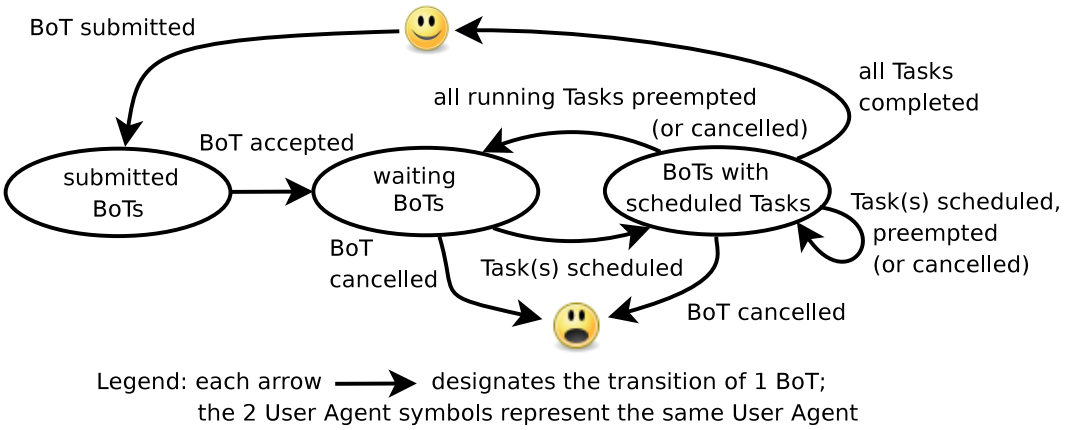


Figure 2.17: Queue of Local BoTs (in a Peer, + interactions with User Agents).

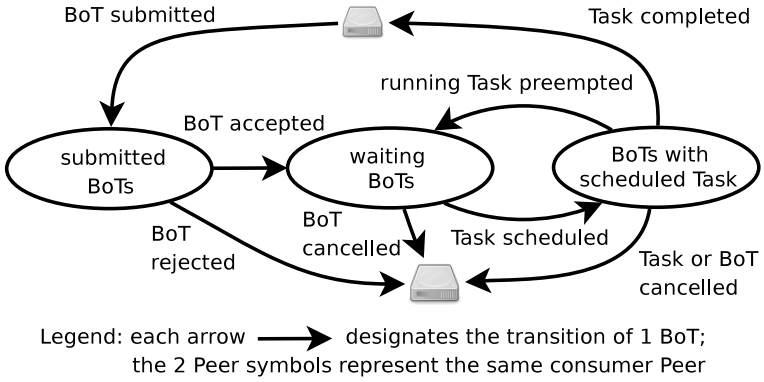


Figure 2.18: Queue of Supplying BoTs (in a Peer, + interactions with consumers); each Supplying BoT is composed of exactly 1 Task.

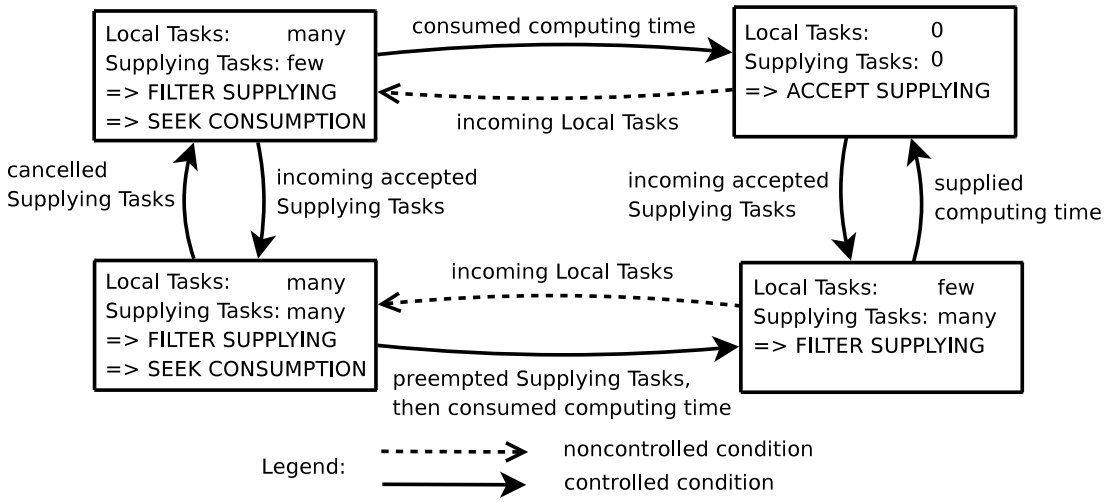


Figure 2.19: Peer bartering states (in function of the length of BoTs queues).

When switching to the $(many, few)$ state from the initial state, a Peer may initially have enough Resources to schedule all the queued Local and Supplying Tasks. But other Peers may submit many Supplying Tasks. If these are not filtered, the Peer switches to the $(many, many)$ state.

In the $(many, few)$ state, a Peer seeks to complete the Local Tasks first, before any of the few Supplying Tasks that could be present (if the Peer was previously in the $(many, many)$ state). Indeed, the purpose of the Peer is to achieve short response times of Local Tasks (see Section 2.1.2). If not enough of its own Resources are available, a Peer should both seek to consume computing time from other Peers and deny other Peers to consume its own computing time. This may help to shorten the transition back to the $(0, 0)$ state.

It is undesirable for a Peer to be in the $(many, many)$ state in the sense that an extended period of queueing of Supplying Tasks contributes to long response times. The Peer is therefore going to be perceived as a slow supplier by other Peers. The Peer can try to complete all Tasks as fast as possible:

1. It first can use the computing time of its Resources;
2. It can then preempt running Supplying Tasks (thus increasing even more the response times of Supplying Tasks) to free some of its busy Resources and use their computing time;
3. Finally, it can try to consume computing time from other Peers.

This will eventually make the Peer switch to the $(few, many)$ state. The Peer can also cancel rather than preempt the Supplying Tasks (either only the running

Supplying Tasks or also the waiting Supplying Tasks). But in this case, the Peer is perceived as unreliable by the consumer Peers.

In the $(few, many)$ state, a Peer seeks to complete the few queued Local Tasks first, and supplies most of its computing time to Supplying Tasks. Maybe counter-intuitively, it is desirable for a Peer to be in this state, in the sense that it builds a consumption potential with other Peers. In this state, a Peer can accept many Supplying Tasks up to a limit, so as to not impact their response times. If Local Tasks are submitted, the Peer switches to the $(many, many)$ state. When there are no more queued Tasks, the Peer returns to the initial state.

2.9.4 Scheduling Model

Task scheduling consists in matching queued Tasks and available Resources (or supplier Peers), which typically involves a Task selection policy and a Resource (or consumer Peer) selection policy. Policies of interest are those that ensure short MBRTs under most circumstances.

The Scheduler is the Peer component responsible for the selection and matching of Tasks and Resources. It relies on the RMS (Section 2.7.6) and the Queue Manager (Section 2.9.3).

Scheduling Policy Decision Points

We propose a formal scheduling model based on control points where scheduling decisions are taken to influence the flow of Tasks through the Peer [312]. This constitutes an original contribution over existing scheduling models in P2P Grids.

In each Peer, there are basically two queues of computational requests: the Local BoTs queue and the Supplying BoTs queue. BoTs flow through a Peer from Users/other Peers and to Resources/other Peers, depending upon Tasks types (see Section 2.6.1).

Policy-based decisions are made at five control points, called the *scheduling policy decision points* (scheduling PDP). Figure 2.20 illustrates the PDPs that control the flow of Tasks flowing through the queues of a Peer. Figures 2.21- 2.23 highlight for which fault-management mechanisms the PDPs are intended.

The scheduling PDPs are:

1. Local Tasks scheduling: Matches Local Tasks and Resources;
2. Consumption Tasks scheduling: Matches Local Tasks and supplier Peers, and schedules the former to the latter, as Consumption Tasks;

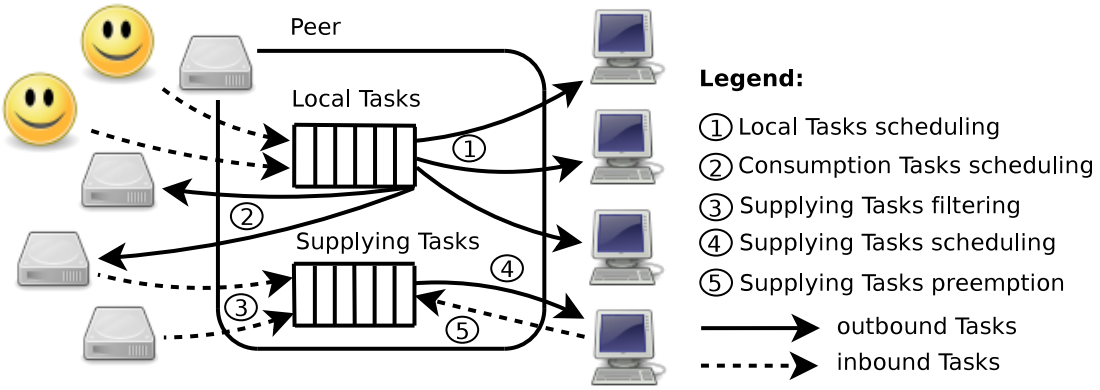


Figure 2.20: Scheduling model.

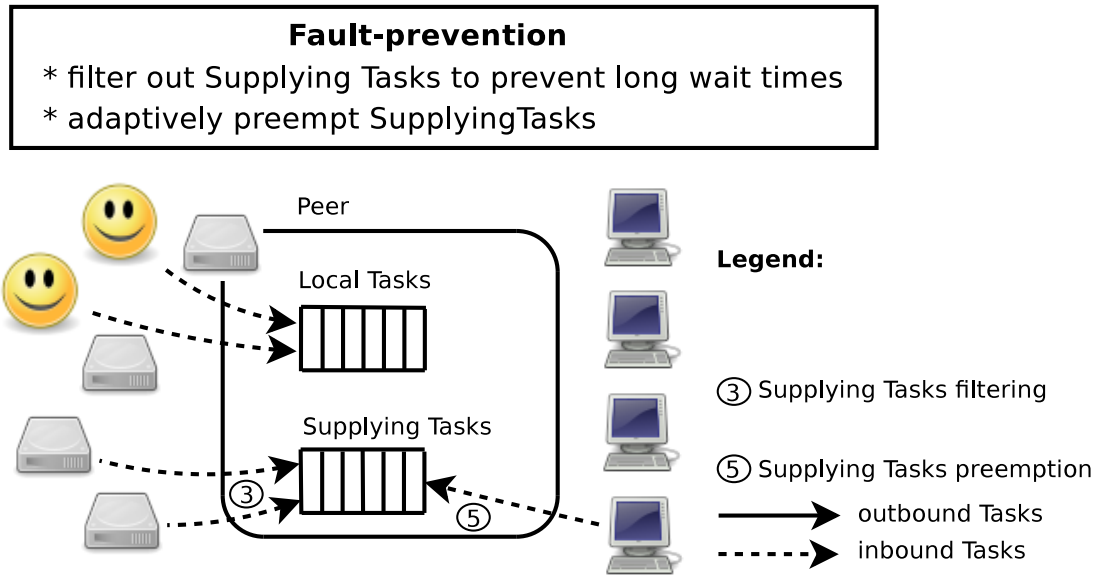


Figure 2.21: Scheduling model (fault-prevention mechanisms).

3. Supplying Tasks filtering: Filters submitted Supplying Tasks[18];
4. Supplying Tasks scheduling: Matches Supplying Tasks and Resources, i.e. supplies computing time to consumer Peers;
5. Supplying Tasks preemption: Preempts Supplying Tasks from Resources so as to instantly recover a full computational power.

Scheduling policies are proposed for each policy decision point, in Chapter 4.

[18] As bartering is conducted at the Task level (see Section 2.6), Supplying BoTs are always composed of exactly one Supplying Task.

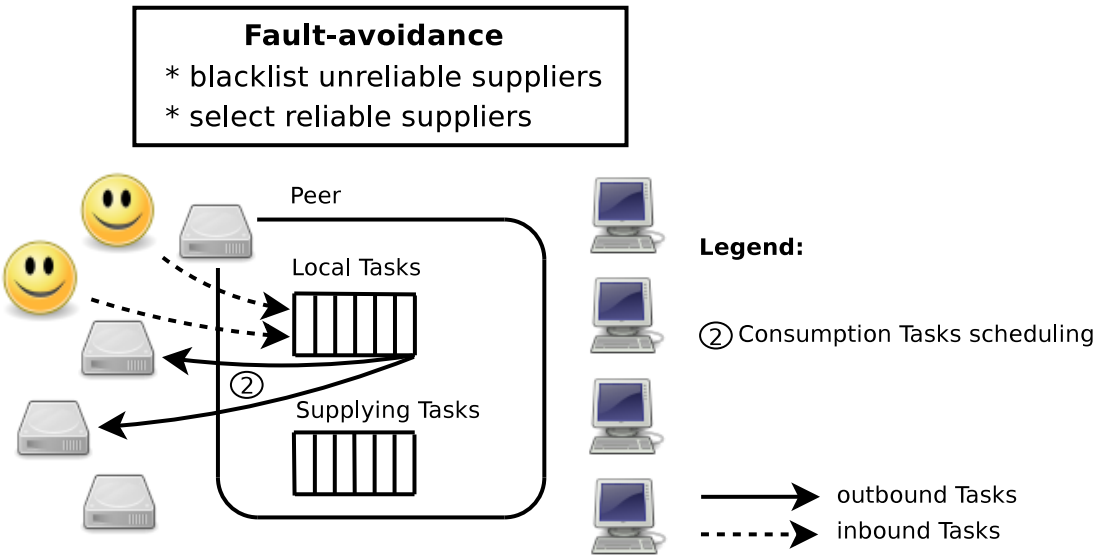


Figure 2.22: Scheduling model (fault-avoidance mechanisms).

The Supplying Tasks filtering policy is activated immediately upon reception of a submitted Supplying BoT. This filtering prevents the Peer to be flooded by too many simultaneous submissions of Supplying Tasks. It also protects the Peer from middleware-level Denial of Service attacks by malicious Peers.

The three Tasks scheduling policies perform the scheduling of the three Tasks types (see Section 2.6.1). They are activated following the various external events processed by the Peer service (see Section 2.9.2), such as submission of a Local BoT, completion of a Supplying Task, availability of additional Resources, ... They may also be activated following internal events, such as time-outs, as will be explained in Section 2.9.6. A complete mapping of events to scheduling policies is available in Appendix C.2.

Local Tasks are scheduled first to the Peer's own Resources because failure of local Resources is likely than preemption of Consumption Tasks by the supplier Peers. BoT selection at the queue-level is FIFO, Task selection at the BoT-level is FIFO or ranking-based, Resource selection is random or ranking-based.

Local Tasks are scheduled before, and possibly instead of Supplying Tasks, because of the objective a Peer, which is to complete as soon as possible its own BoTs. If a preemption policy is activated, Supplying Tasks are preempted so as to immediately schedule a maximum of Local Tasks.

Even after preemption, there are often not enough Resources to schedule all waiting Local Tasks. If possible, these are then scheduled as Consumption Tasks to

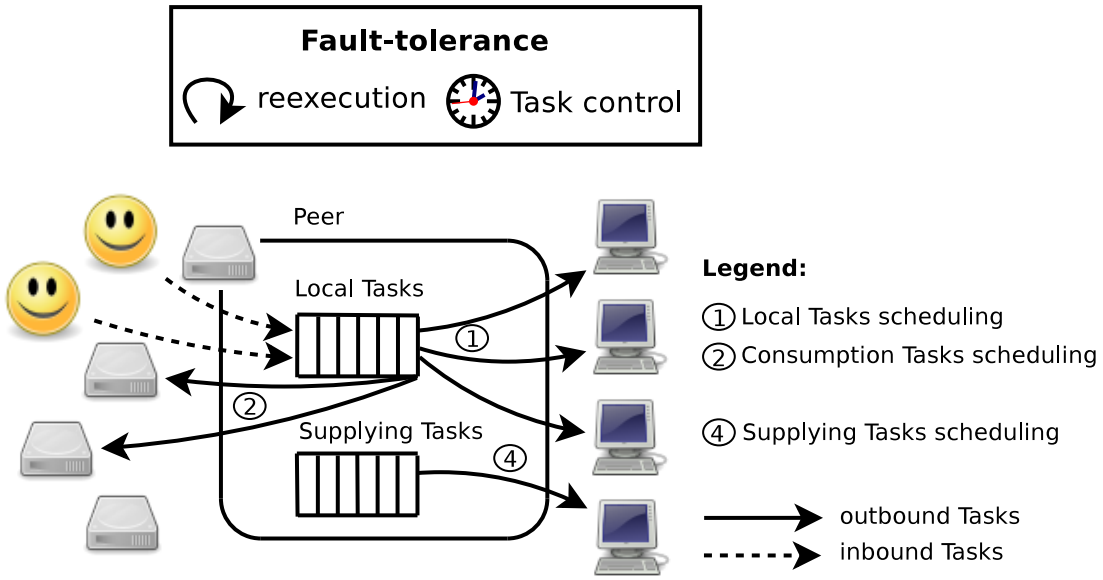


Figure 2.23: Scheduling model (fault-tolerance mechanisms).

other Peers with FIFO BoT selection at the queue-level, FIFO or ranking-based Task selection at the BoT-level, random or ranking-based consumer Peer selection.

If some Resources are still available after all the Local Tasks have been scheduled, waiting Supplying Tasks are scheduled with FIFO or ranking-based BoT selection at the queue-level (there is only one Task Supplying Task in each Supplying BoT), random or ranking-based Resource selection.

If there are not enough available Resources to schedule Supplying Tasks, some of them remain in the Supplying BoTs queue until scheduling is next triggered. In LBG, a Peer does not subcontract Supplying Tasks to other Peers, which means that Supplying Tasks are not forwarded to other Peers as (fake) Consumption Tasks.

In LBG, forwarding of Supplying Tasks may indeed lead to pathological situations such as cyclic bartering. It would also increase the complexity and reduce the robustness of the bartering and scheduling models. For example, in the situation where an intermediate Peer in a bartering chain suddenly would unexpectedly go offline, communication would be lost between endpoints, resulting in a cascade of Task execution failures. Such a situation would often occur in practice because bartering operations are long-running as opposed to, say, the transmission of a single TCP packet. We thus leave subcontracting and Task forwarding aside.

The works of Beaumont et al. [38] and Kreaseck et al. [199] are based on a scheduling model that supports Task forwarding and that is also based on the PDPs to control the

flow of Tasks. However, their work, although relevant and full of insights that would be worthwhile to consider in the context of this dissertation, cannot be applied to P2P Grids because it is situated in the context of hierarchically organized Grids, with strong assumptions of trust between Peers (as opposed to the hypothesis of informational opacity between Peers in LBG).

Reexecution of failed Tasks is a classic fault-tolerance mechanism [71, 37, 21] that is provided in the LBG architecture. Tasks requeued following Task execution failure are always[19] reexecuted until they are eventually completed.

Task replication [86, 308] consists in having each Peer schedule multiple replicas of Local Tasks, either to its own Resources or to suppliers Peers as Consumption Tasks. It clearly brings fault-tolerance but at a cost. It is currently not supported in the LBG architecture.

Finally, our proposed scheduling model compares to the scheduling model of OurGrid as follows. In OurGrid, there is no Supplying Tasks filtering PDP as there is no Supplying Tasks queue. For the same reason, the Supplying Tasks scheduling and preemption PDPs are very basic. Furthermore, there is no possibility to plug new policies into PDPs in OurGrid, as there is no explicitly defined PDP in the OurGrid scheduling model.

2.9.5 Negotiation Model

We suggest that each consumer Peer should first probe the availability of supplier Peers before submitting Consumption Tasks. Indeed, although a supplier Peer with few available Resources may directly reject the submission of a Consumption Task, it may also accept it and queue it. In the latter case, the submitted Task may therefore wait a long time before being actually scheduled.

To avoid as much as possible the submission of Consumption Tasks to busy Peers, a simple (i.e. not involving counter-bids) negotiation protocol is introduced. Consumers send *supplying requests* to suppliers (as a number of Tasks to submit). As a reply, suppliers may send *consumption grants* (as a number of currently available Resources) to signal the availability of Resources. Consumption grants represent the state of the Resources of a Peer, at negotiation time. Upon reception of consumption grants, a consumer selects to which supplier to submit Consumption Tasks.

[19] As a practical measure, a limit could be set to cancel BoTs with Tasks that repeatedly never complete, for example due to bugs. This is also discussed in Section 2.9.6.

Negotiation is a memoryless process. Both the submission of supplying requests and the evaluation of consumption grants have an informative, non-binding purpose. They thus can be activated at will. Negotiation is both intentionally simple and optional. As no estimates of Tasks runtimes are required of Grid application developers (see Section 2.6.3) and as Peers do not have direct access to one another's metadata, the negotiation object is a number of Tasks to submit. Negotiation is optional because a P2P Grid can operate without it (with the support of a timer-based Task control mechanism, as will be explained in Section 2.9.6), and because we want to keep it lightweight (see Section 2.5). These reasons also motivate the choice to not support an *advanced reservations* mechanism [281].

Negotiation Protocol

The Negotiator is the Peer component responsible for the emission and evaluation of supplying requests and consumption grants. Each time a Peer needs access to external Resources, its Negotiator sends to other Peers one supplying request for s Resources. The Peers that were interrogated may reply with a consumption grant which is a number c of currently available Resources, with $0 < c \leq s$. A Peer evaluating supplying requests distributes as most as many consumption grants as there are available Resources, i.e. $c \leq a$. This implies that if $a > 0$ then $0 < c \leq \min(a, s)$.

More recent information is also likely to be more relevant. Thus if multiple supplying requests from a given Peer are received before all pending supplying requests are evaluated, only the most recent supplying request is taken into account, while the previous ones are discarded. The same holds for consumption grants. As supplying requests and consumption grants provide a summarized representation from the state of a Peer, more recent information is also likely to be more relevant.

Peers do not forward supplying requests or consumption grants for the same reasons that they do not forward Tasks (see Section 2.9.4). Moreover, such forwarding would require a secure protocol like SHARP [82] to avoid malicious interference.

Figure 2.24 illustrates a typical bartering sequence between two Peers, with negotiation. The arrows between the two Peers symbolize GNMP messages communicating external events (see Appendix C.2).

Negotiation Policy Decision Points

The control points where negotiation decisions influencing the flow of Tasks through the Peer are now determined.

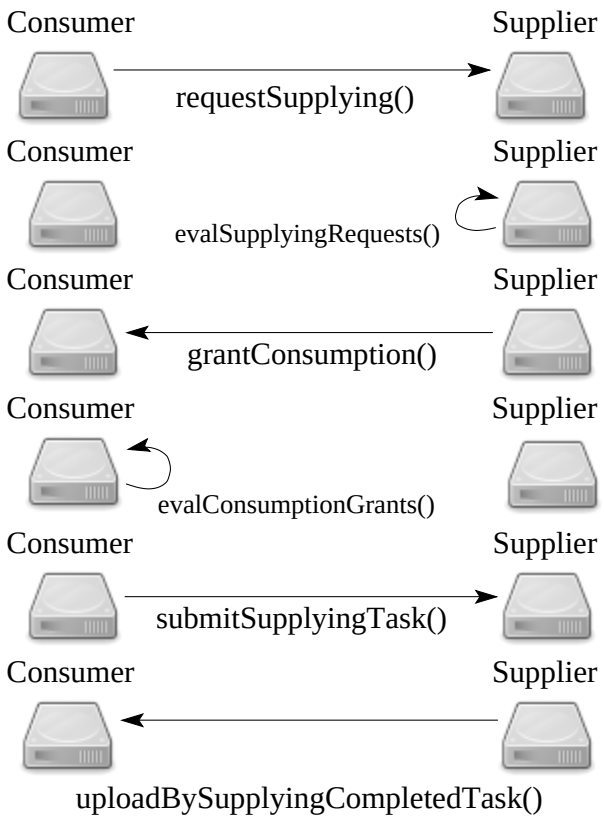


Figure 2.24: Typical bartering sequence between two Peers, with negotiation.

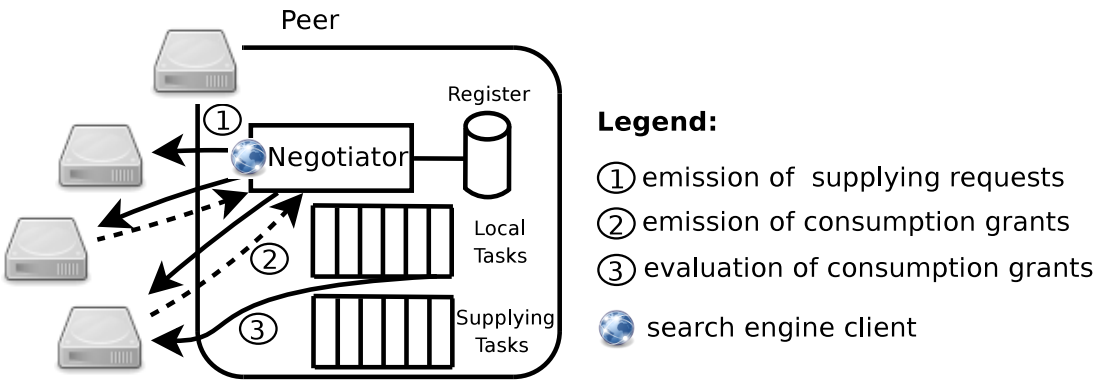


Figure 2.25: Negotiation model.

Policy-based decisions are made at three control points, called the *negotiation policy decision points* (negotiation PDP). They are symbolized on Figure 2.25:

1. Emission of supplying requests: Sends supplying requests to other Peers so as to locate suppliers with available Resources;
2. Evaluation of supplying requests: Evaluates received supplying requests, and may reply with the emission of consumption grants;
3. Evaluation of consumption grants: Evaluates received consumption grants; actually is the Consumption Tasks scheduling PDP (see Section 2.9.4).

A complete mapping of events to negotiation policies is available in Appendix C.2.

Negotiation Policies

Emission of Supplying Requests Only one policy is currently implemented for the emission of supplying requests. When the Peer cannot schedule a number of Local Tasks, it sends a supplying request for this number of Tasks to potential suppliers. To obtain Peer handles of potential suppliers, the Negotiator of the Peer relies on the Grid-level Peer discovery mechanism that is discussed in Appendix D.2.

Before emitting supplying requests, a Peer has first to obtain handles from other Peers (see Section 2.8). The Peer relies on a component - the Search Engine client - that regularly downloads lists of Peers handles from one Search Engine (see Section 2.1.4). It tries and ensures the continuous availability of a specified number[20] of Peer handles. In the current implementation [99], the Search Engine client connects to only one Search Engine. Moreover, multiple Search Engines do not exchange Peer handles with one another. These two restrictions - which can easily be removed as part of future work - give rise to P2P Grids where Peers are not aware of Peers registered in other Grids. This may or may not be desirable; See Appendix D.2 for a longer discussion.

Evaluation of Supplying Requests Four policies are currently implemented: no emission of consumption grants, unlimited emission of consumption grants, random-based bounded emission, and bounded emission to Favors-ranked potential consumers.

Two policies distribute as many consumption grants as there are Resources available. With random-based bounded emission, the available consumption grants

[20]Typically 40 in the current implementation, but insights from related research in P2P networks would be relevant here.

are distributed randomly among the potential consumers. With bounded emission based on favors ranking (see Section 2.3.4) of potential consumers, the available consumption grants are distributed among potential consumers in decreasing order of favor balance. This is similar to OurGrid's default policy [233, 13, 84].

The unlimited policy always sends as many consumption grants as were requested; this policy is implemented for performance comparisons only. Finally, one policy has no effect, meaning that no consumption grants are ever sent.

2.9.6 Internal Events Processor

Peer operations are typically activated following the reception of external events by the Peer service (see Section 2.9.2). However, some Peer operations are activated following other Peer operations or time-outs, which are called *internal events*.

Negotiation control operations and Task control operations, which are activated following internal events, are now discussed. Negotiation control is concerned with the efficiency and safety of the negotiation protocol. To avoid degradations of performance caused by (apparently) never-ending Tasks, a Peer may perform Task control operations, i.e. proactive Task preemption or cancellation.

Negotiation Control - Generation of Supplying Requests

Supplying Requests are generated when a Peer cannot schedule all Local Tasks to its own Resources. A Peer sends supplying requests to other Peers, so as to ensure potential Resource availability, before submitting the scheduled Local Tasks as Consumption Tasks. The emission of supplying requests thus occurs following Local Tasks scheduling rather than external events (see Sections 2.9.4 and 2.9.2).

It is possible that a consumer Peer that has emitted supplying requests receives no consumption grants in return after an extended period of time. To prevent this issue, a timer is introduced to regularly enable the emission of supplying requests without waiting for the next activation of the Local Tasks scheduling PDP.

The timer is reinitialized with each emission of supplying requests. After a defined amount of time has elapsed, if Local Tasks are still unscheduled, new supplying requests are emitted and the timer is reinitialized. Waiting a defined amount of time before emitting new supplying requests also prevents network overloads[21] due

[21] Of course, a malicious Peer could attempt Denial of Service attacks on other Peers by flooding them with control messages, but this issue is specific neither to the emission of supplying requests nor even to P2P Grids.

to floods of supplying requests. The timer threshold can be configured by the human Peer administrator.

Negotiation Control - Evaluation of Negotiation Events

A Peer can immediately evaluate a received supplying request or consumption grant. However, this does not allow to compare and rank multiple Peers communicating negotiation events in a short time span. To accommodate network delays or small temporal variations in emission time, evaluation of negotiation events takes place only when the number of Peers which sent negotiation events (of a given type) exceeds a given threshold. The quantity thresholds (one for consumption grants, one for supplying requests) can be configured by the human Peer administrator.

Supplying requests or consumption grants may be received infrequently or the quantity thresholds may be set too high. In these cases, received supplying requests or consumption grants are never evaluated. To prevent this form of deadlock, timers also introduced to regularly evaluate the received supplying requests or consumption grants.

The timer for received consumption grants is reinitialized with each reception of a first consumption grant. After a defined amount of time has elapsed, the received consumption grants are evaluated and the timer is reinitialized. The operation of the timer for received supplying requests is similar. The thresholds of the timers can be configured by the human Peer administrator.

Consumption Task Control

Consumption Task control is the automatic cancellation of Consumption Tasks that take too long to complete.

A consumer Peer schedules Consumption Tasks (i.e. submits some of its Local Tasks) to supplier Peers because it estimates that they will be completed faster than if they remained queued until some of its own Resources are available. However, the supplier Peer, even if absolutely reliable, may occasionally be very slow (i.e. much slower than the consumer Peer), or accept Supplying Tasks despite having long BoTs queues. In such cases, it may be more beneficial to the consumer Peer to cancel some of its Consumption Tasks and schedule them locally on its own Resources.

A time-out is defined by the human Peer administrator to enable Consumption Task control. After the defined amount of time has elapsed since a Consumption

Task has been submitted, the consumer Peer cancels and requeues it into its Local BoTs queue. After one or more Consumption Tasks have timed-out, both Local and Consumption Tasks scheduling are activated immediately to try and reschedule the cancelled Consumption Tasks.

Supplying Task Control

Supplying Task control is the automatic preemption of Supplying Tasks that take too long to complete.

Cancelling Supplying Tasks that take too long to complete may be useful to protect the Peer against middleware-level Denial of Service attacks. Indeed, the absence of automatic verification of important properties of the code [88, 223] of submitted Tasks (see Section 2.7.4), cancelling (seemingly) never-ending Local Tasks may be useful.

For example, a malicious consumer Peer (or malicious User Agent of a well-behaved consumer Peer) could submit Tasks based on code with critical defects[22] such as never-ending loops. It should be remarked that this issue can arise on a regular basis without any malicious intention. In the case of parameter sweeps applications, where the code is shared between all Tasks, a single, critical bug in a Grid application would be sufficient to "paralyze" a whole Peer if there were no support for Task control.

A time-out is defined (typically a very long value) by the human Peer administrator to enable Supplying Task control. After the defined amount of time has elapsed since a Supplying Task has been scheduled, the supplier Peer preempts and requeues it into its Supplying BoTs queue. After one or more Supplying Tasks have timed-out, both Local and Supplying Tasks scheduling are activated immediately to try and reschedule the preempted Supplying Tasks, as well as any newly queued Local Task.

A supplementary mechanism also forces the cancellation of a Supplying Task after a fixed number of preemptions initiated by the Task control mechanism.

Local Task Control

Local Task control is the automatic preemption of Local Tasks that take too long to complete.

[22] It must be noted that the Resource middleware severely restricts access to O.S.-level resources. It also limits the quantity of RAM available to the Task. See Section 2.7.4.

The motivations raised for Supplying Task control, such as protection against code with never-ending loops, also apply. Moreover, preempting (seemingly) never-ending Local Tasks protects the Peer against Resources that are unexpectedly slow, e.g. due to faulty hardware. Furthermore, the absence of a mechanism allowing the human Peer administrator to select which Resources can be registered (such as authenticated GNMP messages or a simple web-based interface) enables a potential site-level attack: Some malicious Resource human administrators (when different from the Peer human administrators of a site) could register with the Peer (see Section 2.7.6) some exceedingly slow Resources, effectively slowing down the completion times of any Task running on them.

A time-out is defined by the human Peer administrator to enable Local Task control. After the defined amount of time has elapsed since a Local Task has been submitted, the Peer preempts and requeues it into its Local BoTs queue. After one or more Local Tasks have timed-out, Local Tasks scheduling is activated immediately to try to reschedule the preempted Consumption Tasks.

2.9.7 Concurrency Management

Peer Threading Model

As the interactions between Grid nodes are asynchronous, a Peer may interact concurrently with multiple other Grid nodes. Internal events are also triggered asynchronously. Moreover, concurrently running Peer components may increase the efficiency of internal Peer operations. A Peer can thus certainly be considered as a multithreaded, event-driven system.

Figure 2.26 illustrates the threading model of a Peer. Peer threads and their interdependencies are represented. The arrows represent execution control. The service thread receives incoming GNMP messages (see Figure 2.15 for a close-up of the Peer service) and passes them to a pool of helper threads [99] for processing. Internal events are periodically processed in the internal events processor thread. The Scheduler and the Negotiator are both run in their own separate thread. A console thread enables limited interactions with the human administrator of the Peer, e.g. logs displaying. The embedded BitTorrent and FTP software that manage data transfers also run multiple threads (not illustrated).

Code Synchronization

Code synchronization in the LBG middleware, though being an implementation issue, is briefly discussed given its high importance in practice [103, 102]. The

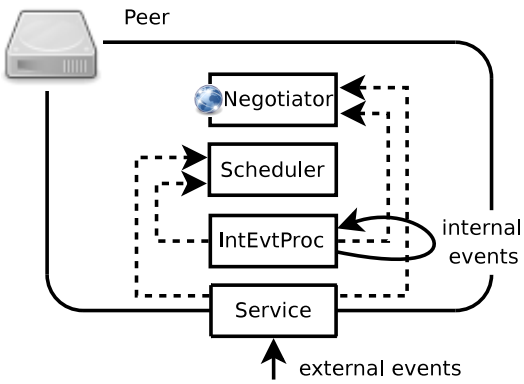


Figure 2.26: Threading model of Peer middleware.

components of the Peer middleware are thread-safe. Methods are synchronized on their respective owner object using the Java method-level synchronization mechanism [178, 49]. Many Peer operations are further synchronized on the RMS (see Section 2.7.6). Indeed some Peer components - such as the Scheduler - first evaluate the state of the Resources of the Peer, then select and finally perform operations on some of them through the RMS. For example: List idle Resources, select one Resource according to a given metric, then run a Task on this Resource. Using the RMS may thus be a multi-step operation in some cases. In these cases, the synchronization level on the RMS could be reduced to smaller code chunks, thus increasing performance of Peer components, if a transactional mechanism at a finer level were implemented.

Control of the Execution of Scheduling and Negotiation Policies

Multiple scheduling PDPs may be activated concurrently. Each of them can also be activated multiple times in a short time span. Moreover, some scheduling policies may be long-running. A Scheduler controller is introduced to control their activation and prevent deadlocks.

A *Scheduler controller* is defined as a possibly-time-delayed communication channel. It makes a set of control flags available to other Peer components. There are two types of atomic operations on a set of control flags: (1) raise one or more flags, (2) read and lower all flags. Each control flag corresponds to a request of activation of a corresponding scheduling PDP (see Section 2.9.4). Multiple policies may be activated at the same time by raising the appropriate flags.

When at least one flag is raised, the Scheduler controller notifies the Scheduler, which was waiting. When awoken, the Scheduler reads the activation flags of the controller - which resets them - and proceeds to execute the requested schedul-

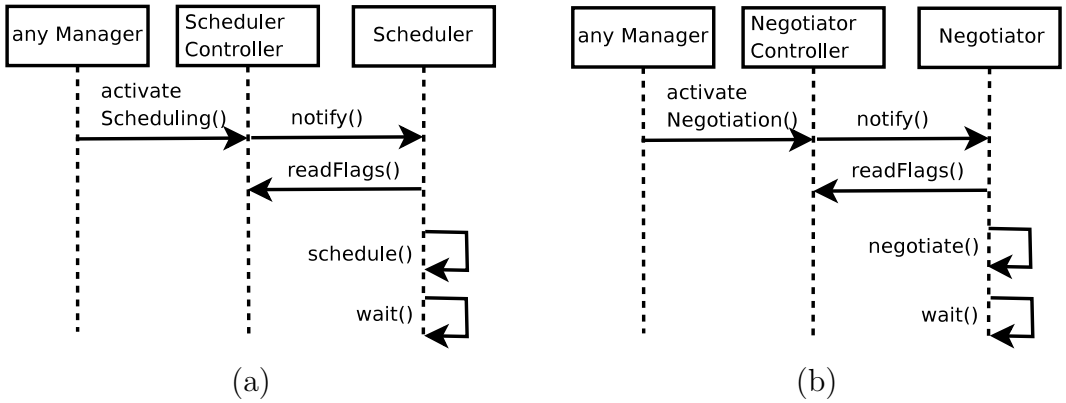


Figure 2.27: (a) Scheduler and (b) Negotiator controller operations.

ing policies in its own thread. Additionally, when the Scheduler returns from a scheduling PDP, the Scheduler controller checks its flags in case some were raised when the Scheduler was running.

A *Negotiator controller* is also introduced to control the activation of negotiation PDPs. Its interactions with the Negotiator are similar to those between the Scheduler controller and the Scheduler. A summary of Scheduler and Negotiator controllers operations is presented in Figure 2.27.

The service and internal events processor activate a scheduling or negotiation PDP through, respectively, the Scheduler controller or Negotiator controller. In turn, this controller activates the Scheduler thread or Negotiator thread. Importantly, an activated scheduling or negotiation PDP is therefore not necessarily activated immediately.

2.9.8 Peer Data Management

The Peer Data Manager, sometimes called Storage Service in the literature, is the Peer component responsible for managing the input data of the Peer's own[23] Tasks [56, 57], i.e. Local and Consumption Tasks. Each Peer Data Manager is equipped with a data cache to provide data storage support. Each Peer Data Manager is equipped with BitTorrent and FTP software components to share the input data files of Tasks submitted by User Agents: It is discussed in Chapter 5 (Section 5.2.2), as data-related operations are independent from other operations.

[23] For scalability, management of input data files of Supplying Tasks is done by the consumer Peer, never by the supplier Peer, as will be discussed in Chapter 5.

2.10 Summary of the Contributions

In this chapter, we have proposed a new P2P Grid architecture - the **Lightweight Bartering Grid architecture [59] (LBG)**. The selected Grid application model is the Bag of Tasks developed in Java. Grid nodes consist of Resources that contribute computational power to the Grid, Peers that manage the sharing of Resources' computing time, and User Agents that submit computational requests. The distinction between Peers and Resources enables to build a **2-levels P2P Grid based on bartering**. A scalable data transfer architecture for LBG is introduced in Chapter 5.

An original contribution in the design of architectures for bartering-based 2-levels P2P Grids is a scheduling model that supports queueing. Queueing of Supplying Tasks enables batch-mode scheduling and it also enables the possibility to reschedule preempted Supplying Tasks. Our proposed scheduling model for Local Tasks, i.e. without bartering, is similar to the classic WQ-R model [71] (WorkQueue with Restart), augmented with support for knowledge-free (i.e. not depending on runtime estimates), ranking-based Task selection and ranking-based Resource selection. When introducing bartering, scheduling is not as straightforward. Our proposed scheduling model for Consumption Tasks and Supplying Tasks depends on several policy decision points (PDPs) that enable many combinations of policies to address various concerns, in particular data placement. The proposed Grid Node Messaging Protocol (GNMP), used by Grid nodes to communicate control messages, facilitates the virtualization and simulation of Grid nodes, as will be discussed in Chapter 3. This, in turn, facilitates the debugging and testing of the LBG middleware as well as the development and performance evaluation of bartering policies, a few of which are proposed in Chapter 4.

Worthwhile areas of further research include the support for Task replication and Task benchmarking in the scheduling model, as well as compliance of GNMP with standard Grid protocols (see Appendix E.2).

Chapter 3

Software Engineering and Simulation of P2P Grids

What is the Matrix? Control.
The Matrix is a computer-generated
dream world ...

- Morpheus

Reproducible testing of P2P Grid middleware is extremely challenging given the distributed nature of the involved software and faults. To address this challenge, Grid nodes are first virtualized and run in a fully controlled environment that simulates a real computer network. Lengthy operations, such as Task execution and timers, are then abstracted so as to enable temporally-scalable testing, i.e. compacting to a few minutes the time to simulate a few days of operations. This abstraction is realized through a discrete-event simulator that is weaved into the code of the virtualized middleware. As a consequence, most of the code of the middleware and simulator implementations is shared and reused. By enabling to run a virtualized version of a P2P Grid on a single computer, reproducible executions become not only possible but also temporally-scalable. Experiments are performed using the LBG simulator.

In Chapter 2, we proposed P2P Grid software, the Lightweight Bartering Grid architecture, with comments on the implementation of the corresponding middleware (composed of the User Agent, Peer and Resource middlewares). This Grid middleware is designed to operate in a P2P environment which, by nature, is distributed and unreliable. Debugging and evaluating the performance of software components in such an environment is therefore challenging.

Feeding synthetic or trace workloads to bartering policies within a controlled environment is a common and useful technique for the evaluation of new bartering

policies. Simulation of a P2P Grid can abstract the environment of Grid nodes as well as their time-consuming operations, such as Task execution. Simulation allows to run a whole Grid in a controlled environment, running on a single computer rather than on many real computers communicating over the Internet. This enables to reproduce and observe the behavior and interactions of Grid nodes in a controllable and reproducible fashion.

An original contribution of our research is the first large-scale, top-down application of the *code once, deploy twice* pattern [63, 59], introduced by the contemporarily proposed GRAS (Grid Reality And Simulation) component [253] of the SimGrid [78, 278] middleware. We propose to virtualize Grid nodes so that the P2P Grid middleware and the simulator of P2P Grid share most of their code, bringing several interesting benefits. In particular, this enables the easy testing and debugging of most of the P2P Grid middleware code (not only bartering policies) within the simulator environment.

This chapter is structured as follows. We first motivate our research by describing challenges in the performance evaluation of scheduling algorithms and in the software engineering of P2P Grid middleware. Design objectives for the development of a discrete-event P2P Grid simulator based on the *code once, deploy twice* pattern are then given, along with expected use cases. Background information is provided and related work is reviewed. We explain how to weave together the simulator code, the code of simulated Grid nodes and the bartering code: Grid nodes are virtualized so that most of their code (particularly Peer bartering code) can be reused in a simulated environment. The proposed simulator of P2P Grid and the simulation parameters are then described. Finally, experimental results are presented and the contents of this chapter are summarized.

3.1 A Simulator of P2P Grid

3.1.1 Purpose and Benefits of Simulation

Performance evaluation of bartering (i.e. scheduling, negotiation,...) algorithms is very difficult, if not impossible, to achieve through analytical models given the complexity of the modelled system. As scheduling is a computationally hard problem, most scheduling algorithms are heuristics. Their performance is usually compared and evaluated experimentally, according to the values of metrics such as mean BoT response time (MBRT), utilization, cancellation history, ... It is consequently important to be able to reproduce at will a given Grid environment so that comparison of such algorithms is meaningful.

Simulation attempts to *predict aspects of the behavior of some system by creating*

an approximate mathematical model of it [208]. The purpose of a discrete-event system simulator is to provide a controlled environment to evaluate a system, or some process or interactions between a set of entities. In general, simulators are used to evaluate the performance of algorithms or optimize their parameters.

The advantage of simulation is that it suffers from none of the stated issues associated with analytical models or testbeds: It is easier to implement a simulator than to develop analytical models (which might not even be feasible for a P2P Grid) and using a simulator makes experiments totally controllable and reproducible.

Moreover, the temporal cost of simulation can be very small compared to the real execution of a system because time-consuming operations, e.g. Task executions and network transfers, are abstracted. It is thus possible to reduce real execution times of several days down to a few minutes. A discrete-event P2P Grid simulator is thus an efficient tool to evaluate the behavior of new bartering policies.

3.1.2 Software Engineering Challenges in P2P Grids

A P2P Grid environment is typically very dynamic and uncontrollable, making it exceedingly difficult to reproduce even basic behavior of Grid nodes (Peers, Resources, User Agents) in a controllable and reproducible manner. Additionally to issues related to performance evaluation - which can be alleviated through simulation - this introduces challenges in the software engineering of a P2P Grid.

Testing and debugging the implementation of bartering policies and associated P2P Grid middleware is difficult because of the multiple sources of failure that can arise in a distributed environment. A recent report [52] on the software engineering of the OurGrid middleware [233, 13, 84, 286] confirms this is a major issue that complicates the development of P2P Grids. It is thus valuable to be able to reproduce at will the conditions that lead to an unexpected outcome or to a failure of the P2P Grid middleware.

3.1.3 Design Objectives of a P2P Grid Simulator

To address the stated software engineering challenges in the development of P2P Grid middleware, we propose to develop a discrete-event simulator of P2P Grid based on the *code once, deploy twice* pattern. We set out two design objectives.

The first objective is to maximize the accuracy of the P2P Grid simulator. This is an obvious requirement. However, we propose to achieve it in a way that has been little explored in the Grid domain: by directly embedding the simulator into the

system to simulate, rather than including components of the system - most often simplified - into the simulator. Consequently, the same bartering policies are used in both the middleware and the simulator.

The first objective is reformulated as embedding the code of the simulator with the bartering code of the Lightweight Bartering Grid architecture, i.e. scheduling and negotiation algorithms, Resource management, queueing, ... It means that both the middleware and the simulator use the same bartering code but some operations, notably interactions between Grid nodes and Task execution, are abstracted in the simulator. A consequence is that the simulator and middleware can be shipped together and deployed from the same software package.

Embedding the code of a discrete-event simulator into the code of a large distributed system - especially if involving efficient management of multithreading - can be challenging. This probably explains why, besides the SimGrid simulator [78, 278], there have been very few attempts so far.

The first objective is covered in Sections 3.2, 3.3 and 3.4.

The second objective is to facilitate the definition of Grid configurations and test new scheduling and negotiation algorithms, which is the basic capability of the intended P2P Grid simulator. A common way to do so is to define a simulation description language that enables the easy and accurate simulation of a P2P Grid with a given configuration.

The second objective is covered in Section 3.5.

In a nutshell, the simulator takes a simulation description file as input, lets Grid nodes interact and provides execution statistics as output. A diagram of the simulator is presented in Figure 3.1.

3.1.4 Expected Use Cases

Coding scheduling and negotiation algorithms once, and deploying them twice is possible by using both simulation and virtualization as software engineering tools. At least three distinct use cases can be envisioned for a P2P Grid simulator based on these ideas.

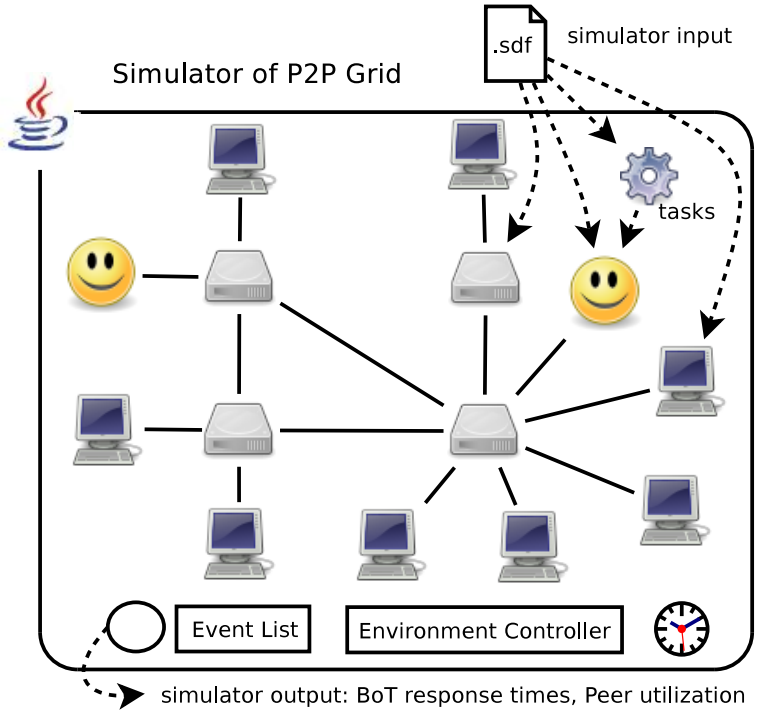


Figure 3.1: Diagram of the discrete-event simulator of P2P Grid. Simulated User Agents, Peers and Resources, as well as the event processor, event list, environment controller and simulation clock are all run within the main simulator thread, on a single computer.

Performance Evaluation of Bartering Policies

The main purpose of a P2P Grid simulator is to facilitate the development and deployment of new scheduling and negotiation algorithms. An algorithm that is available in the simulator can be used right away in the middleware, without any additional coding. Performance evaluation of new scheduling and negotiation algorithms is greatly facilitated as the algorithms are executed in a controlled environment and within a time frame that is orders of magnitude faster than if they were executed directly on real computers.

Teaching of Advanced Topics in Distributed Computing

In the same way network simulators are of interest to the teaching of advanced courses on computer networks [294, 204], using a Grid simulator as an educative tool is of interest in higher education programs offering advanced courses of distributed computing [220, 288] in their curriculum. Describing distributed architectures and algorithms may be challenging because of the multiple levels of abstraction that are involved. Letting students use a simulator to help them ac-

quire a deep understanding of the relationships between software components, as well as their impact on the overall distributed system, is therefore very useful. The accuracy brought by executing the same bartering policies in the simulator as in the middleware is thus of high interest in this context.

Implementation of P2P Grid Middleware

Following the *code once, deploy twice* pattern, it is not needed to build a simulator of P2P Grid separately from a P2P Grid middleware. Unnecessary software engineering efforts are avoided thanks to massive code reuse. The testing and debugging of P2P Grid middleware are greatly facilitated because most of the middleware code can be run in the controlled and reproducible environment of the simulator.

3.2 Related Work

In this section, possible levels of virtualization are first discussed. Discrete-event system simulation is shown to be an excellent trade-off, and then described. Secondly, the limited number of existing Grid simulators and emulators are reviewed.

3.2.1 Virtualization Levels

Virtualization is the injection of an abstraction layer between an application and some Resources used by that application. It provides a logical rather than physical view of data, computing power, storage capacity, and other resources involving the simulation of combined, fragmented, or simplified Resources. There exists multiple levels of virtualization. The following classification of virtualization levels is inspired by Casanova et al.'s [78].

Virtual Machines and Time-Based System Simulators

At a low level, the hardware environment of the system is completely or partially abstracted. So-called virtualization technologies operate at this level. Resources run unmodified in a virtual machine (VM) that is controlled by a virtual machine monitor (VMM), also called hypervisor.

This virtualization level is actually emulation. It does not require any modification of the studied system. In practice, it can enable to easily debug network communications of a middleware. But system simulation within a VM can be very slow,

as the system operates at most as fast as nominal runtime speed. Simulating one hour of operation of a system within a VM takes at least one hour. Simulating Grid operations that span many hours quickly becomes untractable.

Discrete-Event System Simulators

At a higher level, some operations of the system itself are simulated. The system is run, and controlled, by a discrete event system simulator (see Section 3.2.2).

System simulation within a simulator can be fast to very fast, as most time-consuming operations can be abstracted. Moreover, communication between system components is very fast because these components are implemented to run together, using the same memory heap. This heap usually resided in the memory of one computer only, thus abstracting all the communication times. Simulating one hour of operation of a system may be as fast as a few minutes or even seconds.

Mathematical Simulation

At the highest level, most operations of the system are abstracted with an analytical model. The system is controlled by a simple simulator. On one hand, system dynamicity is difficult to take into account, and complex systems are complex to model. On the other hand, it is very fast if a suitable model is available, which is typically not the case for Grids.

3.2.2 Discrete-Event System Simulation

Discrete-event system simulation [36] is the modelling of a system over time through its state and a sequence of events, with the understanding that:

- a system is loosely defined as a group of entities interacting with one another;
- the system state is a set of variables;
- a simulator event[1] represents an asynchronous change in system state and is associated with a timestamp.

There are three classic formalisms to describe the simulated system [179]: Activity Scanning, Event Scheduling and Process Interaction.

[1] Simulator events are distinct from Grid nodes events introduced in Section 2.9.2, even though they may be related to them.

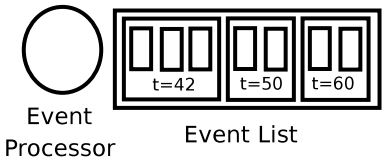


Figure 3.2: Event processor and event list (with 7 events stored at 3 timestamps).

- Activity Scanning *"is a form of rule based programming, in which a rule is specified upon the satisfaction of which a predefined set of operations is executed."* [179]
- With the Event Scheduling formalism, events are defined *"at which discontinuous state transitions occur"* and *"can cause, via scheduling, other events to occur."* [179]
- With a Process Interaction formalism, *"each process in a simulation model specification describes its own action sequence."* [179]

Event Scheduling is the most appropriate formalism to build a simulator of P2P Grid in the context of this dissertation because it enables to express the simulation problem in a natural way. Furthermore, simulators based on the Event Scheduling formalism are usually considered faster than those based on the other two formalisms [179]. On the downside, it can be much harder to implement.

The operations of a discrete-event system simulator based on the Event Scheduling formalism (in the following: simulator, for short) are organized around the management of an event list. The event list is the data structure that maintains future events ordered by increasing timestamp. Events with a similar timestamp are grouped together into an event set: See Figure 3.2, where three sets are represented. Events are inserted into an event list with respect to their timestamp.

Management of the event list is often performed with the Event Scheduling/Time Advance algorithm [36]. Basically, the main simulator loop extracts events from the event list, one at a time:

- The simulator sequentially extracts events at the head of the event list;
- When a simulator event is extracted from the event list, the simulator updates the global system time to the value of the timestamp of this event;
- To process an extracted event, the event processor updates the system state and may insert new simulator events into the event list, in correct temporal order.

If multiple events happen at the same time, i.e. they have the same timestamp, they are inserted into/extracted from the event list in an arbitrary, but reproducible, order among other events with the same timestamp.

A simulation is started by initializing system state and system time, and also by inserting one or more initial events into the event list. The simulator then enters into its main loop to process events one by one.

A simulation could run forever as long as it is fed new simulator events. Typically, criteria to stop simulation include [120]: the simulated system time has exceeded a certain value; the number of events inserted into the system has exceeded a certain threshold; some (possibly indirect or composed) measure of system state has reached a certain value.

Statistics on the state of the simulated system are collected regularly by the simulator, and constitute its output data.

3.2.3 Review of Existing Grid Simulators

Time-based Grid simulators and discrete-event Grid simulators are now reviewed. An overview of the SimGrid discrete-event Grid simulator is then given, which leads to the rationale of developing a new P2P Grid simulator.

Time-Based Grid Simulators

Firstly, time-based Grid simulators are reviewed. These are actually emulators as the Grid middleware code to be simulated is actually run as-is in an emulated environment, the virtual machine. Their purpose is to increase the ease and accuracy of the simulation, at the cost of a huge performance penalty. Indeed, running a time-based Grid simulator takes nearly as much time to simulate as it would take to actually run this software in a real environment.

OptorSim [70] is a Grid emulator that targets Grids processing massive amounts of data and in particular dynamic data replication policies.

MicroGrid [316] is a Grid emulator that is built on top of several existing simulation packages. It targets large-scale Grid deployments.

P2P Realm [198, 197] is a P2P emulator targeted specifically for the P2PDisco [28, 196] P2P Grid middleware. It is oriented essentially towards network-level simulation rather than middleware-level or application-level simulation. As it is not scalable, a true discrete-event system simulator is under development to simulate P2P Disco.

Discrete-Event Grid Simulators

Secondly, discrete-event Grid simulators [36] are reviewed. Their performance may be orders of magnitude faster than emulators, as explained in Section 3.2.1. In the following review, only the global simulation model of the simulators is given; for more complete analysis of the simulation models, we refer the interested reader to the excellent review provided by Casanova et al [78].

Historically, Bricks [3] has been the first Grid simulator. It was thus still oriented towards centralized Resource sharing (see Section 1.1.3).

ChicSim [257, 81] is a Grid simulator built on top of Parsec [239]. It targets Grids that process massive amounts of data and, specifically, helps to study the performance of dynamic data replication policies.

GangSim [128] is a Grid simulator built on top of Ganglia [217]. It targets negotiation of Service Level Agreements. It has a great focus on Grid monitoring given that it is actually a heavily modified version of Ganglia.

A P2P Grid simulator has been released to support the Resource usage accounting research [266, 265] of the OurGrid middleware.

GridSim [69] is a Grid simulator built on top of SimJava [169], a Java toolkit to build simulators. It targets the simulation of the Grid economy, or market-based negotiation and scheduling algorithms. Interestingly, it features a user interface that makes it easy to write simulation scenarios [288]. It is currently deployed as a tool to support teaching of distributed computing courses [220].

GSSim [201] is a Grid simulator built on top of GridSim. It shares with SimGrid and our proposed Grid simulator the goal to virtualize a Grid in order to use the same code both in the simulator and the middleware code. Like SimGrid, it targets the evaluation of scheduling algorithms. However, code reuse is limited to the scheduling algorithms. A related web portal [162] acts as a repository of trace workloads and scheduling algorithms.

SimGrid [78, 278] is a very advanced and flexible Grid simulator and middleware. It targets the evaluation of scheduling algorithms. Code reuse is an important design goal. It also offers a rich API allowing developers to easily simulate, and also run as part of a middleware, code that is built on top of SimGrid components. SimGrid is programmed in C but Java and C++ interfaces are available. SimGrid popularity has been steadily increasing: It has recently been used to simulate the BOINC Volunteer Grid middleware [277].

	Simulation model	Technology	Project activity
Bricks	discrete-event	Java	inactive
ChicSim	discrete-event	C, Parsec	inactive
GangSim	discrete-event	C, Ganglia	inactive
GridSim	discrete-event	Java, SimJava	active
GSSim	discrete-event	Java, SimJava	active
MicroGrid	time-based	C, *aSSF	active
OptorSim	time-based	Java	inactive
OurGrid sim	discrete-event	Java	inactive
P2P Realm	time-based	Java, Chedar	active
SimGrid	discrete-event + code 1, deploy 2	C + Java, C++ interfaces	active

Table 3.1: Key features of reviewed Grid simulators.

Table 3.1 summarizes key features of the reviewed Grid simulators. Existing Grid simulators are mostly discrete-event system simulators. Half have been developed in Java, and half have been developed in C. Not all of them have stood the test of time and remain the object of active research projects.

Overview of SimGrid

Thirdly, the discrete-event system simulator most closely related to our work, SimGrid [78, 209, 253, 77, 75, 278], is reviewed more extensively. GRAS [253, 252] is a component of SimGrid that enables to *code once, deploy twice* Grid decision making capabilities (see Section 3.1.4). It can enable the easy deployment of bartering policies as part of a Grid middleware and as part of a simulator.

GRAS targets *applicative overlay* applications[2]: It exposes an API composed of a set of low level primitives suitable for communications in P2P Grids. GRAS would correspond to LBG's Grid Node Messaging Protocol (GNMP) services and handles (see Section 2.8) packaged into an API, augmented with timing management.

Weaving simulator code into the LBG middleware code is quite different from the GRAS software development model. GRAS requires to adhere to its API when developing decision making code. The LBG simulator, on the other hand, is tailored to the LBG middleware, which is more natural given the exploratory nature of our research.

[2] *"GRAS is not a grid middleware in the common understanding of the world, but rather a tool to constitute the building bricks of such a middleware. GRAS is thus a sort of 'underware'." [175]*

All P2P Grid middlewares that were reviewed in Section 2.5.2 are developed in Java, probably because Java is particularly suitable to develop Internet-facing software. GRAS is developed in C, which is a good choice in itself. Given that LBG is developed in Java, using GRAS through JNI would be possible but performance penalties would have to be evaluated.

Rationale for a new P2P Grid Simulator

In the course of our research work we independently arrived at the conclusion that *code once, deploy twice* is a powerful idea that can really help the software engineering of P2P Grids. We have followed a top-down approach to the virtualization and simulation of Grid middleware that can be retrospectively considered as the first application to a complete P2P Grid middleware of the *code once, deploy twice* pattern that was contemporarily proposed by GRAS/SimGrid [253, 252].

GRAS/SimGrid is an exciting project that is probably on its way to become a reference tool in the universe of Grid simulators. To the best of our knowledge, it is the only existing Grid simulator to support the *code once, deploy twice* pattern. Put simply, there is no other approach besides GRAS/SimGrid that addresses the issues discussed in this chapter.

Had we started our research in 2008, GRAS/SimGrid would have constituted an excellent choice to enable the virtualization and simulation of LBG. However, the integration of GRAS/SimGrid with the LBG middleware code would have required more effort than directly weaving simulator code with the LBG middleware due to the development model and different implementation language. More importantly, it is only recently [78] that the simulation of multithreaded code was supported through the SimIX component of SimGrid.

Other simpler, Java-based Grid simulators, e.g. GridSim [69], might have been considered to implement the simulation core itself (orthogonally to the *code once, deploy twice* pattern), but rare are those that provide an expressive simulation description language [153]. Indeed, one strength of the LBG simulator is its simulation description language (introduced in Section 3.5) that is specifically tailored to the configuration of Peer policies. This tight integration makes it easy to run large parameter sweeps of tens of thousands simulation configurations (as discussed in Section 6.1).

3.3 Grid Nodes Virtualization

We now explain how Grid nodes of our proposed P2P Grid architecture are virtualized. Simulated Grid nodes are instantiated during the initialization of our proposed Grid simulator (see Figure 3.1). The code of these simulated Grid nodes is loaded (using the Java VM class loader) and shared among all instances, but each simulated Grid node has its own separate data structures (those of Peers grow over time due to the storage of metadata about interactions with other Peers).

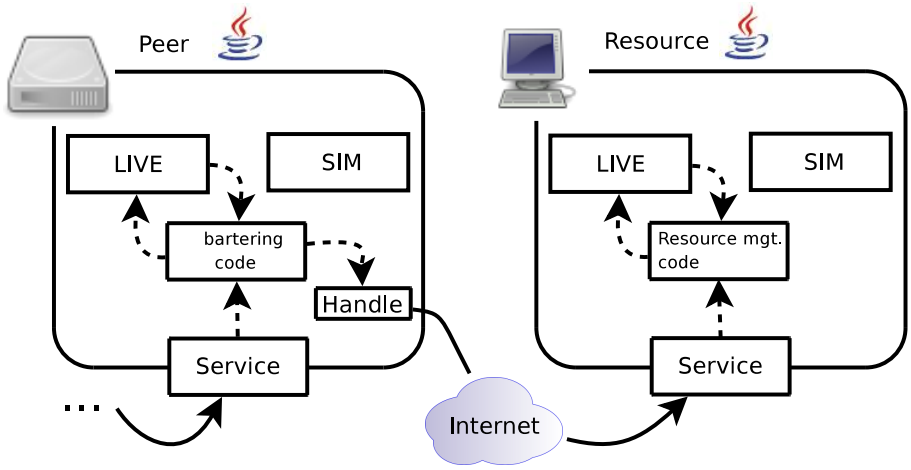
The code of the simulator implementation of Grid nodes is identical to the code of the the middleware implementation. However, some parts of the code have two distinct implementations: One is activated in the simulator, the other one is activated in live Grid nodes. The dual-implemented Java classes are those involved in communications between Grid nodes, multithreading activities and Task execution. The simulated Grid nodes have to be virtualized, i.e. isolated from their environment. They should have no awareness of the fact that they are being run within the same thread of the same Java VM and interact with a fully controlled, virtualized environment. This corresponds to the virtualization of the Fabric, Connectivity and Resource layers [150] in Foster et al.'s Grid architecture.

The virtualization of the Grid nodes code from the middleware implementation is described in this section. The virtualization of communications between Grid nodes is first discussed, followed by the virtualization of Grid nodes themselves. The virtualization of multithreading activities is essentially done for Peers, while the virtualization of actual Task execution is essentially done for Resources. Figure 3.3 illustrates the differences in execution paths in the case of a live Grid and of a simulated Grid.

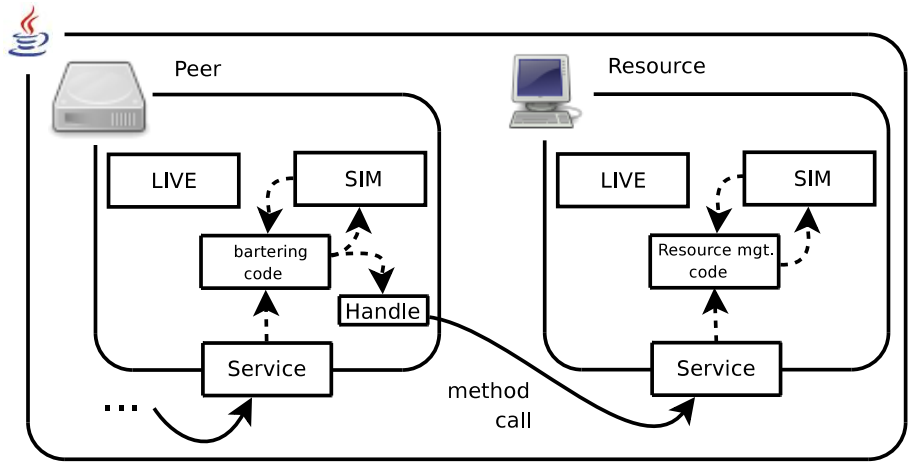
3.3.1 Grid Nodes Messaging Virtualization

To achieve the isolation of Grid nodes from their simulated environment, interfaces should be placed at the locus of minimum data flow with the environment. In the LBG architecture, this corresponds to the Grid Node Messaging Protocol (GNMP) services interfaces (see Tables A.2, A.3 and A.4) as GNMP messages pass through handles and services (see Section 2.8).

In the middleware implementation, a handle performs a network call to send a message to the corresponding service. In the simulator implementation, the network call is replaced by a method call as all handles and services reside in the same Java VM; the transmission of simulated GNMP messages is considered to be infinitely fast and does not cause the system time to be updated. This is a reasonable assumption given the context of a P2P Grid, which is intended to process mostly



(a) Live Grid nodes



(b) Virtualized (simulated) Grid nodes

Figure 3.3: Peer-to-Resource interaction illustrating the difference in execution paths between (a) live Grid nodes, (b) virtualized (simulated) Grid nodes.

long-running (i.e. at least a few seconds), Grid applications. The processing of the contents of GNMP messages by Peers is identical in the simulator and in the middleware implementations of Peers. The processing of the contents of GNMP messages by Resources and User Agents varies between their simulator and the middleware implementations because Task execution and interactions with human users are virtualized, i.e. interactions with entities outside of the middleware.

3.3.2 Resource Virtualization

The simulated and the middleware versions of a Resource share some common code. The common code encompasses the update of the state of the Resource and the management of Tasks metadata. In both implementations, Tasks are handled similarly within the Resource state space.

The two implementations are also different with respect to Task execution and cancellation. Data transfers and data storage on a Resource are not simulated.

Simulation of Task Execution

The middleware implementation of the `runTask()` operation (see Table A.3) consists of launching within a helper thread an execution module that asynchronously starts and controls a new Java VM to run the Task (see Section 2.7.1). This is done in order not to block the Resource service thread, which had invoked the `runTask()` operation.

The simulator implementation of `runTask()` essentially consists of computing the completion time of the simulated Task in function of the computing power of the simulated Resource, and of the Task nominal runtime. In the simulation description file (see Section 3.5), the simulated Resources are configured with a computing power, by the human user of the simulator. Also in the simulation description file, the description of the synthetic workload (to submit by simulated User Agents) associates a nominal runtime to the simulated Tasks. This nominal runtime is given for a Resource with a computing power of one unit. It is thus straightforward to compute the completion time of a Task of given nominal runtime on a Resource of given computing power.

Simulation of Task Cancellation

The middleware implementation of the `cancelTask()` operation on a Resource (see Table A.3) consists of asking the execution module to destroy the Java VM where the Grid application is running.

The simulator implementation of the `cancelTask()` operation on a Resource is not as straightforward because the state of the simulated Resource must be updated in case of simulated Task execution failure. A Task, whether submitted to a live P2P Grid or in the simulator of P2P Grid, may be cancelled because:

- the owner Peer of the Resource decides to cancel the Task because
  - ◇ the Task has timed-out (see Section 2.9.6),
  - ◇ or it is a Supplying Task cancelled by its consumer Peer (see Section 2.9.6);
- (in the simulator implementation only) the simulated Resource cancels a Task to simulate Task execution failure, which causes the simulated Resource to signal its owner Peer to initiate preemption of the running Task;
- the owner Peer has received a preemption signal from the Resource and accordingly updates its internal state, then signals back the Resource to update its internal state too (the Resource expects this message from the Peer because it is important that the Peer updates its internal state).

Simulation of Task Execution Failure

To test Peer behavior and the performance of scheduling algorithms in presence of Resource failure, simulated Resources can be configured to exhibit an unreliable behavior. If a simulated Resource is configured to be unreliable, its `runTask()` operation fails some executions by inserting a FailedTaskEvent instead of a CompletedTaskEvent into the event list (see Section 3.4.2). As Task execution failures due to Resource failure are supposed to be rare events, they can be modelled by a Poisson process [36], i.e. in practice with a negative exponential distribution [135].

3.3.3 Peer Virtualization

The Peer middleware and simulator implementations differ only in how multithreading is implemented. Figure 3.4 illustrates the middleware and the simulator implementations of Peer threads.

In the middleware implementation, many threads (including the service, internal events processor, Scheduler, Negotiator and data management threads) are started when the Peer comes online.

In the simulator implementation, a time-delayed communication channel for activation signals - the environment controller - is instantiated and no thread is started. The Scheduler, Negotiator, internal events processor and service threads

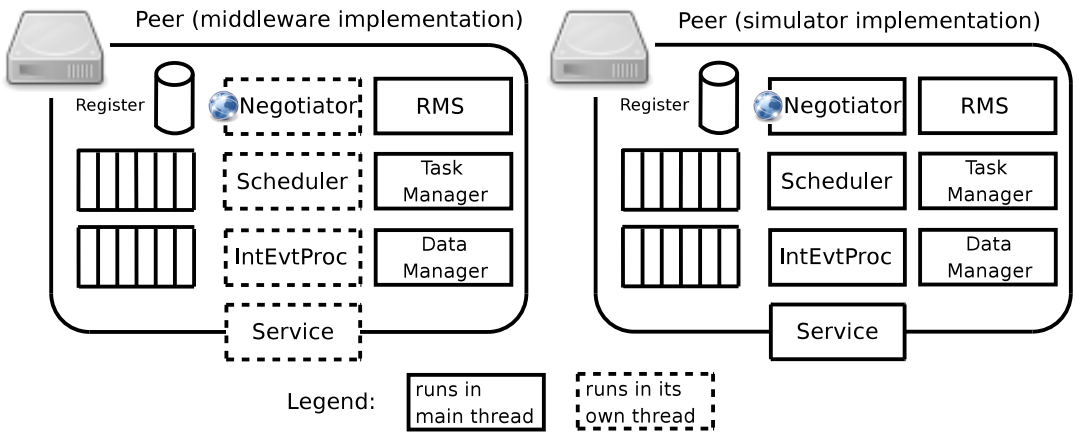


Figure 3.4: (a) Middleware, (b) simulator implementation of Peer threads.

are simulated. The Peer service, Scheduler controller (not shown) and Negotiator controller (not shown) are reimplemented to simulate multithreading.

Multithreading Virtualization

In the middleware implementation, a large number of threads are running in every Grid node. Although the service (see Sections 2.8 and 2.9.2) of every Grid node is simulated, service threads are not. Indeed, the services are purely reactive devices. The simulation of helper threads used for Task execution and data transfers is straightforward as these operations are abstracted into a very simple model.

The challenge therefore consists of simulating the scheduler and internal events threads of every simulated Peer. Running all Scheduler, Negotiator and internal events threads within the simulator would be possible but would degrade the simulator scalability as the number of threads would be linear with the number of simulated Peers. The Schedulers, Negotiators and internal events processor should therefore not run within their own, dedicated threads. We propose that the simulator regularly activates every Scheduler, Negotiators and internal events processors to simulate the multithreading activities of the middleware implementation.

Environment Controller

To simulate the multithreading of the Scheduler and Negotiator threads, we introduce a device called environment controller. It is defined as a time-delayed communication channel used by the Schedulers and Negotiators of Peers, through

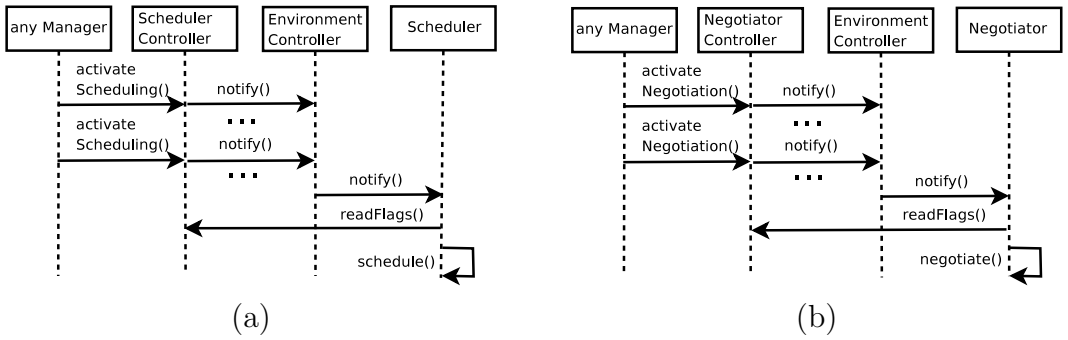


Figure 3.5: (a) Scheduler-related, (b) Negotiator-related Environment Controller operations.

the Negotiator and Scheduler controllers (see Section 2.9.7, in particular Figure 2.27), to activate Scheduling and/or Negotiation policies of this Peer.

When scheduling or negotiation policies are activated, a signal is sent to the appropriate Scheduler or Negotiator controller. In the simulator implementation, this signal is not immediately forwarded to the Scheduler or Negotiator. Instead, notification is simulated by the update of the state of the environment controller, through the raising of corresponding flags (see Figure 3.5).

Multithreading is simulated after all simulator events at the current timestamp have been processed (with all reactive interactions between Grid nodes completed). At this point the environment controller is activated. In turn, it activates the Scheduler, Negotiator and internal events processor of every Peer in arbitrary order. The environment controller reads - and resets - its state for every Peer: When activation signals have been stored for a given Peer, the corresponding Scheduler or Negotiator is activated. The internal events processor is systematically activated for each Peer every simulated time unit, i.e. after all events with a given timestamp have been processed (see also Appendix C.3).

The environment controller is activated after the processing of simulator events with the same timestamp. This is done to guarantee the completion of all interactions between Grid nodes that result from events with that timestamp. The environment controller could be activated after the processing of each simulator event. This would require supplementary coding in the current implementation for probably limited benefits. The environment controller could also be activated during the processing of each simulator event. That would require simulator-level multithreading, with a number of threads linear with the maximum number of simulated Peers that could be involved in cascade interactions. However, running a large number of threads in the same Java VM would not be tractable in practice.

3.3.4 User Agent Virtualization

The simulator implementation of a User Agent is simple: It only counts completed Tasks and BoTs, and does not have to retrieve output data files.

3.4 Simulator Implementation

The implementation of major simulator components (main simulator loop, simulator event processor and simulator clock - see Figure 3.1) is described in this section.

3.4.1 Main Simulator Loop

One Iteration of the Main Simulator Loop

If an event extracted from the event list has a timestamp strictly greater than the timestamp of the previously extracted event, the simulator updates the system-wide clock to simulate the advance of time.

The processing of each event consists of updating the state of the simulated system. During the processing of a simulation event, a new simulation event may be generated, and inserted into the event list. The processing of a simulator event may lead to Grid nodes exchanging (simulated) GNMP messages with one another. Given the *code once, deploy twice* pattern, the **processing of the contents of GNMP messages** by Peers is **identical in the simulator and in the middleware implementations**.

Figure 3.6 gives the skeleton of the algorithm (presented with a Java syntax) of the main simulator loop (see Section 3.2.2). As can be seen, scheduling and negotiation operations are activated by the environment controller (cf. previous section), if the two following conditions are verified:

- All the events happening at the current timestamp have been processed;
- Scheduling or negotiation signals have been communicated by some Peer managers to some Scheduler or Negotiator controller during the processing of the events at the current timestamp.

Termination of the Main Simulator Loop

The main simulator loop stops when the event list is empty. As Tasks complete their simulated execution, simulator events are eventually removed from the event list. Nonetheless, new simulator events are inserted into the event list as long as

simulated User Agents submit Tasks to Peers. The simulated User Agents must thus be configured to submit a finite amount of Tasks to Peers, so that the simulator stops after a finite period of time.

As explained in Sections 2.9.2, 2.9.7 and 3.3.3, a Peer service is a purely reactive device. It may use some Peer managers, such as the Queue Manager (see Section 2.9.3). These may in turn try to activate scheduling or negotiation operations. But contrarily to what happens in the middleware implementation, these signals are not immediately communicated to the Scheduler and Negotiator controllers (see Section 2.9.7, in particular Figure 2.27).

In the simulated implementation, the environment controller (see Section 3.3.3) is notified instead. The environment controller stores scheduling or negotiation signals, but does not process them immediately, independently of the number of managers of a given Peer that have requested the activation of scheduling or negotiation operations. It is only after all events at a given timestamp have been processed that any pending scheduling or negotiation signal is processed by the Scheduler or Negotiation controller of the corresponding Peer. Together with the guarantee that the timestamps of all newly created events are set in the future[3], i.e. are not inferior or equal to the current simulator time, this ensures that there is no possibility for scheduling or negotiation signals to trigger an infinite cycle of processing and emission of Peer events between Peers.

3.4.2 Simulator Events

Simulator events are inserted into the event list by the simulated versions of Grid nodes and processed by the main simulator loop. There are currently four supported types of simulator events (they are related to Peer events, see Section 2.9.2):

- job, i.e. BoT, submission (SubmittedJobEvent),
- completion of Task execution (CompletedTaskEvent),
- failure of Task execution (FailedTaskEvent),
- internal event (InternalEvent).

The event processor extracts simulator events from the event list. For each event, the event processor calls code that, in the middleware implementation, would be called from a Grid node following a signal from its environment, e.g. status of Task

[3]This implies that Tasks running for less than one simulated time unit cannot be accurately simulated. This is not an issue, as Tasks are considered to be long-running.

```
EventList el = new EventList();
EnvironmentController env_ctrlr = new EnvironmentController();

Event first_evt = new SubmittedJobEvent();
el.insert(first_evt);

while (el.isEmpty() == false) {
        Event evt = el.extractFirstEvent();
        processEvent(evt);
        if (el.nextEventHasGreaterTimestamp()) {
                if (env_ctrlr.canScheduleAndNegotiate()) {
                        env_ctrlr.scheduleAndNegotiate();
                }
        }
}
```

Figure 3.6: Main simulator loop.

execution or input from a human user.

The processing of simulator events is summarized in Figure 3.7, where the simulator event processor and the event list (see also Figure 3.2) are represented for each simulator event type, along with the relevant Grid nodes. The processing of each type of simulator events is described in the remainder of this section.

Processing of SubmittedJobEvent

Each simulated User Agent is activated by the simulator to submit new jobs to the Peer it is using. When submitting a job, it simultaneously inserts a new event into the event list so that, at the expected timestamp, the simulator activates it (until the configured number of BoTs has been submitted). A new job is then submitted and this cycle goes on. Inserting a SubmittedJobEvent into the event list can be seen as a form of callback mechanism.

Processing of CompletedTaskEvent and FailedTaskEvent

Simulating a simulator event related to Task execution is a two-step process:

- When a Task is sent to a Resource for execution, the simulated Resource makes a random decision about its reliability and inserts (with high probability) a

CompletedTaskEvent or (with low probability) a FailedTaskEvent into the event list.

- A simulated Resource is activated by the simulator when a Task execution simulator event occurs. In the middleware implementation, this would happen when the Grid application run by the Resource actually either completes or fails its execution. Upon completion or failure of simulated Task execution, the simulated Resource - activated by the event processor - uploads a dummy output data file to its owner Peer, or notifies it that the running Task has been preempted. In either case, the Peer state is correctly updated.

Processing of InternalEvent

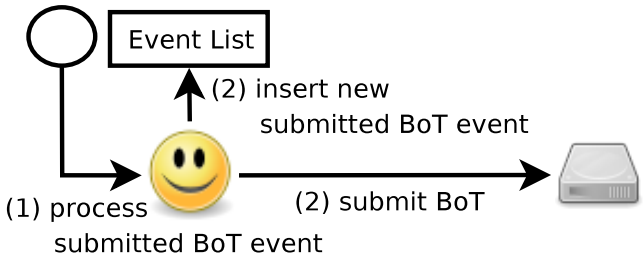
The concurrent (i.e. between Grid nodes) execution of the internal events processor of each Peer is simulated by activating it every simulated time unit, so as to simulate the internal events processor thread (see Section 2.9.6). The processing of an InternalEvent simulator event consists of activating, sequentially within the main simulator thread, the internal events processor of each Peer. The environment controller performs this activation, as it has full access to Peer components.

The activation of internal events processors of all Peers is triggered by one InternalEvent simulator event. After an event has been processed, a new one is immediately inserted into the event list with a timestamp set one simulated time unit later. When an inserted InternalEvent is the only event in the event list, it means that the simulation is nearing completion. This triggers the activation of a counter. InternalEvents continue to be inserted for a certain number of iterations. When the counter meets a threshold, no more InternalEvent is inserted and the simulation stops. This threshold is computed so that any running timer has the possibility to eventually time-out. The counter is reset if a time-out of a running timer leads to the insertion of other events into the event list. The activation of the counter ensures that the main simulator loop (see Section 3.4.1) eventually completes.

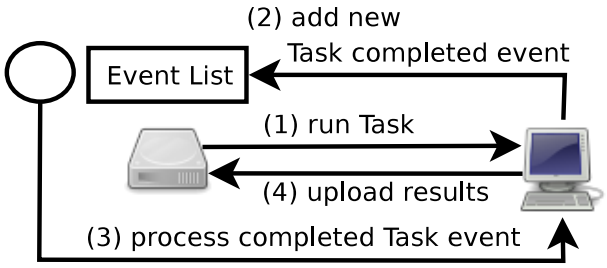
Event List Implementation

The event list of the simulator is implemented with a 2-levels tree-based data structure. A balanced binary tree maintains an ordered list of timestamps when at least one simulator event is happening. Keys are timestamps, each of which is mapped to a set of simulator events. Each of these sets is backed by its own balanced binary tree, where keys are unique event identifiers.

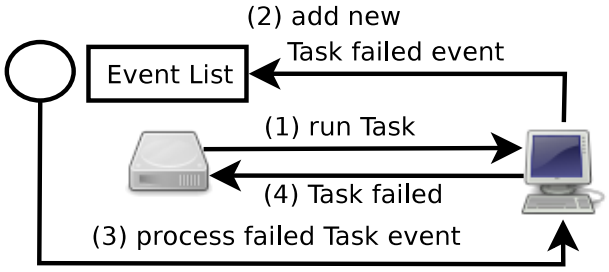
(a) SubmittedJobEvent



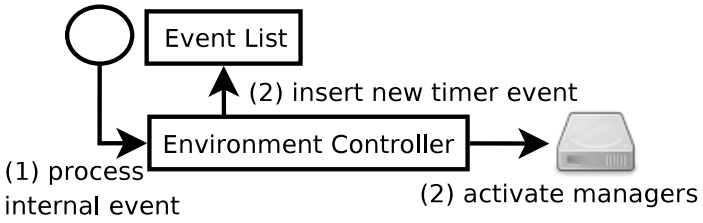
(b) CompletedTaskEvent



(c) FailedTaskEvent



(d) InternalEvent



Legend: numbers (x) describe the execution order of operations.

Figure 3.7: Processing of simulator events: (a) simulated job submission, (b) simulated Task execution completion, (c) simulated Task execution failure, (d) simulated internal event.

3.4.3 Time Management

The middleware and the simulator implementations use the same interface to read the current time. In the middleware implementation, the time is provided by the Java VM and is updated by the computer clock.

In the simulator implementation, the returned time does not come from the computer clock. The returned time is read from a simulator-wide simulated clock instead. The clock is initialized to zero simulated time units and updated by the event list only, when all simulator events happening at a given timestamp have been processed. The temporal resolution of the simulated clock, i.e. the value symbolized by one simulated time unit, is currently one second.

In the middleware implementation, time desynchronization between clocks of Grid nodes does not give rise to major issues. Indeed, a P2P Grid is designed to operate in a fully decentralized way. For example, a Peer which would not acknowledge (by counting supplied favors) its debts towards other Peers would risk to be penalized in the long term, as if it were a free rider.

In the simulator implementation, time synchronization between simulated clocks of Grid nodes must be enforced so that the simulator can correctly compute simulation statistics. As simulated Grid nodes transparently share the same simulated clock, continuous time synchronization is guaranteed.

There is however a small, yet systematic bias in time simulation: The time spent by the bartering code of Peer themselves is neglected in the time management process. This is acceptable as there are many orders of magnitude between the time taken by Peer managers to process an incoming event (a few milliseconds) and the time taken to execute a Task on a Resource (several minutes to several hours). Indeed, the total time to simulate the internal operations of a Peer to manage a Task (including scheduling and negotiation) is independent of the length of the Task execution (which is abstracted).

3.4.4 Limits of the Current Implementation

Transfers of input/output data files between Grid nodes are currently not simulated. The time taken to complete simulated Bags of Tasks is inevitably shorter than what it should be, especially for Data-Intensive Bags of Tasks. We are aware of this issue which will require considerable amounts of work in order to be fully addressed. Recent research works in this direction include those of Al Kiswany et al. [4, 5], Eger et al. [131], Casanova, Legrand et al. [78, 209], Yang [322] and Nussbaum et al. [226].

The activities of multiple Peers, notably scheduling, are sequentialized in the simulator implementation because Grid nodes are no longer running independently from one another. GNMP messages (see Section 2.8) sent concurrently by live Grid nodes running the middleware implementation are also sequentialized.

The processing order of GNMP messages sent simultaneously by multiple Grid nodes thus vary from what it would be for live Grid nodes running the middleware implementation. Indeed, the order of multithreading simulation as well as the order of insertion and extraction of simultaneous simulator events are currently both arbitrary. Moreover, multithreading simulation takes place only after all simulator events at the current timestamp have been processed. Future work is definitely required to investigate the impact of the order of multithreading simulation.

The processing order of single GNMP messages also vary slightly from what it would be for live Grid nodes running the middleware implementation. Small hardware- or network-level variations, e.g. high CPU load or network delays, will always be present and constitute a form of background noise that would be exceedingly difficult to reproduce.

Simulating the network transfers of GNMP messages or introducing true parallelism in the simulation of multithreading would require further analysis. Maintaining acceptable performance so that simulations would remain tractable is likely to be a major challenge. Indeed, the goal of simulation is to abstract reality at a relatively high level to get the benefits of a tool usable in practice (see Section 3.2.1).

3.5 Simulation Description Language

The purpose of this section is to describe the simulation description language that is used in the experiments presented in the next section.

3.5.1 Simulation Description Language

A small simulation description language has been defined. It enables to control a simulation by describing:

- Grid configuration (number of Peers, number and power of Resources for each Peer, ...);
- User Agents configuration (number and metadata of submitted BoT);

```
<scenario description file> ::= <list of scenario properties>

<list of separators> ::= <list of separators> <separator> | <separator>

<list of scenario properties> ::=
   <list of scenario properties> <list of separators> <scenario property> |
   <scenario property>

<scenario property> ::= <key> = <value>

<key> ::= < reserved keyword>

<value> ::= <scalar value> | <vector value>

<scalar value> ::= <boolean> | <int> | <float> | <string>

<vector value> ::= { <list of scalar values> }

<list of scalar values> ::= <list of scalar values> , <scalar value> |
                            <scalar value>
```

Figure 3.8: BNF grammar of a simulation description file.

- Peers Scheduler and Negotiator configuration (selection of scheduling and negotiation policies for each Peer);
- Peers negotiation control;
- Task control;
- data management configuration;
- configuration of some parameters of the simulator itself.

A simulation description file (.sdf), also called a scenario, does not need complex language constructs. It can be structured as a classic properties file, where each property has the form key = value. Each key is a string. Each value may be a scalar or a vector, where a scalar or vector element is either a boolean, an integer or a floating point literal, or a string.

Figure 3.8 presents a partial BNF grammar of a simulation description file. The complete grammar is defined in Appendix B.2.

3.5.2 User Agents Configuration

Multiple simulated User Agents are implicitly defined in a simulation, with exactly one User Agent assigned to each Peer. More User Agents could be easily simulated, but the current focus of the Lightweight Bartering Grid simulator is to study Peer-to-Peer interactions rather than User Agents-to-Peer interactions.

User Agents configuration enables to generate synthetic workloads. The number of BoTs that each User Agent submits, the lower and upper bounds of the BoT inter-arrival time distribution (i.e. time between two consecutive BoT submissions), and an initial time shift before beginning to submit BoTs, have to be defined.

The number of Tasks per BoT, the lower and upper bounds of Task runtime distribution (for a Resource with a power equal to one), and the number of simulated input data files per Task, have to be defined. **Importantly, to reflect the LBG architecture, these runtime (exact) estimates are used by the simulator only (to simulate Task execution and compute execution statistics), not by the scheduling and negotiation policies.**

3.5.3 Grid Configuration

As the purpose of the simulator is to study interactions in P2P Grids, the concept of Peer group is introduced to facilitate the writing of scenarios with multiple Peers which have the same behavior. A Peer group is a group of Peers identically configured, except for their identifier. In a scenario, there must be at least one Peer group and at least one Peer in each Peer group.

Each simulated Resource is assigned a power [266], (in an absolute unit) which is a multiple of a known base power. The power of several Resources relative to one another can be easily adjusted. The Peer power is the sum of its Resources' individual powers. It can be split explicitly or randomly between Resources. In each Peer group, the Peer power, number of Resources per Peer and power of each Resource have to be defined with nonnegative values. The algorithm to distribute the power of a Peer between its Resources is given in Figure 3.9. The storage capacity of the Resource (see Section 5.2.5), or cache size, has also to be specified.

The Resource Mean Time Between Failure (see Section 3.3.2) must also be specified, so as to enable the simulation of Task execution failure on Resources.

Finally a refresh time-out must be defined for the Search Engine client. It is used to indicate the period after which it must refresh its cache and download recently added Peer handles from the Search Engine.

// res_count: flag controlling the number of Resources to configure
// peer_power: flag controlling the total power assigned to the Peer
// lo, hi: minimum, maximum power for Resources (may be undefined)

- (res_count == 0) and (peer_power > 0) →
  random number of Resources and random Resource power in [lo..hi]
- (res_count > 0) and (peer_power > 0) →
  fixed Resource number and power
- (res_count > 0) and (peer_power == 0) →
  fixed number of Resources, random Resource power in [lo..hi]
- (res_count == 0) and (peer_power == 0) →
  no Resource... complete free rider

Figure 3.9: Power repartition algorithm for a simulated Peer.

3.5.4 Peers Policies Configuration

Parameters to configure Peers policies have to be defined, including scheduling, negotiation (see Chapter 2) and data management parameters (see Chapter 5).

Six scheduling policies have to be defined (see Section 2.9.4): Local Tasks scheduling, Consumption Tasks scheduling, Supplying Tasks scheduling, Supplying Tasks Filtering and waiting Supplying Tasks Preemption, running Supplying Tasks Preemption. Two negotiation policies have to be defined (see Section 2.9.5): emission and evaluation of supplying requests. An accounting policy also has to be defined.

A threshold and a time-out have to be defined for received supplying requests and consumption grants, as well as a time-out for Request Supplying (see Section 2.9.6).

Task control and filtering parameters also have to be defined:

- a time-out specifying the period after which to cancel a Consumption Task that takes too long to complete (see Section 2.9.6);
- a queue length threshold to determine if submitted Supplying Tasks should be filtered out and a queue length threshold to determine if queued Supplying Tasks should be preempted (see Section 2.9.4);
- a flag and a ratio to activate blacklisting of unreliable supplier Peers.

Finally, the data management policies [56, 57] (BitTorrent-awareness of Task selection, data-awareness of Resource selection, activation of proactive data replication) also have to be defined (see Sections 5.3.1, 5.3.2, 5.4).

3.5.5 Simulation Configuration

There are currently two simulation configuration parameters: the initial seed of the simulator master random number generator, and a flag to enable the cloning [178, 49] of Java objects that represent GNMP messages.

The initial seed of the simulator master random number generator (also called the random seed) enables to control the randomness of all simulated quantities, such as BoT submission inter-arrival times.

Simulated GNMP messages (see Section 2.8) all reside in the same memory heap. They are passed directly from one method of a handle to a method of a service, rather than being serialized and transferred over a network. The simulator can be configured to clone GNMP messages before they are passed from one simulated Grid node to another, instead of passing them as-is.

There is a performance penalty[4] associated with this cloning, but it is more safe[5]. Without cloning, implementation mistakes in the handling of the contents of GNMP messages can lead to incorrect-in-subtle-ways and hard-to-reproduce runtime behaviors.

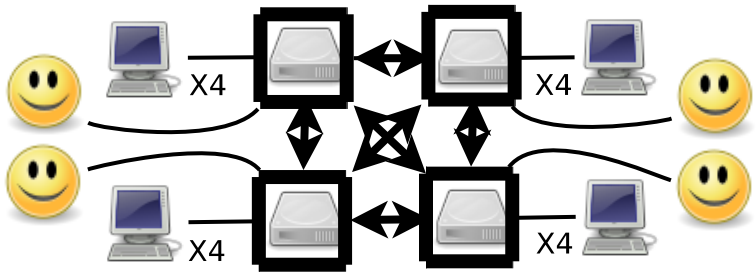
An example of such an implementation mistake is the confusion between the L-value and the R-value of a Java object that is compared to another Java object, i.e. using the `==` keyword (which tests equality of memory references) instead of the `.equals()` method (which tests equality of data fields). With cloning, the crashes of the simulator can be reproduced and the bugs can be easily isolated.

3.6 Experimental Results

3.6.1 Simulator Accuracy

Two scenarios are both run in the LBG simulator and run on a live deployment of the LBG middleware. The results are compared with one another, as well as results from the literature [14] about a comparable live deployment of the OurGrid

[4]Typically less than 2% of execution time, up to 20% for short, extreme simulations.
[5]If Task replication (see Section 2.9.4) were supported, cloning would be mandatory.



Grid topology: 4 Peers (managing 4 Resources each).

Figure 3.10: Grid topology for simulator accuracy experiments.

middleware. The experiments presented in this section have been conducted on 21 x86 PC (Intel P4 CPU with 512 MB RAM).

Both considered scenarios have been proposed in related work [14] as baseline scenarios to discuss the benefits of bartering under contention[6] for Resources, in the OurGrid [233, 84, 286] middleware.

The topology of both considered scenarios is identical. It consists of a Grid of 4 peers that manage 4 Resources each (see Figure 3.10). Each peer must process 60 Bags of 40 Tasks with no input data files. All Resources are identical. Each Task can be completed in exactly 1 minute by any Resource. The inter-arrival time of submitted BoTs is a random variable (uniform distribution) with a value between 1 minute and 20 minutes. If a Peer uses its own Resources only, and in absence of queueing delays, the optimal mean BoT response time (MBRT) is 10 minutes.

It could be argued that it would be interesting to simulate more than 20 Grid nodes. The two considered scenarios were selected first because they could be validated against existing results published for another similar middleware. A major bottleneck to the scalability of reproducible experiments involving live deployments of the full middleware is that many computers have to be dedicated for extended periods of time. The two considered scenarios are also sufficiently small that it was possible for us to reserve enough dedicated computers for the duration of the experiments presented in this section.

The two scenarios differ on the policies used by the Peers. There is one scenario with bartering and there is one scenario without bartering. In the scenario without bartering, Peers do not exchange computing time. In the scenario with bartering, Peers can barter computing time. They are configured to match as closely as possible the default policies in OurGrid, so that our results can be compared with re-

[6]Considering that the set of all Resources of the P2P Grid constitute a shared medium that Peers want to access as largely as possible when they must process Local Tasks.

Scenario	Peer 1 MBRT (s)	Peer 2 MBRT (s)	Peer 3 MBRT (s)	Peer 4 MBRT (s)	Grid MBRT (s)
LBG sim, bartering	573	973	805	893	811
LBG sim, no bartering	820	1202	2216	829	1267
LBG live, bartering	576	907	1153	806	860
LBG live, no bartering	839	1253	2314	850	1314
OurGrid live, bartering	436	453	427	462	445
OurGrid live, no bartering	N/A	N/A	N/A	N/A	1571
optimum, no bartering	600	600	600	600	600

Table 3.2: Mean BoT response times for both scenarios for simulated LBG runs (simulated time), live LBG runs (real time) and live OurGrid runs (real time).

lated results that were obtained from the observation of a real deployment (i.e. not simulated) of the OurGrid middleware [14]: preemptive Local Tasks scheduling, queue-aware NoF-based Supplying Tasks scheduling, Supplying Tasks limited cancellation (when a Peer reclaims its Resources, it cancels rather than preempts the minimum number of Supplying Tasks), data-aware Consumption Tasks scheduling, very short negotiation time-outs, and minimal queueing of Supplying Tasks beyond capacity.

Each scenario has been run in the LBG simulator. The resulting workload traces, i.e. BoTs inter-arrival times, have been recorded. Each scenario has subsequently been run on a live deployment of the LBG middleware in a controlled environment (this constitutes an error-prone and time-consuming process). The delays between BoTs submitted to the live P2P Grid were set by the workload traces produced by the simulator. The results, i.e. mean BoT response times (MBRT), are given in Table 3.2, along with results from the literature [14] about a comparable live deployment of the OurGrid middleware. The results are an average for the first 55 BoTs, when no Peer had yet completed all 60 BoTs.

A first observation is that bartering considerably shortens the MBRT, as expected (further experiments are presented in Chapter 4). Utilization of the Resources is typically around 85% with bartering and around 90% without bartering. However, utilization for Local Tasks is only around 66% with bartering (utilization for Supplying Tasks is thus around 19%), which confirms that Peers are able to acquire computing time from other Peers.

A second observation is that the MBRT (averaged over the four Peers) predicted through simulation is 6% (860 vs. 811) and 3.7% (1314 vs. 1267), respectively, shorter than the MBRT observed through observation of a live P2P Grid. It means that **our proposed simulator is able to explain around 95% of the MBRT for the evaluated scenarios with a total execution time of around**

20 hours (of real, wall clock time). In other variants of these experiments, we never observed a divergence of more than 10%. In Section 3.4.4, arose the hypothesis that the simulation of GNMP messages and multithreading are possible sources of divergence. From our experience watching the log files of these experiments, we tend to believe that most of the divergence comes from not taking into account the time to transfer GNMP messages across the network, and subsequently process them (a few milliseconds each time, that add up with a large number of exchanged messages). It would be interesting to measure a large number of runs of a live P2P Grid in a controlled environment. This requires many computers to be both fully dedicated and operational for many months: If all goes well - i.e. 100% of sustained reliability of computers, network, operating systems and middleware - each of all planned experiments is completed with 21 computers (4 Peers, 16 Resources, 1 Search Engine) in a little less than a day; larger scenarios involving more Peers require additional time and computers to run the middleware.

A third observation is that some MBRTs are shorter than 600 seconds, which is the optimal MBRT without queueing delays and without bartering. This happens with some "well-spaced" workloads where BoTs are submitted at the longest possible intervals of time, i.e. 20 minutes rather than 1 minute. With such workloads, the request peaks of Peers are temporally complementary, i.e. one Peer's busyness often corresponds to other Peers' idleness.

A fourth observation is that OurGrid seems to outperform LBG. Several hundreds simulations of the scenario with bartering have been run (the details are discussed in Chapter 6), in order to explore the behavior of the simulator. We found that LBG can also achieve an equally short MBRT, e.g. 437 seconds, when processing a well-spaced workload. We thus hypothesize that the related results [14] were provided for a well-spaced workload. On the other hand, our results in Table 3.2 are given for typical, average workloads.
Finally, we highlight again that the results observed for the simulation of the two considered scenarios have been doubly validated with real executions of the LBG middleware and with real executions of the OurGrid middleware. Future work should of course be undertaken to evaluate to what extent these promising results can be generalized to other scenarios.

3.6.2 Simulation Bias

The time spent by the Peers bartering code is neglected in the time management process. This small, yet systematic, bias in the simulation of the time is however acceptable. There are many orders of magnitude between the time taken by Peers to process the contents of an incoming GNMP message (several milliseconds) and the time taken to execute a Task on a Resource (several minutes to several hours).

On a typical desktop computer (Intel P4 CPU with 512MB RAM), small typical test scenarios may take less than 1 minute of wall clock time to simulate about 3 hours of operations of a medium-sized Grid (15 Peers, 500 Resources). Let r_s = 3 × 3600 seconds = 10800 seconds be the simulated runtime of a given scenario. Let r_p = (60 seconds / 15 Peers) = 4 seconds be the average runtime of Peer code (which is identical in both middleware and simulator implementations). For this typical test scenario, the time bias has an upper bound of much less than 1% of the simulated execution time ($r_p/r_s = 0.04\%$). Furthermore, this represents an overly large upper bound on simulation bias, considering that some wall clock time is spent by the simulator to initialize its data structures, manage the event list and perform environment controller operations.

3.6.3 Simulator Performance

The performance of the simulator is evaluated according to two metrics: simulator runtime (wall clock time expressed in seconds), and MBRT (simulated time expressed in simulated time units). The scenario of Section 3.6.1 is used as a base scenario: 60 BoTs of 40 Tasks are submitted to each of the 4 Peers that manage 4 Resources each.

The simulator runs presented in Table 3.3 vary according to one of three parameters: number of submitted BoTs, number of Peers in the Grid and number of Resources per Peer. The experiments presented in this section have been conducted on a single core of a 64bits PC (quad-core Intel Xeon CPU, with 15GB RAM available).

The observed simulator runtimes are linear with the number of submitted BoTs. The "BoTs ×100" experiment - which involves nearly 1 million Tasks (2400 Tasks submitted to 4 Peers, multiplied by 100) - exhibits performance asymptotically of the same order of magnitude as what is achieved by SimGrid [78]. However, it must be kept in mind that different things are simulated: The scheduling model of LBG is probably more complex, while simulation of data transfers constitutes the main activity of SimGrid.

The observed simulator runtimes are more than linear with the number of Peers. However, memory consumption becomes unbearable beyond a few thousands Peers, precluding larger simulations. Indeed, LBG Peers - whether simulated or live - memorize a lot of data about their interactions with other Peers: Even if the interaction history is strictly bounded, the memory requirements of the simulator are quadratic with the number of simulated Peers. Memory management can be optimized by both exploiting secondary storage and tuning the Peer discovery pro-

Variations from base scenario			Runtime (s) simulator	MBRT (s) simulation
BoTs	Peers(*)	Resources(**)		
× 1	× 1	× 1	7	607
× 10	× 1	× 1	10	979
× 100	× 1	× 1	163	955
× 1000	× 1	× 1	1605	1030
× 10000	× 1	× 1	18089	1051
× 1	× 10	× 1	106	451
× 1	× 100	× 1	1085	237
× 1	× 200	× 1	3457	298
× 1	× 400	× 1	7963	256
× 1	× 800	× 1	23271	297
× 1	× 1	× 10	6	60
× 1	× 1	× 100	7	60
× 1	× 1	× 1000	11	60
× 1	× 1	× 10000	69	60
× 1	× 1	× 100000	611	60
× 10	× 10	× 10	114	60
× 100	× 100	× 100	16935	60

(*) the total number of Resources in the Grid varies with the total number of Peers
(**) i.e. Resources per Peer

Table 3.3: Simulator runtimes and mean BoT completion runtimes for multiple scenarios derived from the base scenario.

cess. We believe that controlling memory consumption can dramatically improve the simulator performance in scenarios involving large numbers of simulated Peers.

The observed simulator runtimes are less than linear with the number of Resources per Peer, as expected. Such scenarios lead to less simulator iterations because the increased computational power of the simulated Peers enables to process BoTs much faster. Nonetheless, the management of several hundred thousands Resources per Peer induces a small performance penalty. This penalty impacts the simulation itself, but is negligible for individual Grid nodes.

After the observed simulator runtimes, the observed MBRT of the BoTs from the considered scenarios is now discussed. The MBRT is expectedly higher - but almost constant - with the number of BoTs. Interestingly, it is not that much higher, thanks to bartering. System utilization is around 85% in the base scenario and around 97% in the 1000× BoTs scenario, which demonstrates a high level of Resource sharing. The MBRT is sharply decreasing with an increasing number of

Peers. Such scenarios increase the number of Peers with temporally complementary requests patterns, in turn leading to more opportunities of Resource sharing. The MBRT is constant and optimal with the number of Resources per Peer.

Finally, when the number of Peers, Resources per Peer and submitted BoTs are all increased by the same factor, the larger number of additional Resources compensate for the larger number of BoTs and thus does not increase the MBRT, so the total simulated time also remains constant. The MBRT is also optimal. The simulator performance expectedly decreases, as the larger number of Peers increases the simulation time.

The $O(.)$ complexity of the simulator is $O(n_{evt} \times c_{evt} + t_{sim} \times n_{peers} \times c_{barter})$, where:

- n_{evt}: total number of simulator events injected in the event list;
- c_{evt}: cost of processing of one event (which depends on the event type);
- t_{sim}: total simulation time (in simulated time units);
- n_{peers}: total number of Peers of the scenario;
- c_{barter}: cost of bartering operations for one Peer (influenced by many factors).

The presented results are compatible with this complexity model: More submitted BoTs - i.e. more injected simulator events - yield a linear decrease in simulator performance; more Peers yield a more-than-linear decrease in simulator performance; More Resources - i.e. an increased processing time of events and bartering operations - yield a less-than-linear decrease in simulator performance.

3.6.4 Simulator and Middleware Co-Development

Chronology of Development

In this dissertation, the middleware implementation of the Lightweight Bartering Grid architecture has been presented before the simulator implementation. This order of presentation was selected to provide an easier or more intuitive reading and understanding of both implementations.

In practice however, most common components (i.e. bartering code) have been implemented concurrently with the simulator, in a main development branch (April-August 2006). The middleware implementation (network and data management code) was later implemented mainly by another developer [99] in a separate, secondary development branch, that has been regularly synchronized with the main development branch (September 2006-April 2007). The implementation of multithreading in the middleware has been completed in the main development branch

during the winter 2006-2007. In parallel with the middleware implementation, the development of supplementary common components, simulator features, and middleware multithreading code continued in the main development branch (essentially September 2006-May 2007). Multiple bugs have been fixed and additional features have been added to the main development branch thereafter.

Benefits Derived from Simulator and Middleware Co-Development

The simultaneous implementation of the middleware and the simulator - the *code once, deploy twice* pattern - has been made possible thanks to the structuring of communications following the handle/service pattern (see Section 2.8). The development overhead due to the synchronization of the two development branches was actually very small. This development process has worked very well in practice and has demonstrated the following benefits:

- separation of concerns → simplification of the development of both implementations, as well as of the common code;
- behavior of Peer components thoroughly simulated, and late addition of the middleware multithreading code → faster and easier testing and debugging of the middleware implementation, as the common code could be considered reliable when multithreading was added;
- large common code base with cleanly designed interfaces → smaller middleware implementation, that is easier to maintain, or to upgrade in order to support future standard Grid protocols [26];
- parallel development of the simulator and the middleware → reduced time span between the development of new algorithms and their deployment.

3.6.5 Comparison of Software Engineering Practices

The co-development of the middleware and the simulator implementations has enabled us to develop a P2P Grid middleware about half the size (in terms of lines of code and probably features and ease of use as well) of the current state of the art, operational middleware (OurGrid 3.3.2), with probably only a fraction of the time and effort (see Table 3.4).

OurGrid

OurGrid results from a multi-year, multi-developers effort. Distributed development by multiple small teams introduced some challenges of code integration [103,

	OurGrid 3.3.2 (all)	OurGrid 3.3.2 (w/o UT)	Lightweight Bartering Grid 0.4.1
Development	since 2003	since 2003	since April 2006
Classes	772	643	508
Lines	102341	79256	70528

Table 3.4: Comparison of OurGrid (full version, version without unit testing code), and Lightweight Bartering Grid.

52]. Aspect-Oriented Programming [23], which is concerned with so-called *cross-cutting concerns*, is a software engineering pattern that has provided OurGrid with a development methodology enabling to refactor initial implementations into a stable OurGrid 2.2 release [103, 102].

Other software engineering patterns that proved to be very useful for OurGrid are event-based modelling of Grid nodes and unit testing (i.e. automated tests of independent units of source code).

Lightweight Bartering Grid

Both implementations of the Lightweight Bartering Grid (the middleware and the simulator) are also event-driven [62] (see Section 2.9.7), which helped to separate concerns. However, if the *code once, deploy twice* pattern had not been followed, we believe it would not have been possible to develop to an operational status a completely new P2P Grid middleware within about a year. Every bug in the bartering code was isolated in the controlled environment of the simulator, either before it could appear or right after it had appeared during execution of the middleware.

The use of Grid computing itself to improve the quality of Grid computing software - through the completion of a large number of test cases - is discussed in Chapter 6. Complementarily, when an error or unexpected exception occurred when executing the middleware, we used the simulator with identically configured Grid nodes to trace the cause of the problem. The difficulty of testing code on a real network of Resources has thus been avoided.

However, a run-time monitoring tool [325] providing the state and stack of all threads of target Grid nodes had to be used to isolate distributed deadlocks. Indeed, a couple of distributed deadlocks in the Peer middleware were detected only when it was deployed on a few dozen Grid nodes. Specifically, the Negotiator of one Peer and the filtering and queueing components of another Peer were waiting for one another to release the lock on their own RMS, although the RMS was

locked by operations depending on other Peers. This was an implementation issue that forced to check that Grid-facing Peer components were indeed asynchronous. The domains of systematic unit testing, Aspect-Oriented Programming [102] and Formal Verification [313] can also certainly provide efficient tools at different levels of automation to prevent such issues that arise from massively multithreaded code.

3.7 Summary of the Contributions

Software engineering challenges arise from the distributed and unreliable nature of the P2P Grid environment, where many elements are beyond the control of P2P Grid administrators and researchers. A discrete-event system simulator of P2P Grid can be a very helpful **software engineering tool facilitating the development, testing, debugging and performance evaluation of P2P Grid software.** To this end, we have contributed the first large-scale, top-down application of the *code once, deploy twice* pattern [63, 59], introduced by the contemporarily proposed GRAS (Grid Reality And Simulation) component [253] of the SimGrid [78, 278] middleware.

Grid nodes are virtualized, which consists of abstracting their environment as well as the time-consuming operations (e.g. Task execution). The code of a discrete-event P2P Grid simulator is weaved into the code of the virtualized middleware at boundaries between Grid nodes and their environment. This contrasts with the contemporarily proposed GRAS component [253] of SimGrid [78, 278], which is designed as an "underware" [175] to be used in Grid middlewares. Moreover, support for the simulation of multithreaded code - which is required for the virtualization and simulation LBG - was added only recently to GRAS/SimGrid [78] through the SimIX component.

The virtualization of Grid nodes enables the execution of a whole P2P Grid in a controllable and reproducible way on a single computer, thus allowing full observation of their bartering behavior. **Given the massive code reuse introduced by the *code once, deploy twice* pattern, and given the speed of discrete-event simulation** (that can be several orders of magnitude faster than its execution on real computers), **new bartering policies can be easily simulated, integrated and deployed** as they are run similarly in both the middleware and the simulator implementations.

In practice, we first implemented the basic bartering code of the Lightweight Bartering Grid architecture, along with the simulator implementation. The actual implementation of the middleware implementation started several months later and was greatly facilitated because Peer bartering policies - which are a complex

part of the P2P Grid - could always be tested and validated in the simulator. As separation of concerns was largely achieved, it was possible to focus on network-related issues when developing the middleware.

There are open questions and possibilities to extend the features of the proposed discrete-event P2P Grid simulator. Data transfers are currently not taken into account. Simulating data transfers is certainly the next step in the evolution of the simulator, as this will considerably improve simulator accuracy when dealing with Data-Intensive BoT (which are investigated in Chapter 5). Multithreading is simulated but its implementation may be coupled too tightly to design choices specific to the Lightweight Bartering Grid architecture. Recent work on the automated testing of multi-threaded code [102] should be investigated and possibly integrated.

It would also be interesting to investigate the convergence between our proposed approach of *code once, deploy twice* - which implies the weaving of the simulator code into the middleware code - and a pattern centered around the development of a low level API, proposed for SimGrid [78, 278]. Of practical interest would be the standardization of the description of Grid configurations and deployment metadata [78, 278], as well as the interface of bartering algorithms [201] and trace workloads [80, 140, 238, 163]. This would enable multiple research teams to easily plug one another's algorithms into their own P2P Grid simulator and evaluate these for well-known workloads on well-known Grids. Also of practical interest, a graphical user interface would enable the easy editing of Grid configurations [288, 69].

Finally, there are several possibilities to scale up the proposed simulator. Limited multithreading of the environment controller could improve simulator performance on multi-core CPU. Structuring of the simulator itself as an Iterative Stencil application (see Section 2.4, Chapter 6) could lead to a fully distributed simulator. This would involve simulating in parallel the execution of Grid nodes that should run concurrently. The communication costs between the distibuted chunks of the simulator might however be a limiting factor.

Chapter 4

Bartering Guidelines

Opportunities are a tricky crop,
with tiny flowers that are difficult to see
and even more difficult to harvest.

- in Legends of Dune,
by Brian Herbert & Kevin Anderson

A challenge in a middleware as complex as P2P Grid is to design efficient scheduling policies. The Peer scheduler and the Peer negotiator models are structured around Policy Decision Points (PDP). This enables to systematically explore a large number of different possibilities and ideas. This constitutes a mechanism that enables to combine basic policies in a multitude of ways. Several policies are thus proposed for each PDP. Within the scope of the experiments performed using the LBG simulator, it is confirmed that bartering (i.e. cooperation between Peers) is efficient, and that filtering the submitted Supplying Tasks is required to prevent queue wait times to grow out of control. The adaptive preemption policy performs best for the conducted experiments. It is also observed that, despite attempts to model the reliability of supplying, random selection of suppliers is most efficient, which confirms design choices arbitrarily proposed in related works. Experiments are performed using the LBG simulator.

Task execution may fail for multiple reasons. Resources may fail at the hardware-level, at the O.S.-level or (despite the software engineering tools introduced in the previous Chapter) at the middleware-level. Single Resources or even whole Peers can suddenly leave the Grid. Resources are edge computers, thus intrinsically unreliable. Preemption of Task execution is another major cause of Task execution failure: Supplying Tasks can be preempted or cancelled[1] when a Peer reclaims the computational power of its Resources to process its own Local Tasks.

[1]Reminder: cancellation = preemption without subsequent requeueing.

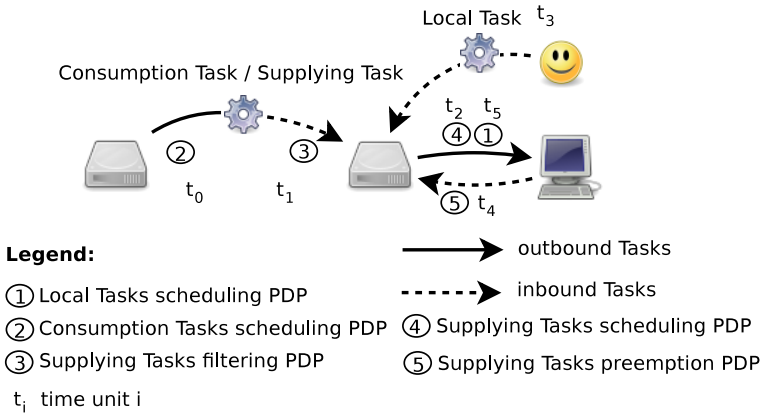


Figure 4.1: Typical interactions leading to the preemption of a Supplying Task.

These interactions are illustrated on Figure 4.1 (see also Figure 2.20 for the Peer scheduling model): the consumer Peer (on the left) submits a Consumption Task to the supplier Peer (on the right). If this Task - perceived as a Supplying Task by the supplier Peer - is not filtered out, it is queued and eventually run on a Resource of the supplier Peer. When a User Agent submits new Local Tasks to the supplier Peer, the latter can preempt or cancel the running Supplying Task. If the Supplying Task is preempted, the supplier Peer requeues it. If the Supplying Task is cancelled, the supplier Peer notifies the consumer Peer of the cancellation. This cancellation is perceived as a Task execution failure by the consumer Peer. Given the opacity between Peers, the consumer Peer does not know whether it was caused by preemption or Resource failure.

In this chapter, scheduling policies are introduced in order to prevent, or at least reduce, the loss of computing time due to Task execution failures. The purpose of designing robust scheduling policies is maintain short MBRTs in the presence of frequent Task execution failures. Even though some of the proposed fault-avoidance and fault-prevention mechanisms may seem straightforward at first sight, they are actually not trivial to design because of the informational opacity between Peers. To also enable Peers to not barter computing time, appropriate policies, e.g. “no scheduling”, are defined for each policy decision point (PDP). A Peer should use together either all or none of these policies designed to prevent bartering.

This chapter is structured as follows. Policies for the Local Tasks scheduling, Supplying Tasks scheduling and Supplying Tasks filtering PDPs are first introduced. A model, worthwhile metrics, metadata storage and policies for the Consumption Tasks scheduling PDP are successively introduced; related work is also reviewed. Policies for the Supplying Tasks preemption PDP are finally introduced. Bartering guidelines are derived from experimental results and the chapter is summarized.

NonpreemptiveLocalScheduling	preeption deactivated
PreemptiveLocalScheduling	preeption activated

Table 4.1: Local Tasks Scheduling Policies.

4.1 Local Tasks Scheduling

Local Tasks scheduling consists in matching a queued Local Task with an available Resource. Two Local Tasks scheduling policies are currently implemented: with and without preemption. With preemption, the Supplying Tasks preemption policy is activated when there are queued Local Tasks and no available Resource. The preemption of running Supplying Tasks enables the scheduler to reclaim additional Resources for the scheduling of Local Tasks.

BoT selection follows a FIFO policy (see Section 2.6.2), i.e. Task selection is applied to all unscheduled Task of the oldest - i.e. least recently submitted - queued BoT, then to the next one, and so on towards the newest BoT. By default, Task selection also follows a FIFO policy, but a data-aware Task selection policy is introduced in Chapter 5. By default, Resource selection follows a random policy, but a data-aware Resource selection policy is also introduced in Chapter 5.

Table 4.1 lists the proposed Local Tasks scheduling policies. Task selection and Resource selection are orthogonal to the selected Local Task scheduling policy.

4.2 Supplying Tasks Scheduling

Supplying Tasks scheduling consists in matching a queued Supplying Task with an available Resource. Three Supplying Tasks scheduling policies are currently implemented: “no scheduling”, FIFO-based and favors-based. By default, Resource selection follows a random policy, but a data-aware Resource selection policy is also introduced in Chapter 5. Task selection and BoT selection are the same operation as computing time is exchanged at the Task level.

With the FIFO-based policy, the oldest - i.e. least recently submitted - Supplying Task in queue is selected. With the favors-based policy, the Supplying Task is selected according to the Network of Favors model (see Section 2.3.4): The Supplying Task submitted by the consumer Peer with the currently highest favor balance is selected; ties are broken arbitrarily. Finally, one policy has no effect, i.e. Supplying Tasks are not scheduled; a Peer can be configured with this policy to prevent the sharing of the computing time of its Resources.

FIFOSupplyingScheduling	FIFO-based consumers ranking
FavorsSupplyingScheduling	favors-based consumers ranking
NoSupplyingScheduling	no bartering

Table 4.2: Supplying Tasks Scheduling Policies.

Table 4.2 lists the proposed Supplying Tasks scheduling policies. Resource selection is orthogonal to the selected Supplying Task scheduling policy.

4.3 Supplying Tasks Filtering

Supplying Tasks filtering consists in accepting or denying the queueing of a Supplying Task submitted by a consumer Peer. The purpose of filtering is to prevent the Supplying Tasks queue to grow when no Resource is available. Five policies are currently implemented: "no filtering" (i.e. all Supplying Tasks are always accepted), total filtering (i.e. Supplying Tasks are never accepted), one FIFO-based filtering policy and two favors-based filtering policies.

The FIFO-based as well as both favors-based filtering policies accept Supplying Tasks until the number of outstanding Tasks exceeds a threshold configured by the human Peer administrator, typically to a very low value. The number of outstanding Tasks of a supplier Peer is defined as the number of waiting Local Tasks, plus the number of waiting Supplying Tasks, minus the number of idle Resources. Intuitively, it gives the queue length - expressed as a number of Tasks - that a newly accepted Supplying Task would find in front of itself.

The FIFO-based filtering policy only considers the number of outstanding Tasks, without inspecting the contents of the Supplying Tasks queue, as opposed to the two favors-based filtering policies. Both favors-based filtering policies follow the Network of Favors model (see Section 2.3.4): The favor balances of all the consumer Peers with queued Supplying Tasks, as well as that of the consumer of the Supplying Task under consideration, are evaluated. If the balance of the incoming Supplying Task has the smallest value, it is filtered out. The rejection of the Supplying Task is communicated to the consumer, but if the GNMP message were lost, the Consumption Tasks control of the consumer Peer (see Section 2.9.6) would automatically initiate the cancellation of the Task. Otherwise, it is accepted and the queued waiting Supplying Task associated with the smallest favor balance is dequeued (ties are broken by dequeueing the most recently submitted - and still waiting - Task among those submitted by the consumer Peer with the smallest favor balance).

FIFOSupplyingFiltering	FIFO-based consumers ranking
NoSupplyingFiltering	no bartering
RelaxedFavorsSupplyingFiltering	relaxed favors-based consumers ranking
StrictFavorsSupplyingFiltering	strict favors-based consumers ranking
UnlimitedSupplyingFiltering	no filtering

Table 4.3: Supplying Tasks Filtering Policies.

The two favors-based filtering policies differ in that one of them strictly enforces the NoF model: Running Supplying Tasks are also considered in the filtering process. The favors-based filtering policies are very important to prevent free-riding: Supplying Tasks submitted by consumer Peers with a positive favor balance will replace in the queue any Supplying Task submitted by consumer Peers with a null favor balance, i.e. free-riders.

Table 4.3 lists the proposed Supplying Tasks filtering policies.

4.4 Metrics for Consumption Tasks Scheduling

4.4.1 Requirements for Metadata

A consumer Peer has no knowledge of the state of other Peers, in particular, queue state, scheduling policies, peaks of Local Tasks, reliability of Resources. If such metadata were transmitted and trusted, as in Desktop Grids, ensuring their accuracy and freshness would be difficult at best. Moreover, the architecture would tend to become centralized as Peers would process large amounts of metadata about many other Peers, which would limit scalability.

Instead, we propose [60, 58] that Consumption Tasks scheduling policies select supplier Peers based on historical metadata collected following past completed interactions with supplier Peers. Trust[2] between Peers is built over many completed interactions rather than a priori. As consumer Peers rely only on their own perceptions, the number of exchanged control messages is minimized.

The bartering process of a consumer Peer can be modelled as a decide-interact-memorize loop (see Figure 4.2). This constitutes an evolution of the Network of Favors (NoF) bartering model [13] (see Section 2.3.4) used in the related OurGrid [233, 84, 286] middleware. The original NoF model is concerned with equi-

[2] According to Sober and Wilson [284, 142]: *"Trust is the consequence or state when one or more members of a network perform according to mutual expectation. It is not an abstract moral virtue, but a network property, byproduct of the quality of interactions between parties."*

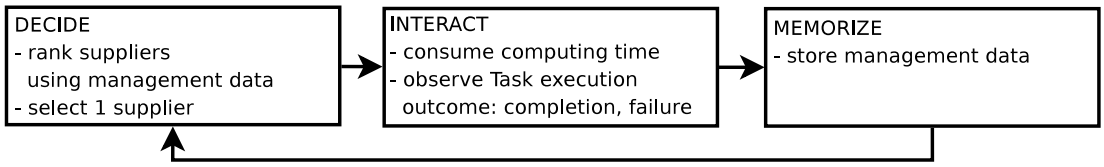


Figure 4.2: Decide-interact-memorize loop. Decisions are made by the Consumption Tasks scheduling PDP, based on information collected from completed interactions.

table supplying, i.e. resistance to free-riding, and relies on Task replication [86] as a mechanism to compensate Task execution failures. It therefore does not study preemption and cancellation [14]. To the contrary, we take preemption and cancellation into account.

Several basic metadata and metrics are now examined. They are depending on bartering-related, Task-related, negotiation-related and Resource-related information. Some of them are selected, others are discarded, based on their computability and storability by desktop computers.

4.4.2 Bartering-related Metadata

Each consumer Peer can store as metadata the outcome of the execution of Tasks (completed, failed[3], filtered out). The time elapsed between Task submission and Task execution outcome can also be stored, either as an execution time or a time-to-failure.

Metadata is collected by a consumer Peer upon Peer external events (see Section 2.9.2):

- submission of a Consumption Task,
- uploading to a consumer Peer of results from a Consumption Task completed by a supplier Peer,
- consumer-initiated or supplier-initiated cancellation of a Consumption Task.

Task Completion Ratio (TCoR) A first, obvious metric to estimate supplier Peer reliability is the Consumption Task Completion Ratio (TCoR). The TCoR of Consumption Tasks is defined as the ratio between the number of Consumption Tasks that were successfully completed and those that were accepted by the

[3] Given the opacity between Peers, a consumer Peer cannot determine if the cancellation of a Task by a supplier Peer has been caused by Resource failure or preemption.

supplier Peer and for which the interaction is completed. The TCoR is a dimensionless metric. A high TCoR indicates good reliability of the supplier Peer, but not necessarily good performance. This type of simple metric is typically used in Volunteer Grids [134] to estimate the availability of worker nodes because it is simple to compute and store.

Task Cancellation Ratio (TCaR) A complementary metric of TCoR is the Consumption Task Cancellation Ratio (TCaR). The TCaR of Consumption Tasks is defined as the ratio between the number of Consumption Tasks that were cancelled and those that were accepted by the supplier Peer. The TCaR is a dimensionless metric. A high TCaR indicates either poor reliability of the supplier Peer, or very poor performance, as the recorded cancellations might have been self-initiated by the consumer Peer subsequently to a time-out (the origin of the cancellations is not stored).

Mean Correlated Consumption Cancellations Count (MC4) Preemption often leads to multiple correlated Task execution failures as a supplier Peer reclaiming its Resources to complete Local Tasks would typically need more than one of them. To the contrary, Task execution failures caused by failures of individual Resource are typically independent, except in special cases such as when multiple Resources are deployed on a multi-core computer. Following the hypothesis that *"large-scale correlated failures are common"* [320], the Mean Correlated Consumption Cancellations Count (MC4) metric is thus introduced.

MC4 is defined as the mean of the sizes of correlated cancellations sets. The timestamps of the most recent cancellations of Consumption Tasks are stored by each consumer Peer. When a set of these cancellations occur within a short time span, they are considered to be correlated. If one cancellation is not correlated with any other cancellation, it is considered as a set of size one. The MC4 is a dimensionless metric. A high MC4 indicates a tendency to correlated cancellations.

Mean Time Between Cancellations (MTBC) The Mean Time Between Cancellations (MTBC) measures the average time elapsed between successive cancellations of Consumption Tasks by a supplier Peer. The MTBC is defined as the mean of the time spans delimited by successive cancellations of Consumption Tasks. The MTBC is expressed as a number of time units, i.e. seconds. The MTBC can be seen as a measure of availability inspired by a classic metric in reliability engineering, the Mean Time Between Failure (MTBF). The MTBC can be used as an estimation of the amount of "uninterrupted attention" that a supplier Peer gives to a Consumption Task. A high MTBC indicates that a Consumption Task

has a higher probability of being successfully completed, i.e. not being cancelled.

Results from related work [320], interpreted in the context of this dissertation, indicate that a high TCoR does not necessarily lead to a high MTBC. Indeed, most Consumption Tasks may complete successfully, with only a small percentage being cancelled (high TCoR), while the time spans between the few cancellations (low MTBC) may be short. Moreover, according to the same related work [320], while the MTBC has some relevance, estimating the time of the next cancellation appears not so trivial. Another limitation of the MTBC is its sensitivity to Task runtimes.

Mean Completion Stride (MCoS) A stride is defined as a sequence of identical Task events, i.e. Task completion or cancellation, and its length is its number of events. The Mean Completion Stride (MCoS) of Consumption Tasks is defined as the mean length of strides of successfully completed Consumption Tasks. The MCoS is expressed as a (real) number of completed Consumption Tasks. The MCoS provides an indirect estimation of the length of the time spans during which cancellations of Consumption Tasks do not occur. Like the MTBC, the MCoS can be used as an estimation of the amount of "uninterrupted attention." The MCoS can be seen as a measure of availability of a supplier Peer in terms of bursts of successfully completed Tasks.

The MCoS and the MTBC are closely related metrics. Combining them may yield a better estimation of availability. Both rankings can be combined by assigning the worst normalized rank (out of the MCoS-based and the MTBC-based ranks) for each supplier Peer. In practice, a set of supplier Peers can be ranked based on the MCoS and the MTBC, thus producing two independent rankings. Each rank of both rankings can then be normalized, i.e. divided by the number of ranked supplier Peers. Finally, the worst of its two normalized ranks is assigned to the supplier Peer. We call this metric the conservative time/stride.

Mean Time To Cancellation (MTTC) The Mean Time To Cancellation (MTTC) is defined as the mean of the time spans that occur between the submission of a Consumption Task and its cancellation (the MTTC is thus defined for cancelled Consumption Tasks only). It is a good measure of the loss of time incurred by cancellations. The MTTC is a variant of the Mean Time To Failure, which is widely used in reliability engineering[4].

[4] A Weibull distribution [16] could also be used to compute it instead of the mean, but the estimation of its parameters is not trivial and could be computationally expensive.

Relying on the MTTC may help to minimize the cost of a cancellation when selecting among several supplier Peers of equal unreliability or unavailability. On one hand, selecting a supplier Peer with a high MTTC increases the probability of successful Task completion. On the other hand, selecting a supplier Peer with a low MTTC may also improve performance, in the sense that the time lost to a cancellation is minimized if it happens early. Our intuition is that it is efficient to select supplier Peers with a high MTTC when they are mostly reliable, and to select Peers with low MTTC when they are mostly unreliable. Such a strategy increases the probability of Task completion when Task completion is probable, and it decreases the cost of Task execution failure when Task execution failure is probable.

4.4.3 Task-Related Metadata

Task-related dynamic data can be used as metadata. Task execution outcomes, e.g. bartering-related metadata, can be used for reliability-aware supplier Peer selection, as explained in the previous paragraphs. Metadata related to transfers of input data files by Resources can be used for data-aware supplier Peer selection, as will be further discussed in Chapter 5.

Task-related static data, such as input parameters (see Section 2.6.3) or filenames of input data files, can be used as metadata. Such metadata, although they can be put to good use [188], are highly application-dependent. Moreover, they provide no information if a Task has no input parameter or no input data file. For example, no input data file is associated to web crawling Tasks or more generally to Tasks that download their data from an out-of-Grid data server (see Section 6.2). Runtime estimates (see Section 2.6.3), if they were available, could also be used.

4.4.4 Negotiation-related Metadata

The proposed negotiation model (see Section 2.9.5) is designed so that supplier Peers can provide hints of their availability to the Consumption Tasks scheduling PDP of consumer Peers. The number of consumption grants sent by a supplier Peer can certainly be used as metadata by consumer Peers. The number of supplying requests might also be used, but it is certainly of less interest to consumer Peers than it is to supplier Peers.

4.4.5 Resource-related Metadata

Resource-related static metadata, i.e. hardware parameters such as the CPU clock rate, are certainly useful and constitute an important source of metadata in Desktop Grids [193]. The NodeWiz Grid Information Service [53], discussed previously in this section, could provide such information. However, the opacity between Peers does not allow one to determine with certainty to which Resource a Task is effectively scheduled. The use of Resource-related static metadata is thus limited to the filtering of supplier Peers that do not match specific hardware requirements, but it is still up to the supplier Peers to schedule Supplying Tasks to Resources.

Resource-related dynamic metadata could be used as metadata, if available. Metadata communicated by supplier Peers should not be trusted in P2P Grids. Furthermore, communicating up-to-date dynamic metadata would limit the scalability of the P2P Grid architecture. Actual CPU availability [16, 71, 260], a widely used metadata, thus cannot be used in the P2P Grid context, especially in 2-levels P2P Grids.

Dynamic performance benchmarks [83, 114] could be initiated by consumer Peers. The Peer middleware of a consumer Peer could periodically schedule benchmarking Tasks to supplier Peers. However, besides the obvious cost in computing time, the opacity between Peers does not allow one to determine with certainty to which Resource a Task is actually scheduled.

Nonetheless, there are two very specific forms of dynamic benchmarking that could be useful. Firstly, a Task could perform its own dynamic benchmark before starting its intrinsic execution, before deciding how much data to process (this will be further discussed in Section 6.2). This requires, however, supplementary work on the behalf of the Grid application developer. Secondly, a consumer Peer could perform dynamic benchmarks using Tasks of fixed runtime (i.e. elapsed time), e.g. sleep 2 minutes. This can be used to estimate the queue length of supplier Peers, but the proposed negotiation protocol (see Section 2.9.5) has been specifically designed to remove the need for such benchmarking. It should be noted that both this benchmarking with Tasks of fixed runtime and the proposed negotiation protocol could equally be fooled by a malicious supplier Peer; however, the cost of repeated negotiations is certainly less than the cost of repeated benchmarking.

4.4.6 Metadata Storage

The *neighborhood* of a Peer is defined as the set of Peers with which interactions either have taken place or could take place, i.e. Peers whose handles (see Section 2.8) have been communicated by the Search Engine (see Sections 2.1.4 and 2.9.5). Each

Peer stores metadata that summarizes its perception of the recent state of the Grid. These metadata are collected implicitly only through the interactions of each Peer with Peers belonging to its neighborhood, without any explicit communication of metadata.

The high number of interactions with other Peers during a Peer's lifetime, i.e. online time, precludes the storage of all metadata about every interaction. Therefore, metadata should be stored either in aggregate form, or only within a window of recent interactions. It should be noted that even if all metadata could be stored, their retrieval should be efficient enough to prevent the scheduling operations to become a performance bottleneck.

The Peer register is the Peer component that stores metadata about supplier Peers belonging to the neighborhood of the Peer running the Peer register. The Peer register stores a profile of each supplier Peer belonging to the Peer's neighborhood, as well as a Grid-level profile. A Peer profile stores collected and aggregated bartering-related metadata communicated by the Task Manager (see Appendix C.1), i.e. completion, cancellation and failure of Supplying Tasks, about the profiled supplier Peer. The metadata required to compute the MC4, MTBC, MCoS, MTTC metrics must be stored for a window of the last K completed interactions; K is currently arbitrarily chosen as 100. The other metadata can be stored in aggregated form if only the mean of the stored values needs to be known.

A Peer profile also stores bartering-related metadata about Supplying Tasks submitted by consumer Peers; this can be used by the Peer when it acts in a supplier role (i.e. as a supplier Peer). It also stores negotiation-related metadata (see Section 2.9.5), i.e. supplying requests and consumption grants sent to, and received from, other Peers. The Grid-level profile stores aggregated bartering-related and negotiation-related metadata, as well as metadata on the input data files of queued Tasks. The implementation of the Peer register is currently not persistent; as future work, persistence support will be added to the Peer register.

Finally, it should be noted that, in the absence of collected metadata, the selection of supplier Peers is reduced to a random choice (i.e. the decide step of Figure 4.2 makes random selections when the interact and memorize have not been activated for the considered suppliers). This is important as it enables the bootstrapping of bartering in the P2P Grid.

4.5 Consumption Tasks Scheduling Policies

Consumption Tasks scheduling consists in matching a queued Local Task with a supplier Peer that sent consumption grants; preferred supplier Peers are those with good reliability and availability. Fifteen Consumption Tasks scheduling policies are currently implemented. These policies use historical metadata about computational resources but no metadata on Tasks; this corresponds to the (H, U) type in Iosup et al.'s classification of scheduling policies [176]. The random policy randomly selects supplier Peers. One policy has no effect, meaning that no Consumption Tasks are scheduled. The other policies are described in the next sections.

4.5.1 Reliability-Aware Consumption Tasks Scheduling

Some authors argue that stability is a form of robustness [210]; an interesting observation is that *"adding capacity and reducing variability are, in some sense, interchangeable options"* [210]. Following this perspective, several policies are now proposed with an objective of fault-avoidance. The reliability of a set of supplier Peers is estimated based on metrics introduced in the previous section. These metrics depend only on the recent behavior of the supplier Peers, not on the Tasks; this is consistent with the previous decision not to require runtime estimates from the Grid application developers (see Section 2.6.3). The value of a given metric is computed for each supplier Peer; a high value indicates a higher probability of successful completion of Task execution. The supplier Peers can be ranked according to the computed values. The supplier Peers are then selected following this ranking - most reliable (i.e. higher-ranked) first - until all available consumption grants have been spent.

The MC4-based, MCoS-based, MTBC-based, MTTC-based, TCaR-based and conservative time/stride-based policies rank and select supplier Peers using the eponym metrics presented in the previous section.

The grants-based policy ranks and selects supplier Peers by decreasing amount of consumption grants they sent during the most recent negotiation interaction, i.e. supplier Peers that declare explicitly a high availability are ranked higher. This, of course, is not consistent with the lack of trust between Peers.

The TCoR, TCaR and MC4 metrics can be made adaptive to blacklist supplier Peers at the Peer-level. An adaptive TCoR policy is introduced based on the rationale discussed in the previous section: The MTTC of reliable supplier Peers should be high and the MTTC of unreliable supplier Peers should be low. After supplier Peers are ranked, those with a high TCoR (i.e. more than 50% of success-

ful completion of Task execution) and a low MTTC (i.e. less than two minutes) are blacklisted. The supplier Peers with a low TCoR (i.e. less than 50% of successful completions) and high MTTC (i.e. more than two minutes) are also blacklisted.

An adaptive MC4 policy is introduced. Our intuition is to minimize the impact of multiple execution failures The supplier Peers are first ranked with the MC4 metric. The supplier Peers with a high MC4, i.e. at least two simultaneous Task execution failures on average, are scheduled in a round-robin fashion over multiple activations of the Consumption Tasks scheduling policy during a given activation of the Peer scheduler, i.e. until all waiting Local Tasks have been scheduled or all recently received consumption grants have been used. This tends to reduce the number of Consumption Tasks to supplier Peers with a high MC4.

A BoT-level blacklisting mechanism is added to the ranking computation of all metrics. Supplier Peers with a very low probability of successful Task execution at the BoT-level are blacklisted, i.e. excluded from the ranking (see Section 4.6.2 for a discussion of the rationale). A supplier Peer is estimated not to be able to successfully complete the execution of a Consumption Task if the number of execution failures of Consumption Tasks from the same BoT than the Task to schedule exceeds a fixed threshold that can be configured by the human Peer administrator, typically 5 [233]. This mechanism prevents to repeatedly reschedule Consumption Tasks to a supplier Peer that consistently exhibits execution failures. This mechanism also implicitly discards supplier Peers which are, in some sense, incompatible with the Tasks from the considered BoT, e.g. not enough RAM actually available, without requiring a complex, dynamic matchmaking mechanism.

A Peer-level blacklisting mechanism is also added to the ranking computation of several metrics: TCaR, TCoR, Adaptive MC4 and Adaptive TCoR. Supplier Peers with extreme TCoR/TCaR/MC4 values are blacklisted. In order to bootstrap the bartering process between Peers and to allow previously unreliable supplier Peers to increase their reputation of reliability, the blacklisting is probabilistic. If an unreliable supplier Peer is selected for blacklisting, it is actually blacklisted only if a random variable is below an activation threshold; this threshold is configured by the human Peer administrator, typically a high value such as 0.8.

4.5.2 NoF-Aware Consumption Tasks Scheduling

For the evaluation of supplying requests (see Section 2.9.5), the Network of Favors model (NoF, see Section 2.3.4) can be used by a supplier Peer to rank consumer Peers based on their bartering reputation. We hypothesize that the NoF can also be used by a consumer Peer to rank supplier Peers through their favor balance and favors history. Supplier Peers could thus be ranked based on the amount of

computing time they supplied in a recent past. The favors-based policy follows the NoF model to rank and select supplier Peers by decreasing amount of supplied favors, i.e. supplier Peers that supplied a lot in the past are ranked higher.

4.5.3 Reciprocity-Aware Consumption Tasks Scheduling

A consumer Peer could also estimate its own capacity of reciprocal supplying to other Peers. This could enable this consumer Peer to select as supplier Peers those to which it has itself been a reliable supplier Peer in a recent past. Those supplier Peers that have reliably received computing time from this consumer Peer in a recent past (when the consumption and supplying roles were reversed) rank it higher when asked to reciprocate. Doing so tends to reinforce the bartering relationships of Peers that have temporally complementary peaks of local requests (see also Appendix D.3.6 for additional discussion).

For instance, a consumer Peer could examine its own history of preemptions and cancellations of Supplying Tasks; using, e.g. the TCaR metric (see Section 4.4.2), it could try to detect if it had to repeatedly preempt or cancel the Supplying Tasks of some of its consumer Peers. This Peer can then consume preferably from Peers to which it was able to reliably supply computing time. In the current implementation, no Consumption Tasks scheduling policy takes the capacity of reciprocal supplying into account but this certainly constitutes interesting future work.

4.5.4 Performance-Aware Consumption Tasks Scheduling

As in OurGrid [84], supplier Peers ranking currently does not take into account *"how quickly or slowly the work was performed"* [14]. Correctly predicting the computational performance of supplier Peers may be difficult because the expected runtime of Tasks is not provided by Grid application developers (see Section 2.6.3), and also because Consumption Tasks are submitted to supplier Peers, not directly to their Resources. For these reasons, it may not be possible in a P2P Grid to ever support advance reservations (with guaranteed nonpreemptible execution) or systematically meet hard QoS deadlines, without a huge cost overhead, e.g. massive Task replication.

However, ranking supplier Peers on their estimated computational performance can decrease response times if scheduling Consumption Tasks to supplier Peers with good (or at least not extremely low [193]) computational performance.

One Consumption Tasks scheduling policy based on performance ranking - thus similar to DFPLT [176] - is implemented. Performance is estimated based on the

mean completion time of Consumption Tasks recently completed by the considered supplier Peers.

Supposing that this policy is systematically used, each consumer Peer would tend to systematically consume computing time from supplier Peers with better computational performance than itself, if available. These would not tend to do the same, as they would first avoid supplier Peers slower than themselves. The Grid-wide impact of such a performance-based policy is thus not certain. Adding support for awareness of reciprocal supplying would probably be of high interest.

4.5.5 Data-Aware Consumption Tasks Scheduling

Supplier Peers can also be ranked according to the expected availability of input data files on their Resources. Response times can be decreased if Tasks are scheduled to supplier Peers that have Resources where input data files are already stored. Data-aware scheduling policies are introduced in Chapter 5. In particular, one Consumption Tasks scheduling policy is introduced to take into account the expected availability of input data files on supplier Peers. A variant of this policy is also introduced: Supplier Peers with a very low TCoR, i.e. less than 20%, are blacklisted even if it is expected to store all input data files required by the Consumption Task to schedule.

4.5.6 Resource Ranking

Some of the mechanisms proposed for Consumption Tasks scheduling in the previous sections could be applied to Local Tasks and Supplying Tasks scheduling. When selecting a Resource, a Local Tasks or Supplying Tasks scheduling policy could rank the Peer' Resources. In particular, the performance and reliability of Resources could be taken into account in future work. Existing work by Kondo et al. [192, 193], Ren et al. [261, 260], Dinda [118], Kapadia et al. [188], Smith et al. [282], for instance, could be integrated in order to improve site-level scheduling. The availability of input data files on Resources is systematically taken into account, as explained in Chapter 5.

4.5.7 Summary of Consumption Tasks Scheduling Policies

Table 4.4 lists the proposed Consumption Tasks scheduling policies. The policies involving on the MC4, favors and data-aware metrics are 100% original, while the others are derived from existing metrics.

AdaptiveMC4	adaptive MC4-based suppliers ranking
AdaptiveTCoR	adaptive TCoR-based suppliers ranking
ConservativeTimeStride	combined MCoS/MTBC-based suppliers ranking
Data	data-aware suppliers ranking
Favors	favors-based suppliers ranking
Grants	consumption grants-based suppliers ranking
MC4	MC4-based suppliers ranking
MCoS	MCoS-based suppliers ranking
MTBC	MTBC-based suppliers ranking
MTTC	MTTC-based suppliers ranking
NoConsumptionScheduling	no bartering
Performance	performance-based suppliers ranking
Random	random suppliers ranking
Reliable	TCoR-based then data-aware suppliers ranking
TCaR	TCaR-based suppliers ranking

Table 4.4: Consumption Tasks Scheduling Policies.

4.6 Related Work

Typical fault-management mechanisms in distributed systems include fault-tolerance, fault-avoidance and also fault-prevention. Fault-tolerance mechanisms enable a distributed system to resume its operations after the occurrence of a fault. Fault-avoidance and fault-prevention mechanisms steer the operation of the distributed system away from potential faults; fault-avoidance mechanisms try to avoid outcomes that are predicted to be disrupted by faults, while fault-prevention mechanisms try to prevent faults from occurring. Fault-tolerance, fault-avoidance and fault-prevention are thus complementary.

4.6.1 Acquisition of Metadata

Gil et al. have stated that *"without a knowledge-rich infrastructure,"* e.g. a Grid Information Service such as the Network Weather Service [314], *"fair and appropriate use of Grid environments will not be possible"* [157]. In a P2P Grid, the challenge, for a consumer Peer, resides in accumulating knowledge on the behavior of supplier Peers in a totally decentralized way.

Distributed Peer discovery services, such as the recent, fully-distributed, kd-tree-based NodeWiz Grid Information Service [53] are extremely useful, but of very limited relevance to the current context. The NodeWiz overlay distributes metadata that are relatively static, such as hardware configuration. Moreover, these metadata must be provided by supplier Peers themselves, and cannot be independently acquired. Additional notes on Peer discovery services are provided in Appendix D.

The recent AssessGrid [24, 25, 307] project is partially similar to ours in its intent, i.e. assessing risk and reliability of the supplying of computing time. In AssessGrid, however, consumption and supplying are not linked. Furthermore, AssessGrid is not intended to be as lightweight and autonomous as a P2P Grid.

4.6.2 Fault-Tolerance

Fault-tolerance can be achieved through a variety of proactive mechanisms, such as checkpointing and Task replication, and reactive mechanisms such as Task control and Task reexecution.

Task reexecution [71, 37, 21] - also called *eager scheduling* [37] - consists of reexecuting a Task upon failure of its execution, until it is eventually completed. The number of such re-executions can be limited to a configured value. Failure-prone Resources can be blacklisted [233, 191], i.e. Tasks from a BoT with failed Tasks on a given Resource should not be executed on this Resource.

Blacklisting [233, 191, 193] consists of not using, for a BoT, a supplier Peer or Resource that is considered as unreliable, e.g. that failed the execution of multiple Tasks of the BoT [233]. Selecting excessively low thresholds for blacklisting can unduly exclude reliable supplier Peers or Resources, thus leading to lower response times than what could be achieved optimally.

Task replication [86, 308], as explained in Section 2.9.4, consists of having each consumer Peer simultaneously executing multiple replicas of each Task. It clearly brings fault-tolerance but at a cost.

Task control, presented in Section 2.9.6, consists of the preemption/cancellation of Tasks that take too long to complete [37].

Checkpointing, as will be further explained in Section 6.2.3, consists of periodically replicating the state of Tasks to replicas stores. Upon Task execution failure, the state of the failed Task is loaded from one of the stored replicas, so that its execution is restarted at the last checkpoint event.

4.6.3 Fault-Avoidance

Fault-avoidance requires (see Figure 4.2) the acquisition and the storage of metadata, the estimation of the availability and performance of supplier Peers Peers,

as well as their ranking.

Our proposed decide-interact-memorize loop is related to the concept of *autonomic computing* [243, 64] where a system constantly monitors and models its environment to adapt its behavior accordingly.

Rankings can be used to exclude supplier Peers with Resources exhibiting extremely low availability or performance [134, 193]. Rankings can also be used to compare the availability and performance of a set of Resources [296]. Rankings can be generated automatically, or interactively with human assistance [296]. Rankings can be computed following a centralized [8, 43], semi-decentralized [100, 307] or fully decentralized [58, 134] organization, the latter being relevant to P2P Grids.

Migration is a fault-avoidance mechanism [306] that consists of first suspending the execution of a Task when an imminent failure is predicted, then moving the suspended Task to another Resource so that it can be restarted. It depends on both ranking and checkpointing.

4.7 Supplying Tasks Preemption Policies

4.7.1 Preemption of Running Supplying Tasks

Supplying Tasks preemption consists in preempting or cancelling queued Supplying Tasks; it is activated by the Local Tasks scheduling policy as needed. Six Supplying Tasks preemption policies are currently implemented for running Supplying Tasks: minimum preemption, full preemption, minimum cancellation, full cancellation, adaptive preemption, and "no preemption".

The minimum preemption policy preempts and requeues as few running Supplying Tasks as needed, i.e. potentially all of them if there are more waiting Local Tasks than running Supplying Tasks. The minimum cancellation policy preempts but does not requeue - i.e. cancels - as few running Supplying Tasks as needed. The full preemption policy preempts all running Supplying Tasks. The full cancellation policy cancels all running Supplying Tasks. The adaptive preemption policy adaptively selects whether Supplying Tasks should be preempted or cancelled, and if they should be offered a second chance, i.e. a short grace period during which they cannot be preempted; the adaptive preemption policy preempts or cancels as few Supplying Tasks as needed. Finally, one policy has no effect, i.e. Supplying Tasks are not preempted; a Peer can be configured with this policy to explicitly prevent the preemption of Supplying Tasks.

Orthogonally to the choice of preemption policy, a cost-aware Resource selection algorithm that we call PSufferage[5] is introduced. PSufferage selects for preemption/cancellation the Supplying Task that would suffer the least from preemption, i.e. the most recently running one.

4.7.2 Fault-Prevention Through Adaptive Preemption

The perception by consumer Peers of the reliability or the performance of a supplier Peer decreases with the cancellation or preemption of Supplying Tasks; This is a well-known issue [14], but it has received little attention in other P2P Grids middlewares. Our intuition is that fault-avoidance - that is initiated by consumer Peers through efficient Consumption Tasks scheduling - can be complemented by fault-prevention, that can be initiated by supplier Peers.

We propose a fault-prevention mechanism based on an adaptive Supplying Tasks preemption policy. It is based on the following observations:

- the cost (in terms of response time) of preemption/cancellation is not identical to all Supplying Tasks;
- the impact of preemption/cancellation on the reliability and performance of a supplier Peer depends on the length of its Local Tasks queue.

The response time of a Supplying Task selected for preemption/cancellation is not greatly affected if it had been running for a short time only. Conversely, its response time is greatly affected if it had been running for a long time. We thus introduce the concept of *second chance*: When a long-running Supplying Task is selected for preemption/cancellation, it is given a short grace period in which, it is hoped, it will eventually complete its execution. After this short grace period during which the Supplying Task is nonpreemptible/noncancellable, the Supplying Task is finally preempted/cancelled if it is still running.

By introducing some small delay in the reclaiming of Resources by a supplier Peer, it is expected that Supplying Tasks that are about to complete their execution are not preempted/cancelled. This introduces a small, but of course systematic, performance penalty in the execution of Local Tasks. The second chance operation is idempotent, i.e. trying to activate it more than once for a running Supplying Task has no effect, so that no Task is graced repeatedly in a potentially infinite loop. Moreover, a graced Supplying Task is protected from preemption/cancellation during the grace period.

[5]The term PSufferage is selected by analogy to the XSufferage scheduling algorithm [74].

Supplying Task	queued Tasks	action	delay
short runtime	few waiting	preempt	immediately
short runtime	many waiting	cancel	immediately
almost completed	few waiting	preempt	after grace period
almost completed	many waiting	cancel	after grace period

Note: almost completed = expected to be soon completed, as the Task's nominal runtime is not provided by the Grid application developer.

Table 4.5: Adaptive Supplying Tasks preemption policy.

Preempting or cancelling a Supplying Task has a different impact on its response time depending on the length of the Task queues of the supplier Peer. If many queued Tasks are waiting (e.g. because a new Local BoT has been submitted to the supplier Peer) preempting a Supplying Task preserves the reliability of the supplier Peer (because there is no cancellation) but decreases its performance (because queueing delays are introduced and increase the response time). If many queued Tasks are waiting, cancelling a Supplying Task allows the consumer Peer to try and reschedule this Task immediately. We thus propose that Supplying Tasks be preempted if there are few waiting Local Tasks, and cancelled if there are many waiting Local Tasks. The threshold of waiting Local Tasks is configured by the human Peer administrator, typically to a very low value.

Using the queue length as an estimation of the waiting time is a classical [285], even if not optimal, strategy. Indeed, accurate estimates of the waiting time would necessitate runtime estimates for the waiting Local Tasks, a prediction of the requeueing of cancelled Consumption Tasks, a prediction of the availability of the Peer's Resources (i.e. additional Resources may become available, others may crash), and a prediction of the availability of supplier Peers (i.e. for a fixed number of supplying requests, the number of supplier Peers that send consumption grants and the number of consumption grants both fluctuate over time). Existing results for the prediction of queue wait time [283] are certainly not applicable to P2P Grids because the metadata required to compute predictions, e.g. queue parameters, are not available due to the opacity between Peers.

Our proposed adaptive Supplying Tasks preemption policy is summarized in Table 4.5. Three parameters influence its behavior:

- a threshold (few vs. many waiting queued Tasks) to select preemption or cancellation,
- a threshold (short runtime of Supplying Task vs. Supplying Task almost completed) to decide if a second chance should be offered,

- a grace period during which the Supplying Task is protected from preemption/cancellation if a second chance has been offered.

The threshold of waiting queued Tasks can be arbitrarily small or large. It should likely be a small number of Tasks. The grace period is configured by the human Peer administrator. It should be small, and likely be no more than a few minutes so that the performance penalty on the completion of Local Tasks remains acceptable. Another idea would be to dynamically adapt the grace period proportionally in function of the Task's current runtime (i.e. elapsed time since the beginning of the Task execution on the Resource).

The threshold of the Supplying Task's current runtime can be arbitrarily chosen, but it would be difficult to estimate it as an absolute value. As the grace period and this threshold of the Supplying Task's current runtime are related, we propose another approach to determine if a Supplying Task should be given a second chance. If a Supplying Task is expected to complete its execution within the grace period, it can be given a second chance. An estimation of the completion time of the Supplying Task is thus required. To compute this runtime estimation, we propose a modified version of the history conservative prediction algorithm [321, 269]. The history conservative prediction algorithm is simple to implement and consistently yields acceptable results [321]. The algorithm produces an estimate by adding the mean and the variance of recent values of the quantity to estimate: Our version is based on the completion times of the most recently (the window size for metadata storage is 100 Supplying Tasks, see Section 4.4.6) completed Supplying Tasks that were submitted by the consumer Peer of the Supplying Task of interest.

4.7.3 Dequeueing of Waiting Supplying Tasks

The Supplying Tasks filtering policy prevents other Peers to submit too many Supplying Tasks. However, it does not preclude the Supplying Tasks queue to grow upon preemption and subsequent requeueing of running Supplying Tasks. Dequeueing a Supplying Task corresponds to cancel the corresponding waiting (i.e. unscheduled) Supplying BoT, as illustrated in Figure 2.18.

A policy to dequeue waiting Supplying Tasks (thus without scheduling them) should thus be defined along with a policy to preempt running Supplying Tasks. Four Supplying Tasks dequeueing policies for waiting Supplying Tasks are currently implemented: FIFO-based, favors-based and full dequeueing of waiting Supplying Tasks, as well as no dequeueing of waiting Supplying Tasks.

The FIFO-based dequeueing policy dequeues the most recent waiting Supplying Task. The favors-based preemption policy follows the Network of Favors model (see Section 2.3.4): It dequeues the waiting Supplying Task associated with the

AdaptivePreemption	adaptive preemption
FullCancellation	full cancellation
FullPreemption	full preemption
LimitedCancellation	limited cancellation
LimitedPreemption	limited preemption
NoPreemption	no preemption

Table 4.6: Running Supplying Tasks Preemption Policies.

FIFOWaitingDequeueing	FIFO-based consumers ranking
FavorsWaitingDequeueing	favors-based consumers ranking
FullWaitingDequeueing	unlimited dequeueing
NoDequeueing	no dequeueing

Table 4.7: Running Supplying Tasks Dequeueing Policies.

smallest favor balance (ties are broken by dequeueing the most recently submitted Task among those submitted by the consumer Peer with the smallest favor balance). The full dequeueing policy dequeues all waiting Supplying Tasks. Finally, one policy has no effect, i.e. waiting Supplying Tasks are never dequeued.

4.7.4 Summary of Supplying Tasks Preemption Policies

Table 4.6 lists the proposed running Supplying Tasks preemption policies. Resource selection is orthogonal to the selected running Supplying Task preemption policy.

Table 4.7 lists the proposed waiting Supplying Tasks dequeueing policies.

4.8 Experimental Results

4.8.1 Methodology

The 4-Peers base scenario of Section 3.6.1 - that has been proposed in related work [14] - is considered again. The topology is fixed; it consists of a Grid of 4 peers that manage 4 Resources each (see Figure 3.10). All Resources are identical. The workload is also fixed. Each peer must process 60 Bags of 40 Tasks with no input data files. Each Task can be completed in exactly 1 minute by any Resource. The inter-arrival time of submitted BoTs is a random variable (uniform distribution) with a value between 1 minute and 20 minutes. If a Peer uses its own Resources only, and in absence of queueing delays due to contention, the optimal

scenario.sdf	PDP_{cts}	PDP_{sts}	PDP_{lts}	PDP_{stf}	PDP_{rstp}	PDP_{wstp}	PDP_{emr}	PDP_{evr}	MBRT
strategy00001	p^1_{cts}	p^1_{sts}	...						?
strategy00002	p^2_{cts}	p^2_{sts}	...						?
strategy00003	p^3_{cts}	p^3_{sts}	...						?
strategy00004	p^4_{cts}	p^1_{sts}	...						?
...	...	...							?
strategy00015	p^{15}_{cts}	p^3_{sts}	...						?
strategy00016	p^1_{cts}	p^1_{sts}	...						?
...	...	...							?
strategy16850	...	...	...						?

Figure 4.3: Strategy matrix (each line is a combination of policies).

mean BoT response time (MBRT) is 10 minutes.

Combinations of the bartering policies, presented in this chapter[6] for the five scheduling policy decision points (PDP), as well as for the two negotiation policy decision points (see Section 2.9.5), can be systematically enumerated. Such a combination of eight[7] bartering policies is called a strategy. Figure 4.3 illustrates the strategy matrix. Given the currently implemented policies, only 16850 strategies are meaningful[8], and only these are actually evaluated (see also Section 6.1.1 for additional details on the generation of the strategy matrix).

Each strategy of the strategy matrix must be completed with policy parameters, as well as descriptions of the workload and P2P Grid topology to consider. All of these are described in a common scenario description file (see Section 3.5), that is associated with each strategy. The policy parameters of the scenario assume constant, standard values, except for the grace period of the adaptive Supplying Tasks preemption policy, which varies (1, 2, 3, 5 and 10 minutes); the values of all parameters are provided in Appendix B.2.2.

[6]In this section, the naming convention of policies is *metric-then-PDP-label*, e.g. AdaptivePreemption, RelaxedFavorsSupplyingFiltering, RandomConsumptionScheduling, ...

[7]Two policies are needed for the Supplying Tasks preemption PDP, see Section 4.7.

[8]Many combinations are of little interest, e.g. "no scheduling" Supplying Tasks scheduling policy with a Supplying Tasks preemption policy; 16850 strategies out of 1080000 are meaningful.

4.8.2 Evaluation of Strategies (4-Peers Base Scenario)

The optimal mean BoT response times (MBRT) that can be achieved without bartering, i.e. each Peer relies on its own Resources only, is 600 seconds (10 times 4 Tasks running in parallel on 4 Resources for 1 minute each). The optimal MBRT that might be achieved with bartering but without contention, i.e. Peers barter computing time and have temporally complementary workloads, is 180 seconds (2 times 16 Tasks, then 1 time 8 Tasks, running in parallel on 16 Resources for 1 minute each). After running the 16850 simulations corresponding to the strategies of the strategy matrix, we found that, over all strategies, the average MBRT is 2617 seconds, but large variations are observed: The best (shorter) MBRT is 381 seconds and the worst (longer) MBRT is 33197 seconds, i.e. there is a factor of 100 between the best and the worst strategies.

Furthermore, in the context of the experiments from Section 3.6.1, the MBRT claimed to be achieved with the OurGrid middleware is 445 seconds; the best MBRT obtained from the results presented in this section is is 358 seconds. The evaluation of a large number of strategies has enabled, for the considered scenario, to identify 36 strategies that lead to lower MBRTs than the strategy equivalent to OurGrid's default strategy.

In this section, several figures provide the average MBRT obtained over all strategies of the strategy matrix; the given MBRTs are thus an average of 16850 MBRTs. As some policies are underperforming and lead to unacceptably long MBRTs, it can be useful to recompute the average MBRT without taking them into account. This enables to discriminate policies leading to strategies with good-but-never-optimal MBRTs from policies leading to strategies with good-and-very-variable MBRTs.

To this end, the average MBRT is computed five times: over 100% of strategies of the strategy matrix, as well as over the top 80%, 20%, 5% and 1% best strategies. The best strategies are those that achieve the lowest MBRTs. Computing the average MBRT of the strategy matrix for the top k% best strategies consists of sorting the strategy matrix by increasing MBRT values and computing the average MBRT over the top k% strategies.

Consumption Tasks Scheduling PDP

Figure 4.4 gives the MBRTs of Consumption Tasks scheduling policies. A first observation is that among the worst results, some strategies with underperforming policies considerably influence the results of most Consumption Tasks scheduling policies. With the "no scheduling" policy, bartering is not used; it is thus not influenced by underperforming policies of other PDPs. Random scheduling

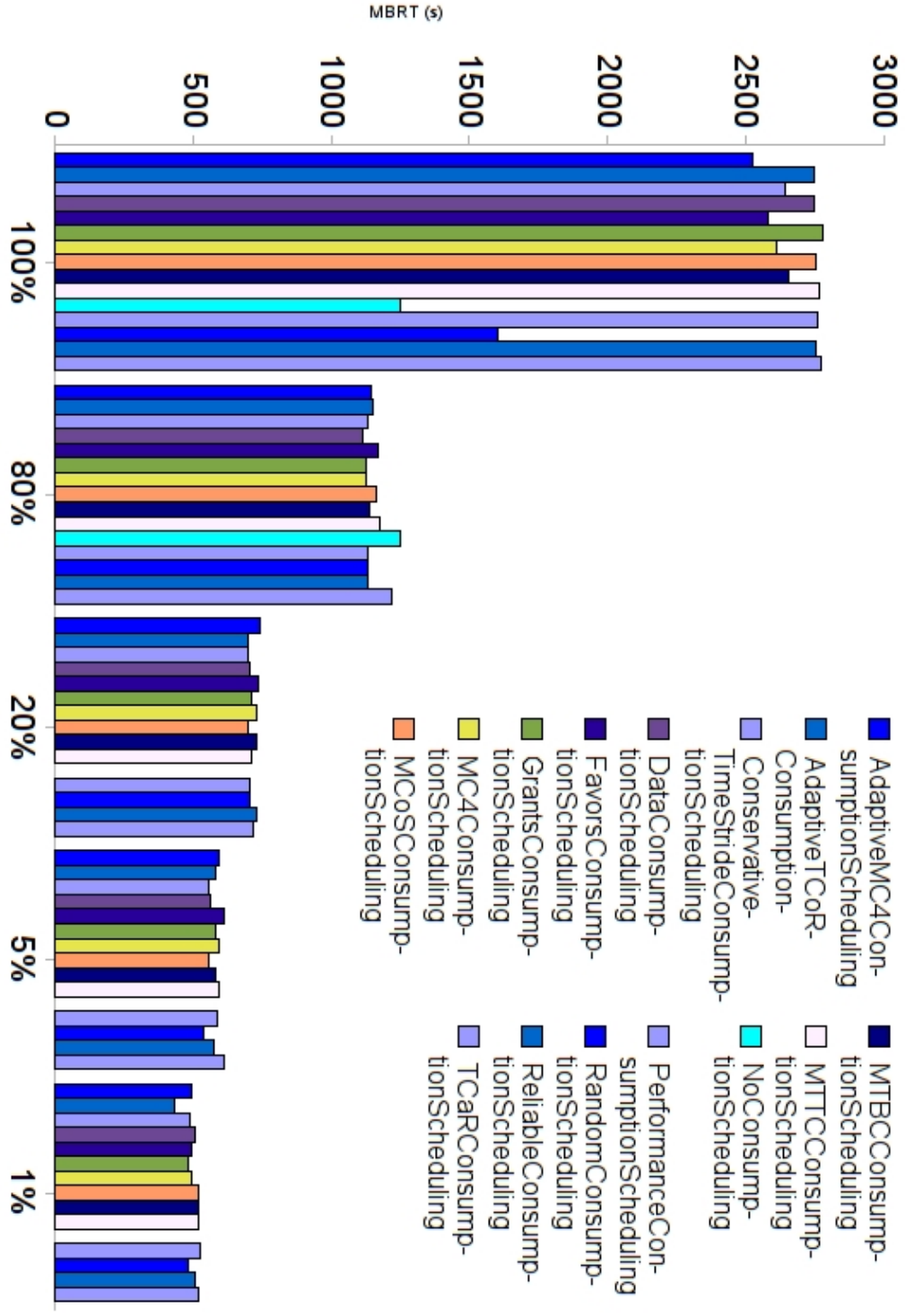


Figure 4.4: MBRT of Consumption Tasks scheduling policies for the 4-Peers scenario.

gives better results over all strategies, and similar results than other policies when considering over the top 80% strategies. Random scheduling is not influenced by underperforming policies of other PDPs either. It is important to remark that a consumer Peer, in the scenario considered in this section, has very little choice: At most 3 supplier Peers to choose from, at any time. This can explain why random consumption choices give better results for strategies when associated with underperforming policies of other PDPs

A second observation is that the strategy with the "no scheduling" policy disappears from the best 20% results. Its MBRT is actually[9] 1251 seconds, which is higher than the optimal 600 seconds because of contention, i.e. frequent arrival of Local BoTs less than 600 seconds apart. The rank of the strategy with the "no scheduling" policy is 8293, which means than a little less than half of the strategies with bartering perform better. This confirms the potential speedup of bartering. Indeed, after identifying underperforming policies for all PDPs and removing the corresponding strategies, most of the remaining strategies perform better than the one without bartering. Consequently, even if it proves elusive to isolate a priori the best strategy for a given Grid topology and a given workload, most strategies that exclude really underperforming policies lead to an acceptable MBRT.

Local and Supplying Tasks Scheduling PDPs

Figures 4.5 and 4.6 give, respectively, the MBRTs of Supplying Tasks scheduling policies and the MBRTs of Local Tasks scheduling policies. Similarly to Consumption Tasks scheduling, a first observation is that the "no scheduling" strategy behaves better than strategies with any other Supplying Tasks scheduling policies associated with underperforming policies for other PDPs, but is not among the best 20% strategies.

A second observation is that strategies that may involve preemption can be on average as good as those that never involve preemption, except in underperforming strategies where preemption is important. In the latter case, our interpretation is that preemption acts as a form of queue length control, preventing queueing delays.

Supplying Tasks Filtering PDP

Figure 4.7 gives the MBRTs of Supplying Tasks filtering policies. Same comments as for other PDPs apply to the "no filtering" policy, i.e. when there is no bartering between Peers.

[9]It can be remarked that the results of the simulations provided in this section are - as expected - compatible with those presented in Chapter 3.

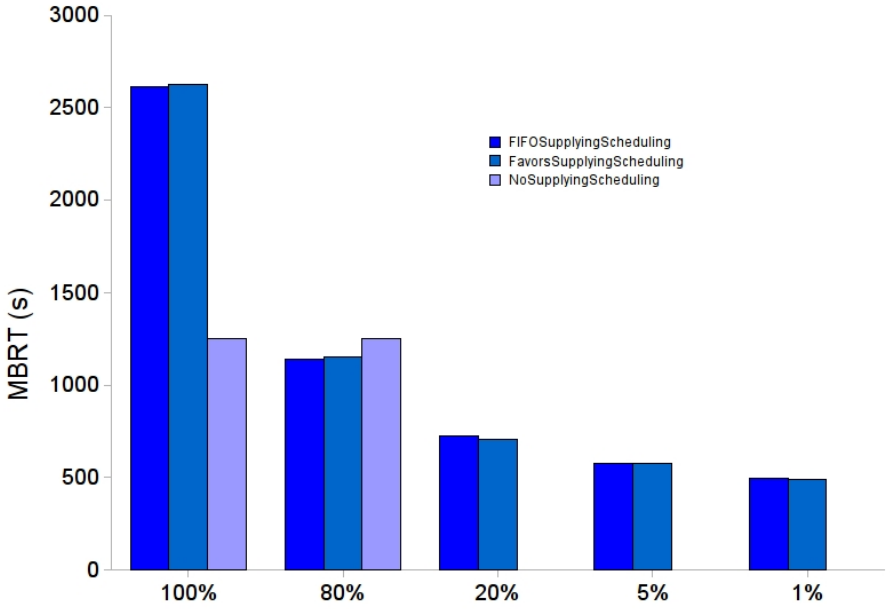


Figure 4.5: MBRT of Supplying Tasks scheduling policies for the 4-Peers scenario.

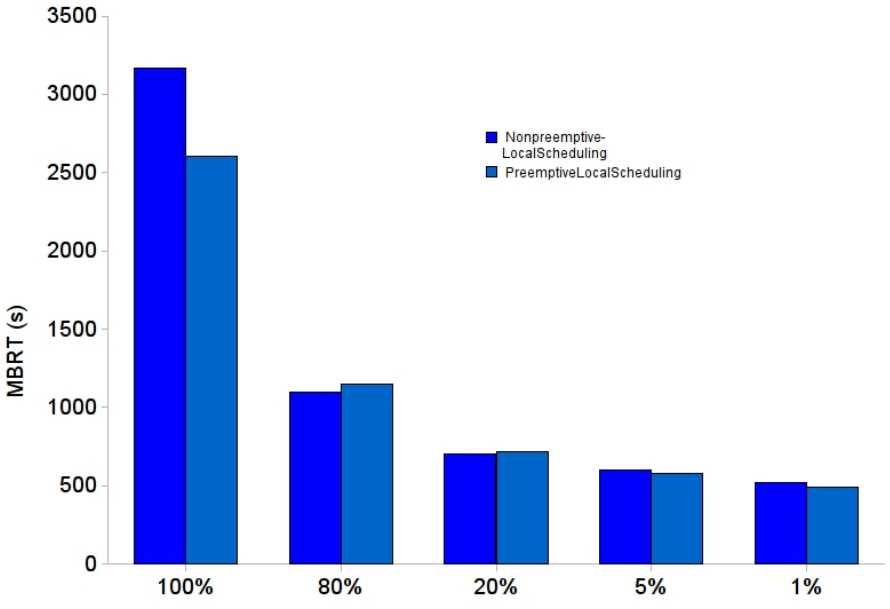


Figure 4.6: MBRT of Local Tasks scheduling policies for the 4-Peers scenario.

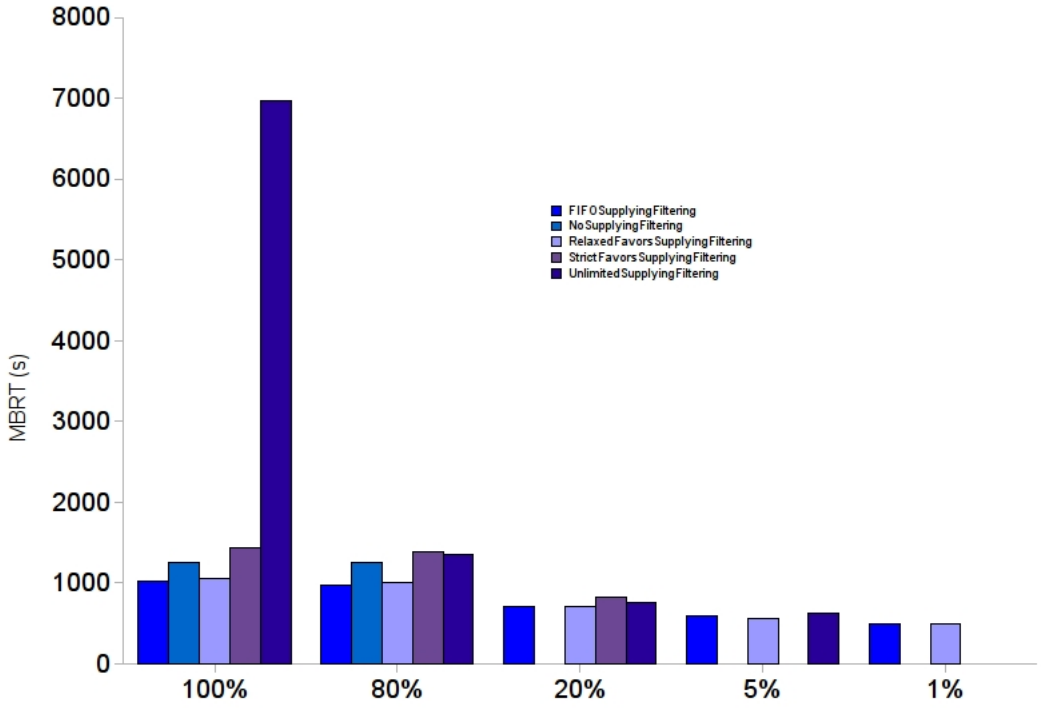


Figure 4.7: MBRT of Supplying Tasks filtering policies for the 4-Peers scenario.

A first observation is that, when bartering is enabled, it is really important to filter out incoming Supplying Tasks so that the Supplying Tasks queue does not grow out of control, i.e. causing queueing delays when there are not enough available Resources. Even if efficient preemption policies are used, the unlimited filtering policy should be avoided. Indeed, accepting all Supplying Tasks and preempting or dequeueing most of them is likely to have a major impact in the middleware implementation, essentially due to the cost of GNMP message transfers. However, this cost is not taken into account in the current simulator implementation; that constitutes one of the rare instances we identified where the simulator is not accurate enough. The impact of this limitation is likely to be quite limited in practice, as the unlimited filtering policy is intended for comparison purposes rather than for production deployment. As the response times obtained with an unlimited filtering policy are already measured to be overly long, it may not be that informative to confirm that they are even longer in a real deployment.

A second observation is that the strict favors-based filtering policy consistently leads to higher MBRTs. The strict favors-based filtering policy preempts running Supplying Tasks if required, as opposed to the relaxed favors-based filtering policy. The strict policy is closer to the NoF model. However, as will be seen in the next few paragraphs, preemption is useful but does not always lead to better results.

Supplying Tasks Preemption PDP

Figure 4.8 gives the MBRTs of running Supplying Tasks preemption policies. Same comments as for other PDPs apply to the "no preemption" policy, i.e. when there is no bartering between Peers.

A first observation is that the adaptive Supplying Tasks preemption policy[10] consistently leads to better results.

A second observation is that both non-adaptive preemption policies (full preemption and limited preemption) are underperforming. This may seem counter-intuitive but can be interpreted as follows: When Supplying Tasks are preempted to schedule newly submitted Local Tasks, they are requeued. If there are many queued Local Tasks, or if even a few Local Tasks have a long runtime, the requeued Supplying Tasks experience large queueing delays. Cancellation policies (full cancellation and limited cancellation) intuitively seem harsher than preemption policies. In a sense, they are, as the adaptive preemption policies mitigates the side effects of cancellation by cancelling only when needed (and preempting otherwise) and only after a grace period. But, on the other hand, cancelling Supplying Tasks enables consumer Peers to try and reschedule these Tasks to other Peers.

If Task replication (see Section 2.9.4) were supported, the side effect of preemption (i.e. queueing delays) might be compensated by the scheduling of multiple instances of Tasks to multiple Peers, of course at a greater computational expense. However, in large-scale deployments of distributed systems [107, 108], it has been observed that Task replication is best used in endgames, i.e. when there remains only a few Tasks to compute and there are many Resources available. Consequently, in this context queueing delays would be counter-productive. Additionally, cancelling Supplying Tasks enables the consumer Peers to recognize the unreliability of supplier Peers: This enables consumer Peers to avoid submitting Consumption Tasks, in particular the last few ones, to heavily-loaded supplier Peers. Thus, preemption, if not adaptive, should not be used. Cancellation or adaptive preemption should be preferred.

Figure 4.9 gives the MBRTs of waiting Supplying Tasks dequeueing policies. Same comments as for other PDPs apply to the "no preemption" policy, i.e. when there is no bartering between Peers.

[10] Grace periods of more than 1 minute have no impact in this scenario given the Task length of 1 minute. These are shown as a baseline for the scenario considered in the next section.

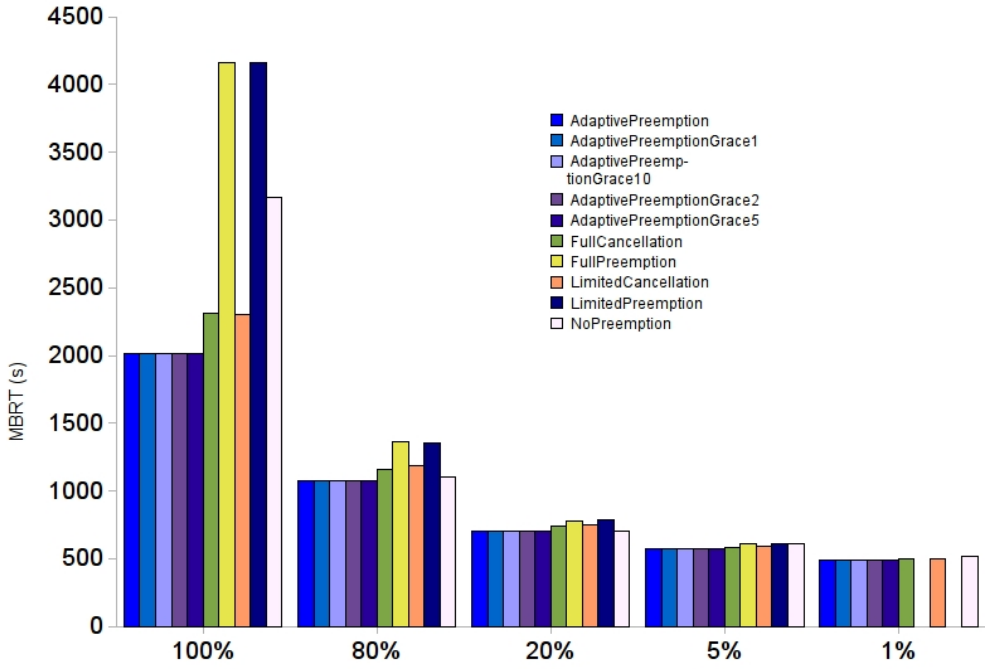


Figure 4.8: MBRT of running Supplying Tasks preemption policies for the 4-Peers scenario.

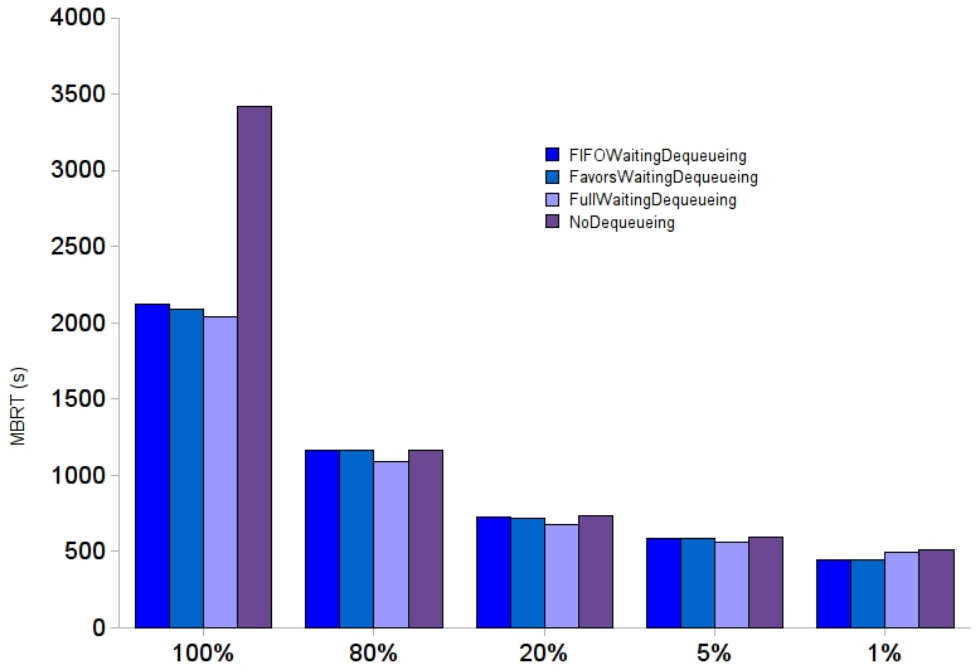


Figure 4.9: MBRT of waiting Supplying Tasks dequeueing policies for the 4-Peers scenario.

The only observation is that dequeueing waiting Supplying Tasks - upon preemption of running Supplying Tasks and requeueing of cancelled Local or Consumption Tasks - is very important in underperforming strategies. Indeed, this consistently leads to lower MBRTs, including for these underperforming strategies. It should thus be enabled.

Emission and Evaluation of Supplying Requests PDPs

Figures 4.10 and 4.11 give, respectively, the MBRTs of the policies for the emission of supplying requests and the MBRTs of the policies for the evaluation of supplying requests. Same comments as for other PDPs apply to the "no emission" and to the "no evaluation" policies, i.e. when there is no bartering between Peers.

As expected, Figure 4.10 simply shows that the emission of supplying requests is influenced by underperforming policies of other PDPs.

Figure 4.11 shows that distributing an unlimited number of consumption grants, i.e. as many consumption grants as requested by consumer Peers, leads to underperforming strategies. Similarly to the unlimited Supplying Tasks filtering policy, the unlimited evaluation of supplying requests policy should not be used in practice. Indeed, even if an efficient preemption policy is used, the frequent dequeueing of nearly all accepted Supplying Tasks is likely to have an impact on performance in the middleware implementation.

4.8.3 Evaluation of Strategies (4x10-Peers Scenario)

The base scenario of the previous section is modified by multiplying by 10 the number of Peers, i.e. there are 40 Peers in each considered P2P Grid (with 4 Resources each), so as to increase opportunities when scheduling Consumption Tasks. Figure 4.12 gives the MBRTs of Consumption Tasks scheduling policies. On average over all strategies, the mean MBRT is 23063 seconds, the best (minimum) is 158 seconds and the worst (maximum) is 602949 seconds; there is a factor of $\sim$4000 between the best strategy and the worst strategy.

As a reminder, the minimum MBRT over all strategies was 381 seconds in the 4-Peers base scenario. It means that increasing the opportunities of bartering helps Peers to achieve a lower MBRT.

The remarks about the bartering policies that were presented in the previous section still hold for the 4x10-Peers scenario (as they also do for an intermediate number of Peers or variable Task runtimes), save for two differences.

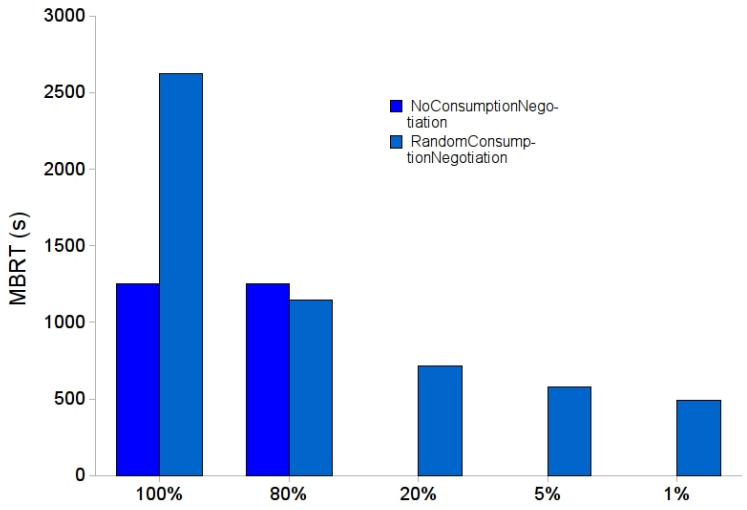


Figure 4.10: MBRT of emission of supplying requests policies for the 4-Peers scenario.

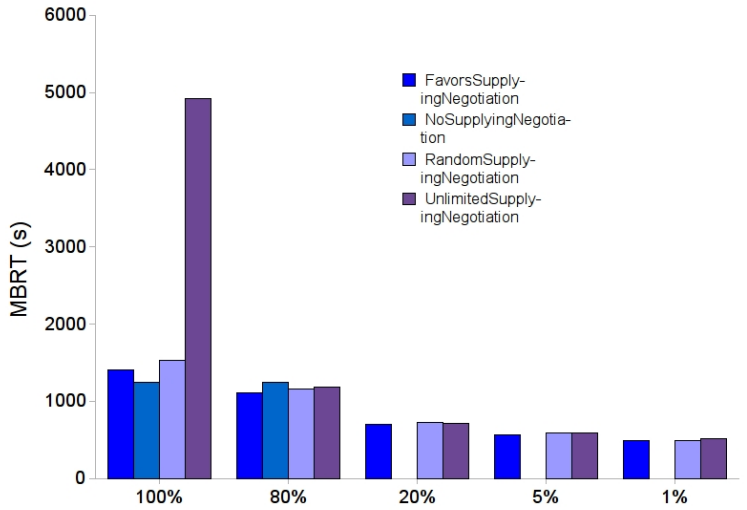


Figure 4.11: MBRT of evaluation of supplying requests policies for the 4-Peers scenario.

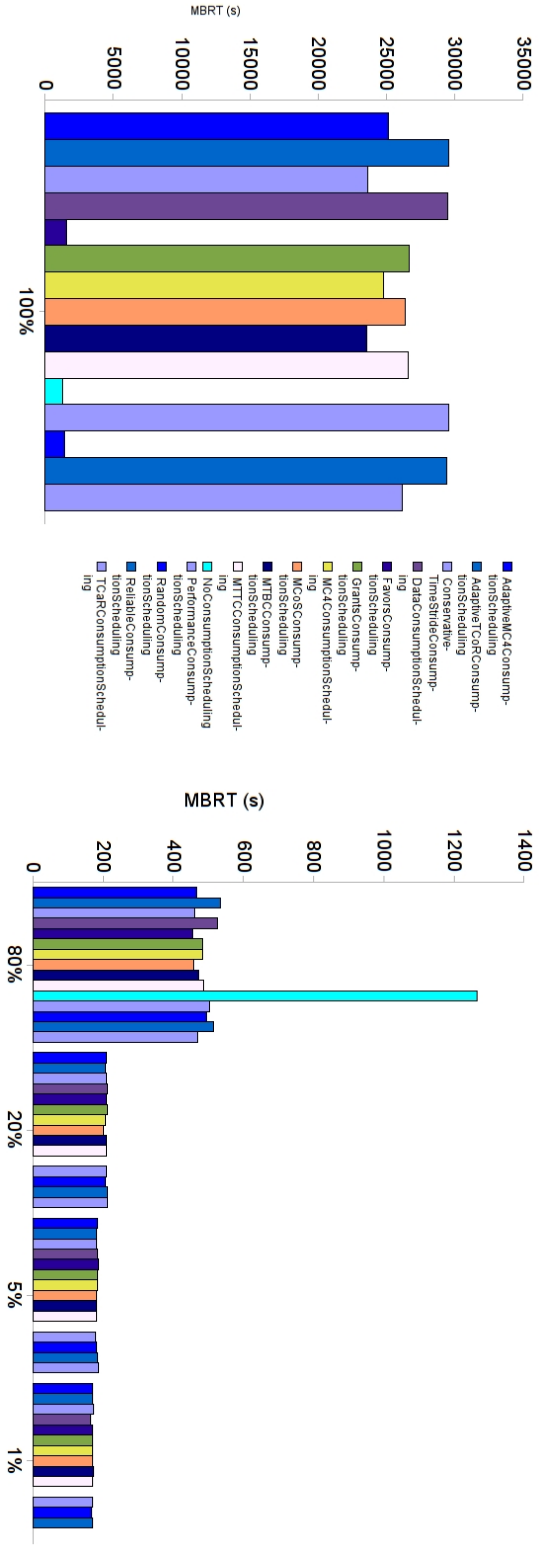


Figure 4.12: MBRT of Consumption Tasks policies for the 4x10-Peers scenario.

Firstly, the gap of performance between underperforming strategies and regularly performing strategies has increased, e.g. the MBRTs achieved with underperforming strategies among 100% of the strategies is more than 10 times as long as those achieved among the best 80% of the strategies (while it is about 2 times as long for the base scenario).

Secondly, the random, the "no scheduling" and also the favors-based Consumption Tasks scheduling policies perform very well when combined with underperforming policies for other PDPs, i.e. they lead to significantly lower MBRTs when considering 100% of the strategies. The other Consumption Tasks scheduling policies do not perform well when considering 100% of the strategies; they do not perform better than the three mentioned ones either, when considering only the best strategies.

Simple Consumption Tasks scheduling policies can lead to strategies that perform very well. RandomConsumptionScheduling is the Consumption Tasks scheduling policies used in the OurGrid middleware. With the experiments presented in this section, we showed that lower MBRTs than those achieved with the OurGrid default strategy can be achieved. But these experiments also show that, for the Consumption Tasks scheduling PDP, a simple policy is sufficient.

Nonetheless, we still believe that ranking-based policies can be designed to take advantage of the specificity of some workloads. Our proposed P2P Grid architecture, bartering, scheduling and negotiation models constitute a framework to easily test new policies. Now that these foundations have been laid, we envision the following roadmap for future research in ranking-based Consumption Tasks scheduling policies:

1. Study trace workloads from operational, large-scale P2P Grids deployments, using more advanced statistical tools so as to identify typical workloads;

2. Tune our proposed Consumption Tasks scheduling policies so that they are adapted to some typical workloads;

3. Design a policy switching algorithm that can identify typical workloads;

4. Using this policy switching algorithm, activate on-the-fly the appropriate ranking-based policy (instead of a simple policy such as RandomConsumptionScheduling) for BoTs that constitute typical workloads.

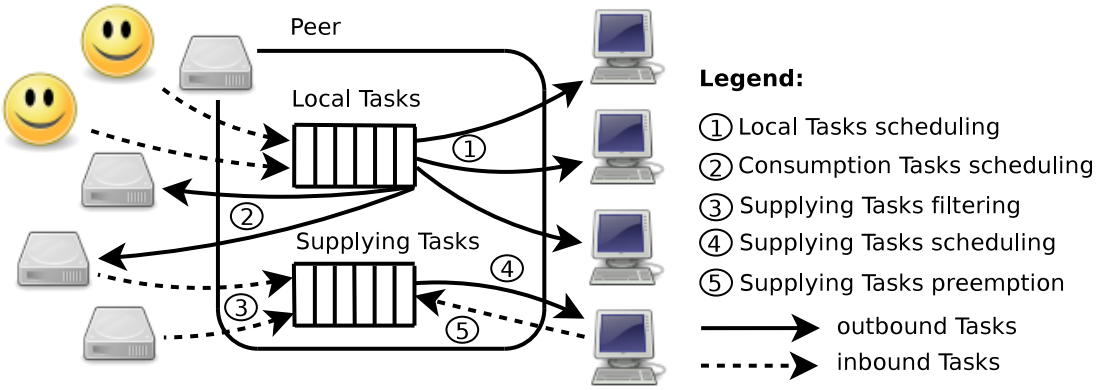


Figure 4.13: Scheduling model.

4.9 Summary of the Contributions

4.9.1 Discussion

Our proposed P2P Grid architecture and the weaving of the simulator code into the bartering code constitute a framework that have enabled us to systematically study ten of thousands of bartering strategies. Scheduling is a computationally hard problem, thus heuristics are needed for tractability of the implemented systems. With our approach based on Policy Decision Points (see Figure 4.13 for a reminder of the scheduling model), it is possible to evaluate offline a large number of simple heuristics, and subsequently select any of them when deploying the P2P Grid middleware. There are clearly limits to such a combinatorial approach, but it nonetheless constitutes an improvement over the evaluation of only a handful of policies at a time.

The simulations presented in this chapter are expected to be reasonably accurate, based on results described in Section 3.6.1. However, also as stated in Section 3.6.1, future work should be undertaken to evaluate the simulator accuracy for various scenarios.

The two PDPs from which important conclusions can be made are the Consumption Tasks scheduling PDP and the Supplying Tasks preemption PDP.

We have proposed several ranking-based Consumption Tasks scheduling policies, but all failed to outperform random chance, probably because they are too narrowly specific to some workloads. This does not prove that it is impossible to achieve better-than-random results. Nonetheless, this is **compatible with the implicit**[11] **selection of Consumption Tasks scheduling policy made in the**

[11] To the best of our knowledge.

OurGrid middleware (RandomConsumptionScheduling). Even though a random choice of supplier seems efficient in the experiments that were conducted, the Consumption Tasks scheduling PDP is only one out of several; there exist better-than-random policies for the other PDPs.

We have also proposed an **adaptive Supplying Tasks preemption policy that, coupled with a cost-aware Resource selection algorithm that we call PSufferage, leads to efficient and robust strategies**. It is intrinsically in a Peer's best interest to preempt Supplying Tasks to reclaim its own computational power in order to process its own workload first. Preemption with subsequent requeueing, however, is not the most efficient approach. Cancellation without subsequent requeueing leads to better strategies. However, even if cancellation gives very good results on average, some consumers are individually penalized as some of their BoTs experience unwelcome delays, even if their overall MBRT is low. To prevent this side effect of cancellation, the adaptive preemption policy offers a second chance to some of the Supplying Tasks that have to be preempted. A short grace period is offered to those Supplying Tasks that are estimated to complete shortly: During this grace period, the selected Supplying Tasks are protected from preemption or cancellation. By accepting to slightly delay the reclaiming of its computational power, a supplier Peer can greatly improve its reliability with consumer Peers whose Supplying Tasks are offered a second chance.

4.9.2 Guidelines

Finally, we propose the following guidelines for an efficient and robust strategy that can be deployed in practice:

- Consumption Tasks scheduling PDP: RandomConsumptionScheduling or DataConsumptionScheduling;
- Supplying Tasks scheduling PDP: FavorsSupplyingScheduling;
- Local Tasks scheduling PDP: PreemptiveLocalScheduling;
- Supplying Tasks filtering PDP: RelaxedFavorsSupplyingFiltering;
- running Supplying Tasks preemption PDP: AdaptivePreemption;
- waiting Supplying Tasks dequeueing PDP: FavorsWaitingPreemption or FullWaitingPreemption;
- emission of supplying requests PDP: RandomConsumptionNegotiation;
- evaluation of supplying requests PDP: FavorsSupplyingNegotiation.

Chapter 5

P2P Data Transfers

The biggest difference between time and space is that you can't reuse time.

- Merrick Furst

The massive size and amount of data files to be transferred across the Grid can cause delays in the completion of Tasks. In practice, sets of data files often present repetitive patterns, in the sense that some files are repeatedly processed over time in multiple Bags of Tasks, or some files are processed in a variety of ways within the same Bag of Tasks. A fully decentralized data transfer architecture is proposed to take advantage of these redundancies; importantly, it is designed to enable worker nodes to collaborate beyond Peer boundaries. Temporal redundancy is addressed by relying on P2P data transfers based on the BitTorrent P2P file sharing protocol; this applies when there are enough Resources available and Tasks depending on identical input data files can be scheduled together. Spatial redundancy is addressed by a distributing caching mechanism combined with data reuse; this applies when input data files required by Tasks are already cached by available Resources. Temporal redundancy that is implicit can still be taken advantage of: When input data files of Tasks to schedule are cached by Resources not available for scheduling these Tasks, the decentralized data transfer architecture enables these busy Resources to share the cached data files with the available Resources where the Tasks are scheduled. Experiments are performed using the LBG middleware.

High parallelism in Task execution, leading to shorter overall BoT response times, requires the simultaneous transfer of potentially large input data files. These data transfers may rapidly lead to performance bottlenecks in the case of so-called *Data-Intensive Bags of Tasks* [267, 113, 76, 184, 257], that are Bags of Tasks processing large input data files (see Section 2.4). Scheduling Data-Intensive BoTs is not trivial because transfers of large data files may become a performance bottleneck in the P2P Grid. In turn, this has a negative impact both on the overall response

times of the Tasks and on the willingness of supplier Peers to actually supply computing time (very slow downloads of required input data files might discourage suppliers).

A classic solution to this issue is data reuse. It entirely avoids the problem of multiple simultaneous data transfers by making use of opportunistically cached or proactively replicated data. On the downside, Tasks - essentially those sharing the same input data files - have to be scheduled sequentially, rather than simultaneously, to benefit from the input data files that were downloaded for other Tasks. This may lead to higher BoT response times and it becomes more difficult to offer nontrivial QoS [151]. Furthermore, it is not always possible in practice to massively use data caching. As P2P Resources are nondedicated edge computers, the availability of storage space is highly variable and potentially limited.

Task execution parallelism and data reuse both have benefits as well as downsides. In existing research, they are usually presented as mutually exclusive (see Figure 5.1 for an intuitive illustration). We propose [56, 57] to actually integrate them, so as to achieve good performance in most situations.

A highly scalable and fully distributed data transfer architecture is introduced: Grid nodes are equipped with data caches and data transfer (download) as well as data sharing (upload) softwares. The goal is to transparently increase the number of data servers and to spread the load caused by data transfers, across the P2P Grid. Input data files required by subsequent BoTs benefit from the caching, provided that Resource selection takes data location into account when scheduling. To enable Task execution parallelism in Data-Intensive BoTs that exhibit some redundancy in the required input data files, we propose to transfer data with the BitTorrent P2P file sharing protocol. In order to exploit the efficiency of BitTorrent, we also propose a novel Task selection scheduling algorithm, *Temporal Tasks Grouping* (TTG) to - counter-intuitively - maximize the temporal coincidence of the downloads of large, identical input data files required by different Tasks of a BoT. The combination of BitTorrent and TTG ensures an efficient download of uncached data: Every input data file must obviously be downloaded at least once into the P2P Grid, but our algorithms are designed to reduce the cost of downloading identical - i.e. redundant - copies of input data files, be they needed simultaneously or over time.

This chapter is structured as follows. The state of the art in scheduling of Data-Intensive Bags of Tasks and in BitTorrent data transfer architectures is first reviewed in Section 5.1. Our proposed data transfer architecture and Task scheduling policies are then introduced, in Sections 5.2 and 5.3, respectively. They are evaluated and experimental results are discussed in Section 5.5. Proactive, asynchronous data replication is also briefly discussed in Section 5.4.

(a) Data reuse (b) Task execution parallelism

Figure 5.1: (a) Data reuse vs. (b) Task execution parallelism.

5.1 Related Work

In this section, the state of the art is reviewed in the scheduling of Tasks depending on large data files, and also in mechanisms enabling multiple simultaneous data transfers, in particular the BitTorrent P2P file sharing protocol.

5.1.1 Data Replication

A possibility to avoid that massive data transfers degrade performance is to perform them, as much as possible, when this does not increase BoT response times. Proactively replicating data on multiple Resources may increase the efficiency of subsequent data-aware Task scheduling, but with a possibly large storage cost due to data caching.

In predictable environments, more typical of Desktop Grids rather than of P2P Grids, it is possible to complete all data replication before the scheduling of a Bag of Tasks. A recent example of state of the art synchronous data replication in a Desktop Grid, in the context of life sciences applications [113], proposes an integer programming scheduling algorithm. It is limited to a steady state context. It is therefore not applicable in the context of P2P Grids because of the unreliability and constant change of P2P environments.

It has been shown [257] that data replication can be performed proactively and asynchronously from Task scheduling, with simple and cost-efficient algorithms, as long as the Task scheduler is aware of data placement. As proactive data replication in a very dynamic environment seldom achieve perfect data placement, a trade-off [45] must be found between creating new replicas in order to balance the computational load, or using existing replicas in order to avoid data transfers.

5.1.2 Data Reuse

Synchronous data replication may not be applicable to P2P Grids. Moreover, the costs incurred by data replication can be very high in terms of transfer times. They can also be high in terms of infrastructure overload, i.e. a Grid Peer may stop operating due to thrashing caused by the handling of an excessive number of data management threads [98]. Explicit data replication should thus be avoided and substituted with data caching, i.e. implicit data replication.

Data reuse is a general term to designate the combination of data caching and awareness of data placement. If each Resource is equipped with a data cache, only the first Task (scheduled to this Resource) requiring a given data file triggers its transfer. Subsequent Tasks (scheduled to this Resource) requiring the same data file do not trigger its transfer, assuming this data file is still present in the data cache. When prior availability of data on Resources is taken into account, Task scheduling is said to be data-aware. Parallelism in the execution of Tasks depending on the same input data files is of course reduced when using only a data reuse mechanism.

There exist Task scheduling algorithms [98], taking advantage of data caching. One algorithm sequentially schedules to the same Resources the Tasks requiring identical input data files. Another algorithm schedules to a single Resource all Tasks requiring the same set of large data files. Of course, this decreases execution parallelism but may be very efficient in P2P Grids connected with slow data links, or when the data files are so large that it would not be practical to transfer them more than once [44].

A data-aware Workflow scheduling algorithm has been designed [254] around a statically precomputed schedule of data reuse and cache cleaning operations. It is not relevant to this dissertation because it is not adaptive to Resource availability.

The Storage Affinity [267] Task scheduling algorithm, implemented in the OurGrid [233, 84, 286] P2P Grid middleware, dynamically takes data placement into account and is thus perfectly adapted to P2P Grids. It schedules Tasks first to Resources where most of the required input data files are already available (as measured by the *Storage Affinity* metric, which gives its name to the algorithm) before considering other Resources where the unavailability of some input data files requires data replication. The data transfer architecture presented with the Storage Affinity [267] algorithm relies on a single Peer-level data cache on each Peer; as opposed to our work, it thus does not rely on multiple Resource-level data caches per Peer. The Peer-level data cache is accessed by Resources through an NFS file system, which is not highly scalable.

As it is operating in a P2P environment, the Storage Affinity algorithm has to deal with the unavailability of accurate data about computational times. Task replication

(see Section 2.9.4) is proposed as a heuristic to find good Task-to-Resource assignments. Redundancy brings excellent tolerance to Resource faults and variability in performance, but at a cost [86]. It can be remarked that Task replication has the side effect that any BoT implicitly becomes Data-Intensive, in the sense that its input data files are transferred multiple times, thereby increasing the amount of network traffic.

Data reuse, which depends on the presence of data caching support, can prevent unnecessary data transfers. Among the reviewed related works, the Storage Affinity [267] algorithm is the most suitable to our purposes, provided that it is adapted to a fully distributed data transfer architecture and storage model. In particular, the storage capacity available on Resources should be considered as limited rather than infinite due to the nondedicated nature of Resources in a P2P Grid.

5.1.3 Task Execution Parallelism

Task selection can increase the parallelism of Task execution on multiple Resources, but this may cause a so-called flash crowd. A *flash crowd* is a large number of downloaders simultaneously downloading the same file, thus overwhelming the original data server with download requests and causing a performance bottleneck.

Without efficient handling of flash crowds, Task execution parallelism is difficult to achieve. The BitTorrent P2P file sharing protocol is able to prevent such performance bottlenecks and has been proposed in several recent works [57, 56, 184, 309, 310], including ours [57, 56, 184], to address this issue. These works are discussed in a next section, after an introduction to BitTorrent and a discussion on its relevance.

5.1.4 Overview of BitTorrent

BitTorrent [87, 207, 46] is a file sharing protocol that enables computers to exchange files with one another in a P2P fashion. Built-in mechanisms enforce the cooperative behavior of a group of downloaders that has the common interest of downloading the same file at the same time.

BitTorrent Architecture

A BitTorrent Peer is any computer running the BitTorrent client [46] software. As opposed to what happens in other P2P file sharing protocols, a BitTorrent P2P network comes into existence for a single file only, i.e. a given BitTorrent Peer concurrently downloading two files is a member of two BitTorrent P2P networks.

In this chapter, to avoid confusion in the use of the terms Grid Peer and BitTorrent Peer, the term BitTorrent Peer is substituted with the term BitTorrent node, in order to disambiguate the usage of the term Peer. The term Peer is also always prefixed, i.e. Grid Peer. To avoid unnecessary complexity when referring to the bartering role of a Grid Peer, the terms Grid consumer Peer and Grid supplier Peer are substituted with the terms consumer Peer and supplier Peer, respectively.

A BitTorrent node that shares a complete file with other BitTorrent nodes is called a *seeder*. A BitTorrent node originally sharing a file must first split the file to share into pieces and also create a so-called torrent file. This sharing is at the piece-level, i.e. individual file pieces are requested and transferred. This metadata file describes the file to share, as well as the location of the tracker that is used.

The BitTorrent *tracker* [48] software is the file-level BitTorrent node discovery service[1]. It introduces to one another BitTorrent nodes interested in a given file. A BitTorrent node originally sharing a file must either launch its own tracker or use a publicly available tracker.

BitTorrent Protocol

Each BitTorrent node that wants to download a given file - i.e. each downloader - must first contact the tracker. From this point on, it belongs to the BitTorrent network for this file. Downloaders exchange pieces of the file with one another, and also download pieces from seeders. A BitTorrent node invites other BitTorrent nodes to cooperate by uploading to them (allowing them to download) the pieces it has already downloaded.

The BitTorrent protocol has a built-in incentive mechanism to incite reciprocity among BitTorrent nodes. Each BitTorrent node does not serve most of the downloaders that are interested in pieces it has already downloaded. At any time, a BitTorrent node only allows a few BitTorrent nodes to download pieces. In a tit-for-tat fashion, the accepted downloaders are those that have uploaded the most in a recent past.

To bootstrap the system, another policy, called the *optimistic unchoking* [87, 207] policy, also regularly selects at random one BitTorrent node to unchoke. This allows BitTorrent nodes to initially acquire a few pieces of the file, so that they can eventually start contributing to the BitTorrent network. Initially, a downloader waits to be selected a few times by the optimistic unchoking algorithm of other BitTorrent nodes, so that it can acquire a few pieces. The downloader then starts sharing these pieces with other

[1]Trackerless versions of the BitTorrent protocol are beginning to appear [117]. Given their recency, it is not clear yet whether they are resistant to sabotage by malicious BitTorrent nodes attempting to corrupt or to slow down the propagation of information that would be communicated by the BitTorrent tracker.

BitTorrent nodes. It continues exchanging pieces until the download is complete. Finally, it continues to share (serve pieces of) the file even after the download is complete.

BitTorrent Properties Relevant to Grid Computing

BitTorrent P2P networks are unstructured and need not be centrally deployed.

Collaboration between downloaders starts very early, as pieces of files, rather than whole files, are exchanged. As opposed to several other P2P file sharing protocols [222, 160], BitTorrent nodes do not have to wait for a file to be completely downloaded to begin uploading some of its pieces to other BitTorrent nodes. Every BitTorrent node downloading a file also can begin acting as an uploader as soon as it has downloaded at least one piece of the file (as opposed to: as soon as it has downloaded the whole file).

The BitTorrent protocol also specifies the default behavior that each BitTorrent node continues to act as uploader after its role as a downloader is finished. The sharing of a given file thus continues after its download is completed. This behavior is very useful in the context of P2P Grids, as will be explained in the next sections.

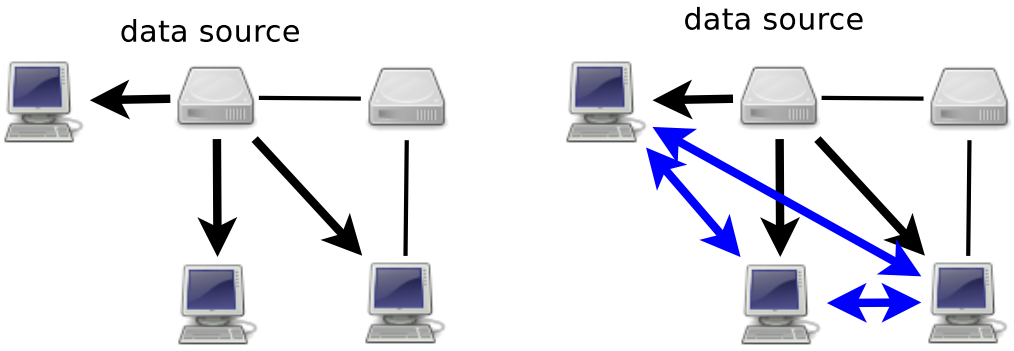
With BitTorrent, as opposed to what happens with direct file transfer protocols, network links between BitTorrent nodes are exploited (see Figure 5.2): As each downloader is also an uploader, the network load is removed from the original seeder and distributed among all downloaders and seeders. BitTorrent is able to exploit so-called orthogonal network bandwidth, which is made up of the *"physical network paths not included in a source-rooted application level tree."* [4]

The resulting advantage over direct file transfer protocols, e.g. File Transfer Protocol (FTP), is that the total transfer time of a file by multiple downloaders increases slowly with their number [309], i.e. less than linearly. This property makes any BitTorrent node able to efficiently handle a flash crowd of downloaders.

5.1.5 Why BitTorrent Is Relevant

Recent studies [4, 5, 189, 328] on the use of BitTorrent in Grid environments show that BitTorrent performs nearly as well as mechanisms based on explicitly built overlays[2] in over-provisioned network cores.

[2] *"An overlay network is a computer network which is built on top of another network. Nodes in the overlay can be thought of as being connected by virtual or logical links, each of which corresponds to a path, perhaps through many physical links, in the underlying network. For example, many P2P networks are overlay networks because they run on top of the Internet."* [234]



(a) centralized (e.g. FTP) data sharing (b) P2P (e.g. BitTorrent) data sharing

Figure 5.2: A Grid Peer (top) shares a file with 3 Resources. (a) With FTP data sharing, only the network links with the Grid Peer are exploited. (b) With BitTorrent data sharing, the 4 BitTorrent nodes (1 Grid Peer + 3 Resources) cooperate with one another: Network links between all downloaders are also exploited, leading to file download times essentially independent of the number of downloaders.

More importantly, these studies also indicate that BitTorrent sustains *"equivalent undegraded performance"* as the available network bandwidth decreases. In other words, BitTorrent performs well in managed and over-provisioned networks and, as opposed to explicit overlay-based techniques, maintains good performance in bandwidth-constrained networks that are more typical of P2P Grids environments.

Transferring data with the BitTorrent P2P file sharing protocol in a P2P Grid opens a very interesting perspective: As more Tasks sharing the same input data files are scheduled at the same time, the overall time of the multiple simultaneous data transfers remains close to the time of transferring it only once.

The relevance of using BitTorrent in P2P Grids is now discussed.

- First, BitTorrent is both highly scalable [287] and highly adaptive to unreliable network conditions, making it suitable for P2P Grids.
- Second, BitTorrent's incentive mechanism, i.e. the choking policy, discourages malicious Grid nodes to run misbehaved BitTorrent nodes.
- Third, data exchanges at the piece-level enable that multiple copies of a given input data file (required by multiple Tasks scheduled concurrently) are simultaneously transferred with a very high temporal efficiency, whereas data exchanges at the file-level would be much less temporally efficient.

- Fourth, the default behavior upon completion of a download is to continue sharing the downloaded file. This behavior actually encourages Grid Peers to schedule Tasks using Temporal Tasks Grouping as this has the effect of maintaining a large number of data sources even if a flash crowd is not totally synchronized, i.e. if Tasks needing the same data are not scheduled at the exact same time, or if network performance of Resources are highly variable, leading systematically to delayed response times on some of them. It is important to remark that, when supplementary storage space is needed, a Grid node will eventually stop sharing a completely downloaded input data file not needed by the currently running Task, thus effectively precluding downloaded files from saturating the storage capacities of a P2P Grid.
- Fifth, the BitTorrent node discovery service, i.e. the tracker, operates at an appropriate level of granularity, as each Grid Peer can operate its own BitTorrent tracker to manage a discovery service for its own files. Indeed, as explained in Section 5.1.4 there is no need for a global Grid-level tracker that would manage a BitTorrent node discovery service for all files of a P2P Grid.

Introducing QoS in BitTorrent [15, 194] has been proposed in recent research work in Content Networks. This research is very relevant and may lead to BitTorrent variants enhanced for P2P Grids.

Extracting the basic concepts from BitTorrent and using them to design new, lightweight BitTorrent-like protocols, such as GridTorrent [189] or Overhaul [240], may also be a promising avenue of research.

5.1.6 Why GridFTP Is Not Relevant

Classic Grids often exhibit certain features that can be taken advantage of, such as high speed networks and the presence of multiple network interfaces on many Grid nodes. Some direct data transfer protocols have been modified specifically for use in Grids with the goal to speed up massive data transfers.

GridFTP [165, 7] is an extended FTP protocol specifically designed for high performance, highly managed Grids. GridFTP [165] is *"a high-performance, secure, reliable data transfer protocol optimized for high-bandwidth wide-area networks. It is based upon the Internet FTP protocol, and it implements extensions for high-performance operation that were either already specified in the FTP specification but not commonly implemented or that were proposed as extensions by [the Globus alliance]."*

Historically, GridFTP has targeted high performance, controlled environments, in particular those involving cluster-to-cluster file transfers. One of its key strengths is the support for striping, i.e. parallel data transfers of a file through several network

interfaces. *"Data striping refers to the segmentation of logically sequential data, such as a single file, so that segments can be assigned to multiple physical devices [...] and thus written concurrently"* [104]. Using data striping can improve the performance of data transfers even with a single network interface [214]. However, data striping support - which is a key strength of GridFTP - cannot be exploited to its full potential in a P2P Grid context, if only because P2P Grid Resources rarely come with more than one active network interface.
The security features of GridFTP [235] ensure authentication of the parties as well as the encryption of the transferred data. This is hugely important in practice. However, we hypothesize that the support of most of these features (except maybe delegation of credentials) can be added transparently to other data transfer protocols as well.

Importantly, GridFTP does not exploit the network links between downloaders. In contrast, this is the key strength of BitTorrent.

Despite all its strengths, because the exploitation of the so-called orthogonal bandwidth is a requirement to maintain fully decentralized architecture, GridFTP should thus not be used as-is as the baseline protocol of the data transfer architecture of a P2P Grid. However, GridFTP could certainly be used as a secondary protocol to optimize some BitTorrent operations. As Allcock et al. [7] have pointed out, GridFTP *"could be used to good effect as a data transfer tool in"* BitTorrent to augment the reliability and performance of TCP/IP connections between BitTorrent nodes, as point-to-point data transfers would be performed with GridFTP.

5.1.7 BitTorrent Support in Existing Grid Middlewares

The idea of using the BitTorrent file sharing protocol in distributed computing systems has been proposed in several research works. BitTorrent has first been introduced in stable distributed environments (Desktop Grids and clusters) usually as loosely coupled, script-based implementations [184, 47], but also as a well-integrated, but specially modified, BitTorrent library [310, 309]. BitTorrent has also been considered, but not yet implemented, in mainstream Grids and Volunteer Grids. We have recently proposed to use BitTorrent in P2P Grids [56, 57], with a deeply integrated, off-the-shelf BitTorrent library [30].

Desktop Grids and Clusters

Using BitTorrent in a Desktop Grid has been recently and independently proposed in two studies, one by Wei, Fedak and Cappello [310], and one by us [184].

In the first mentioned study [310], a model of BitTorrent transfer times is described (building upon previous work [309]), along with a BitTorrent data transfer architecture and BitTorrent-aware Task scheduling.

Several BitTorrent-aware versions of classic, knowledge-based scheduling heuristics (BT-MinMin, BT-MaxMin, BT-Sufferage) are proposed. The BT-X knowledge-based scheduling heuristics [310] are designed to operate in a cluster or Desktop Grid environment. They require *"knowledge about communication performance and CPU load performance."* However, this data is not generally easy to obtain, and it is specifically not to be trusted in a P2P environment.

Furthermore, the proposed heuristics use a modified version of BitTorrent, where the standard choking policy is replaced by Predictive Communications Ordering (PCO): All Resources must follow a precomputed and inflexible data transfer schedule. This is only possible when Grid Peers downloading input data can be centrally managed, which is not the case in P2P Grids. For these two reasons (requirement of hard-to-obtain knowledge and PCO), the BT-X heuristics cannot be applied to the context of P2P Grids.

This very interesting work is, to the best of our knowledge, the first published proposal to couple Task scheduling and BitTorrent data transfers.

In the second mentioned study [184], which is part of the work leading to this dissertation, a Computer Vision Learning problem, structured as a Data-Intensive Bag of Tasks application, is presented. This BoT is shown to be successfully computed with a non-dedicated Desktop Grid running a basic, proprietary middleware. BitTorrent is used to simultaneously transfer half-gigabyte-sized data to several dozens of Resources. However, Task scheduling is very basic and unadapted to P2P Grids [184].

Finally, loosely coupled BitTorrent support has been implemented in a computer cluster [47] but, again to the best of our knowledge, there is no relevant publication.

Volunteer Grids

BitTorrent support for data transfers in Volunteer Grids has been considered [12] but, to the best of our knowledge, there is no relevant publication that proposes a complete mechanism or architecture.

BitTorrent support is planned to be implemented in the well-known BOINC middleware [51, 12, 92, 93] by the end of 2007. *"With BitTorrent fully in place by clients and servers late-2007, great savings are expected in the telecommunication cost structures of the current server user base."* [50].

Mainstream Grids

Recent research [4, 5] has shown through simulation that BitTorrent is indeed an excellent choice of data transfer technology in networks that are typical of a P2P Grid environment. It has also been shown that in the near future BitTorrent will become an excellent choice in networks that are typical of mainstream Grids.

The main goal of these recent studies [4, 5] was to investigate the possibility of using BitTorrent in mainstream Grids, i.e. high performance, highly managed Grids such as those running gLite [158] in the context of the CERN LHC experiments. A conclusion is that BitTorrent support will be required in future mainstream Grids as the deluge of data to process increases year after year and will eventually overcome the capacity of every existing network. This conclusion is supported by a recent BitTorrent-like protocol, called GridTorrent [189], that is designed specifically for mainstream Grids.

P2P Grids

The environment of P2P Grids is different from the stable environment of Desktop Grids, clusters and mainstream Grids. A data transfer architecture based on BitTorrent is even more relevant in P2P Grids.

Our recent work [56, 57] has proposed and shown how to use the BitTorrent file sharing protocol in P2P Grids. A Java implementation [55] has been publicly released in May 2007.

The main contributions reported in this publication are presented in the remainder of this chapter: (1) A scalable data transfer architecture, (2) an efficient Task scheduling policy designed to benefit from BitTorrent's efficiency at handling flash crowds and (3) the description of an operational software implementation tailored for P2P Grids. Our implementation is easily and automatically deployable, in a fully decentralized way. It does not need a centrally controlled and explicitly constructed data transfer overlay. It does not need Predictive Communications Ordering. It is deeply integrated with a BitTorrent library. All these features constitute improvements over the few early implementations of BitTorrent-based data transfer architecture in Grids.

Distributed, long-term storage of torrents, i.e. BitTorrent metadata files, is relevant as it may add help develop data persistence support in P2P Grids. Nodezilla [224], self-described as a Grid network, proposes reliable distributed storage for torrents.

5.1.8 Other Group-based Data Transfer Mechanisms

Besides multicast, which is difficult to implement both reliably and scalably [146], distributed cooperation and multi-sourcing, i.e. having multiple sources of data sharing (which may be seen as a variant of data replication), constitute the two main mechanisms of group-based data transfers.

Using a set of data caches scattered over a P2P Grid leverages the bandwidth of several Peers and partially redistributes the transfer load across the P2P Grid. Recent work includes the Super-Peer model [95] and the File Mover overlay network [17]. They however both require the explicit deployment of Peer-independent data caches as well as of a routing substrate, or overlay.

Having simultaneous downloaders of a given data file act cooperatively as a group with a common interest is another, efficient possibility. The BitTorrent [87, 207, 46] P2P file sharing protocol is a prime example of this approach.

5.2 Data Transfer Architecture

We propose to augment the P2P Grid architecture with a fully distributed and highly scalable data transfer architecture. Two overlays are coexisting: the P2P Grid overlay and the data transfer overlay. To achieve efficient data transfers across the P2P Grid:

- each Resource manages and downloads all required input data files as needed;
- each Grid Peer serves its own data files;
- Grid Peers are both BitTorrent nodes and FTP servers;
- Resources are also BitTorrent nodes;
- importantly, **Resources also act as transient data servers** to other Resources: **Data files downloaded by a Resource are automatically shared with BitTorrent.**

The most appropriate data transfer protocol (BitTorrent or FTP) is selected adaptively for each input data file downloaded by a Resource.

The data management, sharing, transfer and storage software components of our architecture are described in this section, as well as the data paths taken by transferred files between Grid nodes in the data transfer overlay. The scalability of the architecture is also discussed. Task scheduling and the data transfer protocol selection algorithm are covered in Section 5.3.4.

5.2.1 Data Scheduling

The data transfer mechanism is the following: Input data files of a given Task are transferred synchronously upon scheduling of the Task to a given Resource. When a Task is scheduled to a Resource, this triggers the Resource to initiate the downloading of the required, uncached input data files. It is the Resource that actually initiates and controls the file download, similarly to what is proposed in the Super-Peer model [95]. The data transfer mechanism is thus pull-based.

The timing of data transfers, or data scheduling, is automatically synchronized with the timing of Tasks scheduling, which is explained in Section 5.3.

5.2.2 Data Managers

A software component called Data Manager equips each Grid Peer and each Resource. Each Data Manager relies on its own data cache (will be described in Section 5.2.5) to manage the storage of data files. Each Data Manager is also deployed with data sharing and data transfer softwares.

Grid Peer Data Manager

A Data Manager running on a Grid Peer manages the input data files of its own Tasks only. The Grid Peer Data's Manager is used to download input data files from its User Agents, to store and to share these files with the Resources that process Tasks depending on them. Each Data Manager running on a Grid Peer is deployed[3] with a data cache, a BitTorrent tracker, a BitTorrent client and an FTP server. There are potentially as many BitTorrent trackers as there are Grid Peers. Each BitTorrent tracker offers a discovery service for all files shared by the Grid Peer with which it is deployed.

Resource Data Manager

A Data Manager running on a Resource manages input data files of Tasks belonging to the Grid Peer it is working for, as well as input data files belonging to any of the consumers of this Grid Peer. The Resource's Data Manager purpose is to download (from the Grid Peer by which it is managed or from other Resources) and store input data files. It also shares the stored input data files with other Resources running Tasks depending on these input data files. Each Data Manager running on a Resource is deployed with a data cache, a BitTorrent client and an FTP client.

[3] Data Managers running on Grid Peers may also run an FTP client. Along with the BitTorrent client, this enables a Grid Peer to use either protocol to get input data files from User Agents if these files are not embedded within Tasks.

Responsibility	(consumer) Peer deployed software	Resource deployed software
data storage	data cache	data cache
BT data tracking	Azureus BitTorrent tracker	- -
BT data sharing	Azureus BitTorrent client	Azureus BitTorrent client
FTP data sharing	Apache FTP server	- -
BT data downloading	- -	Azureus BitTorrent client
FTP data downloading	- -	edtFTPj FTP client

Table 5.1: Software deployed with Data Managers of Grid Peers and Resources. Grid Peers acting in a supplier role do not make use of the deployed software.

Deployed Softwares

The responsibilities of a Data Manager running on a Grid Peer and on a Resource overlap but are not equal. The sets of data sharing and data transfer softwares deployed on different Grid nodes are thus different (see Table 5.1). In practice, all the data sharing and data transfer softwares used in our architecture are available as Java libraries, allowing a 100% Java implementation: Azureus [30] (version 2.5.0.4), Apache FTP server [20] (version 20061027, slightly patched to enhance security by denying several FTP commands), and edtFTPj [130] (version 1.5.3). They are thus embedded into the LBG middleware [62, 57], can be deployed as-is and automatically on a great number of platforms, and do not require any extra support.

5.2.3 Data Preprocessing

When a BoT is submitted to a Grid Peer by a User Agent, all required input data files are transferred with the BoT to the Grid Peer. Before the Grid Peer queues the BoT, it preprocesses the input data files embedded within the BoT: They are removed from data structures of the BoT and stored into the data cache of the Grid Peer. Metadata are generated for each input data file and added to the data structures of the BoT's Tasks. The metadata of an input data file [99] consist of an automatically assigned file name based on a simple hierarchical scheme (in order to provide Grid-wide naming unicity that is required when inserting a file into a data

cache), a digital fingerprint [4] of the file and location metadata (FTP URL or BitTorrent torrent metadata including the address of the BitTorrent tracker deployed with the Grid Peer). With this preprocessing, Tasks can be transferred rapidly across the P2P Grid, as they contain only metadata. Resources where Tasks are scheduled know where to download the required input data files by consulting the metadata embedded within the Tasks.

5.2.4 Data Paths

Data Transfer Endpoints

A Grid Peer may schedule its Tasks to its own Resources (i.e. Local Tasks), or to other Grid Peers (i.e. Consumption Tasks). A Grid Peer may also schedule Tasks from other Grid Peers to its own Resources (i.e. Supplying Tasks). Endpoints of a data transfer in a P2P Grid vary according to the type of Task for which it is required.

A data transfer is initiated similarly for Local Tasks and Supplying Tasks. It is the Resource where the Task is scheduled that initiates, and constitutes the sink of, the data transfer. The source of a data transfer for a local Task is the Grid Peer managing the Resource running the Task, which acts as both consumer and supplier Peer. The source of a data transfer for an external Task is its consumer Peer, not the supplier Peer that owns the Resource running the Task.

A consumer Peer consumes computing time from other Grid Peers but, interestingly, it "supplies" its input data files to Resources (the computing and data transfer relationships are inverted). Input data files are not transferred immediately upon submission of a Consumption Task to a supplier Peer. Data transfers take place only after the supplier Peer has scheduled this Task - perceived as a Supplying Task - to one of its Resources. Such transfers never involve the supplier Peer and are strictly between the consumer Peer and supplied Resources.

Path of Data Transferred with FTP

If a Resource selects the FTP data transfer protocol to download a given input data file, this file is directly downloaded from the Grid Peer that shares it.

Path of Data Transferred with BitTorrent

If a Resource selects the BitTorrent data transfer protocol to download a given input data file, this file is simultaneously downloaded from, and shared with, other

[4] The integrity of downloaded files is checked with a SHA-256 hash.

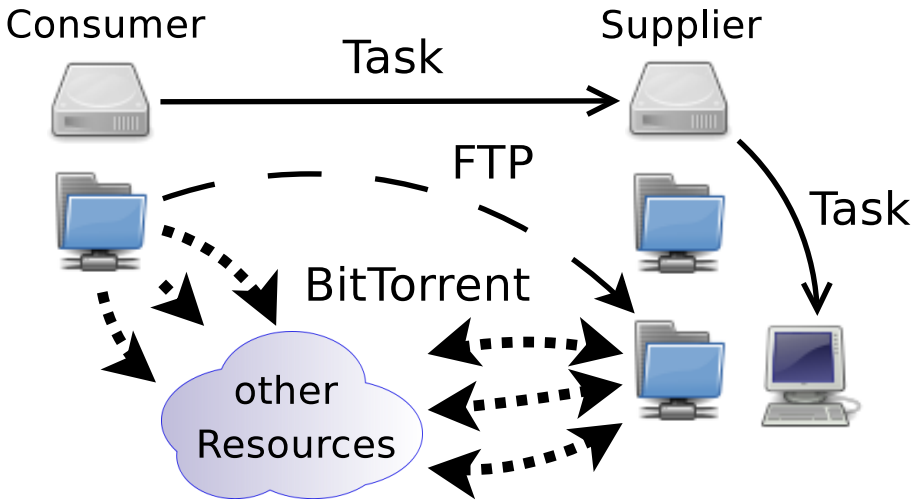


Figure 5.3: Possible data paths for the transfer of a single file from a consumer Peer (top left) to a supplier Peer (top right) and finally to a Resource of the latter.

Grid Peers and Resources already sharing or downloading it. The data path of a given file in the Grid overlay, from the Grid Peer that shares it to the Resource that downloads it, may therefore not be direct. Moreover, a Resource may download an input data file using BitTorrent entirely from other Resources which have already downloaded it, and not at all from the Grid Peer that shares it.

Path of Metadata

As explained in Section 5.2.3, metadata describing a given input data file of a given Task are actually embedded within the given Task, and thus transferred to a Resource along with the given Task. In particular, the BitTorrent torrent metadata of a given input data file - which is used to locate the BitTorrent tracker for the given file, as explained in Section 5.1.4 - are communicated in this way. The path and transfer timing of metadata are thus identical to those of the Tasks they are embedded into.

Summary of Possible Data Paths

Figure 5.3 shows the multiple data paths of input data files and, implicitly, of metadata in the Grid overlay. Plain lines represent the path of a Task from a consumer Peer to a supplier Peer, and then to the Resource where it is actually scheduled. Plain lines thus also represent the path of metadata, which are embedded within Tasks. It should be noted that when a Grid Peer schedules the Task directly to one of its own Resources, the representations of the consumer Peer and

supplier Peer should be collapsed on Figure 5.3. Long-dashed lines represent the data path of FTP-transferred input data files. Short-dashed lines represent the data path of BitTorrent-transferred input data files. Resources in the cloud of Resources represented at the bottom of Figure 5.3 may belong either to the consumer Peer, to the supplier Peer, or to other supplier Peers that are not involved in the represented Resource sharing but that have previously acted as supplier Peers to the represented consumer Peer.

5.2.5 Data Caches

Each Data Manager is equipped with a data cache that manages the storage of data files. A Grid Peer data cache is very basic, while more requirements are imposed on a Resource data cache.

A data cache running on a Grid Peer has an unlimited file capacity because it stores input data files of its own Tasks and a Grid Peer is supposed[5] to be able to accept Tasks submitted by its User Agents.

The available storage capacity on Resources is considered to be finite, as opposed to related works [267]. A data cache running on a Resource has a bounded capacity because Resources in a P2P Grid are typically edge computers. This capacity is the maximum number of cacheable files, as well as the maximum number of cacheable bytes.

Data Cache Parameters

The only supported operation on a data cache running on a Resource is the synchronization with a set of files, called *working set*, communicated by the Grid Peer that owns the Resource. A Resource's data cache is controlled by three parameters: (1) a cache size, (2) a working set and (3) a cache replacement policy.

Cache Size

The cache size bounds both the maximum number of files and the number of bytes that can be stored. It is configured statically by the human administrator of the Resource. Except where specified otherwise, the term *cache size* designates the maximum number of files allowed to be cached.

[5]Of course, as a practical measure, a maximum quota could be imposed on the number or byte count of input data files of the BoTs of each User Agent.

Working Set

Each Grid Peer maintains a separate working set for each of its Resources. A working set is the set of files that the corresponding Resource data cache must have in storage. The metadata of the files in the working set are communicated by the Grid Peer each time[6] a Task is scheduled to the Resource (see Section 5.2.1).

Upon reception of the working set metadata, a data cache synchronizes its contents with these metadata. The data cache first verifies which files are already cached using the hash and Grid-level name of the file that are contained in the metadata (see Section 5.2.3).

When a file of the working set is not stored in the data cache, the Resource downloads it from the Grid Peer sharing it; this Grid Peer is either the owner Peer of the Resource (in the case of Local Tasks), or the consumer Peer that submitted the Task requiring this file (in the case of Supplying Tasks). As already mentioned, if BitTorrent is used, the Resource may also download the file from other Resources that have already downloaded it.

Each Grid Peer guarantees a property for each of the working set it maintains for its Resources: The working set maintained for a given Resource always includes the files needed by the Task running on, or scheduled to, the Resource. Each Grid Peer also guarantees for each of its Resources that the associated working set can always be fully stored in the Resource's data cache. Consequently, a Grid Peer never schedules a Local Task or a Supplying Task to a Resource with insufficient cache size, either in term of available bytes or in term of available file slots.

Cache Replacement Policy

The cache replacement policy selects which files to eject from the cache when the insertion of files from the working set causes an overflow, e.g. the byte count or the number of files exceeds the cache size. Files not part of the working set are ejected from the cache following a Least Recently Used (LRU) policy until the working set is fully stored: Files the least recently used (and not in the working set) are ejected from the cache. As a consequence of the cache replacement policy, an input data file cannot be ejected from a Resource data cache as long as it is needed by the currently running Task.

As long as a file is stored in a data cache, it remains shared with BitTorrent. The size of a BitTorrent network for one given input data file, i.e. the number of

[6] They may also be communicated to the Resource asynchronously from Task scheduling (see Section 5.4).

BitTorrent nodes sharing or downloading a given file, depends upon the contents of Resource data caches across the P2P Grid. As long as there is at least one BoT yet to be completed that depends on a given input data file, the size of the BitTorrent network for this file is at least one. The Grid Peer to which the BoT was initially submitted continues to share this file until it is not needed by any Task of its queued BoT.

5.2.6 Data Trackers

Each Grid Peer is equipped with a software component called Data Tracker. It is not related to the BitTorrent tracker although it derives its name from a related purpose. It has the responsibility to continuously track the contents of the data caches of its Resources. It is actually a reverse mapping[7] of input data files to the Resources where they are actually stored.

The metadata stored in the data tracker can be used to:

- locate all Resources storing a given (set of) input data file(s);
- rank all Resources according to the number of bytes of the input data files of a given Task are already cached (ties are broken by selecting the Resource with the less filled data cache in order to balance storage among Resources);
- locate the most replicated of all tracked input data files, as well as the least replicated one.

5.2.7 Scalability of the Data Transfer Architecture

As explained in the previous sections, the BitTorrent-based data transfer overlay is distinct from the Grid overlay. Endpoints from both are connected (see Section 5.2.4). A P2P Grid is intrinsically scalable, by design: How scalable does it remain when transfers of large input data files occur?

Our proposed data transfer architecture is scalable for BitTorrent transfers. The load of BitTorrent data transfers is almost entirely on Resources, not at all on supplier Peers, and only minimally on consumer Peers (see Figure 5.3 again for an illustration).

Our proposed use of BitTorrent can be classified as following the BitTorrent hybrid model [15], where the original data source, i.e. the consumer Peer serves data

[7] It consists of a 2-levels data structure backed by balanced binary trees: The first level maps each file to a Resources set, each of which is stored in a separate data structure.

along with other BitTorrent nodes, i.e. Resources. Consequently, a consumer Peer sharing data files indeed experiences some load, but considerably less than with FTP transfers.

To completely remove the load from consumer Peers, it could be imagined that each consumer Peer first updates the working set of a handful of its Resources with replicas of the input data files that are going to be needed by one of its Tasks soon to be scheduled. The consumer Peer would not schedule any Task to these Resources so that they are used exclusively to share input data files on its behalf. Immediately after these Resources have downloaded the input data files to share, the consumer Peer could entirely stop to share these files, effectively removing from itself any load due to their sharing.

Our proposed architecture of data transfers is not immediately scalable for FTP transfers. The load of FTP data transfers is totally on consumer Peers, not at all on supplier Peers and only minimally on Resources (see Figure 5.3 again). More accurately, the load due to FTP transfers is completely on the FTP servers co-located with consumer Peers. So our proposed architecture could be made even more scalable by using so-called Content Distribution Networks [145, 242, 279] (CDN). This would however come at the expense of maintaining multiple FTP servers. Interestingly, BitTorrent could be used to synchronize multiple CDN servers efficiently. Synchronization would become very scalable and the resulting network load would be spread among the CDN servers.

5.3 Task Scheduling

Scheduling one Task involves operations to be made data-aware (see Figure 5.4):

- Task selection: selecting a Task in the Local Tasks queue (Local Tasks and Consumption Tasks scheduling) or Supplying Tasks queue (Supplying Tasks scheduling);
- Resource selection: selecting on which Resource to schedule the selected Task (Local Tasks and Supplying Tasks scheduling);
- supplier Peer selection: selecting on which supplier to schedule the selected Task (Consumption Tasks scheduling);
- data transfer protocol selection: selecting for each input data file of the Task which data transfer protocol (BitTorrent, FTP) to use (Local Tasks and Consumption Tasks scheduling).

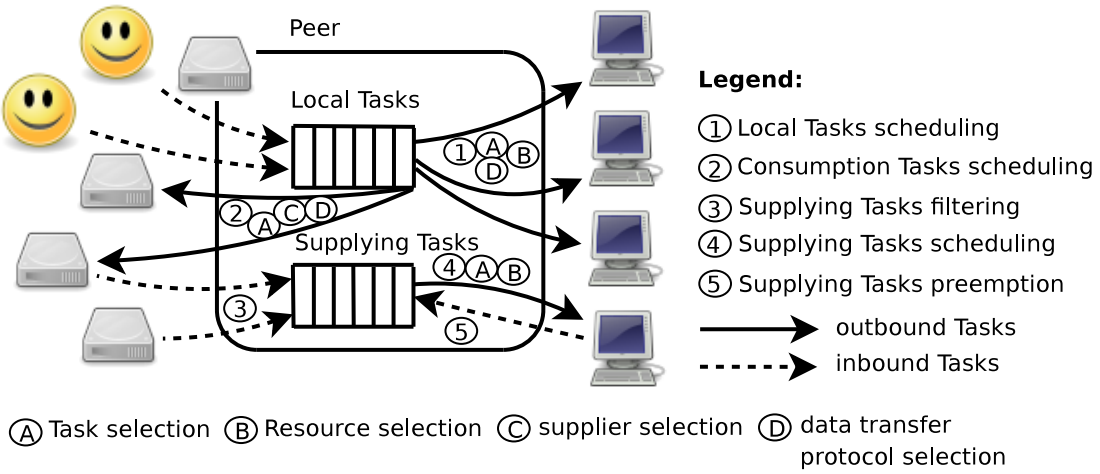


Figure 5.4: Scheduling model (extended version of Figure 2.20).

5.3.1 Task Selection

We propose a novel Task selection algorithm, called Temporal Tasks Grouping (TTG). It computes a schedule that determines the order in which Local or Supplying Tasks should be selected for execution. It is based on the observation that any Data-Intensive BoT may be partitioned into several groups of Tasks, with each group depending on separate subsets of redundant input data files.

Let $\theta = \theta_0, \ldots, \theta_{n-1}$ be a Bag of n Tasks. Let Δ_i be the set of input data files of Task θ_i, Δ_i^j its j^{th} input data file and $|\Delta_i^j|$ the size (in bytes) of the j^{th} input data file. Two Tasks θ_i, θ_k are said to be *related* when they have at least one input data file in common, i.e. $\exists j, l : \Delta_i^j = \Delta_k^l$. A set of Tasks θ is said to be *connected* if every θ_i is related to at least one other Task, i.e. $\forall i \exists k : \theta_i, \theta_k$ are related. Any BoT can be partitioned into disjoint connected sets of Tasks by repeatedly applying a transitive closure algorithm.

A *schedule* $\sigma(\theta)$ of a set θ of Tasks is an ordering of θ. When scheduling Tasks of a BoT, Task selection consists in following the schedule computed for the BoT. A *subsequence* $\bar{\sigma_s}(\theta)$ is a section of this ordering, and its length (expressed as a number of Tasks) is noted $|\bar{\sigma_s}(\theta)|$. The (statically computed) distance between the sets of input data files of the two Tasks θ_i, θ_k is the sum of the sizes of the input data files of θ_k that are not identical with θ_i: $d(\Delta_i, \Delta_k) = \sum_l |\Delta_k^l|, \forall l : \Delta_k^l \notin \Delta_i$. This distance is not symmetric.

To schedule at the same time Tasks sharing some input data files, it could be efficient to minimize the sum of distances between subsequent Tasks within schedules. This would require to take into account the variability of the number of available

Resources, and to solve an asymmetric[8] Travelling Salesman Problem [203] for each schedule, which is computationally hard. Instead, we propose to schedule Tasks sharing input data files at the same time.

Two Tasks are said to be *data-equal* when they have all their input data files in common, i.e. $d(\Delta_i, \Delta_k) = 0$. A *data-equal subsequence* of θ is an extensive subsequence of data-equal Tasks: The Tasks immediately before and after the subsequence are not data-equal to those of the subsequence. Temporal Tasks Grouping consists in sorting by decreasing length $|\bar{\sigma_s}(\theta)|$ the data-equal subsequences of a schedule (see Table 5.5). Multiple data-equal subsequences of similar length may then be sorted using a nearest neighbor algorithm [203]; the metric can be the distance $d(\Delta_i, \Delta_k)$ between two Tasks of neighbor subsequences, as the input data files of one Task are representative of those of all Tasks in a subsequence.

Importantly, computing a schedule with TTG can be performed in batch-mode [215, 71], i.e. it considers a set of Tasks, but statically at the submission time of the BoT, rather than dynamically, as it is decoupled from the Resource selection algorithm. TTG takes into account only the BoT itself; it does not need any information from other Grid Peers. Selection of Local Tasks or Supplying Tasks to schedule consists in following the schedule computed for the BoT.

The main benefit of Temporal Tasks Grouping is to ensure that the scheduling of data-equal Tasks is temporally grouped so as to maximize efficiency of BitTorrent transfers. Another benefit of TTG is that the largest groups of data-equal Tasks are scheduled first (the length of the scheduled data-equal subsequences is necessarily decreasing). Indeed, the Peer negotiator (see Section 2.9.5) tries to obtain as many consumption grants as possible as soon as a BoT is scheduled, which contributes to the synchronization of the scheduling of Tasks belonging to the largest data-equal subsequences.

As future work, it might be interesting to provide to the negotiator the size of the next group of data-equal Tasks to schedule, so that it can hold the received consumption grants until a certain number of consumption grants have been received. This could increase the temporal synchronization of the scheduling of data-equal Tasks, if the supplier Peers are not so busy that consumption grants are representative of their state only for short periods of time.

[8] City = data cache. Tour = schedule of Tasks, with input data files cached as a side effect. It is an asymmetrical TSP because the impact on data availability of scheduling θ_i then θ_j to a Resource may not be the same as scheduling θ_j then θ_i, given the finite size of data caches.

Original Tasks input data files of Bag of Tasks θ

Δ_0	Δ_1	Δ_2	Δ_3	Δ_4	Δ_5	Δ_6	Δ_7
$\{g\}$	$\{i\}$	$\{g\}$	$\{r\}$	$\{g\}$	$\{g\}$	$\{d\}$	$\{r\}$

Sorted Tasks input data files of Bag of Tasks θ

Δ_0	Δ_2	Δ_4	Δ_5	Δ_3	Δ_7	Δ_1	Δ_6
$\{g\}$	$\{g\}$	$\{g\}$	$\{g\}$	$\{r\}$	$\{r\}$	$\{i\}$	$\{d\}$

Figure 5.5: Tasks of BoT θ (here with 1 input data file per Task) are grouped into data-equal subsequences, which are sorted by decreasing length $|\bar{\sigma}_s(\theta)|$.

5.3.2 Resource Selection

The Resource selection algorithm selects a Resource to schedule Local and Supplying Tasks. We propose to adapt an existing supplier Peer selection algorithm, Storage Affinity [267], that explicitly takes data placement into account. Storage Affinity must be adapted in the sense that the LBG data transfer architecture is fully distributed.

Let Δ_{R_x} be the contents of the data cache of Resource R_x. At a given time, it contains input data files accumulated from previous Task executions.

When a Grid Peer is scheduling a Local or Supplying Task θ_i, a Resource is located by minimizing the (dynamically computed) distance $d(\Delta_i, \Delta_{R_x})$ between the input data set Δ_i of the Task to schedule and the data cache of each Resource. This distance, computed dynamically, represents the transfer cost of scheduling θ_i on R_x. It requires data tracking support: Each Grid Peer knows the contents of the data cache of each of its Resources at any time, which is made possible by the way data caches operations have been designed, e.g. the contents of a data cache of a Resource is synchronized on the working set communicated by the Grid Peer that owns the Resource.

After a few Tasks have been scheduled, the minimum distance is expected to be small. There is probably at least one Resource with a data cache already storing most of input data files of Δ_i due to the execution of previous Tasks. This minimization is equivalent to maximizing the Storage Affinity [267] metric of the given Task with the contents of the data caches of the Grid Peer's Resources.

If Resources of a given Grid Peer exhibit large variation in reliability or performance, the data-aware Resource selection should be complemented with a performance- or reliability-aware [58] Resource selection algorithm (see Chapter 4), or with Task replication [267] (see Section 2.9.4).

5.3.3 Supplier Peer Selection

Data placement also has to be taken into account when selecting a supplier Peer to schedule Consumption Tasks. Another variant of an existing supplier Peer selection algorithm, Storage Affinity [267], is now introduced as supplier Peer selection algorithm. Given the context of P2P Grids, the state of the data caches of a supplier Peer is not accurately known to other Grid Peers. Awareness of data placement can be taken explicitly into account, but based on estimates (as opposed to Storage Affinity) rather than on actual knowledge (similarly to Storage Affinity).

The state of the data caches of suppliers is estimated based on metadata that a Grid Peer can acquire independently. For each Local BoT, a Grid Peer maintains the list of supplier Peers that have supplied Resources for at least one Task of the BoT; the Grid Peer also maintains a reverse mapping of suppliers to input data files. These mappings become inaccurate when input data files are ejected from the data caches of suppliers' Resources. Even if accurate, a supplier Peer may schedule a Supplying Task poorly, if only because the Resources whose data caches contain the required input data files are busy at scheduling time. As future work, it will be useful to quantify the data storage reliability of supplier Peers, i.e. to estimate the ejection rate of files from data caches of supplier Peers' Resources. This is not straightforward, as an input data file that has to be downloaded again may simply have been cached by a busy Resource; therefore, multiple downloads of a given input data file by Resources of a supplier Peer do not necessarily indicate the ejection of this file from the supplier Peer's data caches.

To schedule a Consumption Task, a Grid Peer selects a supplier Peer among a set of suppliers. Those that already processed other Tasks of the same BoT are ranked first. The ranking is performed using the same distance that was defined in Section 5.3.2 (Resource selection), except that it is applied to the supposed contents of the set of data caches of the target Grid Peer's Resources instead of the data cache of one Resource.

An important consequence of the organization of our proposed data transfer architecture is that the impact of ineffective data-aware scheduling decisions is mitigated by the availability, in other Resources of the P2P Grid, of cached copies of a file to download. If the required input data file has never been downloaded into the P2P Grid before, or if it has been downloaded a long time ago (thus, not cached anymore), all scheduling decisions are equivalent, as far as data-awareness is concerned. If the required input data file is already cached on one Resource, an ineffective scheduling decision would consist either in scheduling a Consumption Task to a supplier Peer with no Resource having this file in cache, or in scheduling

a Local Task or Supplying Task to a Resource that does not have this file in cache. Following the cache miss, this file can be downloaded with BitTorrent from at least one other source than the original data server (i.e. consumer Peer); this mitigates bottlenecks arising at the original data server.

Finally, it can be remarked that data-aware supplier Peer selection by a consumer Peer and data-aware Resource selection by the selected supplier Peer, even if based only on estimates, are both needed to provide end-to-end data-aware scheduling. Data reuse is thus a fully distributed, cooperative effort. It should be reminded that the supplier Peer is motivated, by the incentives of the bartering mechanism (see Section 2.3.4), to offer good response times to the consumer Peer. This implies that suppliers Peer should seek to download data diligently.

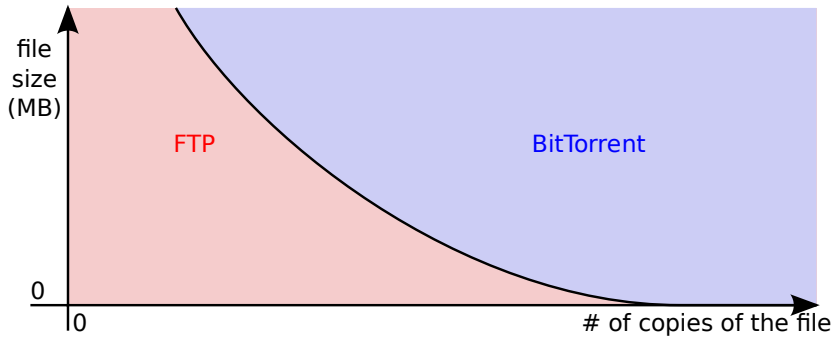
5.3.4 Data Transfer Protocol Selection

The research presented in this section results from cooperative work with our colleague Xavier Dalem.

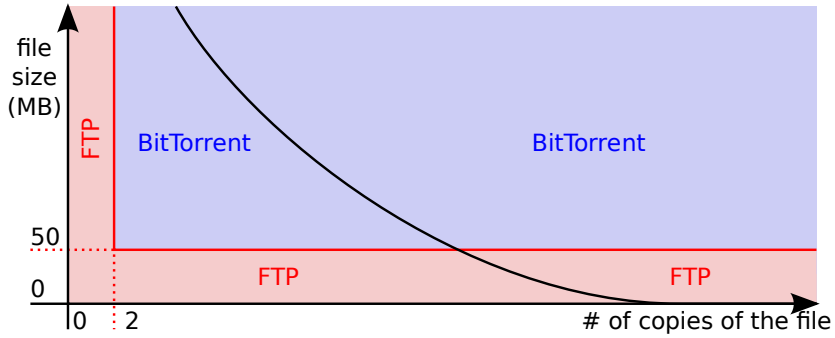
Neither BitTorrent nor FTP achieves the shortest download times in all situations. In most setups, a given input data file is downloaded in a shorter time by a group of BitTorrent nodes rather than by a group of FTP clients. However, there exist conditions that may lead to shorter download times with FTP. Specifically, a very small file size or a very small number of potentially concurrent downloaders lead to shorter download times with FTP. This is caused by the larger overhead of the BitTorrent protocol: BitTorrent nodes must initially wait to be selected a few times by the optimistic unchoking algorithm of other BitTorrent nodes, as explained in Section 5.1.4. On Figure 5.6a, the black curve separates the setups such that BitTorrent or FTP leads to shorter download times. The zone below the curve consists of all setups where FTP should be preferred.

We propose a variant [56, 57, 99] of an existing algorithm. The existing algorithm [310, 309] is based on an analytical model of download times in function of the file size, network bandwidth, latency of the data transfer protocol and also of a factor that is logarithmic in the number of downloaders. To make a decision, the output of the analytical model for each protocol (BitTorrent, FTP) is compared and the fastest protocol (for the given setup) is selected.

Our data transfer protocol selection algorithm, illustrated on Figure 5.6b, determines if a given input data file should be shared with BitTorrent or with FTP, based on the size of the file (following Wei et al.'s [310, 309] results) and on its level of redundancy within the BoT and in data caches over the P2P Grid (the latter is obtained indirectly from the BitTorrent tracker, as a consumer Peer has



(a) Optimal data transfer protocol selection
(for a data transfer load that is low relative to the capacity of the P2P Grid)



(b) Actual data transfer protocol selection

Figure 5.6: Data transfer protocol selection algorithm for one file.

no access to the data caches of supplied Resources). It is applied once to all files of a Local BoT. On one hand, it is simpler, thus less accurate, than the existing algorithm. On the other hand, it is easier to calibrate, thus to implement, as it does not use estimates of the available network bandwidth (which would have to be periodically updated given the variability in the performance of P2P networks). Furthermore, as opposed to the existing algorithm, it does not rely on Predictive Communications Ordering (PCO).

In the 100 Mbps switched Ethernet network used in our experiments, two concurrent downloads of a data file and a file size of ∼50 MB, are sufficient for BitTorrent to outperform FTP. As illustrated on Figure 5.6b, there is a zone where BitTorrent is selected but FTP should be preferred. The data transfer load of the P2P Grid influences the area of this zone, which tends to shrink under heavy load. In situations of high contention, BitTorrent should be preferred so that the proposed algorithm leads to lower or equal download times. In situations of low contention, the absolute temporal penalty is limited, as the number and size of downloads are intrinsically limited in the FTP-preferred/BitTorrent-selected zone. Predicting the

situations of high contention would also be difficult, as it depends on BoTs submitted across the P2P Grid, as well as out-of-Grid network traffic. This is why we propose a simple heuristic that behaves well in the worst cases and offers reasonably degraded performance under low contention. Finally, the human administrators of Resources and Grid Peers can always override the data transfer protocol selection algorithm. This is useful if one of the protocols is filtered at the network level.

5.4 Proactive Data Replication

In this section, a recent bartering-based data sharing mechanism that could benefit our proposed data transfer architecture is first reviewed. It motivates idle BitTorrent nodes to help busy BitTorrent nodes download data files, with an expectation of future reciprocity. We then propose a proactive data replication mechanism. It is designed to speculatively cache input data files in idle Resources of a given Grid Peer by proactively replicating the most popular input data files of recently scheduled Supplying Tasks.

5.4.1 Bartering Bandwidth with Idle BitTorrent Nodes

A variant of the incentive mechanism of BitTorrent, called 2Fast [155], has been proposed as part of Tribler [295, 250], which is a classic P2P network where only data files are exchanged. Tribler itself relies on the standard BitTorrent [87, 207, 46] protocol to manage data exchanges between BitTorrent nodes interested in a given file. 2Fast operates on top of Tribler following a bartering model similar to the Network of Favors [13] (see Section 2.3.4). 2Fast seeks to harness the bandwidth of BitTorrent nodes not belonging to the BitTorrent network of a given file. 2Fast relies on a byte-level accounting of data transfers between BitTorrent nodes to motivate those not related to the file to download it nonetheless.

2Fast provides incentives to idle BitTorrent nodes by making their contributions recognized and memorized. Idle BitTorrent nodes are thus motivated to help busy BitTorrent nodes to complete their ongoing transfers. BitTorrent has indeed the property that additional downloaders of a given file further help to spread and balance the network load. The standard BitTorrent protocol could be seen as enabling an immediate, restricted form of bartering, with 2Fast enabling a delayed, more general form of bartering, similarly to the Network of Favors.

The 2Fast [155] protocol could be added to the LBG data transfer architecture to leverage the bandwidth of idle Resources. 2Fast could even be extended so that idle Resources contributing their bandwidth also take account of the bartering (of

computing time) relationships of their owner Grid Peer. Adding 2Fast to LBG would contribute to the increasing of bandwidth utilization and to the decreasing, in the P2P Grid, of all download times when BitTorrent is used.

5.4.2 Replicating Data to Idle Resources

With our proposed data transfer architecture, input data files stored in Resource data caches continue to be shared for some time, until they are ejected because they have not been needed in a recent past. This property can be leveraged by a Grid Peer to augment the cache hits of its Resources in a near future. We propose that a Grid Peer proactively makes its idle Resources download input data files that are predicted to be required by Supplying Tasks in a near future.

To this end, we propose a proactive, asynchronous (from Task scheduling) data replication mechanism [56]. Proactive replication benefits the owner Peer of these idle Resources when it schedules Supplying Tasks requiring these input data files. Indeed, not having to download these files contributes to increasing utilization and decreasing response times of the owner Peer. Proactive replication also benefits to other Grid Peers. Indeed, they benefit from supplementary sources of sharing, which contributes to decreasing download times using BitTorrent. Proactive data replication can also be first applied to queued Local Tasks before it is applied to queued or expected Supplying Tasks.

The mechanism works as follows: A supplier Peer triggers some of its idle Resources to download, using BitTorrent, the most popular input data files of Supplying Tasks, that are already cached (or in the process of being cached) by other Resources. The activation of the replication process can occur at any time. The replication of a selected input data file can be triggered immediately after it has begun to be downloaded by Resources where scheduled Tasks require it. This replication could also be triggered at a later time, which promotes bandwidth utilization anyways.

The contents of a Resource data cache are controlled through its working set (see Section 5.2.5). Each time a Grid Peer schedules a Task one of its Resources, this Grid Peer communicates the metadata of the working set associated to the Resource. These metadata must include the metadata of the input data files required by the scheduled Task. They also include the metadata of cached files not ejected from the cache by the files required by the scheduled Task. The Grid Peer can also asynchronously make the Resource synchronize its data cache by communicating working set metadata that also include the metadata of files selected for data replication.

The selection by a Grid Peer of input data files to be replicated works as follows:

- Following a spatial locality principle, we hypothesize that popular input data files - i.e. files that are cached by a large number of the Grid Peer's Resources - are also good candidates for replication. Moreover, each copy of a popular input data file also constitutes an additional source of sharing, which is an advantage when using BitTorrent. A popularity metric is defined by counting occurrences of input data files in Resource caches. It is similar to the metric proposed by Ranganathan's [257] which counts the occurrences of input data files in recent scheduling decisions.

- Following a temporal locality principle, we hypothesize that, if several Tasks (of a given consumer Peer) sharing the same input data files have been run in the recent past, more of them will be run in the near future, at nearby locations. To take recency into account, our popularity metric uses a sliding window to take account only of the input data files of the K most recent scheduled Supplying Tasks.

- In a spirit similar to BitTorrent's optimistic unchoking policy [87, 207], adding some randomness augments the diversity of the replicated data, which in turns - we hypothesize - increases the probability of moderately popular data to be replicated.

The input data files replicated to a given Resource includes the most popular input data files (exploitation of high utility), as well as a few randomly selected input data files (search for high utility). The replicated files should never eject already cached files, but only fill empty slots. Our proposed data replication mechanism is therefore controlled by the following parameters: timing of the activation of data replication, value of the sliding window, number of files to replicate, proportion (i.e. weight) of random files among the files to replicate. As opposed to our proposed data transfer architecture and Task scheduling algorithms, the data replication mechanism has yet to be experimentally evaluated, as future work.

Our proposed data replication mechanism is similar to related state of the art work by Ranganathan et al. [257], but is slightly different: Our mechanism is *pull-based*, i.e. initiated by Resources of supplier Peers, rather than *push-based* and our popularity metric also uses a sliding window.

A data replication mechanism based on the *replicator* concept has recently been proposed [218] to provide an augmented version of BitTorrent. The replicators are BitTorrent nodes artificially introduced to support a BitTorrent network, unlike our mechanism. It is pull-based, similarly to our mechanism. It is not targeted towards P2P Grid and is specifically limited to symmetrical-bandwidth networks.

5.5 Experimental Results

In this section, the proposed combination of algorithms and protocols are evaluated with a real application using a full deployment of the complete middleware. Useful metrics are first introduced. Important implementation details are discussed. The hardware environment and Grid configurations are then presented. Experiments evaluating various parameters are then described.

5.5.1 Useful Metrics

Data Diversity Ratio

A measure of the diversity (and, conversely, of the redundancy) of input data files between Tasks of a BoT is the *Data Diversity Ratio* (DDR). It corresponds exactly to the Shared Data Ratio that has been used by Wei et al. in recent work [310], but has been renamed to reflect more accurately its semantic.

The DDR is defined as the size (in bytes) of all the distinct input data files among all Tasks of the BoT, divided by the total size (in bytes) of all input data files of all Tasks of the BoT: $\text{DDR} = \frac{number\ of\ bytes\ of\ distinct\ files}{number\ of\ bytes\ of\ all\ files}$.

If there is no sharing of data, DDR = 1. The DDR decreases towards 0 as the level of sharing increases. It has a lower bound that is the size of the smallest file of the BoT divided by the number of bytes of all files in the BoT; this bound simplifies itself to $\frac{1}{file\ count\ in\ BoT}$ when all files are identical. The following holds:

$$(0 < \frac{number\ of\ bytes\ of\ smallest\ file}{number\ of\ bytes\ of\ all\ files} \leq \text{DDR} \leq 1)$$

For example, the DDR of a Bag of three Tasks depending on the same input data file is 1/3, which also corresponds to the lower bound.

Inter-Task Data Sharing

There is *Inter-Task data sharing* within a BoT whenever two or more Tasks share at least one input data file. It is equivalent to the condition $DDR < 1$. This metric was introduced by Santos et al. as *"inter-task data reutilization"* [267].

Inter-BoT Data Sharing

There is *Inter-BoT data sharing* between BoTs whenever there is data sharing between two subsequently submitted BoTs, i.e. at least one input or output data file of the firstly submitted BoT is also included as an input data file of the secondly submitted BoT. In the following, the definition is restricted to input data files only

because there currently is no support for the caching of output data files in LBG. This metric was introduced by Santos et al. as *"inter-job data reutilization"* [267].

Mean Cache Hit Ratio

It can be useful to estimate the efficiency of Task scheduling expressed as a measure of good matching of Tasks and Resources such that a large number of input data files are cached by the selected Resources. The *cache hit ratio* of a Resource is the number of cache hits, divided by the number of cache queries, of the Resource's data cache. It is thus a real number between 0 and 1 included.

The cache hit ratio of a Resource is updated each time a file is required by the synchronization of the data cache with the working set, i.e. when the input data files of a scheduled Task are either retrieved from the cache or downloaded from the P2P Grid. If scheduling decisions are efficient, the cache hit ratio is high, many input data files are cached and few input data files have to be downloaded.

The *mean cache hit ratio* of a Grid Peer is the mean of the cache hit ratios of Resources of this Grid Peer. It is thus also a real number between 0 and 1 included. Human administrators of Grid Peers typically expect high mean cache hit ratios, which constitute an indicator of efficient system utilization.

5.5.2 Deployment

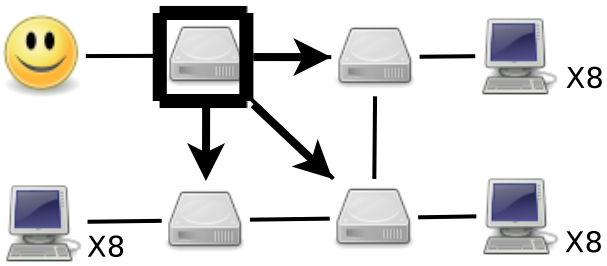
Hardware Environment

The experiments presented in this chapter have been conducted on 30 x86 PC (Intel P4 CPU 3GHz with 1GB RAM), all equipped with standard hard drives, and connected with a 100 Mbps switched Ethernet network.

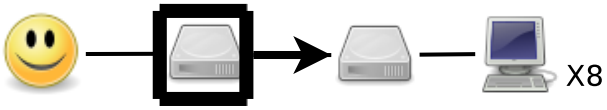
The P2P Grids considered in the experiments presented in the next sections 4 Grid Peers (1 Grid Peer acting in a consumer role, 3 Grid Peers acting in a supplier role), 24 Resources (8 assigned to each supplier Peer), 1 User Agent (submitting Bags of Tasks to the consumer Peer) and 1 Search Engine. 24 PC (out of 30 available) are used as Resources. 6 PC are used as other Grid nodes: 4 Grid Peers, 1 User Agent and 1 Search engine.

Grid Configuration

In order to avoid interferences arising from queueing issues in the presented experiments, only one consumer Peer is deployed and it has no Resource. For the same reason, only the single consumer Peer submits Tasks to the supplier Peers, which



(a) Grid topology I: 1 consumer, 3 suppliers (managing 8 Resources each).



(b) Grid topology II: 1 consumer, 1 supplier (managing 8 Resources).

Figure 5.7: Grid topologies for P2P data transfer experiments.

Original Tasks input data files of Bag of Tasks θ

Δ_0	Δ_1	Δ_2	Δ_3	Δ_4	Δ_5	Δ_6	Δ_7
$\{g\}$	$\{r\}$	$\{i\}$	$\{d\}$	$\{g\}$	$\{r\}$	$\{i\}$	$\{d\}$

Figure 5.8: Data-equal Tasks of submitted BoTs are ordered as far as possible from one another, so that FIFO Task selection results in temporal ungrouping of data-equal Tasks.

have no User Agent and thus are not submitted any Local Task. Figure 5.7 illustrates the two Grid configurations that are deployed for the following experiments.

In order to control the impact of Task runtimes on the overall BoT response times, the real application that is considered is a simple *Hello, Grid*-type Bag of Tasks. The code of each Task first sleeps for 2 seconds (wall-clock time) then hashes the input data file associated with the Task.

One large input data file is associated to each Task of each submitted BoT. In case of data redundancy between Tasks, i.e. DDR < 1, Tasks within a submitted BoT are ordered so as to maximize the distance between data-equal Tasks defined in Section 5.3.1 (see Figure 5.8). This initial order of Tasks within a submitted BoT is selected because it degrades BitTorrent performance the most if a FIFO Task selection algorithm is used instead of TTG. There may exist several such permutations for a given BoT.

Several parameters are taken into account:

- number of submitted BoTs,
- number of Tasks of each submitted BoT,
- size of the processed input data files (bytes count),
- DDR,
- data transfer protocol,
- Resource selection algorithm,
- Task selection algorithm,
- cache size (capacity expressed as a number of files).

Mean BoT response times (MBRT) and (when applicable) mean cache hit ratios, are evaluated for each.

5.5.3 Varying Data Protocol, Task Selection, Caching

Multiple combinations of data transfer protocols, Task selection algorithm and caching support are evaluated. Grid topology I (see Figure 5.7) is used. One BoT of 100 Tasks with a medium DDR of 0.25 is submitted, i.e. 25 groups of 4 Tasks depending on an identical input data file, with Tasks depending on the same input data file spread as much as possible in the submission order. Results presented in Figure 5.9 and Table 5.2 show that in this setup BoT response times are much better, i.e. shorter, with both BitTorrent data transfers and Temporal Tasks Grouping activated.

If Temporal Tasks Grouping is activated, the caching mechanism cannot bring any benefit in this experiment. There are 25 sets of 4 data-equal Tasks: With TTG, all 4 Tasks of any given set are scheduled at the same time to different Resources. The cache hit ratios thus remain equal to 0.0, as none of the files is already cached.

If TTG is not activated, increasing the cache size leads to increasing cache hit ratios and decreasing BoT response times, as expected. Also as expected, the performance is degraded with BitTorrent data transfers when there is a small cache size and no TTG. If the data transfer protocol selection algorithm were activated, BitTorrent would be selected for all setups in this experiment: As FIFO Task selection does not degrade the performance of FTP data transfers, activating TTG ensures that the performance of BitTorrent data transfers is not degraded.

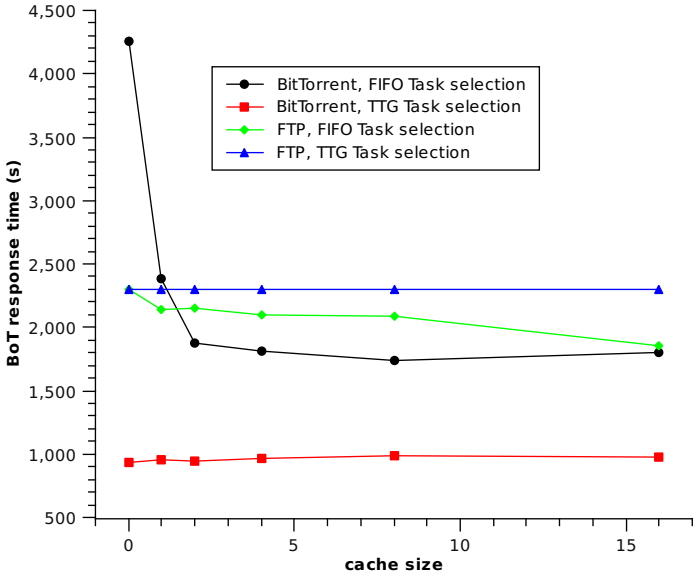


Figure 5.9: Varying data protocol, Task selection, caching.

Fixed parameters	Varying parameters
• 1 BoT of 100 Tasks • file size = 256 MB • DDR = 0.25 • Resource selection = data-aware	• data transfer = { BitTorrent | FTP } • Task selection = { TTG | FIFO } • cache size (# files) = { 0 | 1 | 2 | 4 | 8 | 16 }

data transfer = BitTorrent, Task selection = FIFO

cache	0	1	2	4	8	16
cache hit ratio	0.0	0.02	0.07	0.09	0.10	0.11
MBRT (s)	4260	2388	1874	1814	1739	1806

data transfer = BitTorrent, Task selection = TTG

cache	0	1	2	4	8	16
cache hit ratio	0.0	0.0	0.0	0.0	0.0	0.0
MBRT (s)	935	950	948	961	984	972

data transfer = FTP, Task selection = FIFO

cache	0	1	2	4	8	16
cache hit ratio	0.0	0.05	0.06	0.07	0.08	0.18
MBRT (s)	2297	2137	2146	2093	2092	1850

data transfer = FTP, Task selection = TTG

cache	0	1	2	4	8	16
cache hit ratio	0.0	0.0	0.0	0.0	0.0	0.0
MBRT (s)	2297	2298	2296	2298	2296	2298

Table 5.2: Varying data protocol, Task selection, caching.

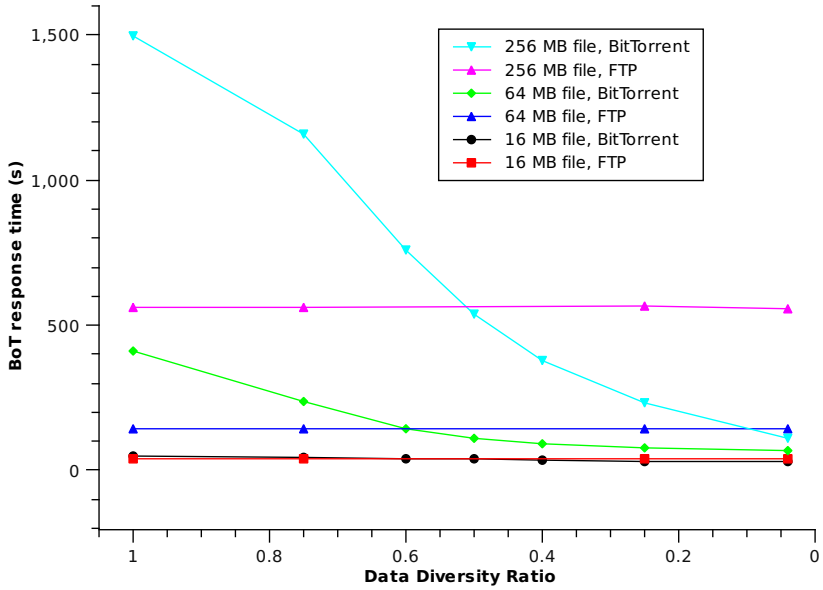


Figure 5.10: Varying data protocol and data configuration.

5.5.4 Varying Data Protocol and Data Configuration

The impact of file size and DDR on BitTorrent and FTP data transfers is now evaluated. One BoT of 24 Tasks, i.e. exactly one Task per Resource, is submitted so that caching and Resource selection play no role. Resource selection is done with Temporal Tasks Grouping. Grid topology I (see Figure 5.7) is used.

Results presented in Figure 5.10 and Table 5.3 show that BitTorrent data transfers lead to excellent performance in most configurations. They also show that there is a threshold in the range of DDR values beyond which BitTorrent data transfers become more interesting than FTP data transfers (independently of any extra gain from an increased cache size). In this experiment, this threshold is expected to be 0.5 and is indeed close to 0.5. A DDR of 0.5 means that there are 12 independent input data files, with exactly 2 Tasks depending on each of them. A high DDR leads to mediocre performance with BitTorrent data transfers, as there is no opportunity to exploit orthogonal bandwidth. As expected, the results also show that the DDR has no impact on FTP data transfers. If the data transfer protocol selection algorithm were activated, BitTorrent would be selected for a given file when its number of copies would be two, which corresponds here to a DDR of 0.5. This experiment demonstrates that this algorithm, even if simple, gives acceptable results. This does not preclude, however, the need of further study of its performance in a broader range of networks.

Fixed parameters	Varying parameters
• 1 BoT of 24 Tasks • Resource selection = data-aware • Task selection = TTG • cache size (# files) = 1	• file size = {16 MB | 64 MB | 256 MB} • DDR = { 0.04 | 0.25 | 0.40 | 0.50 | 0.60 | 0.75 | 1.0 } • data transfer = { BitTorrent | FTP }

file size = 16 MB, data transfer = BitTorrent

DDR	1.0	0.75	0.60	0.50	0.40	0.25	0.04
MBRT (s)	49	43	40	38	35	31	30

file size = 16 MB, data transfer = FTP

DDR	1.0	0.75	0.25	0.04
MBRT (s)	38	38	38	38

file size = 64 MB, data transfer = BitTorrent

DDR	1.0	0.75	0.60	0.50	0.40	0.25	0.04
MBRT (s)	413	235	144	108	93	75	66

file size = 64 MB, data transfer = FTP

DDR	1.0	0.75	0.25	0.04
MBRT (s)	145	142	141	141

file size = 256 MB, data transfer = BitTorrent

DDR	1.0	0.75	0.60	0.50	0.40	0.25	0.04
MBRT (s)	1497	1160	760	539	377	232	109

file size = 256 MB, data transfer = FTP

DDR	1.0	0.75	0.25	0.04
MBRT (s)	562	562	565	558

Table 5.3: Varying data protocol and data configuration.

5.5.5 Inter-Task Data Sharing, Varying Task Selection

The impact of Temporal Tasks Grouping is now evaluated. A BoT of 48 Tasks (twice the number of available Resources) is submitted. Grid topology I (see Figure 5.7) is used. Different levels of data sharing are tested, with and without activation of TTG. Two observations can be made about results presented in Figure 5.11 and Table 5.4.

First, all else being equal, BoT response times obtained with TTG-based Task selection are much better, i.e. shorter, except for a very low DDR. In this latter case, caching plays a greater role, leading to BoT response times roughly equivalent between TTG-based and FIFO-based Task selection. The difference between the two policies is that network bandwidth is traded for storage space. If the data transfer protocol selection algorithm were activated, BitTorrent would be selected except with a DDR of 1.0.

Second, caching has a huge impact, even when achieving only very low levels of cache hit ratios. Deploying Resources with cache, even of small size, has thus a strong practical interest due to the combination of Data-Intensive BoT and the use of edge Resources.

5.5.6 Inter-BoT Data Sharing, Varying Resource Selection

A common and widespread scenario is that of a scientist who sequentially submits multiple times the same BoT with high DDR - i.e. there is little or no inter-Task data sharing - **and the same algorithm run by all Tasks**. Only the input parameters or the algorithm are modified between consecutive executions of the BoT after an analysis of computed results, i.e. an experiments session. The input data files of a given Task thus remain the same from one execution of the BoT to another.

An experiment is run for this scenario that is highly relevant in practice. Eight Bags of 24 Tasks are submitted sequentially to the consumer Peer. Grid topology I (see Figure 5.7) is used. Data-aware Resource selection is varying between fully activated, activated for Grid Peers only, activated for Resources only, and completely deactivated.

Results in Figure 5.12 and Table 5.5 show that the cache hit ratio increases with the level of data-awareness in the Resource selection. It is optimal, i.e. 7/8 in this experiment, when both Grid-Peer-level and Resource-level data-aware Resource selection algorithms are activated. BoT response times also decrease sharply when the cache hit ratio increases.

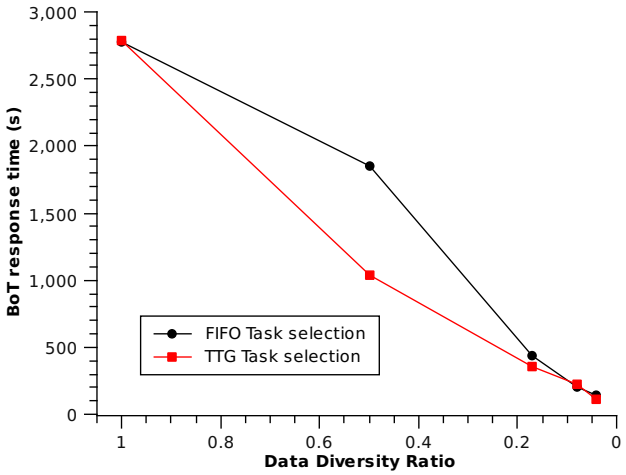


Figure 5.11: Inter-Task data sharing, varying Task selection.

Fixed parameters	Varying parameters
• 1 BoT of 48 Tasks • file size = 256 MB • data transfer = BitTorrent • Resource selection = data-aware • cache size (# files) = 6	• DDR = {0.04 | 0.08 | 0.17 | 0.50 | 1.0} • Task selection = { TTG | FIFO }

Task selection = FIFO

DDR	1.0	0.50	0.17	0.08	0.04
cache hit ratio	0.0	0.02	0.07	0.33	0.61
MBRT (s)	2775	1854	442	201	142

Task selection = TTG

DDR	1.0	0.50	0.17	0.08	0.04
cache hit ratio	0.0	0.0	0.0	0.04	0.38
MBRT (s)	2783	1039	361	223	113

Table 5.4: Inter-Task data sharing, varying Task selection.

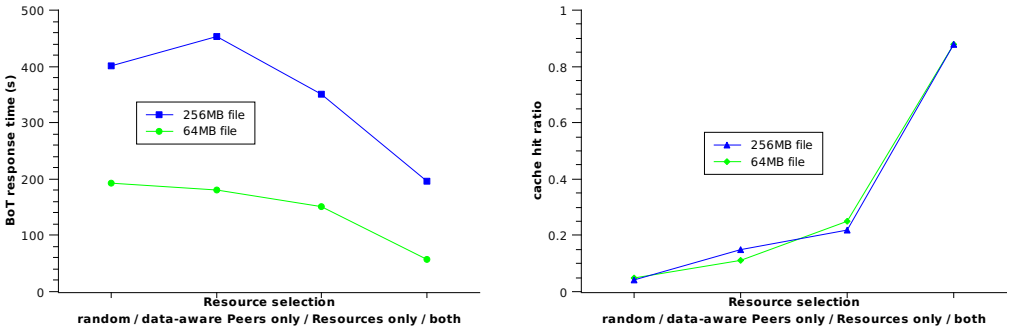


Figure 5.12: Inter-BoT data sharing, varying Resource selection.

Fixed parameters	Varying parameters
• 8 BoT of 24 Tasks each • DDR = 1.0 • data transfer = BitTorrent • Task selection = TTG • cache size (# files) = 1	• file size = { 64 MB | 256 MB } • Resource selection = { random | data-aware for Grid Peers only | data-aware for Resources only | data-aware }

file size = 64 MB

Resource selection	random	data-aware (Grid Peers only)	data-aware (Resources only)	data-aware
cache hit ratio	0.05	0.11	0.25	0.88
MBRT (s)	192	181	151	56.5

file size = 256 MB

Resource selection	random	data-aware (Grid Peers only)	data-aware (Resources only)	data-aware
cache hit ratio	0.04	0.15	0.22	0.88
MBRT (s)	401	453	350	196

Table 5.5: Inter-BoT data sharing, varying Resource selection.

Fixed parameters	Varying parameters
• file size = 256 MB • data transfer = BitTorrent • Resource selection = data-aware • Task selection = TTG	• { 1×12 Tasks (0 cached file) | 1×6 then 1×12 Tasks (6 cached files) | 1×12 then 1×12 Tasks (12 cached files) } • DDR = minimal for each BoT • cache size (# files) = { 0 | 1 }

Tasks in preparatory BoT Tasks in studied BoT	0 12	6 18	12 24
# cached files (*) # files to download (*)	0 0	6 6	12 12
cache size (# files)	1	1	1
cache hit ratio (*)	0.0	0.33	0.50
MBRT (s) (*)	107	80	72

(*) for the studied BoT only

Table 5.6: BitTorrent Performance with Multiple Seeders.

5.5.7 Scalability of the Data Transfer Architecture

The impact of the availability of multiple BitTorrent seeders due to the presence of cached copies of a given input data files is now evaluated. Grid topology I (see Figure 5.7) is used. Bags of Tasks with a minimal DDR - i.e. input data files are all identical - are submitted so that there are always 12 copies of a common, identical input data file that must be downloaded concurrently.

A Bag of, successively, 12, 18 and 24 Tasks is studied three times. All Tasks depend on a common, identical input data file. The BoT response time and cache hit ratio are observed in 3 separate setups. Each setup corresponds to a varying number of cached copies of the common input data file. In the first setup, there exists no cached copy of the file required by the studied Bag of 12 Tasks. In the second setup, a "preparatory" BoT of 6 Tasks is first submitted and completed, so that there exists 6 cached copies of the file when the studied Bag of 18 Tasks is submitted. In the third setup, a "preparatory" BoT of 12 Tasks is first submitted and completed, so that there exists 12 cached copies of the file when the studied Bag of 24 Tasks is submitted.

In all three setups, there are exactly 12 file downloads (of the same file, using BitTorrent) that occur concurrently. The difference between setups resides in the number of supplementary sources of sharing available, due to the number of cached copies of the file. Results in Table 5.6 show that indeed the number of supple-

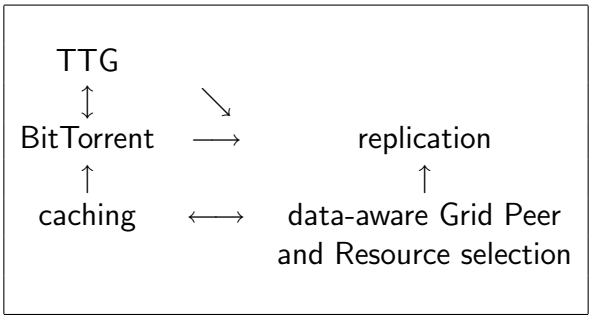


Figure 5.13: Algorithms and protocols dependencies.

mentary sources of sharing improves the performance of the 12 downloads. This demonstrates the scalability of our proposed data transfer architecture, that is designed to distribute the load of data transfers across the P2P Grid, i.e. by having Resources automatically share all downloaded files with BitTorrent.

5.6 Discussion

5.6.1 A Network of Technologies

We have proposed to combine several algorithms and protocols to perform efficient data transfers and data-aware Task scheduling of Data-Intensive BoT in P2P Grids, namely BitTorrent P2P file sharing protocol, data caching, data-aware Resource/supplier Peer selection, Task selection based on Temporal Tasks Grouping, data replication. Figure 5.13 summarizes the dependencies that exist within our proposed network of algorithms and protocols.

Impact of BitTorrent

BitTorrent must be activated for Temporal Tasks Grouping to have any favorable impact on transfer times of identical input data files. Reciprocally, excellent performance of BitTorrent data transfers is achieved only with a Task selection algorithm that temporally groups Tasks with identical input data files.

BitTorrent should not be activated to transfer small files or files of a BoT where the number of identical files is limited. In these setups, FTP overhead is lower.

If no BitTorrent support is available, TTG is not useful and the only gains of performance would come from caching and data reuse.

BitTorrent is necessary to control the cost of our proposed push-based, proactive, asynchronous data replication mechanism. Therefore, in the absence of BitTorrent support, no such replication should take place.

Impact of Temporal Tasks Grouping

The impact of Task selection with the Temporal Tasks Grouping algorithm is to temporally group the scheduling of Tasks sharing the same input data files so as to maximize the number of simultaneous BitTorrent nodes downloading these files, thereby dramatically decreasing the mean file transfer times.

TTG can be activated at all times. On one hand, TTG can greatly decrease the download times achieved with BitTorrent. On the other hand, TTG does not decrease the download times achieved with FTP.

Impact of Caching

Caching of data on Resources naturally brings important benefits as it can prevent unnecessary data transfers. A larger cache capacity brings even greater benefits as it increases the probability of cache hits.

On the downside, caching has a cost in terms of storage space, which might be limited on Resources of a P2P Grid, which are not necessarily high end. However, it is important to highlight that even limited caching support already brings considerable benefits.

When Temporal Tasks Grouping is not activated or has a limited effect, i.e. when processing a BoT exhibiting a high DDR or in the absence of BitTorrent support, caching is the only source of performance gain.

Even in the presence of strong caching support, Temporal Tasks Grouping is very important in case of inter-Task data sharing within a BoT. It is the only means for efficient data transfers of files that are introduced in the Grid for the first time.

It should also be noted that the proposed management of data caches is fully dynamic, and thus more adapted to P2P Grids than existing data cache management policies that are statically precomputed [254].

Impact of the Cache Replacement Policy

Using a pure LRU cache replacement policy is sub-optimal. Better performance would probably be achieved by taking into account the size of the cached files.

For example, a P2P Grid of 24 Resources, each having a cache size of 2 files, can be considered. Those Resources can use two different cache replacement policies: pure LRU (the least recently used file in the cache is ejected first) or weight-based (the smallest file in the cache is ejected first). Task selection is based on Temporal Tasks Grouping and Resource selection is data-aware.

3 BoTs of 24 Tasks (θ_1, θ_2, θ_3) are submitted sequentially. All Tasks of a given BoT depend on an identical file (Δ_1^0, Δ_2^0, Δ_3^0 respectively): The DDR of each BoT is thus $\frac{1}{24}$. As an example, one can assume that Δ_1^0 weighs 200 MB; Δ_2^0 and Δ_3^0 both weigh 1 MB.

With Temporal Tasks Grouping, all Resources hold both Δ_1^0 and Δ_2^0 in their cache after the completion of θ_2. When θ_3 is submitted, depending on the cache replacement policy (LRU or weight-based), either Δ_1^0 (the oldest) or Δ_2^0 (the smallest) is ejected from the caches and replaced by Δ_3^0. At this point, cache hit ratios are all equal to zero (for both policies) since a different file was required for every Task.

If one hypothesizes that θ_1 is then resubmitted, Δ_1^0 has to be downloaded again (which is expensive since it is a 200 MB file) or is loaded from the caches (in this case, yielding a cache hit ratio of 0.25), depending on the cache replacement policy (LRU or weight-based). With LRU, the BoT response time is larger and the perceived performance is lower.

It can be argued that a purely weight-based replacement policy is not optimal either. A hybrid cache replacement policy, maybe also involving additional factors, could lead to even better performance.

Performance of BitTorrent Data Transfers

Temporal Tasks Grouping enables the excellent performance of BitTorrent data transfers in situations of inter-Task data sharing, i.e. when several Tasks of a given BoT share input data files.

Caching support also increases performance of BitTorrent transfers in situations of inter-BoT data sharing, i.e. when Tasks of multiple, successive BoTs share input data files. Let's suppose that a given input data file is cached on several Resources but all these Resources are already running a Task: A new transfer of this file is triggered by the available Resource when the Task that depends on it is scheduled.

In this setup, a BitTorrent data transfer of this file benefits from existing replicas of the file that exist on busy Resources, each of which constitutes a supplementary source of data sharing. Therefore, caching support can be viewed as an indirect way to increase the number of BitTorrent nodes sharing input data files. This is very useful as it supports opportunistic scheduling of Tasks on extra supplementary Resources as they become available, which is important in the unstable Resource environment of a P2P Grid.

To sum up, the effect of Temporal Tasks Grouping on performance of BitTorrent transfers is immediate, while the effect of caching support is temporally delayed.

Impact of Data-Aware Resource/supplier Peer Selection

Data reuse consists in taking data placement into account when selecting a Resource and, if applicable, a supplier Peer to schedule a given Task. The impact of data-aware Resource selection by a given Grid Peer with an algorithm similar to the Storage Affinity [267] algorithm (without Task replication) is to ensure that, when scheduling a given Local or Supplying Task, the Grid Peer selects one of the Resources that stores the maximum number of input data files required by this Task.

Similarly, the impact of data-aware supplier Peer selection by a given consumer Peer is to ensure that, when scheduling a given Consumption Task, the consumer Peer selects one of the supplier Peers that has probably at least one Resource storing the maximum number of input data files required by this Task.

Deactivating data-aware Resource selection or supplier Peer selection severely degrades BoT response times and cache hit ratio when scheduling BoTs with a low DDR, i.e. a high number of identical input data files. Reciprocally, data-aware Resource selection and supplier Peer selection are useless without caching support.

Finally, even for a BoT exhibiting a high DDR, i.e. there is no inter-Task data sharing, data reuse has a strong practical interest in the case of inter-BoT data sharing. **A common and widespread scenario is that of a scientist who runs over and over successive versions[9] of a given algorithm applied to the same set of input data files from one BoT to another** (the files of the set are not necessarily identical, i.e. the DDR may be high). This Grid application, which is the same for all Tasks of one submitted BoT, is modified between executions of BoTs, following a phase of debugging or analysis of computed results. For a given

[9] A *parameter sweep* is a BoT that exhibits inter-Task data sharing and with a low DDR. The kind of BoT discussed here, with different code but data sharing between BoTs and arbitrary DDR, might be named *application sweep*.

execution of the BoT, input data files of different Tasks are very different. For multiple executions of the BoT, input data files of a given Task are always identical.

5.6.2 Impact of Computations

The experiments presented in this Chapter evaluate the behavior of the complete P2P Grid middleware fully deployed on real computers to run a real computation. The application that is run (see Section 5.5.2), is designed to exhibit a very high communication-to-computation ratio (or low computation-to-communication ratio). If the communication-to-computation ratio were low, there would be little interest to reduce the impact of data transfers, as these would intrinsically have little impact. Thus, our proposed data transfer architecture has no impact and little interest when there are few data transfers. When there are massive amounts of data transfers, however, our proposed data transfer architecture can reduce their impact.

5.6.3 Recommended Deployment Options

Maximization of data reuse is enabled by combining caching support and data-aware Resource and supplier Peer selection. Caching support should be configured to use as much storage space as possible on each Resource.

Temporal Tasks Grouping and BitTorrent support enable to maximize parallelism of Task execution. BitTorrent support should be made systematically available, and Temporal Tasks Grouping should be systematically activated. It should be emphasized that the deployment of BitTorrent software does not mean that all transfers are performed with BitTorrent, but only that it is available. Grid Peers can be configured to use BitTorrent either systematically, not at all or following an adaptive data transfer protocol selection algorithm.

5.7 Summary of the Contributions

Our research work in the scheduling of Data-Intensive Bags of Tasks in P2P Grids [57, 56] could be summarized as **combining P2P Grid computing and P2P data transfer technologies. Original contributions include a highly scalable and fully distributed data transfer architecture,** an adaptive use of the BitTorrent P2P file sharing protocol, and BitTorrent-aware and data-aware Task scheduling policies. **Our algorithms rely on caching and data reuse to prevent data transfers, and on BitTorrent to remove the cost of redundant simultaneous data downloads.** To the best of our knowledge [19],

our operational Java implementation [55] is the first to deeply integrate P2P file sharing and P2P Grid computing technologies, relying on a standard, off-the-shelf BitTorrent library. This implementation is easily and automatically deployable, in a fully decentralized way.

Experiments were conducted with a real application using a full deployment of the complete middleware. The experimental results that were collected demonstrate that considerable performance gains are achieved for the evaluated setups, even those with very small cache sizes, for both low and high data sharing within and between BoTs, and also in the special, important case of parameter sweeps. Future work essentially resides in improving the estimation of data availability in supplier Peers, in the study of weight-based cache replacement algorithms, and in a more accurate data protocol selection algorithm.

Chapter 6

Non-expected P2P Grid Applications

It is infinitely improbable that all worker nodes of a co-allocated set will be exempt from failure (especially if reset buttons are not deactivated).

- preface, Hitchhiker's Guide to the Grid

The LBG simulator itself is temporally-scalable, but Chapters 3 and 4 required a massive amount of simulations runs for the presented experiments. Distributed simulation, i.e. running instances of the LBG simulator as Tasks on the LBG middleware, has enabled to address this computational deluge. Experiments are performed using the LBG simulator.

Orthogonally, a tightly-coupled computational fluid dynamics application (structured as a so-called Iterative Stencil application) was considered. Two challenges were addressed to run this application on the LBG middleware. Firstly, one benefit of the LBG middleware is also a drawback to run Iterative Stencil. Resources from other Peers can be dynamically acquired to scale out an application. However, the Task schedule is not known in advance, which makes impossible to load balance the application before it is deployed (which is important as the slowest Task of an Iterative Stencil determines the speed of the whole application). As the considered application performs dynamic benchmarks only at deployment time, locality-awareness of load balancing was implicitly addressed. Secondly, the frequent preemption of external computational requests leads to the constant restart of the whole Iterative Stencil from the beginning (if the execution of one Task fails, the whole application fails). It was thus not expected to be able to ever run an Iterative Stencil on a P2P Grid. Nonetheless, a distributed checkpointing mechanism successfully addresses this huge issue thanks to its P2P-awareness: The checkpoints are stored in a decentralized fashion following a topology that takes into account the possibility of bursts of preemption. Experiments are performed using the LBG middleware.

This chapter covers applications that are not expected (see Section 2.4) in P2P Grids. In Section 6.1, useful, advanced deployment options of the Lightweight Bartering Grid middleware are discussed. In Section 6.2, the robust execution of (heavily-communicating) Iterative Stencil applications is investigated.

6.1 Advanced Deployment Options

In distributed environments, the deployment of software can become challenging [202]. In this section, several useful deployments options of the LBG middleware are discussed. These are side effects - or byproducts - of the *code once, deploy twice* software engineering pattern introduced in Chapter 3. Section 6.1.1 discusses distributed simulation. Section 6.1.2 discusses self-bootstrapping of P2P Grid software. Section 6.1.3 discusses distributed Peer deployment in a P2P Grid.

6.1.1 Distributed Simulation

A large number of simulation runs[1] are required to use the LBG simulator as a software engineering tool as proposed in Chapter 3. We propose a distributed testing process based on policy enumeration (fixed workload, varying bartering policies). We also propose a distributed evaluation process based on scenario randomization and results averaging to smooth out outlier simulation results, (varying workload, fixed bartering policies). In both processes, a Grid Task can be defined for each intended simulation run to be completed. In the following, we refer to such a task as a *SimTask* and also call *BoS* a *Bag of SimTasks*.

Policy Enumerator

Testing and evaluating large-scale distributed software can be very challenging [52]. Testing every valid combination of bartering policies under typical scenarios before major releases of the middleware is useful. It helps ensure that recent changes, e.g. in the bartering code, have not broken the middleware for well-known typical Grid nodes configurations. The number of test cases may be very large, but as they are independent they can be run as SimTasks.

A related example of distributed testing is to certify the installation process of packages in the Debian [109] GNU/Linux distribution. Recompiling and simulating the installation of every software package in Debian can be time consuming. This is a

[1] A *simulation run* refers to one execution of the LBG simulator with a given simulation description file. In other words, it is one simulation of a whole P2P Grid in a fully controlled environment.

typical BoT application for which P2P Grids were designed: It is run infrequently and it exhibits large peaks computational requirements. Structuring this application as a BoT and running it on a Grid [227] has *"help[ed] significantly the search for bugs"*, with a linear speed-up in BoT response time.

Another related example is the Skoll [248] continuous distributed testing project. It proposes techniques to handle extremely large sets of combinations of policies. It relies on distributed testing, but adaptively selects which tests to actually run. A limited number of test cases are initially selected. Some of them complete their execution, others fail in the sense that they exhibit unexpected behaviors, such as exceptions, run-times errors and assertion failures. Such failures are due either to programming mistakes, i.e. bugs, or to intentional checks manually coded by the Grid application developer, i.e. to try and detect inconsistencies in data structures or to enforce desirable properties such as system liveness.

Each test case that fails leads to the testing of neighbor cases. This tends to minimize the number of test cases to run, while maximizing the coverage of the test case space. Such test case selection techniques can certainly be added to the distributed testing process that we propose for LBG. It is important to take note that distributed testing is a form of testing that is complementary with - and does not replace - other quality assurance mechanisms, such as definition and enforcement of invariants, model checking and unit testing.

Scenario Randomizer

The LBG simulator always produces the same output for a given scenario (see Section 3.5.1 and Appendix B.2). As a random seed is defined for each simulation run (see Section 3.5.5), random variations can be added to the Grid environment, including synthetic workloads and in particular the inter-arrival times of submitted BoTs. Some values of the random seed may lead to "limit cases" of input values that can bias the performance of bartering policies.

To address this issue, we propose to run a large number of instances of the LBG simulator as a Bag of SimTasks (BoS). The simulator instances are configured with the same bartering policies but, to a certain extent, with varying[2] workloads. Simulation results accumulated in successive simulation runs (such as peers utilization and mean BoT response times) are averaged, reducing the influence of the "limit cases" and hopefully smoothing out outlier results.

[2]The values of the random seeds are random integer numbers. The values of the job inter-arrival times are drawn from a random variable within a configured interval.

Structuring the LBG Simulator as a Bag of SimTasks

Structuring multiple simulation runs as a Bag of SimTasks is actually straightforward. A simulation description file is given to each SimTask as an input data file. Each SimTask prints simulation statistics to the console. As the standard output stream of a Grid application is automatically returned to the User Agent as a log file, a SimTask does not need to expose its results as a Grid application (see Appendix A). Log files are informally structured but easily parsable; it is straightforward to average multiple log files. Figure 6.1 illustrates a Bag of 3 SimTasks run on a P2P Grid, each of them simulating a whole P2P Grid.

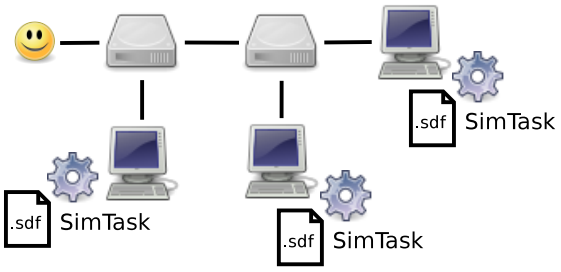


Figure 6.1: Bag of SimTasks run on a P2P Grid.

Support for the structuring of the LBG simulator as a BoS requires to:

- wrap the simulator with a small Java class which implements the Grid application interface (see Appendix A.1), in order to obtain a SimTask (for both the scenario randomizer and policy enumerator);
- generate a set of simulation description files (each assigned to a separate SimTask) from a given simulation description file, each with either a different master random seed (for the scenario randomizer) or a different set of bartering policies (for the policy enumerator);
- average multiple log files (one log contains the statistics of one SimTask) generated by the execution of a BoS (for the scenario randomizer only).

6.1.2 Self-Bootstrapping

Self-bootstrapping is the pattern where a current, stable version of a given system is used to develop the next version of this system. This pattern is common for compilers.

The proposed structuring of multiple simulation runs as a Bag of SimTasks has been inspired by the self-bootstrapping pattern. A P2P Grid is deployed with nodes running the LBG middleware configured with basic bartering policies. New bartering policies can be tested, debugged and evaluated through LBG simulator runs completed by this LBG-based P2P Grid. Self-bootstrapping thus enables our research work to contribute to its own evolution.

A limited form of self-bootstrapping has been used in the development [266, 265] of the OurGrid [233, 84, 286] middleware. Advanced accounting algorithms have been simulated by a special-purpose discrete-event system simulator partially based on code from the OurGrid middleware. This simulator has been run as Tasks of Bags of OurGrid SimTasks on a P2P Grid where a version of the OurGrid middleware equipped with basic accounting algorithms was deployed.

6.1.3 Distributed Peer Deployment

The deployment of a Peer at a given site is currently performed by the human administrators of this site. They select one of the computers at the site to manage the other computers. Even if bartering is fully decentralized at the P2P-Grid-level, this introduces a centralized point of failure at the site-level.

One solution would be to introduce self-organization at site-level, for example with a leader election protocol [195]. If at some point a Resource is unable to establish contact with its owner Peer, it can interrogate other Resources at the same site. Indeed, in case of Peer failure, multiple Resources will be unable to communicate with it. Using this information can enable to detect Peer failure.

Upon detection of Peer failure, Resources can select one of them to become the new Peer. This would require some form of directory support at the site-level, so that User Agents can transparently locate an entry point to the Grid. The delay before the activation of the leader election protocol should be long enough to avoid misinterpreting temporary communications failure as Peer failure.

With the proposed leader election protocol, the elected Resource has to deploy the Peer middleware on the computer where it is running. As the LBG middleware and simulator are packaged into the same jar file, the Peer middleware code can be loaded by any Resource. The elected Resource could thus instruct itself to run the Peer middleware as a (never-ending) Task. In this case, the selected Resource acting as a Peer would not (and, indeed, could not) continue acting as a Resource.

As a benefit, the Peer can be deployed 100% at the middleware-level, not needing support from the underlying O.S.. This contrasts with classic Grid deployment

tools [156], which are far from lightweight and represent a centralized point of failure at the site-level.

6.1.4 Experimental Results

Evaluation of Policy Enumerator

An example of our proposed policy enumerator process is now given. The introduction of the adaptive preemption policy (see Section 4.7.2) combined with some opportunistic general purpose refactoring was done over the course of one week. It brought the code base from 285 source files (391 classes, 53780 lines) to 292 source files (402 classes, 55621 lines), 73 of which were modified (either created or updated).

Five days after the beginning of this operation, we run a Bag of 2892 SimTasks, each with a different combination of bartering policies, with 20 jobs submitted to each of the 15 simulated Peers. The BoS was completed in 53 minutes on 100 Resources (x86 PC with 512MB RAM): 2859 SimTasks were successfully completed, 40 SimTasks failed due to an exception in the Task control code (see Section 2.9.6), and 3 SimTasks failed due to an exception in the RMS code (see Section 2.7.6).

After investigating the stack traces available in the execution logs of the failed SimTasks, we patched the Task control code (the order in which running Tasks were stored by the Task controller was not systematically maintained). We run a new BoS with the updated code on 101 Resources. It was completed in 50 minutes. All 2892 SimTasks were successfully completed.

Of the 2892 SimTasks, the configuration of 482 involved the newly implemented adaptive preemption policy, yet only 43 failed due to the Task control bug. Without the policy enumerator, this subtle issue would probably not have been resolved before it resurfaced unexpectedly a little while later at a more inconvenient time.

The policy enumerator certifies that the software behaves as intended for a specific set of well-known, typical Grid configurations and workloads. Complementary mechanisms should be used but we believe that such distributed testing is a valuable tool in a software engineer's toolbox.

Evaluation of Scenario Randomizer

To smooth out outlier simulation results, as proposed in Section 6.1.1, the simulator can be run multiple times (about 120 Resources were used, composed of x86 PC with 512MB RAM) with the same simulation description, except for the master random seed which is different each time. These multiple simulation runs

# simulation runs	1	10	20	40	80	160	320
mean simulation time (s)	7	9	10	11	8	9	10
mean BoT response time (s)	767	809	817	832	858	879	860

(a) 4-Peers scenario with bartering

# simulation runs	1	10	20	40	80	160	320
mean simulation time (s)	5	8	8	7	8	9	7
mean BoT response time (s)	1249	1436	1449	1429	1416	1415	1412

(b) 4-Peers scenario without bartering

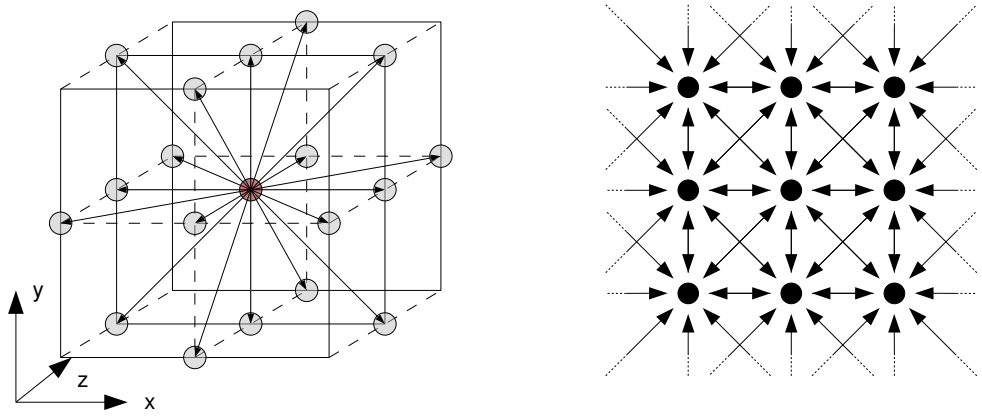
Table 6.1: Simulation runtimes and mean BoT response times achieved with scenario randomizer applied to the 4-Peers scenarios (a) with, (b) without bartering.

can be structured as a Bag of SimTasks that is run by the LBG middleware.

The two 4-Peers scenarios [15] described in Section 3.6.1 are run multiple times. Table 6.1 shows, for varying numbers of simulation runs, the mean runtime of one simulation and the mean BoT response time (MBRT) of all simulated BoTs. The mean runtime of one simulation is expressed in wall clock time while the MBRT is expressed in simulated time. This experiment shows that a single simulation run often leads to results faster than the average of as few as 10 simulation runs. These results hold for scenarios larger than the considered examples. Extensive statistical studies could be conducted to determine the optimal number of simulation runs required to achieve a given level of confidence. Orthogonally, the results for these two scenarios, when compared to one another, also confirm that bartering indeed decreases the MBRT (by $\sim 40\%$ for these specific scenarios).

6.2 LBG-SQUARE

In this section, the deployment and fault-tolerant execution of Iterative Stencil applications on the Lightweight Bartering Grid middleware is investigated. Such a deployment is not straightforward as LBG has been intended to run Bag of Tasks, but was not expected to run this type of Grid application. An application-level checkpointing mechanism providing fault-tolerance support for Iterative Stencils run on P2P Grids is also introduced. The research presented in this section results from cooperative work with our colleague Gérard Dethier, who has introduced LaBoGrid, a generic computational fluid dynamics software (Lattice-Boltzmann method) structured as an Iterative Stencil application.



(a) (b)

Figure 6.2: Lattices: (a) 3D, 19 sites (focus on central site) and (b) 2D, 9 sites.

6.2.1 Iterative Stencil Applications

Iterative Stencil applications have been presented in Section 2.4 and are illustrated in Figure 2.10. These are applications that can be structured as a set of periodically recomputed interdependent computational Tasks involving heavy and/or frequent communication between subsets of them. *"A stencil [application] updates every point in a regular grid with a weighted subset of [nearby points]"*, as if a stencil were figuratively applied [187]. Each Task is connected to a subset of the Iterative Stencil's Tasks, also called its neighbors.

Tasks of an Iterative Stencil are thus tightly interdependent: If any one Task fails, the whole application fails. Iterative Stencils are usually run in stable environments, like parallel computers or clusters connected with high-speed networks, but not in P2P Grids.

LaBoGrid

An example of Iterative Stencil application is Lattice-Boltzmann (LB) simulation. *"Computational fluid dynamics (CFD) is one of the branches of fluid mechanics that uses numerical methods and algorithms to solve and analyze problems that involve fluid flows"* [89]. Lattice-Boltzmann simulation methods constitute a family of CFD methods that can deal with complex models and are easily parallelizable. Figure 6.2[3] represents a typical 3D lattice, as well as one of its projection into a 2D lattice.

[3]Courtesy of Gérard Dethier [114].

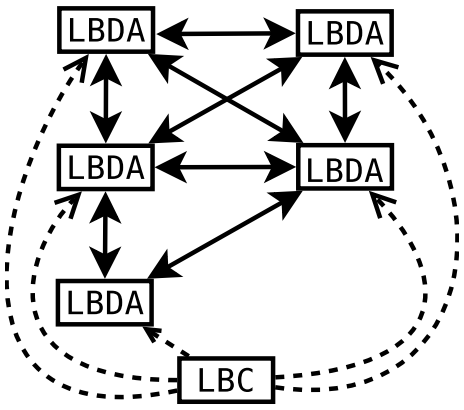


Figure 6.3: LaBoGrid architecture.

The application that motivates the research presented in this section was brought by the Chemical Engineering lab of our alma mater. The motivating application consists of the LB simulation, based on the Lattice-Boltzmann method, of flows in a metallic foam [216]. The goal of the LB simulation is to improve the *"design, efficiency and selectivity of related chemical engineering processes."*

To address the requirements of the motivating application, LaBoGrid [114, 115, 116] (Lattice-Boltzmann Grid), has been introduced. It is a generic software to run Lattice-Boltzmann flow simulations on a computational Grid.

LaBoGrid is based on two components (see Figure 6.3): Lattice-Boltzmann Distributed Agent (LBDA) and Lattice-Boltzmann Controller (LBC). Each LBDA is a Task of the Iterative Stencil application that is run on the Grid. The LBC is the out-of-Grid component that controls the LB simulation according to user-defined parameters.

Basically, the LBC takes the description of a lattice as input parameter, transforms it into a data model, slices the obtained model data into data blocks and finally distributes these data blocks to the LBDAs for processing.

The structure of the metallic foam of interest is obtained by micro tomography as a large matrix of voxels which constitute the flow boundaries. This data is then modelled as a typical D3Q19 lattice which can be given as input to LaBoGrid. As illustrated on Figure 6.2b, periodic boundary conditions are assumed on faces parallel to the flow direction.

Motivation for Running Iterative Stencils on a Grid

Continuous advances in networking infrastructure and the increasing availability of numerous, inexpensive (but not necessarily fast or reliable) computers at the edge of the Internet are motivating the computing of Iterative Stencil applications on P2P Grids (instead of dedicated, high performance clusters).

P2P Grids can automate the large-scale co-allocation of Resources across administrative domains, without any of the administrative burdens typically associated with the deployment of large-scale systems, and at practically no cost beyond the regular operating costs of one's own Resources. This enables smaller organizations to process larger data sets than was previously possible, at a fraction of the cost.

Challenges in Running Iterative Stencils on a P2P Grid

Of course, these stated benefits come at a price. Even if the peak performance of a P2P Grid may be comparable to that of a cluster of equivalent computational power, the mean Iterative Stencil response times will probably be higher on a P2P Grid. Obviously, relying on the global Internet imposes performance penalties on data transfers. Additionally, the unreliable nature of Task execution in P2P Grids introduces challenges of its own.

Deploying an Iterative Stencil, independently of the execution substrate (e.g. cluster or P2P Grid), is intrinsically challenging in presence of high heterogeneity in computational and network performance. As each LBDA is synchronized with several other LBDAs, the performance of the slowest Resource or network link indeed determines the performance of the whole system. Efficient load balancing is thus very important for the overall response times. Additionally, a specific issue with P2P Grids is that the configuration information required for efficient load balancing is not available at submission time, when deploying the LBDAs. Indeed, given the scheduling model in LBG (see Section 2.9.4), the schedule of Tasks is not known in advance and it is very difficult to predict to which Resource of which Peer a given LBDA is going to be scheduled.

6.2.2 Deploying an Iterative Stencil on a P2P Grid

The deployment of LaBoGrid on the LBG P2P Grid middleware is now examined. The following discussion certainly can be applied to other Iterative Stencils and P2P Grid middlewares as well.

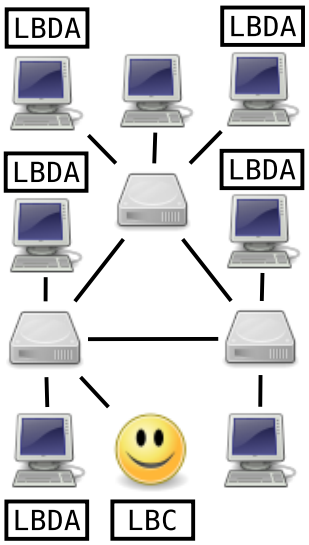


Figure 6.4: Organization of LBG-SQUARE.

Structuring a set of LBDAs as a Bag of (Long-Running) Tasks

Resources are needed to run the LBDAs. They can be provided by structuring a set of LBDAs as a Bag of Tasks submitted to the Lightweight Bartering Grid by the LBC, that is running an embedded version of a User Agent. We call LBG-SQUARE the resulting[4] software system. Figure 6.4 provides an illustration of a typical LBG-SQUARE deployment, corresponding to the typical LaBoGrid deployment shown on Figure 6.3.

Running one distributed computing middleware as one Task of another distributed computing middleware is a common pattern. However, it introduces issues related to firewall traversal. The LBG middleware should be able to ask the underlying O.S.-level and/or network-level security infrastructure to temporarily open TCP ports that can be used by Grid applications. This issue is common to all Grid middlewares, and is further discussed in Appendix F.7.2.

Another issue is that there is no support for communications between Tasks in the Lightweight Bartering Grid. But, as communications are needed between pairs of running Tasks (as opposed to communications between a completed Task and a Task about to be started - which is the case of Workflows), LBDAs can directly communicate with one another. To enable this, the LBC communicates to each LBDA the IP address and TCP port of its neighbors along with the initial data.

[4] Lattice-Boltzmann Grid × Lightweight Bartering Grid software = LBG^2.

Resource Co-Allocation

To maximize parallelization of the processing of data blocks of the simulated lattice, a large number of computers are required. Co-allocation [97, 256, 125] consists in ensuring simultaneous access to multiple computers for a certain duration.

Drost et al. introduced locality-aware co-allocation [125] for Workflow applications in 1-level P2P Grids, i.e. where each Peer manages only one Resource (itself). We target Iterative Stencils rather than Workflows. Their proposed locality-aware co-allocation mechanism is flooding/gossiping, while ours is based on the combination of P2P Grid middleware-level bartering [59, 84] and application-level benchmarking. Our work is more general in the sense that 2-levels P2P Grids are considered. Importantly, we also introduce an application-level fault-tolerance mechanism.

It can be remarked that co-allocation is the opposite of converting space-shared Resources into intermittent Resources [94, 106]. From this perspective, co-allocation can be seen as converting intermittent Resources into virtual space-shared Resources.

Implicit support for co-allocation is available in the Lightweight Bartering Grid: A Peer is designed to compute a BoT as fast as possible, requesting access to Resources of other Peers as needed until the BoT is completed.

Deployment

Once the LBC (Lattice-Boltzmann Controller) is up and running, deploying LaBoGrid essentially consists in co-allocating Resources to deploy all LBDAs (Lattice-Boltzmann Distributed Agents). Once the LBDAs are deployed (i.e. at runtime) and have downloaded initial data from the LBC, LBDAs communicate together and exchange data between each iteration of the LB simulation. Upon completion of the planned number of iterations, all LBDAs upload results to the LBC, which stores them into a database.

Balancing the computational load between LBDAs consists in balancing the repartition of data blocks between them in function of the relative performance of Resources. Our colleague Gérard Dethier recently proposed an algorithm [114] to compute a performance model at runtime. It is based on the dynamic benchmarking of Resources where LBDAs have been deployed.
After the submission of a BoT, once LBDAs have been deployed, they first contact the LBC to signal their availability. The LBC can then systematically benchmark the newly obtained Resources as they become available. Upon completion of these performance benchmarks, the LBC can balance the load without awareness of the execution substrate, and immediately start distributing the initial data blocks to the LBDAs. **The proposed algorithm based on dynamic benchmarking**

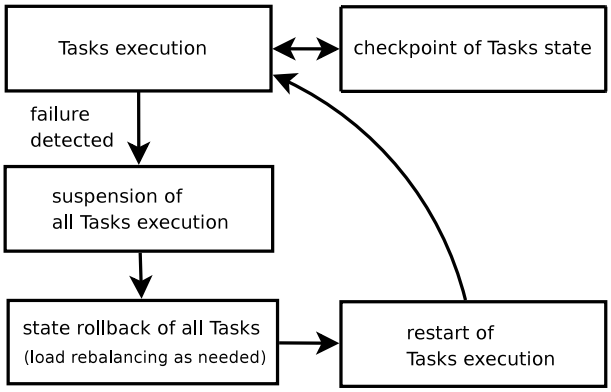


Figure 6.5: Checkpoint/restart mechanism.

thus brings locality-awareness to the co-allocation mechanism provided by the P2P Grid middleware, in a totally transparent way.

As the LBC can continuously request the deployment of additional LBDAs, it also benchmarks additional Resources that become available during the execution of the LB simulation. This enables to rebalance the load if sending LBDAs to these new Resources is predicted to improve LaBoGrid performance. However, load rebalancing should not be done too often because of the high cost of transferring data to all LBDAs.

6.2.3 Fault-Tolerance for Iterative Stencils

Task execution failure does not only delay the completion of an Iterative Stencil (just like a Bag of Tasks), it also suspends the execution of all Tasks (unlike a Bag of Tasks). A common way for Iterative Stencils to deal with Task execution failure is the checkpoint/restart mechanism [115, 116, 35, 132, 133], illustrated in Figure 6.5.
A few generic (i.e. application-independent) fault-tolerance mechanisms for Iterative Stencil applications have been proposed [115, 116, 35, 133, 132]. However, to the best of our knowledge, the study of robustness for execution of Iterative Stencils in P2P Grids has been first explored in our recent work [115, 116].

Checkpointing

The state of every Task is periodically checkpointed, i.e. replicated and stored (checkpoint events), possibly multiple times.

Checkpointing can be controlled at multiple levels: It could be application-level checkpointing, or transparent (i.e. middleware-level or O.S.-level) checkpointing. Application-

level checkpointing must be implemented by Grid application (e.g. LaBoGrid) developers, but enables more control and flexibility in the timing, selection and recovery of replicated data [246], which is why we select it.

Early work [71] in middleware-level checkpointing that could be applied to P2P Grids assumes the availability of a (centralized) checkpoint server. However, it does not provide specific details on its organization or implementation.

Fault Recovery

Upon Task execution failure, the execution of the involved Tasks (which often means all Tasks, not only failed Tasks) is suspended. The last stored consistent state of all Tasks is reloaded from one of the existing replicas to an available Resource. Task execution is then restarted (restart events).

Design Parameters

The checkpointing graph describes the number and location of replicas of the state of every Task.

A first design choice, related to the selection of a checkpointing topology, is whether to dedicate some Resources as replica storage servers or have each Resource also act as a replica storage server. Then, various encodings of replicas among replica storage servers have been proposed, such as the parity encoding [246], mirroring or complex codes like Reed-Solomon coding [247]. Yet another design choice is whether to centralize or distribute replica storage [132, 133].

The degree of fault-tolerance, i.e. the maximum number of simultaneous Task execution failures that can be tolerated, depends on the number of replicas of each Task. There is a trade-off similar to that of the Error-Correcting Codes: Higher fault-tolerance has a higher cost in terms of data redundancy.

Orthogonally, checkpointing can be disk-based or diskless. Diskless checkpointing [247] precludes the longer access times associated with disk-based data storage. The drawback is the need for larger amounts of available memory.

Disk-based checkpointing [115, 116] is selected because:

- the size of LaBoGrid checkpoints can be in the order of 1×10^2 MB per LBDA;
- the amount of RAM available on computers at the edge of the Internet would be in the order of 5 to 10×10^2 MB;
- multiple replicas are stored on each Resource.

We propose an application-level disk-based checkpointing mechanism [115, 116], with a P2P checkpointing topology [132, 133], where each Resource acts as both a consumer and supplier of replica storage (which is well in line with the general perspective of this dissertation), and there is no dedicated replica storage server. Resources use local storage, i.e. playpen directories (see Section 2.7.4), to store state replicas. The checkpointing graph is generated by a central controller. In the context of LaBoGrid, though, the LB Controller can communicate state replication decisions to all Tasks at no extra cost, along with the uploading of initial data to the LBDAs.

Given the robustness/cost trade-off expressed above, every Resource stores a limited number of replicas of its state. This provides the potential for a scalable checkpointing architecture.

6.2.4 Generating Management Graphs of LB Simulations

The model graph describes the slicing of the initial data of a LB simulation into data blocks, and their interconnections.

The Resource graph describes the Resources obtained from the P2P Grid that run an LBDA (see Figure 6.6). Its nodes are weighted with performance cost (i.e. estimated speed resulting from benchmarking [114]).

The computation graph [114] assigns data blocks to Resources. It is a partial subgraph [264] of the Resource graph. A mapping of the nodes of the model graph to the nodes of the Resource graph indicates which Resources are going to run LBDAs. Edges of the Resource graph are selected according to the connectivity defined by the model graph. These edges are then added to link the selected Resources. The computed mapping of the nodes of the model graph to the nodes of the Resource graph minimizes processing time. This optimally balances or rebalances the computing times of all Tasks.

The checkpointing graph [115, 116] describes the number and location of replicas of the state of every Task. It is constructed in two steps: first by balancing at the Peer level the number of replicas stored on each Resource, then by uniformly spreading inbound replicas among the Resources of each Peer.

Figure 6.7 illustrates the three types of management graphs of LB simulations, in correspondence with the P2P Grid illustrated in Figure 6.4 and with the Resource graph illustrated in Figure 6.6. The nodes of the model graph are data blocks. This graph is non-directed. The nodes of the computation graph are the Resources provided by the P2P Grid. This graph is also non-directed. The nodes

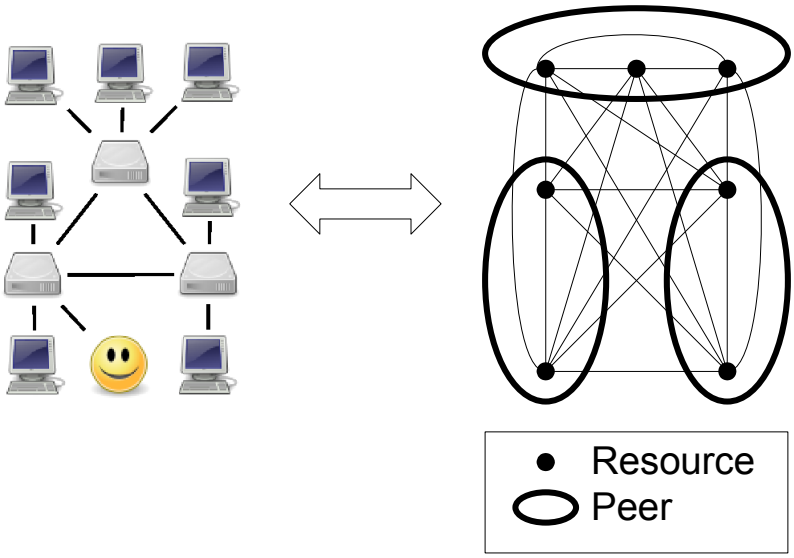


Figure 6.6: Resource graph of a P2P Grid.

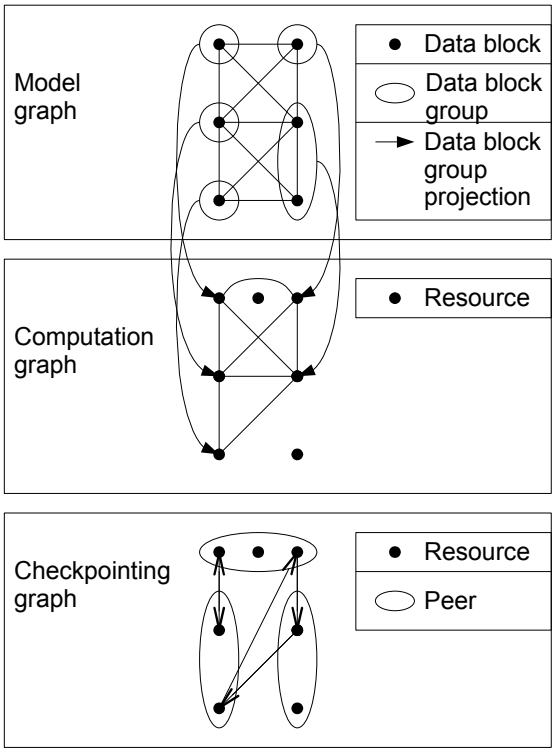


Figure 6.7: Management graphs of LB simulations.

of the checkpointing graph are the same as the computation graph ones. They are grouped by Peer. This graph is directed. The replicas of a Task are on Resources from other Peers.

The Resource graph varies over time because of Task execution failure or deployment of new LBDAs. By definition, the computation and checkpointing graphs must be recomputed.

Task execution failure can arise from crashes of individual Resources, or from the simultaneous preemption of multiple Resources of a supplier Peer which has an urgent need for its own Resources. Therefore, the construction of the checkpointing graph must also ensure that replicas of the state of a Task running on a given Resource of a given supplier Peer are preferably stored on Resources of other supplier Peers. Even if one Peer preempts all Tasks running on its Resources, state replicas are hopefully available on Resources of other Peers.

6.2.5 Implementation of LBG-SQUARE

LBG-SQUARE is the software implicitly resulting from the combination of the Lightweight Bartering Grid and LaBoGrid middlewares. It arises from the structuring of a set of LBDAs as a Bag of Tasks.

Supplementary coding to support LBG-SQUARE was completed in half a day, which is impressive given the complexity of these two separate middlewares. It has essentially involved the embedding of an LBG User Agent into the LBC.

Load Balancing

The essential purpose of the LB Controller is to balance or rebalance the computing load between LBDAs and send them initial data. The LBC starts by generating the model graph, which enables to compute what is the maximum number of Resources that could be used by the LB simulation.

Based on this information, the LBC submits a Bag of Tasks to the Grid, such that each Task is an LBDA. When enough LBDAs have been deployed to Resources and have contacted the LB Controller, the latter generates the Resource graph.

It can then generate the computation and checkpointing graphs. At this point, it uploads the initial data data blocks to each LBDA. It finally waits for results to be uploaded by the LBDA, and subsequently writes them into the results database.

Fault Recovery

In order to provide fault-tolerance, the LB Controller also initiates and coordinates fault recovery operations. Upon detection of Task execution failure, the LB Controller initiates fault recovery (see Section 6.2.3).

The LBC temporizes for a short period after having detected the failure of a Task execution, in order to handle multiple simultaneous Task execution failures. It then submits a new BoT, in order to obtain new Resources.

After a time-out has been exceeded, the computation and checkpointing graphs are recomputed. The LBC then communicates to the newly deployed LBDAs which other LBDAs store a replica of their state so that they can download it. At the same time, the LBC instructs all LBDAs that were not affected by failure to roll back their state to the most recent replica of their own state. When all LBDAs have updated their state, they automatically begin to process data.

The LBC informs each newly deployed LBDA of its neighbors in both computation and checkpointing graphs. A newly deployed LBDA can then download from other LBDAs a replica of the state to assume. LBDAs that survived failure are informed again of their neighbors in both computation and checkpointing graphs when the LBC asks them to roll back to their most recent synchronized state.

LB Controller Flowchart

Figure 6.8 summarizes as a flowchart the operations of an LB Controller. Importantly, the described operations are generic to any Iterative Stencil application; only some implementation details are specific to LaBoGrid. The presented LB Controller could certainly be adapted to other similar applications.

The initial operations of the LBC are intrinsically required because the LBDAs must be launched somehow. After LBDAs have started their computations, the LBC simply waits for results to return.

The LBG-SQUARE architecture is perfectly decentralized under non-faulty conditions. Upon fault detection, the LBC centrally coordinates fault recovery operations. The LBC sends small control messages, which constitute a very low overhead. However, it is not directly involved in the actual transfers of replicas of Task state; the LBDAs perform these transfers. The LBG-SQUARE architecture is thus mostly decentralized under faulty conditions. In any case, the LBG-SQUARE architecture can scale because LBDAs act autonomously and in a P2P fashion.

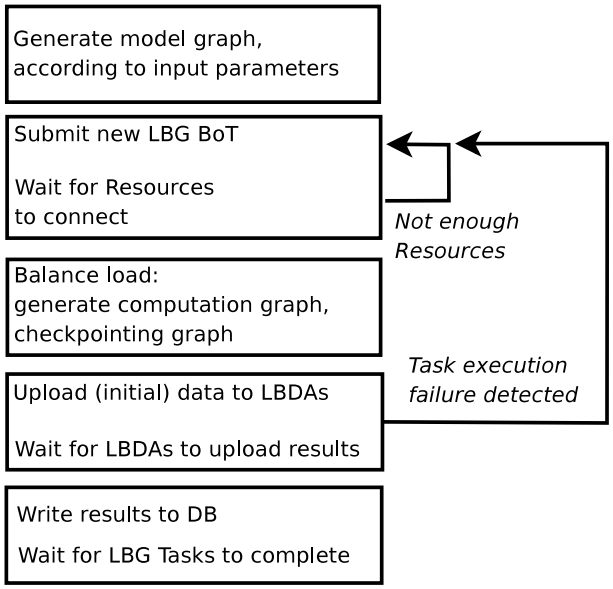


Figure 6.8: LB Controller flowchart.

LBG User Agents

Only very few modifications of the Lightweight Bartering Grid are actually required. The middleware implementation of User Agents has been extended to work as a standalone server, so that the LBC can submit BoTs from another, separate Java VM. This ensures a complete separation of processes between the LBC and the User Agent it relies on.

Beyond this, the Lightweight Bartering Grid and LaBoGrid are independent and can easily cooperate. Load balancing and fault-tolerance are handled by LaBoGrid. Moreover, LBDAs communicate directly with one another, although middleware support could enable cooperation of the LBDAs with the security infrastructure.

6.2.6 Experimental Results

Experimental Setup

The experiments presented in this section have been conducted on 48 x86 PC (Intel P4 CPU with 1GB RAM for Resources, Intel Celeron CPU with 512MB RAM for Peers and the User Agent), all equipped with standard hard drives, and connected with a 100 Mbps switched Ethernet network. Four different Grids are considered (1, 2, 4 and 8 Peers). Each Peer has 5 Resources.

A lattice of 200^3 nodes (referred to as a 200^3-lattice) is considered. The Lattice-Boltzmann simulation (see Section 6.2.1) is run for 100 iterations.

Resources	Deployment time (s)
5	9
10	10
20	20
40	25

Table 6.2: LBDAs deployment times.

The proposed checkpointing mechanism is controlled through its state replication parameters, the replication degree and replication period. The replication degree d (with $d = 0, 1, 2, 3, 4$) means that each Resource saves its state to d other Resources. The replication period p (with $p = 1, 5, 10$) means that a replication is made every p iterations.

Deployment Time

The LaBoGrid deployment time can be separated into: (1) LBDA deployment time, i.e. time between submission of a set of LBDAs to the P2P Grid and their actual deployment on supplied Resources, which depends only on the amount of Resources and (2) LBDA initialization time, which depends only on lattice size given that the limiting factor is the time to upload the initial (fixed-size) data to all LBDAs from the LBC. Table 6.2 shows the measured LBDA deployment time (averaged on 5 runs).

Execution Time without Failure

The execution time of a given LB simulation in a reliable context is affected by the replication degree and period, and the amount of Resources. The total execution time of the LB simulation on an 8-Peers Grid totalling 40 Resources, without state replication, is 113 seconds. Table 6.3 gives the performance penalty coefficient in function of the replication degree and period. This coefficient is obtained by dividing the execution time with state replication by the execution time without state replication.

The performance penalty is linear in function of the replication degree when replicas are checkpointed at every iteration, and less than linear when the replication period increases. The performance gain is roughly linear in function of the replication period, as might be expected.

As observed and as expected, the state replication mechanism heavily increases the execution time. However, without this fault-tolerance mechanism, Iterative

Replication	Period		
Degree	1	5	10
1	15.3	4.0	2.6
2	24.8	5.7	3.6
3	34.8	7.8	4.12
4	44.0	10.1	4.8

Table 6.3: Performance penalty coefficient due to state replication.

Failures	Execution time (s)	Overhead (s)
0	296	0
1	328	32
2	323	27
5	350	54

Table 6.4: Execution time with Failures.

Stencils cannot be run in P2P Grids. Knowing the fault probability and scope (number of failed Task executions), optimal parameters leading to reliable - but not overlong - executions can be found.

Execution Time with Failures

To simulate both types of failures - isolated Resource failure and bursts of preemption (when a supplier reclaims its own computational power for incoming Local Tasks) - a background load generator concurrently submits Local Tasks to a given Peer, so that some LBDAs are preempted. To ensure that the failure in LBDAs execution does not delay the overall runtime of LaBoGrid, each Local Task is a simple *"Hello, Grid"*. Resources from which LBDAs have been preempted are thus unavailable for a moment only, ensuring that LaBoGrid can quickly reacquire all Resources.

The fault-tolerance mechanism is tested with 3 fault types: 1, 2 and 5 Resources from the same Peer are reclaimed once and simultaneously in a given experiment. With 5 Resources reclaimed, the Peer temporarily stops to contribute to the LB simulation. The fault recovery was tested with an LB simulation of a 200^3-lattice with a replication degree 1 and period 10. Table 6.4 gives the execution times compared to the execution times without failure.

The overhead time contains:

- time to detect the failure;
- time to reacquire new Resources and subsequently reconfigure all the available

Resources (the reconfiguration is triggered either by a timer or the end of benchmarking of new Resources);

- time to perform the state rollback itself;
- time to perform the checkpointing according to the new checkpointing graph;
- time to recalculate the iterations that happened[5] after the last checkpointing and before the Task execution failure.

6.2.7 Supplementary Fault-Tolerance Mechanisms

The LB Controller could proactively attempt to acquire supplementary Resources concurrently with waiting for a running LB simulation to complete. To do so, the LBC could regularly submit new BoT in order to deploy additional LBDAs. This could serve two purposes: LBDAs pooling and opportunistic load rebalancing.

LBDAs Pooling

Firstly, the deployed-but-unused LBDAs could be pooled and kept in reserve as spares. Fault recovery of failed Resources would be sped up because of the immediate availability of these spares; precluding the need to acquire new Resources.

Opportunistic Load Rebalancing

Secondly, after several deployed-but-unused LBDAs have become available, the LBC could stop the LB simulation and rebalance the load, so as to opportunistically use the power of the extra Resources. There is a trade-off between the increase in simulation speed and the time lost by the load rebalancing as, clearly, load balancing should be done infrequently. If the processing or network performance of newly available Resources is really low, these should not be used at all in the LB simulation.

To support opportunistic load rebalancing, the load rebalancing mechanism should be enhanced so as to decide if the load should be rebalanced: A threshold must be determined in function of the estimated completion time with or without rebalancing.

[5] This parameter is not controlled in our experiments, which explains variations in the observed time overhead.

6.3 Summary of the Contributions

In this chapter, we have proposed a **distributed simulation mechanism** that is inspired by the self-bootstrapping pattern. It consists of running multiple instances of the LBG simulator implementation, called SimTasks, on the LBG middleware implementation. This enables a **scalable testing and performance evaluation of a large number of bartering policies**.

Another original contribution is the introduction of the **LBG-SQUARE software** resulting from the cooperation of our P2P Grid middleware (Lightweight Bartering Grid) with an **Iterative Stencil application (Lattice-Boltzmann Grid) that is scheduled as a Bag of Tasks**. LBG-SQUARE has been introduced to expand the capabilities of an existing generic software for Lattice-Boltzmann flow simulations on the Grid, that is motivated by a real-world application (study of flows in a metallic foam). LBG-SQUARE has run production workloads, with success.

With a P2P Grid, supplementary Resources can be autonomously and dynamically obtained across organizational boundaries. **Co-allocation is provided through the LBG (middleware-level) scheduling mechanism** and - as it was intended for independent Tasks, not for heavily-communicating Tasks - **it is supplemented with a central (application-level) controller** that performs dynamic Resource benchmarking and dynamically generates load balanced management graphs describing the topology to be assumed by the Iterative Stencil's Tasks.

LBG-SQUARE is designed to withstand frequent failures as a price to pay for running in an undedicated, unreliable environment. **Application-level fault-tolerance is introduced, following a P2P checkpoint/restart pattern.** Tasks autonomously cooperate with one another in a P2P fashion to store replicas of their state into local storage provided by the Resources. **State replication has a strong impact on execution time. But without checkpointing, Iterative Stencils could not be run on P2P Grids. Furthermore, our proposed checkpointing mechanism is adapted to P2P Grids as the checkpointing graph is constructed to be resilient to multiple simultaneous Resource preemptions by any given Peer. The LBG-SQUARE architecture is scalable because the computations and checkpointing are fully distributed.** A centralized organization is assumed only during an initial benchmarking and load balancing phase, upon dynamic load rebalancing and also upon fault recovery.

Chapter 7

Conclusions

P2P Grid computing seeks the convergence of Grid and P2P technologies, as proposed in a paper by Foster and Iamnitchi in 2003 [147]. P2P Grid computing enables independent organizations to barter (i.e. exchange) computing time with one another over the Internet, in order to complete computational requests. Peers are intended to operate autonomously, in a fully decentralized fashion.

To enable organizations to easily share the computational power of their computers and exchange computing time, a P2P Grid middleware is required. Several concerns, notably protection against free-riding, are already addressed by the state-of-the-art P2P Grid middleware, OurGrid [233, 13, 84, 286]. Nonetheless, there remain many concerns to be fully addressed: Data transfers [19] deployment [19], security [19, 52], software engineering/testing [52].

Among these, our dissertation has proposed contributions in data transfers (Chapter 5), testing (Chapter 3), and also deployment (Chapter 6). Furthermore, to investigate new scheduling algorithms and to facilitate further research in scheduling, our dissertation has introduced a new P2P Grid architecture, the Lightweight Bartering Grid (LBG) architecture (Chapter 2). Our dissertation has also proposed a scheduling model (Chapter 2) and several bartering guidelines (Chapter 4), some of which could not have been easily integrated with existing P2P Grid architectures. In particular, we have studied novel data-aware scheduling policies and novel scheduling policies that make consumption decisions based on autonomously acquired information, which requires that Peers make the scheduling decisions, not the User Agents.

7.1 Summary of the Contributions

7.1.1 Scheduling

We have proposed a new P2P Grid architecture, the Lightweight Bartering Grid (LBG) architecture [59], which has led to the introduction of a scheduling model with clearly defined Policy Decision Points. Our scheduling model supports the queueing of external requests (see Section 2.9.4), which allows either online or batch-mode scheduling policies as well as the masking of individual failures of worker nodes to other Peers. Our scheduling model also enables the study of consumption decisions (see Chapter 4). We also have proposed several bartering guidelines following the study of many combinations of bartering policies. In particular, an adaptive Supplying Tasks preemption policy, coupled with a cost-aware Resource selection algorithm that we call PSufferage, has been shown to lead to efficient and robust bartering strategies.

The LBG architecture is intended to be robust to Task execution failures. Through the systematic cooperation between the Peer components and also between the Grid nodes arising from our proposed scheduling model, the reliability of the execution of computational requests is greater than the sum of the reliabilities of worker nodes. Peers can tolerate large-scale Task execution failures, either those arising from the sudden unavailability of individual worker nodes - including transient disruption of communication links - or those arising from preemption/cancellation of external Tasks by Peers.

7.1.2 Data Transfers

In collaboration with our colleague Xavier Dalem, following early work with our colleague Sébastien Jodogne, we have proposed a scalable P2P data transfer architecture [57, 56, 184]. The BitTorrent P2P file sharing protocol [87, 207, 46] is used for data transfers, in order to automatically and efficiently transfer and handle large input data files. Our algorithms are designed to reduce the cost of downloading identical - i.e. redundant - copies of input data files, whether temporally grouped or spatially grouped.

7.1.3 Software Engineering and Testing

The LBG architecture is implemented as a fully operational middleware, and also as a discrete-event simulator. We have shown how to virtualize Grid nodes, i.e. isolate them from their environment, so that a discrete-event simulator can be weaved into the bartering code of the middleware, leading to massive code reuse between implementations. Using virtualization and simulation as software engineering tools [63, 59] opens the possibility of reproducible testing and accurate

performance evaluation of the bartering code of the middleware. This is made possible because Peers of a simulated Grid make the same bartering decisions as Peers deployed on real computers.

Only the contemporarily proposed GRAS component [253] of the SimGrid [78, 278] middleware introduces a mechanism of similar nature. Both approaches are intended as tools to facilitate software engineering but differ in some of their use cases and features. GRAS/SimGrid is designed to be generic (bottom-up approach) while the LBG simulator is specifically tailored to the LBG middleware (top-down approach). The LBG simulator offers a simulation description language in which the configuration of Peer policies can be easily expressed. This tight integration enables to rapidly evaluate new combinations of scheduling policies [59, 63]. The simulation of multithreaded code (of the LBG middleware) constituted a requirement of the LBG simulator, while it is only recently [78] that support for it was added to GRAS/SimGrid through the SimIX component. Had we started our research in 2008, GRAS/SimGrid would have constituted a good starting point to enable the virtualization and simulation of LBG.

P2P Grids of up to one hundred Grid nodes have been deployed on real computers using the LBG middleware, either one Peer with around 100 worker nodes or multiple Peers with a few dozen worker nodes. P2P Grids of up to 1.6 million Grid nodes have been deployed in the LBG simulator (see Section 3.6.3), either around ten thousand Peers with a few worker nodes, or a few Peers with nearly half a million worker nodes, or a few hundred Peers with a few hundred worker nodes. Thousands of different combination of scheduling policies have been evaluated in the LBG simulator - which has been made possible with the application of the *code once, deploy twice* pattern. We thus believe that the LBG architecture can scale very well, and are looking forward to evaluate larger deployments of the LBG middleware.

7.1.4 Deployment

The LBG architecture is lightweight: It is characterized by ease and flexibility of use and deployment that are lacking in most existing Grid technologies. In particular, our proposed data transfer architecture is automatically deployed. It simply requires that human administrators open TCP ports in firewalls and it does not need the explicit construction of an overlay. Very importantly, our proposed data transfer architecture is completely transparent to human users. As a complement to the robustness provided by the scheduling model, Peers can opportunistically use their (dynamically registered) worker nodes or worker nodes supplied by other Peers.

In collaboration with our colleague Gérard Dethier, we have shown how to deploy Iterative Stencil applications on P2P Grids. We have also shown how to run them reliably, which is far from trivial [115, 116]. In particular, we have proposed an application-level P2P-aware P2P checkpointing mechanism.

7.2 Open Questions

Several questions related to our proposed ideas and mechanisms remain open and deserve further research. As a complement to the following list of such open questions, Appendix F systematically lists areas of future work, including security.

7.2.1 Scheduling and Negotiation Models

A challenging issue would be to support reservations and deadlines. To this end, each Peer could update its scheduling policies according to the behavior of the other Peers. The scheduling model could also be augmented with Task replication as a complementary mechanism for the robustness of Task execution.

Another open and intriguing question is to study the concept of variable informational opacity between Peers. By definition, Peers of a P2P Grid do not exchange metadata on their behavior, thus creating an informational opacity. Peers acquire metadata about the P2P Grid only through the observation of the outcome of their interactions with other Peers [60, 58]. However, there might exist multiple degrees of trust between Peers, beyond an all-or-nothing model. For example, metadata communicated to the Peer deployed by a University department could be more trusted if it came from a Peer deployed by another department of the same University, or from an affiliated laboratory from another University, rather than from an unknown source beyond the campus intranet. This observation, also made to suggest scheduling policies biased in favor of Peers with out-of-Grid friendly relationships [14], could lead to a finer trust model. Such a trust model would lead to the use of hybrid metadata of varying quality and provenance within the scheduling model.

7.2.2 Data Transfer and Persistence

Worker nodes that have downloaded files to process, automatically share these files using BitTorrent. Our proposed data transfer architecture could be made even more scalable if each Peer was using its worker nodes to fully distribute its data server, i.e. the Data Manager (see Section 5.2.2). The initial sharing of any file introduced in the P2P Grid could thus be done from any worker node, thus decreasing the load on the Peer.

The sharing of files using FTP could also be offloaded to worker nodes, using our envisioned but yet-to-be-evaluated proactive data replication mechanism (see Section 5.4). An approach recently proposed by Fortino et al. - based on the integration [145] of P2P, Grids and Agents to build Content Distribution Networks [242, 279] (CDN) - is very relevant to this issue and deserves close study.

Extensions to the data transfer architecture are required to support other Task models at the middleware-level. Support for communications between non-running Tasks, i.e. caching of output data files, is required for Workflow applications. Support for communications between running Tasks is required for Iterative Stencils.

Peers should also definitely be augmented with persistence capabilities to make them tolerant to their own individual failures, i.e. reload metadata from persistent storage after a crash of the computer running a Peer.

7.2.3 Simulation and Architecture

Improving the simulation of multithreading and supporting the simulation of data transfers are also two important questions to address in future research devoted to P2P Grid in general, and to LBG in particular.

In a broader perspective, P2P Grids, Desktop Grids and Volunteer Grids are three types of Grids (see Section 2.5) designed to aggregate the computational power of a large number of unreliable worker nodes [61]. The convergence of these three types of Grids would be of high interest.

7.3 General Conclusion

In this dissertation, we have systematically studied what is minimally required to enable bartering between organizations in a P2P fashion. Although there remain many open questions, the Lightweight Bartering Grid (LBG) architecture provides an integrated view and definition of what is a P2P Grid, and also how to build one. Our bartering and scheduling models, in particular, contribute to the understanding of scheduling in fully decentralized systems. Our contributions to software engineering can facilitate the work of Grid middleware researchers and developers. Our data transfer architecture contributes to the understanding of mechanisms to maintain high scalability and reduce the impact on response times when facing high volume data transfers in an unpredictable and fully decentralized computing environment. Finally, beyond these contributions to the foundations of the P2P Grid domain, the practical relevance and usefulness of the LBG middleware has

already been demonstrated in a "real-world" deployment: We have explained how to use LBG as an execution substrate to run a complex, tightly-coupled computational fluid dynamics application on a very unreliable and uncontrollable set of worker nodes.

References

[1] D. Abraham, A. Blum, and T. Sandholm, "Clearing Algorithms for Barter Exchange Markets: Enabling Nationwide Kidney Exchanges," in *Proc. ACM Conference on Electronic Commerce*, San Diego, CA, USA, June 2007.

[2] D. Abramson, J. Giddy, and L. Kotler, "High Performance Parametric Modeling with Nimrod/G: Killer Application for the Global Grid?" in *Proc. IPDPS*, Cancun, Mexico, 2000.

[3] K. Aida, A. Takefusa, H. Nakada, S. Matsuoka, S. . Sekiguchi, and U. Nagashima, "Performance Evaluation Model for Scheduling in Global Computing Systems," in *Int. J. High Performance Computing Applications*. SAGE, 2000, vol. 14, no. 3, pp. 268–279.

[4] S. Al Kiswany and M. Ripeanu, "A Simulation Study of Data Distribution Strategies for Large-scale Scientific Data Collaborations," in *Proc. CCECE*, Vancouver, BC, Canada, April 2007.

[5] S. Al Kiswany, M. Ripeanu, A. Iamnitchi, and S. Vazhkudai, "Are P2P Data-Dissemination Techniques Viable in Today's Data Intensive Scientific Collaborations ?" in *Proc. Euro-Par*, Rennes, France, August 2007.

[6] A. H. Alhusaini, C. S. Raghavendra, and V. K. Prasanna, "Run-Time Adaptation for Grid Environments," in *Proc. IPDPS*, San Francisco, CA, USA, 2001.

[7] W. Allcock, J. Bresnahan, R. Kettimuthu, M. Link, C. Dumitrescu, I. Raicu, and I. Foster, "The Globus Striped GridFTP Framework and Server," in *Proc. SC*, Seattle, WA, USA, 2005.

[8] B. Alunkal, I. Veljkovic, G. von Laszewski, and K. Amin, "Reputation-based Grid Resource Selection," in *Proc. Workshop on Adaptive Grid Middleware*, New Orleans, LA, USA, September 2003.

[9] "Amazon Elastic Compute Cloud." [Online]. Available: http://aws.amazon.com/ec2

[10] K. Amin, G. von Laszewski, and A. Mikler, "Grid Computing for the Masses: An Overview," in *Proc. Workshop on Grid and Cooperative Computing*, Shangai, China, 2003.

[11] D. Anderson and G. Fedak, "The Computational and Storage Potential of Volunteer Computing," in *Proc. CCGRID 2006*, Singapore, 2006.

[12] D. P. Anderson, "BOINC: A System for Public-Resource Computing and Storage," in *Proc. Grid*, Pittsburgh, PA, USA, November 2004.

[13] N. Andrade, W. Cirne, F. Brasileiro, and P. Roisenberg, "OurGrid: An Approach to Easily Assemble Grids with Equitable Resource Sharing," in *Proc. Workshop on Job Scheduling Strategies for Parallel Processing*, Seattle, WA, USA, 2003.

[14] N. Andrade, F. Brasileiro, and W. Cirne, "Automatic Grid Assembly by Promoting Collaboration in Peer-to-Peer Grids," in *Journal of Parallel and Distributed Computing*, 2007, vol. 67, no. 8.

[15] N. Andrade, J. Santana, F. Brasileiro, and W. Cirne, "On the Efficiency and Cost of Introducing QoS in BitTorrent," in *Proc. GP2PC*, Rio de Janeiro, Brazil, May 2007.

[16] C. Anglano, J. Brevik, M. Canonico, D. Nurmi, and R. Wolski, "Fault-aware scheduling for Bag-of-Tasks applications on Desktop Grids," in *Proc. Grid*, Barcelona, Spain, September 2006.

[17] C. Anglano and M. Canonico, "The File Mover: An Efficient Data Transfer System for Grid Applications." in *Proc. CCGrid*, Chicago, IL, USA, 2004.

[18] ——, "Scheduling Algorithms for Multiple Bag-of-Task Applications on Desktop Grids: A Knowledge-Free Approach," in *Proc. PCGrid, IPDPS Workshops*, Miami, FL, USA, April 2008.

[19] C. Anglano, M. Canonico, M. Guazzone, M. Botta, S. Rabellino, S. Arena, and G. Girardi, "Peer-to-Peer Desktop Grids in the Real World: the ShareGrid Project," in *Proc. GP2PC*, Lyon, France, May 2008.

[20] "Apache FTP Server." [Online]. Available: http://incubator.apache.org/ftpserver/

[21] A. Araújo, C. Boeres, V. E. Rebello, and C. C. Ribeiro, "A Distributed Strategy for Autonomic Grid-enabled Cooperative Metaheuristics," Catholic University of Rio de Janeiro, Rio de Janeiro, Brazil, Tech. Rep., 2007.

[22] M. P. Armstrong, M. K. Cowles, and S. Wang, "Using a Computational Grid for Geographic Information Analysis: A Reconnaissance," in *The Professional Geographer*. Blackwell, 2005, vol. 57, no. 3.

[23] "Aspect-Oriented Programming," Wikipedia, the Free Encyclopedia, September 2007. [Online]. Available: http://en.wikipedia.org/wiki/Aspect-oriented_programming

[24] "Assess Grid." [Online]. Available: http://www.assessgrid.eu/

[25] AssessGrid Consortium, "AssessGrid - Deliverable 1.3 System Architecture," AssessGrid Consortium, Tech. Rep., December 2006. [Online]. Available: http://wwwcs.uni-paderborn.de/pc2/fileadmin/AssessGrid/usermounts/publications/deliverables/AssessGrid_D1.3_System_Architecture.pdf

[26] M. Atkinson, D. DeRoure, A. Dunlop, G. Fox, P. Henderson, T. Hey, N. Paton, S. Newhouse, S. Parastatidis, A. Trefethen, P. Watson, and J. Webber, "Web Service Grids: an Evolutionary Approach," in *J. Concurrency and Computation: Practice and Experience.* Wiley, 2005, vol. 17, no. 2-4, pp. 377–389.

[27] "Autonomic Computing - The 8 Elements." [Online]. Available: http://www.research.ibm.com/autonomic/overview/elements.html

[28] A. Auvinen, M. Vapa, M. Weber, N. Kotilainen, and J. Vuori, "Chedar: Peer-to-peer Middleware," in *Proc. IPDPS*, Rhodes, Greece, 2006.

[29] A. Avila-Rosas and M. Luck, "A Direct Reputation Model for VO Formation," in *Proc. CEEMAS*, Budapest, Hungary, September 2005.

[30] "Azureus." [Online]. Available: http://azureus.sourceforge.net/

[31] "Azureus debugging." [Online]. Available: http://metastatic.org/text/Concern/2006/09/30/minor-bug-takes-an-hour-to-reproduce/

[32] F. Azzedin, M. Maheswaran, and A. Mitra, "Trust Brokering and Its Use for Resource Matchmaking in Public-Resource Grids," in *J. Grid Computing.* Springer, September 2006, vol. 4, no. 3.

[33] R. M. Badia, O. Beckmann, M. Bubak, D. Caromel, V. Getov, L. Henrio, S. Isaiadis, V. Lazarov, M. Malawski, S. Panagiotidi, N. Parlavantzas, and J. Thiyagalingam, "Lightweight Grid Platform: Design Methodology," CoreGRID, Institute on Grid Systems, Tools and Environments, Tech. Rep. TR-0020, January 2006.

[34] D. Banerjee, S. Saha, S. Sen, and P. Dasgupta, "Reciprocal Resource Sharing in P2P Environments," in *Proc. AAMAS*, Utrecht, The Netherlands, 2005.

[35] C. Banino-Rokkones, "Algorithmic and Scheduling Techniques for Heterogeneous and Distributed Computing," Ph.D. dissertation, Norwegian University of Science and Technology, Trondheim, Norway, March 2007.

[36] J. Banks, J. Carson, B. Nelson, and D. Nicol, *Discrete-Event System Simulation*, 3rd ed. Prentice Hall, 2000.

[37] A. Baratloo, M. Karaul, Z. M. Kedem, and P. Wyckoff, "Charlotte: Metacomputing on the Web," in *Proc. PDCS*, Dijon, France, September 1996.

[38] O. Beaumont, L. Carter, J. Ferrante, A. Legrand, L. Marchal, and Y. Robert, "Centralized versus distributed schedulers for multiple bag-of-task applications," in *Proc. IPDPS*, Rhodes Island, Greece, April 2006.

[39] P. Beckman, S. Nadella, N. Trebon, and I. Beschastnikh, "SPRUCE: A System for Supporting Urgent High-Performance Computing," in *Proc. IFIP WoCo9 Conference*, Prescott, AZ, USA, 2006.

[40] "BEgrid." [Online]. Available: http://www.begrid.be/

[41] F. Berman, A. J. G. Hey, and G. Fox, Eds., *Grid Computing: Making the Global Infrastructure a Reality*. Wiley, April 2003.

[42] F. Berman, A. Chien, K. Cooper, J. Dongarra, I. Foster, D. Gannon, L. Johnsson, K. Kennedy, C. Kesselman, J. Mellor-Crummey, D. Reed, L. Torczon, and R. Wolski, "The GrADS Project: Software Support for High-Level Grid Application Development," in *Int. J. High Performance Computing Applications*. SAGE, August 2001, vol. 15, no. 4, pp. 327–344.

[43] V. Berten, "Stochastic Approach to Brokering Heuristics for Computational Grids," Ph.D. dissertation, Free University of Bruxelles, Bruxelles, Belgium, June 2007.

[44] M. Beynon, R. Ferreira, T. Kurc, A. Sussman, and J. Saltz, "DataCutter: Middleware for Filtering Very Large Scientific Datasets on Archival Storage Systems," in *Proc. IEEE Symposium on Mass Storage Systems*, College Park, MD, USA, March 2000.

[45] M. Beynon, T. Kurc, A. Sussman, and J. Saltz, "Optimizing Execution of Component-based Applications using Group Instances," in *Proc. CCGrid*, Brisbane, Queensland, Australia, May 15-18 2001.

[46] "BitTorrent," Wikipedia, the Free Encyclopedia, July 2007. [Online]. Available: http://en.wikipedia.org/wiki/BitTorrent

[47] "BitTorrent for Clusters." [Online]. Available: http://www.umiacs.umd.edu/~nedwards/research/bittorrent_for_clusters.html

[48] "BitTorrent Tracker," Wikipedia, the Free Encyclopedia, July 2007. [Online]. Available: http://en.wikipedia.org/wiki/BitTorrent_tracker

[49] B. Boigelot, "Undergraduate Course INFO 062 - Programmation orientée-objet," University of Liège, 2007. [Online]. Available: http://www.montefiore.ulg.ac.be/~boigelot/cours/oop/

[50] "BOINC client-server technology," Wikipedia, the Free Encyclopedia. [Online]. Available: http://en.wikipedia.org/wiki/BOINC_client-server_technology

[51] "BOINC middleware." [Online]. Available: http://boinc.berkeley.edu/

[52] F. Brasileiro, E. Araújo, W. Voorsluys, M. Oliveira, and F. Figueiredo, "Bridging the High Performance Computing Gap: the OurGrid Experience," in *Proc. LAGrid, CCGrid Workshops*, Rio de Janeiro, Brazil, May 2007.

[53] F. Brasileiro, L. B. Costa, A. Andrade, W. Cirne, S. Basu, and S. Banerjee, "A large scale fault-tolerant grid information service," in *Proc. MGC*, Melbourne, Victoria, Australia, 2006.

[54] C. Briquet, "Algorithmes de contournement de barrières," Master's thesis, University of Liège, Liège, Belgium, 2003.

[55] C. Briquet and X. Dalem, "Lightweight Bartering Grid middleware and simulator." [Online]. Available: http://www.montefiore.ulg.ac.be/~briquet/

[56] C. Briquet, X. Dalem, S. Jodogne, and P.-A. de Marneffe, "Scheduling Data-Intensive Bags of Tasks in P2P Grids with BitTorrent-enabled Data Distribution," in *Proc. UPGRADE-CN'07, HPDC Workshops*, Monterey Bay, CA, USA, June 2007.

[57] ——, "P2P File Sharing for P2P Computing," in *Multiagent and Grid Systems*. IOS Press, 2008, to appear.

[58] C. Briquet and P.-A. de Marneffe, "Learning Reliability Models of Grid Resource Supplying," in *Proc. Cracow Grid Workshop*, Cracow, Poland, 2005.

[59] ——, "Description of a Lightweight Bartering Grid Architecture," in *Proc. Cracow Grid Workshop*, Cracow, Poland, 2006.

[60] ——, "Grid Resource Negotiation: Survey with a Machine Learning Perspective," in *Proc. Parallel and Distributed Computing and Networks*, Innsbruck, Austria, 2006.

[61] ——, "What is the Grid ? Tentative Definitions Beyond Resource Coordination," University of Liège, Liège, Belgium, Tech. Rep., 2006.

[62] ——, "Notes on an Event-Driven Implementation of Bartering in P2P Grids," University of Liège, Liège, Belgium, Tech. Rep., June 2007.

[63] ——, "Reproducible Testing of Distributed Software with Middleware Virtualization and Simulation," in *Proc. PADTAD*, Seattle, WA, USA, July 2008.

[64] R. Briquet, *L'immersion linguistique*. Labor, 2006.

[65] R. Buyya, "Economic-based Distributed Resource Management and Scheduling for Grid Computing," Ph.D. dissertation, Monash University, Melbourne, Victoria, Australia, 2002.

[66] R. Buyya, D. Abramson, and J. Giddy, "Economy Driven Resource Management Architecture for Computational Power Grids," in *Proc. Int. Conf. Parallel and Distributed Processing Techniques and Applications*, Las Vegas, NV, USA, 2000.

[67] R. Buyya, D. Abramson, and S. Venugopal, "The Grid Economy," in *Proc. of the IEEE, Special Issue on Grid Computing*, M. Parashar and C. Lee, Eds. New York, NY, USA: IEEE Press, March 2005, vol. 93, no. 3, pp. 698–714.

[68] R. Buyya, H. Stockinger, J. Giddy, and D. Abramson, "Economic Models for Management of Resources in Grid Computing," in *Proc. Int. Symposium Convergence of Information Technologies and Communications*, Denver, CO, USA, 2001.

[69] R. Buyya and M. Murshed, "GridSim: A Toolkit for the Modeling and Simulation of Distributed Resource Management and Scheduling for Grid Computing," in *Journal of Concurrency and Computation: Practice and Experience*. USA: Wiley Press, 2002.

[70] D. Cameron, R. Carvajal-Schiaffino, A. Millar, C. Nicholson, K. Stockinger, and F. Zini, "Analysis of Scheduling and Replica Optimisation Strategies for Data Grids using OptorSim," in *J. Grid Computing*. Springer, 2004, vol. 2, no. 1, pp. 57–69.

[71] M. Canonico, "Scheduling Algorithms for Bag-of-Tasks Applications on Fault-Prone Desktop Grids," Ph.D. dissertation, University of Torino, Italy, Torino, Italy, 2006.

[72] F. Cappello and al., "Grid'5000: A Large Scale, Reconfigurable, Controlable and Monitorable Grid Platform," in *Proc. Grid'2005 workshop*, Seattle, WA, USA, 2005.

[73] A. Carzaniga, G. P. Picco, and G. Vigna, "Is Code Still Moving Around? Looking Back at a Decade of Code Mobility," in *Proc. ICSE*, Minneapolis, MN, USA, May 2007.

[74] H. Casanova, A. Legrand, D. Zagorodnov, and F. Berman, "Heuristics for Scheduling Parameter Sweep Applications in Grid Environments," in *Proc. 9th Heterogeneous Computing Workshop*, Cancun, Mexico, 2000.

[75] H. Casanova, "Simgrid: A Toolkit for the Simulation of Application Scheduling," in *Proc. CCGrid*, Brisbane, Queensland, Australia, May 15-18 2001. [Online]. Available: http://citeseer.ist.psu.edu/casanova01simgrid.html

[76] H. Casanova, T. Bartol, J. Stiles, and F. Berman, "Distributing MCell Simulations on the Grid," in *Int. J. High Perf. Comp. App.*, 2001, vol. 14, no. 3.

[77] H. Casanova, A. Legrand, and L. Marchal, "Scheduling Distributed Applications: the SimGrid Simulation Framework," in *Proc. CCGrid*, Tokyo, Japan, May 12-16 2003.

[78] H. Casanova, A. Legrand, and M. Quinson, "SimGrid: a Generic Framework for Large-Scale Distributed Experimentations," in *Proc. UKSim*, Cambridge, UK, April 2008.

[79] E. Cavalcanti, L. Assis, M. Gaudêncio, W. Cirne, F. Brasileiro, and R. Novaes, "Sandboxing for a Free-to-join Grid with Support for Secure Site-wide Storage Area," in *Proc. Int. Workshop on Virtualization Technology in Distributed Computing*, Tampa, FL, USA, November 2006.

[80] S. J. Chapin, W. Cirne, D. G. Feitelson, J. P. Jones, S. T. Leutenegger, U. Schwiegelshohn, W. Smith, and D. Talby, "Benchmarks and Standards for the Evaluation of Parallel Job Schedulers," in *Proc. Workshop on Job Scheduling Strategies for Parallel Processing*, San Juan, Puerto Rico, April 1999.

[81] "ChicSim." [Online]. Available: http://people.cs.uchicago.edu/~krangana/ChicSim.html

[82] B. Chun, Y. Fu, and A. Vahdat, "Bootstrapping a Distributed Computational Economy with Peer-to-Peer Bartering," in *Proc. Workshop on Economics of Peer-to-Peer Systems*, Berkeley, CA, USA, June 2003.

[83] G. Chun, H. Dail, H. Casanova, and A. Snavely, "Benchmark Probes for Grid Assessment," in *Proc. High-Performance Grid Computing, IPDPS Workshops*, Santa Fe, NM, USA, April 2004.

[84] W. Cirne, F. Brasileiro, N. Andrade, L. B. Costa, A. Andrade, R. Novaes, and M. Mowbray, "Labs of the World, Unite!!!" in *J. Grid Computing*. Springer, 2006.

[85] W. Cirne, F. Brasileiro, J. Sauvé, N. Andrade, N. Paranhos, E. Santos-Neto, and R. Medeiros, "Grid Computing for Bag of Tasks Applications," in *Proc. 3rd IFIP Conference on E-Commerce, E-Business and E-Goverment*, São Paulo, Brazil, September 2003.

[86] W. Cirne, D. Paranhos, F. Brasileiro, L. F. W. Góes, and W. Voorsluys, "On the Efficacy, Efficiency and Emergent Behavior of Task Replication in Large Distributed Systems," in *Parallel Computing*. Elsevier, 2007, vol. 33, no. 3.

[87] B. Cohen, "Incentives Build Robustness in BitTorrent," in *Proc. Workshop on Economics of Peer-to-Peer Systems*, Berkeley, CA, USA, 2003.

[88] C. Colby, P. Lee, and G. C. Necula, "A Proof-Carrying Code Architecture for Java," in *Proc. CAV*, Chicago, IL, USA, July 2000.

[89] "Computational fluid dynamics," Wikipedia, the Free Encyclopedia. [Online]. Available: http://en.wikipedia.org/wiki/Computational_fluid_dynamics

[90] "Computing Center Software." [Online]. Available: http://wwwcs.uni-paderborn.de/pc2/projects/ccs/

[91] "Condor middleware." [Online]. Available: http://www.cs.wisc.edu/condor/

[92] F. Costa, "Extension of BOINC middleware to a Peer-to-Peer Architecture," Master's thesis, University of Coimbra, Coimbra, Portugal, July 2007.

[93] F. Costa, L. Silva, G. Fedak, and I. Kelley, "Optimizing the Data Distribution Layer of BOINC with BitTorrent," in *Proc. PCGrid, IPDPS Workshops*, Miami, FL, USA, April 2008.

[94] L. B. Costa, W. Cirne, and D. Fireman, "Converting Space Shared Resources into Intermittent Resources for use in Bag-of-Tasks Grids," in *Proc. SBAC-PAD*, Rio de Janeiro, Brazil, October 2005.

[95] P. Cozza, C. Mastroianni, D. Talia, and I. Taylor, "A Super-Peer Protocol for Multiple Job Submission on a Grid," in *Proc. CoreGrid Workshop on Grid Middleware, Euro-Par*, August 2006.

[96] K. Czajkowski, I. Foster, C. Kesselman, V. Sander, and S. Tuecke, "SNAP: A Protocol for Negotiating Service Level Agreements and Coordinating Resource Management in Distributed Systems," in *Proc. 8th Workshop on Job Scheduling Strategies for Parallel Processing*, Edinburgh, Scotland, UK, 2002.

[97] K. Czajkowski, I. Foster, and C. Kesselman, "Resource Co-Allocation in Computational Grids," in *Proc. HPDC*, Redondo Beach, CA, USA, August 1999.

[98] F. A. B. da Silva, S. Carvalho, and E. R. Hruschka, "A Scheduling Algorithm for Running Bag-of-Tasks Data Mining Applications on the Grid," in *Proc. Euro-Par*, Pisa, Italy, 2004.

[99] X. Dalem, "Implémentation d'un Grid Peer-to-Peer," Master's Thesis, University of Liège, Liège, Belgium, June 2007.

[100] E. Damiani, S. De Capitani di Vimercati, S. Paraboschi, P. Samarati, and F. Violante, "A Reputation-based Approach for Choosing Reliable Resources in Peer-to-Peer Networks," in *Proc. Conf. Computer and Communications Security*, Washington, DC, USA, November 2002.

[101] A. Dan, C. Dumitrescu, and M. Ripeanu, "Connecting Client Objectives with Resource Capabilities: An Essential Component for Grid Service Management Infrastructures," in *Proc. 2nd Int. Conf. Service Oriented Computing*, New York, NY, USA, 2004.

[102] A. Dantas, W. Cirne, and F. Brasileiro, "Improving Automated Testing of Multi-threaded Software," in *Proc. Int. Conf. Software Testing, Verification and Validation*, Lillehammer, Norway, April 2008.

[103] A. Dantas, W. Cirne, and K. Saikoski, "Using AOP to Bring a Project Back in Shape: The OurGrid Case," in *J. Brazilian Computer Society*, October 2006, vol. 11, no. 3.

[104] "Data Striping," Wikipedia, the Free Encyclopedia. [Online]. Available: http://en.wikipedia.org/wiki/Data_striping

[105] P.-A. de Marneffe, "Undergraduate Course INFO0027-1 - Introduction à l'Algorithmique II," University of Liège, 1998.

[106] C. De Rose, T. Ferreto, R. Calheiros, W. Cirne, L. Costa, and D. Fireman, "Allocation Strategies for Utilization of Space Shared Resources in Bag of Tasks Grids," in *J. FGCS*. Elsevier, 2007, to appear.

[107] J. Dean and S. Ghemawat, "MapReduce: Simplified Data Processing on Large Clusters," in *Communications of the ACM*, 2008, vol. 51, no. 1.

[108] ——, "MapReduce: Simplified Data Processing on Large Clusters," in *Proc. OSDI*, San Francisco, CA, USA, December 2008.

[109] "Debian GNU/Linux." [Online]. Available: http://www.debian.org/

[110] T. A. DeFanti, I. Foster, M. E. Papka, and R. Stevens, "Overview of the I-Way: Wide-Area Visual Supercomputing," in *Int. J. Supercomputer Applications and HPC*. SAGE, 1996, vol. 10, no. 2, pp. 123–130.

[111] "Default Policy Implementation and Policy File Syntax." [Online]. Available: http://java.sun.com/j2se/1.5.0/docs/guide/security/PolicyFiles.html

[112] A. Denis, O. Aumage, R. Hofman, K. Verstoep, T. Kielmann, and H. E. Bal, "Wide-Area Communication for Grids: An Integrated Solution to Connectivity, Performance and Security Problems," in *Proc. HPDC*, Honolulu, HI, USA, June 2004.

[113] F. Desprez and A. Vernois, "Simultaneous Scheduling of Replication and Computation for Data-Intensive Applications on the Grid," in *J. Grid Comp.* Springer, 2006.

[114] G. Dethier, C. Briquet, P. Marchot, and P.-A. de Marneffe, "A Grid-enabled Lattice-Boltzmann-based modelling system," in *Proc. International Conference on Parallel Processing and Applied Mathematics*, Gdansk, Poland, 2007.

[115] ——, "LBG-SQUARE - Fault-Tolerant, Locality-Aware Co-allocation in P2P Grids," in *Proc. PDCAT*, Dunedin, New Zealand, December 2008.

[116] ——, "LBG-SQUARE - Fault-Tolerant, Locality-Aware Co-allocation in P2P Grids," University of Liège, Liège, Belgium, Tech. Rep., 2008.

[117] "DHT-based BitTorrent (draft)." [Online]. Available: http://www.bittorrent.org/Draft_DHT_protocol.html

[118] P. A. Dinda, "Online Prediction of the Running Time of Tasks," in *Proc. HPDC*, San Francisco, CA, USA, August 2001.

[119] P. Dini, W. Gentzsch, M. Potts, A. Clemm, M. Yousif, and A. Polze, "Internet, GRID, Self-Adaptability and Beyond: Are We Ready ?" in *Proc. 15th Int. Workshop on Database and Expert Systems Applications*, Zaragoza, Spain, 2004.

[120] "Discrete-Event System Simulation," Wikipedia, the Free Encyclopedia. [Online]. Available: http://en.wikipedia.org/wiki/Discrete_event_simulation

[121] "distributed.net." [Online]. Available: http://www.distributed.net/

[122] "DMReview Glossary." [Online]. Available: http://www.dmreview.com/rg/resources/glossary.cfm

[123] "Docking@home." [Online]. Available: http://docking.utep.edu/

[124] J. E. Doran, S. Franklin, N. R. Jennings, and T. J. Norman, "On Cooperation in Multi-Agent Systems," in *The Knowledge Engineering Review*. Cambridge University Press, 1997, vol. 12.

[125] N. Drost, R. V. van Nieuwpoort, and H. E. Bal, "Simple Locality-Aware Co-allocation in Peer-to-Peer Supercomputing," in *Proc. GP2P*, Singapore, May 2006.

[126] C. Dumitrescu and I. Foster, "Usage Policy-based CPU Sharing in Virtual Organizations," in *Proc. Grid*, Pittsburgh, PA, USA, 2004.

[127] ——, "GRUBER: A Grid Resource SLA Broker," in *Proc. Euro-Par*, Lisboa, Portugal, 2005.

[128] ——, "GangSim: A Simulator for Grid Scheduling Studies," in *Proc. CCGrid*, Cardiff, UK, 2005.

[129] P. S. Dutta, L. Moreau, and N. R. Jennings, "Finding interaction partners using cognition-based decision strategies," in *Proc. IJCAI*, Acapulco, Mexico, 2003.

[130] "edtFTPj." [Online]. Available: http://www.enterprisedt.com/

[131] K. Eger, T. Hoßfeld, A. Binzenhöfer, and G. Kunzmann, "Simulation of Large-Scale P2P Networks: Packet-level vs. Flow-level Simulations," in *Proc. UPGRADE-CN'07, HPDC Workshops*, 2007.

[132] C. Engelmann and A. Geist, "Super-Scalable Algorithms for Computing on 100,000 Processors," in *Proc. Int. Conf. on Computational Science*, Atlanta, GA, USA, May 2005.

[133] C. Engelmann and G. Geist, "A Diskless Checkpointing Algorithm for Super-scale Architectures Applied to the Fast Fourier Transform," in *Proc. CLADE'03, HPDC Workshops*, Seattle, WA, USA, June 2003.

[134] T. Estrada, D. A. Flores, M. Taufer, P. J. Teller, A. Kerstens, and D. P. Anderson, "The Effectiveness of Threshold-based Scheduling Policies in BOINC Projects," in *Proc. Int. Conf. e-Science and Grid computing*, Amsterdam, The Netherlands, December 2006.

[135] "Exponential Distribution," MathWorld. [Online]. Available: http://mathworld.wolfram.com/ExponentialDistribution.html

[136] "Extensible Messaging and Presence Protocol (XMPP)." [Online]. Available: http://www.xmpp.org/

[137] "Extensible Messaging and Presence Protocol (XMPP): Core," RFC 3920, October 2004. [Online]. Available: http://www.ietf.org/rfc/rfc3920.txt

[138] "Faucets middleware." [Online]. Available: http://charm.cs.uiuc.edu/research/faucets/ZeroSumClusterBartering.html

[139] G. Fedak, C. Germain, V. Néri, and F. Cappello, "XtremWeb: A Generic Global Computing System," in *Proc. CCGrid*, Brisbane, Queensland, Australia, 2001.

[140] D. G. Feitelson, "Workload Modeling for Performance Evaluation," in *Proc. Performance Evaluation of Complex Systems: Techniques and Tools*, Rome, Italy, September 2002.

[141] M. Feldman and J. Chuang, "Overcoming Free-Riding Behavior in Peer-to-Peer Systems," in *ACM SIGecom Exchanges*, July 2005, vol. 5, no. 4.

[142] J. E. Fokker, P. H. Westendorp, and H. de Ridder, "Towards Stimulating Cooperative Behavior In Peer-To-Peer Networks," in *Proc. ICIS*, Las Vegas, NV, USA, December 2005.

[143] "Folding@home." [Online]. Available: http://folding.stanford.edu/

[144] G. Fortino and W. Russo, "High-level Interoperability between Java-based Mobile Agent Systems," in *PDCS*, San Francisco, CA, USA, September 2004.

[145] ——, "Using P2P, GRID and Agent technologies for the development of content distribution networks," in *J. FGCS*. Elsevier, 2008, vol. 24, no. 3.

[146] G. Fortino, W. Russo, and E. Zimeo, "Reliable Multicast Protocols for Java-based Grid Middleware Platforms," in *Proc. PDCS*, Marina del Rey, CA, USA, November 2003.

[147] I. Foster and A. Iamnitchi, "On Death, Taxes, and the Convergence of Peer-to-Peer and Grid Computing," in *Proc. 2nd Int. Workshop on Peer-to-Peer Systems*, Berkeley, CA, USA, 2003.

[148] I. Foster, N. Jennings, and C. Kesselman, "Brain Meets Brawn: Why Grids and Agents Need Each Other," in *Proc. Int. Conference on Autonomous Agents and Multi-Agent Systems*, New York, NY, USA, 2004.

[149] I. Foster and C. Kesselman, Eds., *The Grid 2: Blueprint for a New Computing Infrastructure*. San Francisco, CA, USA: Morgan Kaufmann, 2004.

[150] I. Foster, C. Kesselman, and S. Tuecke, "The Anatomy of the Grid: Enabling Scalable Virtual Organizations," *Int. J. Supercomputer App.*, vol. 15(3), 2001.

[151] I. Foster, "What is the Grid ? A three Point Checklist," *Grid Today*, July 2002.

[152] I. Foster and C. Kesselman, "Globus: A Metacomputing Infrastructure Toolkit," in *Int. J. Supercomputer Applications and HPC*. SAGE, 1997, vol. 11, no. 2, pp. 115–128.

[153] M.-E. Frincu, M. Quinson, and F. Suter, "Handling Very Large Platforms with the New SimGrid Platform Description Formalism," INRIA, University Henri Poincaré, Nancy, France, Tech. Rep., February 2008.

[154] Y. Fu, J. Chase, B. Chun, S. Schwab, and A. Vahdat, "SHARP: An Architecture for Secure Resource Peering," in *Proc. ACM Symposium Operating Systems Principles*, New York, NY, USA, 2003.

[155] P. Garbacki, A. Iosup, D. Epema, and M. van Steen, "2Fast: Collaborative Downloads in P2P Networks," in *6th IEEE Int. Conf. Peer-to-Peer Computing*, Galway, Ireland, September 2006. [Online]. Available: http://pds.twi.tudelft.nl/%7Epawel/pub/2fast_p2p.pdf

[156] Y. Georgiou, J. Leduc, B. Videau, J. Peyrard, and O. Richard, "A Tool for Environment Deployment in Clusters and Light Grids," in *Second Workshop on System Management Tools for Large-Scale Parallel Systems*, Rhodes, Greece, April 2006.

[157] Y. Gil, E. Deelman, J. Blythe, C. Kesselman, and H. Tangmunarunkit, "Artificial Intelligence and Grids: Worflow Planning and Beyond," in *Intelligent Systems, Special Issue on E-Science*. IEEE, 2004, vol. 19, no. 1, pp. 26–33.

[158] "gLite: Lightweight Middleware for Grid Computing." [Online]. Available: http://glite.web.cern.ch/

[159] "The Globus Alliance." [Online]. Available: http://www.globus.org/

[160] "Gnutella," Wikipedia, the Free Encyclopedia. [Online]. Available: http://en.wikipedia.org/wiki/Gnutella

[161] "Grid Café." [Online]. Available: http://gridcafe.web.cern.ch/gridcafe/

[162] "Grid Scheduling Simulator." [Online]. Available: http://www.gssim.org/

[163] "Grid Workloads Archive." [Online]. Available: http://gwa.ewi.tudelft.nl/pmwiki/

[164] "GridBus middleware." [Online]. Available: http://www.gridbus.org/

[165] "GridFTP." [Online]. Available: http://www.globus.org/grid_software/data/gridftp.php

[166] "GridSphere." [Online]. Available: http://www.gridsphere.org/

[167] M. Henning, "A New Approach to Object-Oriented Middleware," in *IEEE Internet Computing*. Piscataway, NJ, USA: IEEE Educational Activities Department, January 2004, vol. 8, no. 1.

[168] M. Hovestadt, O. Kao, A. Keller, and A. Streit, "Scheduling in HPC Resource Management Systems," in *Proc. 9th Workshop Job Scheduling Strategies for Parallel Processing*, Seattle, WA, USA, 2003.

[169] F. Howell and R. McNab, "SimJava: A Discrete Event Simulation Package for Java with Applications in Computer Systems Modelling," in *Proc. Conf. Web-Based Modeling and Simulation*, San Diego, CA, USA, 1998.

[170] "HPC4U Project deliverable - Negotiating on QoS aspects, version 1.2," December 2004. [Online]. Available: http://www.hpc4u.org/docs/HPC4U_D12_NegotiatingQoS.pdf

[171] Y. Hu, Y. Xue, J. Wang, X. Sun, G. Cai, J. Tang, Y. Luo, S. Zhong, Y. Wang, and A. Zhang, "Feasibility Study of Geo-Spatial Analysis Using Grid Computing," in *Proc. ICCS*, Cracow, Poland, 2004.

[172] B. Hudzia, T. Ellahi, L. McDermott, and T. Kechadi, "A Java Based Architecture of P2P-Grid Middleware," in *Proc. PDPTA*, Las Vegas, NV, USA, June 2006.

[173] B. Hudzia, L. McDermott, T. Ellahi, and T. Kechadi, "Entity Based Peer to Peer in a Data Grid Environment," in *Proc. IMACS World Congress Scientific Computation, Applied Mathematics and Simulation*, Paris, France, July 2005.

[174] B. Hudzia, "THON: A Tessellation based Hierarchical Overlay Network for DGET Grid Middleware," in *Proc. ICCCN*, Honolulu, HI, USA, August 2007.

[175] "Introduction to the GRAS framework." [Online]. Available: http://simgrid.gforge.inria.fr/doc/GRAS_tut_intro.html

[176] A. Iosup, O. Sonmez, S. Anoep, and D. Epema, "The Performance of Bags-Of-Tasks in Large-Scale Distributed Computing Systems," in *Proc. HPDC*, Boston, MA, USA, June 2008.

[177] D. Irwin, L. Grit, and J. Chase, "Balancing Risk and Reward in a Market-based Task Service," in *Proc. HPDC*, Honolulu, HI, USA, 2004.

[178] "J2SE 5.0 API Specification." [Online]. Available: http://java.sun.com/j2se/1.5.0/docs/api/

[179] P. H. Jacobs and A. Verbraeck, "Single-Threaded Specification of Process-Interaction Formalism in Java," in *Proc. Winter Simulation Conference*, Washington, DC, USA, 2004.

[180] "Java Remote Method Invocation (RMI)." [Online]. Available: http://java.sun.com/javase/technologies/core/basic/rmi/index.jsp

[181] E. Jeannot, "Improving Middleware Performance with AdOC: an Adaptive Online Compression Library for Data Transfer," in *Proc. IPDPS*, Denver, CO, USA, April 2005.

[182] N. Jennings, P. Faratin, A. Lomuscio, S. Parsons, C. Sierra, and M. Woolridge, "Automated Negotiation: Prospects, Methods and Challenges," in *Int. J. Group Decision and Negotiation*. Springer, 2001, vol. 10, no. 2.

[183] "JNGI." [Online]. Available: http://jngi.jxta.org/

[184] S. Jodogne, C. Briquet, and J. Piater, "Approximate Policy Iteration for Closed-Loop Learning of Visual Tasks," in *Proc. European Conference on Machine Learning*, Berlin, Germany, 2006.

[185] "JXTA." [Online]. Available: http://www.jxta.org/

[186] L. Kale, S. Kumar, J. DeSouza, M. Potnuru, and S. Bandhakavi, "Faucets: Efficient Resource Allocation on the Computational Grid," in *Proc. Int. Conf. on Parallel Processing*, Montreal, Quebec, Canada, 2004.

[187] S. Kamil, K. Datta, S. Williams, L. Oliker, J. Shalf, and K. Yelick, "Implicit and Explicit Optimizations for Stencil Computations," in *Proc. Memory Systems Performance and Correctness*, San Jose, CA, USA, 2006.

[188] N. Kapadia, J. Fortes, and C. Brodley, "Predictive Application-Performance Modeling in a Computational Grid Environment," in *Proc. HPDC*, Redondo Beach, CA, USA, August 1999.

[189] A. Kaplan, G. C. Fox, and G. von Laszewski, "GridTorrent Framework: A High-performance Data Transfer and Data Sharing Framework for Scientific Computing," in *Proc. Grid Computing Environments, Supercomputing Workshops*, Reno, NV, USA, November 2007.

[190] "Koala scheduler." [Online]. Available: http://www.st.ewi.tudelft.nl/koala/

[191] D. Kondo, F. Araujo, P. Malecot, P. Domingues, L. M. Silva, G. Fedak, and F. Cappello, "Characterizing Result Errors in Internet Desktop Grids," in *Proc. Euro-Par*, Rennes, France, August 2007.

[192] D. Kondo, H. Casanova, and A. A. Chien, "Scheduling Task Parallel Applications for Rapid Application Turnaround on Enterprise Desktop Grids," in *J. Grid Computing*. Kluwer, 2007, to appear.

[193] D. Kondo, A. A. Chien, and H. Casanova, "Resource Management for Rapid Application Turnaround on Enterprise Desktop Grids," in *Proc. SC*, Pittsburgh, PA, September 2004.

[194] S. G. M. Koo, C. S. G. Lee, K. Kannan, and S.-w. Kwong, "On the Economics of Peer-to-Peer Content Distribution Systems for Large-volume Contents," in *Proc. SCI*, Orlando, FL, USA, July 2004.

[195] E. Korach, S. Kutten, and S. Moran, "A Modular Technique for the Design of Efficient Distributed Leader Finding Algorithms," in *ACM Transactions on Programming Languages and Systems*, 1990, vol. 12, no. 1, pp. 84–101.

[196] M. Kotilainen, N.and Vapa, M. Weber, J. Töyrylä, and J. Vuori, "P2PDisCo - Java Distributed Computing for Workstations Using Chedar Peer-to-Peer Middleware," in *Proc. IPDPS*, Denver, CO, USA, 2005.

[197] N. Kotilainen, M. Vapa, A. Auvinen, M. Weber, and J. Vuori, "P2PStudio - Monitoring, Controlling and Visualization Tool for Peer-to-Peer Networks Research," in *ACM Int. Workshop on Performance Monitoring, Measurement and Evaluation of Heterogeneous Wireless and Wired Networks*, Torremolinos, Spain, 2006.

[198] N. Kotilainen, M. Vapa, T. Keltanen, A. Auvinen, and J. Vuori, "P2PRealm - Peer-to-Peer Network Simulator," in *Proc. Int. Workshop on Computer-Aided Modeling, Analysis and Design of Communication Links and Networks*, Trento, Italy, 2006.

[199] B. Kreaseck, L. Carter, H. Casanova, and J. Ferrante, "Autonomous Protocols for Bandwidth-Centric Scheduling of Independent-task Applications," in *Proc. IPDPS*, Nice, France, April 2003.

[200] K. Kurowski, J. Nabrzyski, and J. Pukacki, "User preference driven multiobjective resource management in Grid environments," in *Proc. CCGrid*, Brisbane, Queensland, Australia, 2001.

[201] K. Kurowski, J. Nabrzyski, A. Oleksiak, and J. Weglarz, "Grid Scheduling Simulations with GSSIM," in *Proc. 3rd International Workshop on Scheduling and Resource Management for Parallel and Distributed Systems*, Hsinchu, Taiwan, December 2007.

[202] S. Lacour, "Contribution à l'automatisation du déploiement d'applications sur des grilles de calcul," Ph.D. dissertation, University of Rennes, Rennes, France, December 2005.

[203] E. Lawler, J. Lenstra, K. A. Rinnooy, and D. Shmoys, Eds., *The Traveling Salesman Problem: A Guided Tour of Combinatorial Optimization*, ser.

Wiley Series in Discrete Mathematics & Optimization. New York: Wiley, 1985.

[204] G. Leduc, "Graduate Course INFO 031 - Structure des Réseaux Informatiques et Multimédia," University of Liège, 2007. [Online]. Available: http://www.montefiore.ulg.ac.be/~leduc/cours/structure-multimedia.html

[205] C. Lee, Y. Schwartzman, J. Hardy, and A. Snavely, "Are user runtime estimates inherently inaccurate ?" in *Proc. Workshop on Job Scheduling Strategies for Parallel Processing*, New York, NY, USA, June 2004.

[206] C. B. Lee and A. E. Snavely, "Precise and Realistic Utility Functions for User-Centric Performance Analysis of Schedulers," in *Proc. HPDC*, Monterey Bay, CA, USA, June 2007.

[207] A. Legout, G. Urvoy-Keller, and P. Michiardi, "Understanding BitTorrent: An Experimental Perspective," INRIA, Sophia Antipolis, France, Tech. Rep., 2005.

[208] A. Legrand, "Models, Simulation, Emulation, Experimentation for Grid Computing," Seminar at the Grid 5000 school, March 2006. [Online]. Available: http://graal.ens-lyon.fr/~alegrand/articles/slides_g5k_simul.pdf

[209] A. Legrand, M. Quinson, K. Fujiwara, and H. Casanova, "The SimGrid Project - Simulation and Deployment of Distributed Applications," in *Proc. HPDC*, Paris, France, May 2006.

[210] R. Leus, "The generation of stable project plans," Ph.D. dissertation, Catholic University of Leuven, Leuven, Belgium, 2003.

[211] A. Lima, W. Cirne, F. Brasileiro, and D. Fireman, "A Case for Event-Driven Distributed Objects," in *Proc. DOA, OTM Workshops*, Montpellier, France, October 2006.

[212] S. Loureiro, "Mobile Code Protection," Ph.D. dissertation, Euécom Institute, Sophia Antipolis, France, January 2001.

[213] S. Loureiro and R. Molva, "Privacy for Mobile Code," in *Proc. OOPSLA*, Denver, CO, USA, November 1999.

[214] D. Lu, Y. Qiao, P. Dinda, and F. Bustamante, "Modeling and Taming Parallel TCP on the Wide Area Network," in *Proc. IPDPS*, Denver, CO, USA, April 2005.

[215] M. Maheswaran, S. Ali, H. J. Siegel, D. A. Hensgen, and R. F. Freund, "Dynamic Matching and Scheduling of a Class of Independent Tasks onto Heterogeneous Computing Systems," in *Heterogeneous Computing Workshop*, San Juan, Puerto Rico, April 1999.

[216] P. Marchot, D. Beugre, G. Crine, Michel Dethier, A. Léonard, and D. Toye, "Lattice boltzmann 3d flow simulations in a hepa aerosol filter, on a computing grid," in *Proc. European Congress of Chemical Engineering*, Copenhagen, Denmark, September 2007.

[217] M. L. Massie, B. N. Chun, and D. E. Culler, "The Ganglia Distributed Monitoring System: Design, Implementation, and Experience," in *Parallel Computing*. Elsevier, July 2004, vol. 30, no. 7.

[218] M. Meulpolder, D. Epema, and H. Sips, "Replication in Bandwidth-Symmetric BitTorrent Networks," in *Proc. Hot-P2P, IPDPS Workshops*, Miami, FL, USA, April 2008.

[219] "MOAB Grid middleware." [Online]. Available: http://www.gridtoday.com/grid/512100.html

[220] M. Murshed and R. Buyya, "Using the GridSim Toolkit for Enabling Grid Computing Education," in *Proc. Conf. Comm. Networks and Distributed Systems Modeling and Simulation*, San Antonio, TX, USA, 2002.

[221] J. Nabrzyski, J. Schopf, and J. Weglarz, Eds., *Grid Resource Management: State of the Art and Future Trends*. Kluwer Academic Publishers, 2003.

[222] "Napster," Wikipedia, the Free Encyclopedia. [Online]. Available: http://en.wikipedia.org/wiki/Napster

[223] G. C. Necula, "Proof-Carrying Code," in *Proc. POPL*, Paris, France, January 1997.

[224] "Nodezilla." [Online]. Available: http://www.nodezilla.net/wiki/pmwiki.php/Docs/AzureusPlugin

[225] T. J. Norman, A. Preece, S. Chalmers, N. R. Jennings, M. Luck, V. D. Dang, T. D. Nguyen, V. Deora, J. Shao, A. Gray, and N. Fiddian, "CONOISE: Agent-Based Formation of Virtual Organisations," in *Proc. 23rd SGAI Int. Conf. on Innovative Techniques and Applications of AI*, Cambridge, UK, 2003.

[226] L. Nussbaum, "Lightweight Emulation to Study Peer-to-Peer Systems," in *Hot-P2P*, Rhodes, Greece, April 2006.

[227] ——, "Use of Grid Computing for Debian Quality Assurance," in *Proc. FOSDEM*, Bruxelles, Belgium, February 2007.

[228] P. Obreiter and J. Nimis, "A Taxonomy of Incentive Patterns - the Design Space of Incentives for Cooperation," in *Proc. Int. Workshop Agents and Peer-to-Peer Computing*, Melbourne, Victoria, Australia, 2003.

[229] "The Open Grid Services Architecture, Version 1.0." [Online]. Available: http://www.gridforum.org/documents/GWD-I-E/GFD-I.030.pdf

[230] R. Olejnik, B. Toursel, M. Tudruj, and E. Laskowski, "DG-ADAJ: a Java Computing Platform for Desktop Grid," in *Proc. Cracow Grid Workshop*, Cracow, Poland, 2005.

[231] "Open Grid Forum." [Online]. Available: http://www.ogf.org/

[232] D. Oppenheimer, J. Albrecht, D. Patterson, and A. Vahdat, "Distributed Resource Discovery on PlanetLab with SWORD," in *Proc. Workshop Real, Large Distributed Systems*, San Francisco, CA, USA, December 2004.

[233] "OurGrid middleware." [Online]. Available: http://www.ourgrid.org/

[234] "Overlay Network," Wikipedia, the Free Encyclopedia. [Online]. Available: http://en.wikipedia.org/wiki/Overlay_network

[235] "Overview of the Grid Security Infrastructure." [Online]. Available: http://www.globus.org/security/overview.html

[236] "Oxford American dictionary," 2005.

[237] C. Ozturan, "Resource Bartering in Grids," in *Proc. Concurrent Information Processing And Computing*, Sinaia, Romania, 2003.

[238] "Parallel Workloads Archive." [Online]. Available: http://www.cs.huji.ac.il/labs/parallel/workload/

[239] "Parsec." [Online]. Available: http://pcl.cs.ucla.edu/projects/parsec

[240] J. A. Patel and I. Gupta, "Overhaul: Extending HTTP to Combat Flash Crowds," in *Proc. Workshop Web Content Caching and Distribution*, Beijing, China, October 2004.

[241] J. Patel, W. T. L. Teacy, N. R. Jennings, M. Luck, S. Chalmers, N. Oren, T. J. Norman, A. Preece, P. M. Gray, G. Shercliff, P. J. Stockreisser, J. Shao, W. A. Gray, N. J. Fiddian, and S. Thompson, "Monitoring, Policing and Trust for Grid-Based Virtual Organisations," in *Proc. UK e-Science All Hands Meeting*, Nottingham, UK, 2005.

[242] A.-M. K. Pathan, J. Broberg, K. Bubendorfer, K. H. Kim, and R. Buyya, "An Architecture for Virtual Organization (VO)-based Effective Peering of Content Delivery Networks," in *Proc. UPGRADE-CN'07, HPDC Workshops*, Monterey Bay, CA, USA, June 2007.

[243] P. Pattnaik, K. Ekanadham, and J. Jann, *Grid Computing: Making the Global Infrastructure a Reality.* Wiley, April 2003, ch. 13.

[244] "Peer." [Online]. Available: http://www.geo.ulg.ac.be/liege/principaute/peer.htm

[245] "Google Map of Peer." [Online]. Available: http://maps.google.be/maps?q=Peer

[246] J. S. Plank, Y. Kim, and J. J. Dongarra, "Fault Tolerant Matrix Operations for Networks of Workstations Using Diskless Checkpointing," in *J. Parallel and Distributed Computing*. Elsevier, 1997, vol. 43, no. 2.

[247] J. S. Plank, K. Li, and M. Puening, "Diskless Checkpointing," in *IEEE Transactions on Parallel and Distributed Systems*, October 1998, vol. 9, no. 10.

[248] A. Porter, A. Memon, C. Yilmaz, D. C. Schmidt, and B. Natarajan, "Skoll: A Process and Infrastructure for Distributed Continuous Quality Assurance," in *IEEE Transactions on Software Engineering*, 2007, vol. 33, no. 8.

[249] "Postcards From the Edge of the GRIDtoday VIP Summit." [Online]. Available: http://news.taborcommunications.com/msgget.jsp?mid=442401&xsl=story.xsl

[250] J. Pouwelse, P. Garbacki, J. Wang, A. Bakker, J. Yang, A. Iosup, D. Epema, M. Reinders, M. van Steen, and H. Sips, "Tribler: a social-based peer-to-peer system," *Concurrency and Computation (to appear)*, 2007, accepted for publication. [Online]. Available: http://www.pds.ewi.tudelft.nl/pubs/papers/cpe2007.pdf

[251] "ProActive middleware." [Online]. Available: http://www-sop.inria.fr/oasis/ProActive/

[252] M. Quinson, "Découverte automatique des caractéristiques et capacités d'une plate-forme de calcul distirbué," Ph.D. dissertation, ENS Lyon, Lyon, France, December 2003.

[253] ——, "GRAS: a Research and Development Framework for Grid and P2P Infrastructures," in *Proc. Parallel and Distributed Computing and Systems*, Dallas, TX, USA, November 2006.

[254] A. Ramakrishnan, G. Singh, H. Zhao, E. Deelman, R. Sakellariou, K. Vahi, K. Blackburn, D. Meyers, and M. Samidi, "Scheduling Data Intensive Workflows Onto Storage-Constrained Distributed Resources," in *Proc. CCGrid*, Rio de Janeiro, Brazil, May 2007.

[255] R. Raman, M. Livny, and M. Solomon, "Matchmaking: Distributed Resource Management for High Throughput Computing," in *Proc. HPDC*, Chicago, IL, USA, July 1998.

[256] ——, "Resource Management through Multilateral Matchmaking," in *Proc. HPDC*, Pittsburgh, PA, USA, August 2000.

[257] K. Ranganathan and I. Foster, "Decoupling Computation and Data Scheduling in Distributed Data-Intensive Applications," in *Proc. HPDC*, Edinburgh, Scotland, UK, 2002.

[258] C. Rattanapoka, "A fault-tolerant Message Passing Interface Implementation for Grids," Ph.D. dissertation, University Louis Pasteur, Strasbourg, France, April 2008.

[259] "Reed College." [Online]. Available: http://www.reed.edu/

[260] X. Ren, S. Lee, R. Eigenmann, and S. Bagchi, "Resource Availability Prediction in Fine-Grained Cycle Sharing Systems," in *Proc. HPDC*, Paris, France, June 2006.

[261] ——, "Prediction of Resource Availability in Fine-Grained Cycle Sharing Systems and Empirical Evaluation," in *Journal of Grid Computing*. Springer, 2007.

[262] A. Rezmerita, T. Morlier, V. Neri, and F. Cappello, "Private Virtual Cluster: Infrastructure and Protocol for Instant Grids," in *Proc. Euro-Par*, Dresden, Germany, August 2006.

[263] M. Ripeanu, M. Mowbray, N. Andrade, and A. Lima, "Gifting Technologies: A BitTorrent Case Study," in *First Monday*, November 2006, vol. 11, no. 11. [Online]. Available: http://www.firstmonday.org/issues/issue11_11/ripeanu/index.html

[264] M. Roubens, "Graduate Course - Théorie des Graphes," University of Liège, 2000.

[265] R. Santos, A. Andrade, F. Brasileiro, W. Cirne, and N. Andrade, "Accurate Autonomous Accounting in Peer-to-Peer Grids," in *Proc. Workshop on Middleware for Grid Computing*, Grenoble, France, 2005.

[266] R. Santos, A. Andrade, W. Cirne, F. Brasileiro, and N. Andrade, "Relative Autonomous Accounting for Peer-to-Peer Grids," in *Concurrency and Computation: Practice and Experience*. Wiley, 2007, vol. 19, no. 14.

[267] E. Santos-Neto, W. Cirne, F. Brasileiro, and A. Lima, "Exploiting Replication and Data Reuse to Efficiently Schedule Data-intensive Applications on Grids," in *Proc. Workshop Job Scheduling Strategies for Parallel Processing*, New York, NY, USA, 2004.

[268] L. F. G. Sarmenta, "Sabotage-Tolerance Mechanisms for Volunteer Computing Systems," in *Proc. CCGrid*, Brisbane, Queensland, Australia, May 2001.

[269] J. M. Schopf and F. Berman, "Stochastic Scheduling," in *Proc. Supercomputing*, Portland, OR, USA, November 1999.

[270] J. M. Schopf and S. J. Newhouse, "State of Grid Users: 25 Conversations with UK eScience Groups," in *J. Cluster Computing.* Springer, 2004, to appear.

[271] J. M. Schopf and B. Nitzberg, "Grids: The Top Ten Questions," Northwestern University, Evanston, IL, USA, Tech. Rep. #CS-00-05, 2000.

[272] ——, "Grids: The Top Ten Questions," in *Scientific Programming Journal, Special Issue on Grid Computing.* IOS Press, August 2002, vol. 10, no. 2.

[273] "SETI@home." [Online]. Available: http://setiathome.berkeley.edu/

[274] C. Shapiro, "Consumer information, product quality, and seller reputation," in *The Bell Journal of Economics*, 1982, vol. 13.

[275] P. Shivam, A. Iamnitchi, A. Yumerefendi, and J. Chase, "Model-Driven Placement of Compute Tasks and Data in a Networked Utility," in *Proc. ICAC*, Seattle, WA, USA, 2005.

[276] J. F. Shoch and J. A. Hupp, "The "Worm" Programs - Early Experience with a Distributed Computation," in *Communications of the ACM*, 1982, vol. 25, no. 3.

[277] "SimBoinc." [Online]. Available: http://simboinc.gforge.inria.fr/

[278] "SimGrid middleware." [Online]. Available: http://simgrid.gforge.inria.fr/

[279] S. Sivasubramanian, G. Pierre, and M. van Steen, "Autonomic Data Placement Strategies for Update-intensive Web Applications," in *Proc. Int. Workshop Advanced Architectures and Algorithms for Internet Delivery and Applications*, Orlando, FL, USA, June 2005.

[280] S. Smallen, H. Casanova, and F. Berman, "Applying Scheduling and Tuning to On-line Parallel Tomography," in *Proc. SC*, Denver, CO, USA, November 2001.

[281] W. Smith, I. Foster, and V. E. Taylor, "Scheduling with Advanced Reservations," in *Proc. IPDPS*, Cancun, Mexico, May 2000.

[282] W. Smith, V. E. Taylor, and I. Foster, "Predicting Application Run Times Using Historical Information," in *Proc. JSSPP, IPPS/SPDP Workshops*, Orlando, FL, USA, March 1998.

[283] ——, "Using Run-Time Predictions to Estimate Queue Wait Times and Improve Scheduler Performance," in *Proc. JSSPP, IPPS/SPDP Workshops*, San Juan, Puerto Rico, April 1999.

[284] E. Sober and D. S. Wilson, *Unto Others: The Evolution and Psychology of Unselfish Behaviour*. Cambridge, MA, USA: Harvard University Press, 1999.

[285] A. Streit, "Self-Tuning Job Scheduling Strategies for the Resource Management of HPC Systems and Computational Grids," Ph.D. dissertation, University of Paderborn, Paderborn, Germany, 2003.

[286] "Stripped-down grid: A lightweight grid for computing's have-nots." [Online]. Available: http://www.hpl.hp.com/news/2005/jan-mar/grid.html

[287] D. Stutzbach, D. Zappala, and R. Rejaie, "The Scalability of Swarming Peer-to-Peer Content Delivery," in *Proc. IFIP Networking*, Waterloo, Ontario, Canada, May 2005.

[288] A. Sulistio, C. S. Yeo, and R. Buyya, "Visual Modeler for Grid Modelling and Simulation (GridSim) Toolkit," in *ICCS*, Melbourne, Victoria, Australia, 2003.

[289] T. Sundsted, "The practice of peer-to-peer computing: Introduction and history," March 2001. [Online]. Available: http://www.ibm.com/developerworks/java/library/j-p2p/

[290] "Sun Grid Compute Utility." [Online]. Available: http://www.network.com/

[291] J. Tang, "An Agent-based Peer-to-Peer Grid Computing Architecture," Master's thesis, University of Wollongong, Wollongong, New South Wales, Australia, 2005.

[292] "Tango Library." [Online]. Available: http://tango.freedesktop.org/

[293] Y. Tanimura, T. Hiroyasu, M. Miki, and K. Aoi, "The System for Evolutionary Computing on the Computational Grid," in *Proc. Parallel and Distributed Computing and Systems*, Cambridge, MA, USA, 2002.

[294] "The Network Simulator (ns-2)." [Online]. Available: http://www.isi.edu/nsnam/ns/

[295] "Tribler P2P network." [Online]. Available: https://www.tribler.org/wiki/whatIsTribler

[296] G. Tsouloupas and M. D. Dikaiakos, "Grid Resource Ranking using Low-level Performance Measurements," in *Proc. Euro-Par*, Rennes, France, August 2007.

[297] P. Tucker and F. Berman, "On Market Mechanisms as a Software Technique," University of California, San Diego, CA, USA, Tech. Rep. TR CS96-513, 1996.

[298] "UCLA Grid Portal middleware." [Online]. Available: http://www.ucgrid.org/

[299] "Unicore middleware." [Online]. Available: http://www.unicore.eu/

[300] J. Verbeke, N. Nadgir, G. Ruetsch, and I. Sharapov, "Framework for Peer-to-Peer Distributed Computing in a Heterogeneous, Decentralized Environment," in *Proc. Grid*, Baltimore, MD, USA, 2002.

[301] G. Vigna, "Mobile Agents: Ten Reasons For Failure," in *Proc. MDM*, Berkeley, CA, USA, January 2004.

[302] "VirtualBox." [Online]. Available: http://www.virtualbox.org/

[303] G. L. Volpato and C. Grimm, "Dynamic Firewalls and Service Deployment Models for Grid Environments," in *Proc. Cracow Grid Workshop*, Cracow, Poland, 2006.

[304] G. von Laszewski, "The Grid-Idea and Its Evolution," in *J. Information Technology*, 2005, vol. 47, no. 6.

[305] W. Voorsluys and F. V. Brasileiro, "On the design and implementation of an open peer-to-peer grid infrastructure," in *Int. Workshop Real Overlays And Distributed Systems*, Belém, Brazil, June 2007.

[306] K. Voß, "Comparing Fault-Tolerance Mechanisms for Self-Organizing Resource Management in Grids," in *Proc. Conf. Semantic, Knowledge and Grid*, Xi'an, China, October 2007.

[307] ——, "Enhance Self-managing Grids by Risk Management," in *Proc. ICNS*, Athens, Greece, June 2007.

[308] H. Wang, H. Takizawa, and H. Kobayashi, "A dependable Peer-to-Peer computing platform," in *J. FGCS*. Elsevier, 2007, vol. 23, no. 8.

[309] B. Wei, G. Fedak, and F. Cappello, "Collaborative Data Distribution with BitTorrent for Computational Desktop Grids," in *Proc. ISPDC*, Lille, France, 2005.

[310] ——, "Scheduling Independent Tasks Sharing Large Data Distributed with BitTorrent," in *Proc. Grid*, Seattle, WA, USA, 2005.

[311] M. Wellman, W. Walsh, P. Wurman, and J. MacKie-Mason, "Auction Protocols for Decentralized Scheduling," in *Games and Economic Behavior*. Academic Press, 2001, vol. 35, pp. 271–303.

[312] M. Wilkes, "The dynamics of paging," in *The Computer Journal*. British Computer Society, 1973, vol. 16, no. 1.

[313] P. Wolper and B. Boigelot, "Verifying Systems with Infinite but Regular State Spaces," in *Proc. CAV*, Vancouver, BC, Canada, June 1998.

[314] R. Wolski and J. Spring, N. Hayes, "The Network Weather Service: A Distributed Resource Performance Forecasting Service for Metacomputing," in *J. FGCS*. Elsevier, 1999, vol. 15, no. 5-6.

[315] "Xen." [Online]. Available: http://xen.org/

[316] H. Xia, H. Dail, H. Casanova, and A. Chien, "The MicroGrid: Using Emulation to Predict Application Performance in Diverse Grid Network Environments," in *Proc. CLADE'04, HPDC Workshops*, Honolulu, HI, USA, June 2004.

[317] L. Xiao, Y. Zhu, L. Ni, and Z. Xu, "GridIS: an Incentive-based Grid Scheduling," in *Proc. IPDPS*, Denver, CO, USA, 2003.

[318] "XtremWeb middleware." [Online]. Available: http://www.xtremweb.org/

[319] G. Xu, A. Rountev, Y. Tang, and F. Qin, "Efficient Checkpointing of Java Software Using Context-Sensitive Capture and Replay," in *ACM SIGSOFT Symposium on the Foundations of Software Engineering*, Dubrovnik, Croatia, September 2007.

[320] P. Yalagandula, S. Nath, H. Yu, P. B. Gibbons, and S. Seshan, "Beyond Availability: Towards a Deeper Understanding of Machine Failure Characteristics in Large Distributed Systems," in *Proc. Workshop Real, Large Distributed Systems*, San Francisco, CA, USA, December 2004.

[321] L. Yang, J. Schopf, and I. Foster, "Conservative Scheduling: Using Predicted Variance to Improve Scheduling Decisions in Dynamic Environments," in *Proc. Supercomputing*, Phoenix, AZ, USA, 2003.

[322] W. Yang and N. B. Abu-Ghazaleh, "GPS: A General Peer-to-Peer Simulator and its Use for Modeling BitTorrent," in *Proc. MASCOTS*, Atlanta, GA, USA, 2005.

[323] Z. Yang and R. Gunasheelan, "An SNAP-based Resource Management System for Grid Environments," in *Proc. Grid Asia*, Biopolis, Singapore, 2005.

[324] C. Yeo and R. Buyya, "A taxonomy of market-based resource management systems for utility-driven cluster computing," University of Melbourne, Melbourne, Victoria, Australia, Tech. Rep. GRIDS-TR-2004-12, GCDS Laboratory, 2004.

[325] "YourKit Java Profiler." [Online]. Available: http://www.yourkit.com/

[326] J. Yu and R. Buyya, "A Taxonomy of Workflow Management Systems for Grid Computing," in *J. Grid Computing*. Springer, September 2005, vol. 3, no. 3-4, pp. 171–200.

[327] D. Zhou and V. Lo, "Cluster Computing on the Fly: Resource Discovery in a Cycle Sharing Peer-to-Peer System," in *Proc. GP2PC*, Chicago, IL, USA, April 2004.

[328] A. Zissimos, K. Doka, A. Chazapis, , and N. Koziris, "GridTorrent: Optimizing data transfers in the Grid with collaborative sharing," in *Proc. Panhellenic Conference on Informatics*, Patras, Greece, May 2007.

Appendix A

Application and Service Interfaces

A.1 Grid Application Interface

The `set*()` operations of a Grid application are guaranteed to be all activated (in arbitrary order) prior to its execution through `compute()`. After `compute()` has returned, `getResult()` is activated.

`setInputData()`	set input data files
`setParameters()`	set application parameters
`setSupplier()`	communicate the ID of the supplier Peer
`setPlaypen()`	communicate the path of the playpen
`compute()`	activate the computing of the Grid application
`getResult()`	obtain a reference to the output data

Table A.1: Grid application interface.

A.2 Grid Node Service Interfaces

Grid node operations can both be implemented asynchronously and fail without blocking[1] system-level Grid operations.

`notifyCompletedTask()`	send results (output data file) of a completed Task
`notifyCompletedJob()`	send results (output data file) of the last completed Task of a BoT

Table A.2: User Agent Service interface.

[1] Although it is often beneficial to retry the operation at a later time.

`isAlive()`	simple liveness test
`idle()`	query the current Resource status
`runTask()`	ask the Resource to compute a Task
`taskStartTime()`	query the start time of the currently running Task
`cancelTask()`	ask the Resource to cancel the currently running Task
`setWorkingSet()`	communicate a working set (see Section 5.2.5) to the Resource
`cacheCapacity()`	query the total cache capacity of the Resource

Table A.3: Resource Service interface.

`isAlive()`	simple liveness test
`requestSupplying()`	communicate a supplying request
`grantConsumption()`	communicate a consumption grant
`submitSupplyingTask()`	ask the Peer to compute a Consumption Task (which is perceived as a Supplying Task)
`uploadBySupplyingCompletedTask()`	upload results of a completed Supplying Task submitted by the Peer
`cancelLocalTaskOnSupplyingResource()`	tell the Peer one of its Consumption Task has been cancelled
`cancelSupplyingTaskOnLocalResource()`	tell the Peer that it can safely cancel a Supplying Task

(a) External Peer Service interface.

`addResource()`	register with the Peer
`removeResource()`	unregister from the Peer
`uploadLocallyCompletedTask()`	upload results of a completed Task to the Peer
`preemptLocalTaskOnLocalResource()`	tell the Peer that the running Local Task had to be preempted
`preemptSupplyingTaskOnLocalResource()`	tell the Peer that the running Supplying Task had to be preempted

(b) Internal Peer Service interface.

`submitJob()`	submit a Bag of Tasks to the Peer
`cancelJob()`	cancel a Bag of Tasks from the Peer

(c) User Agent Peer Service interface.

Table A.4: Peer Service interface.

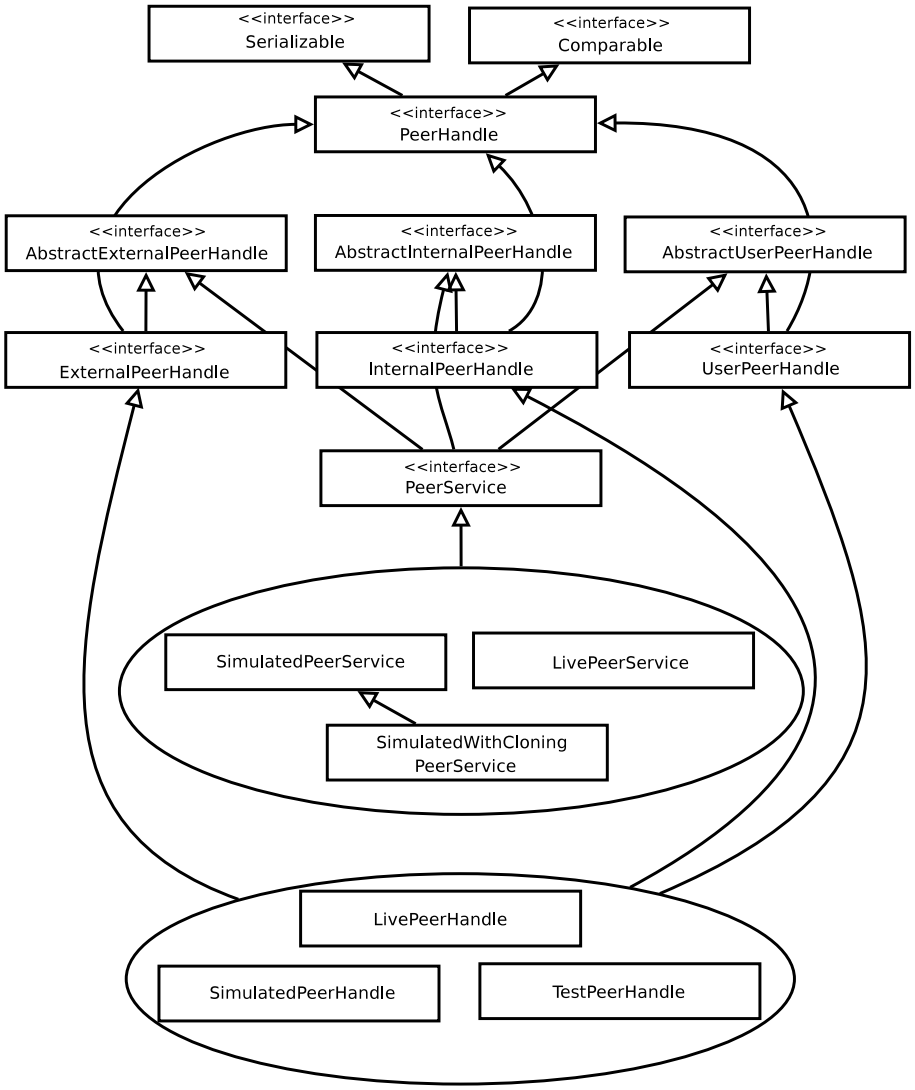


Figure A.1: Peer handle/service class hierarchy.

Figure A.1 illustrates the Peer handle/service class hierarchy. To simplify the diagram, entities with identical relationships have been grouped in circled areas. These are also actually the only entities that are classes; all other entities are interfaces.

The core of the class hierarchy is constituted by the PeerService, ExternalPeerHandle, InternalPeerHandle and UserPeerHandle interfaces. Each of these 3 PeerHandle subinterfaces is used by the corresponding type of Grid node (Peer, Resource, User Agent) to communicate with a Peer through a Peer service. The abstract interfaces layer has been introduced to guarantee that all Peer handle and service interfaces implement all the required operations (presented in Table A.4).

Appendix B

Configuration Languages and Parameters

B.1 Job Description Language

The job description language (JDL) is described and a file example is now given.

B.1.1 JDL BNF Grammar

```
<job description file> ::= <job>

<list of separators> ::= <list of separators> <separator> | <separator>

<job> ::= <header> <list of separators> <bag of tasks>

<header> ::= task_list = <list of labels>

<bag of tasks> ::= <bag of tasks> <list of separators> <task> | <task>

<list of labels> ::= <list of labels> , <label> | <label>

<label> ::= <string>

<task> ::= <jar property> <list of separators>
           <main class property> <list of separators>
           <parameters property> <list of separators> <data property>

<jar property> ::= <label>.jar = <string>

<main class property> ::= <label>.main_class | <label>.main_class = <string>

<parameters property> ::=
   <label>.parameters | <label>.parameters = <list of labels>

<data property> ::= <label>.datas | <label>.datas = <list of labels>
```

To use the Java Properties API [178] to parse job description files, separators are defined as line feeds (i.e. \n). Moreover, #-starting comments are also allowed and more spacing can be added but we chose readability over comprehensiveness.

B.1.2 Job Description File Example

```
# .jdf example

task_list = my_task1, my_task2, my_task3, my_task4, my_task5

my_task1.jar = gisapp.jar
my_task1.main_class = gis.algo.acc.DGSP
my_task1.parameters = 0.42, 19, 1600
my_task1.datas = acc.asc, barr.asc, mnt.asc, pop.asc

my_task2.jar = gisapp.jar
my_task2.main_class = gis.algo.acc.DGSP
my_task2.parameters = 0.42, 19, 1800
my_task2.datas = acc.asc, barr.asc, mnt.asc, pop.asc

my_task3.jar = gisapp.jar
my_task3.main_class = gis.algo.acc.DGSP
my_task3.parameters = 0.42, 18, 2000
my_task3.datas = acc.asc, barr.asc, mnt.asc, pop.asc
my_task4.jar = gisapp.jar
my_task4.main_class = gis.algo.acc.DGSP
my_task4.parameters = 0.42, 18, 2200
my_task4.datas = acc.asc, barr.asc, mnt.asc, pop.asc

my_task5.jar = gisapp.jar
my_task5.main_class = gis.algo.acc.DGSP
my_task5.parameters = 0.42, 20, 2500
my_task5.datas = acc.asc, barr.asc, mnt.asc, pop.asc
```

B.2 Simulation Description Language

A scenario describes the parameters of a simulation. It is written in the simulation description language (SDL), which is described. A file example is then given.

B.2.1 SDL BNF Grammar

The main BNF rules are provided again for convenience. The legal keys and values of specific rules are then systematically presented.

```
<scenario description file> ::= <list of scenario properties>

<list of separators> ::= <list of separators> <separator> | <separator>

<list of scenario properties> ::=
   <list of scenario properties> <list of separators> <scenario property> |
   <scenario property>

<scenario property> ::= <key> = <value>

<key> ::= < reserved keyword>

<value> ::= <scalar value> | <vector value>

<scalar value> ::= <boolean> | <int> | <float> | <string>

<vector value> ::= { <list of scalar values> }

<list of scalar values> ::= <list of scalar values> , <scalar value> |
                            <scalar value>
```

Vector values in scenario properties have been introduced to support the concept of Peer group (see Section 3.5.3). In a given scenario, vector values all have the same length and the i^{th} element of a given vector value corresponds to the i^{th} Peer group. The size of (i.e. the number of Peers in) each Peer group, as well as a common Peer identifier prefix for each Peer group, have to be defined. The number of Peer groups is implicitly defined by the length of the vectors in the scenario, which must all be identical.

To use the Java Properties API [178] to parse simulation description files, separators are defined as line feeds (i.e. \n). Moreover, #-starting comments are also allowed and more spacing can be added but we chose readability over comprehensiveness.

Grid Configuration

The Grid configuration describes the Grid nodes. Of interest are the computational power of Peers and the Resources reliability.

- PEER_BASE_NAME = <string>
- PEER_GROUP_SIZE = {<int>,...}
- PEER_POWER = {<int>,...}
- RES_COUNT = {<int>,...}
- RES_POWER_LO = {<int>,...}
- RES_POWER_HI = {<int>,...}

- RES_CACHE_CAPACITY = {<int>,...}
- RES_MTBF = {<int>,...}
- SEARCH_TIMEOUT = <int>

User Agents Configuration

The configuration of User Agents enables to generate synthetic workloads.

- JOBS_PER_PEER = {<int>,...}
- JOB_INIT_SHIFT = {<int>,...}
- JOB_INTER_TIME_LO = {<int>,...}
- JOB_INTER_TIME_HI = {<int>,...}
- TASKS_PER_JOB = {<int>,...}
- TASK_LEN_LO = {<int>,...}
- TASK_LEN_HI = {<int>,...}
- DATA_PER_TASK = {<int>,...}

Peers Scheduling Policies Configuration

The configuration of the scheduling policies and admissible queue length parameters determines the behavior of the Scheduler.

- PEER_SCHEDULING_LOCAL_POLICY =
  { NonpreemptiveLocalScheduling | PreemptiveLocalScheduling,...}
- PEER_PREEMPTION_SUPPLYING_RUNNING_POLICY =
  { NoPreemption | AdaptivePreemption | FullCancellation | FullPreemption |
  LimitedCancellation | LimitedPreemption,...}
- PEER_PREEMPTION_SUPPLYING_WAITING_POLICY =
  { NoPreemption | NoWaitingPreemption | FavorsWaitingPreemption |
  FIFOWaitingPreemption | FullWaitingPreemption,...}
- PEER_FILTERING_SUPPLYING_POLICY =
  { NoSupplyingFiltering | FIFOSupplyingFiltering |
  RelaxedFavorsSupplyingFiltering | StrictFavorsSupplyingFiltering |
  UnlimitedSupplyingFiltering,...}
- PEER_SCHEDULING_SUPPLYING_POLICY =
  { NoSupplyingScheduling | FIFOSupplyingScheduling |
  FavorsSupplyingScheduling,...}
- PEER_SCHEDULING_CONSUMPTION_POLICY =
  { NoConsumptionScheduling | AdaptiveMC4ConsumptionScheduling |
  AdaptiveTCoRConsumptionScheduling |
  ConservativeTimeStrideConsumptionScheduling |
  DataConsumptionScheduling | FavorsConsumptionScheduling |
  GrantsConsumptionScheduling | MC4ConsumptionScheduling |
  MCoSConsumptionScheduling | MTBCConsumptionScheduling |
  MTTCConsumptionScheduling | PerformanceConsumptionScheduling |
  RandomConsumptionScheduling | ReliableConsumptionScheduling |
  TCaRConsumptionScheduling,...}

- PEER_Q_FILTERING_THRESHOLD = {<int>,...}
- LOCAL_Q_LEN_PREEMPTION_THRESHOLD = {<int>,...}
- PEER_PSUFFERAGE = {<boolean>,...}
- DO_CONSUMPTION_BLACKLIST = {<boolean>,...}
- CONSUMPTION_BLACKLIST_PROBA = {<int>,...}

Peers Negotiation Policies Configuration

The configuration of the negotiation policies determines the behavior of the Negotiator.

- PEER_NEGOTIATION_SUPPLYING_POLICY =
  { NoSupplyingNegotiation | RandomSupplyingNegotiation |
  FavorsSupplyingNegotiation | UnlimitedSupplyingNegotiation,...}
- PEER_NEGOTIATION_CONSUMPTION_POLICY =
  { NoConsumptionNegotiation | RandomConsumptionNegotiation,...}
- PEER_DEFAULT_ACCOUNTANT =
  { NoEvalAccountant | OGPerfectAccountant |
  OGTimeAccountant | OGRelativePowerAccountant |
  LBGTimeAccountant | LBGRelativePowerAccountant,...}

Peers Negotiation Control Configuration

The configuration of negotiation timers and thresholds determines the behavior of the negotiation protocol.

- REQUESTORS_COUNT_THRESHOLD = {<int>,...}
- REQUESTORS_TIME_THRESHOLD = {<int>,...}
- GRANTORS_COUNT_THRESHOLD = {<int>,...}
- GRANTORS_TIME_THRESHOLD = {<int>,...}
- REQUEST_SUPPLYING_TIME_THRESHOLD = {<int>,...}

Peers Task Control Configuration

The configuration of a time-out for Consumption Tasks is required for the Task control mechanism.

- CONSUMPTION_TIMEOUT = {<int>,...}

Peers Data Management Configuration

Data management operations are simulated, while data transfers operations are not. The configuration of data management policies (and previously of a number of input data files per Task in the User Agents configuration) thus enables to test the correctness of data management operations.

- PEER_TTG_POLICY = {<boolean>,...}
- PEER_STORAGE_AFFINITY = {<boolean>,...}
- PEER_DATA_REPLICATION = {<boolean>,...}
- PEER_IDLE_REPLICATION_RATIO = {<float>,...}

Simulator Configuration

- SIMULATION_SEED = <int>
- CLONE_SIMULATED_TRANSFERRED_OBJECTS = <boolean>

B.2.2 Simulation Description File Example

```
# .sdf example:  corresponds to the 4-Peers scenario proposed in
# Andrade, Brasileiro and Cirne, Automatic Grid Assembly [...], JPDC, 2007

PEER_BASE_NAME = og_peer_
PEER_GROUP_SIZE = { 1, 1, 1, 1 }
PEER_POWER = { 4, 4, 4, 4 }
PEER_TTG_POLICY = { true, true, true, true }
PEER_STORAGE_AFFINITY = { true, true, true, true }
PEER_DATA_REPLICATION = { false, false, false, false }
PEER_IDLE_REPLICATION_RATIO = { 0.0, 0.0, 0.0, 0.0 }
PEER_PSUFFERAGE = { true, true, true, true }
PEER_Q_FILTERING_THRESHOLD = { 1, 1, 1, 1 }
RES_COUNT = { 4, 4, 4, 4 }
RES_POWER_LO = { 1, 1, 1, 1 }
RES_POWER_HI = { 1, 1, 1, 1 }
RES_CACHE_CAPACITY = { 24, 24, 24, 24 }
RES_MTBF = { 0, 0, 0, 0 }
CONSUMPTION_TIMEOUT = { 604800, 604800, 604800, 604800 }
LOCAL_Q_LEN_PREEMPTION_THRESHOLD = { 12, 12, 12, 12 }
DO_CONSUMPTION_BLACKLIST = { false, false, false, false }
CONSUMPTION_BLACKLIST_PROBA = { 0.0, 0.0, 0.0, 0.0 }

PEER_SCHEDULING_LOCAL_POLICY =
      { PreemptiveLocalScheduling, PreemptiveLocalScheduling,
      PreemptiveLocalScheduling, PreemptiveLocalScheduling }
PEER_PREEMPTION_SUPPLYING_RUNNING_POLICY =
      { LimitedCancellation, LimitedCancellation,
      LimitedCancellation, LimitedCancellation }
PEER_PREEMPTION_SUPPLYING_WAITING_POLICY =
      { FullWaitingPreemption, FullWaitingPreemption,
```

```
      FullWaitingPreemption, FullWaitingPreemption }
PEER_FILTERING_SUPPLYING_POLICY =
      { FIFOSupplyingFiltering, FIFOSupplyingFiltering,
      FIFOSupplyingFiltering, FIFOSupplyingFiltering }
PEER_SCHEDULING_SUPPLYING_POLICY =
      { FavorsSupplyingScheduling, FavorsSupplyingScheduling,
      FavorsSupplyingScheduling, FavorsSupplyingScheduling }
PEER_SCHEDULING_CONSUMPTION_POLICY =
      { DataConsumptionScheduling, DataConsumptionScheduling,
      DataConsumptionScheduling, DataConsumptionScheduling }

PEER_NEGOTIATION_SUPPLYING_POLICY =
      { FavorsSupplyingNegotiation, FavorsSupplyingNegotiation,
      FavorsSupplyingNegotiation, FavorsSupplyingNegotiation }
PEER_NEGOTIATION_CONSUMPTION_POLICY =
      { RandomConsumptionNegotiation, RandomConsumptionNegotiation,
      RandomConsumptionNegotiation, RandomConsumptionNegotiation }

PEER_DEFAULT_ACCOUNTANT =
      { LBGRelativePowerAccountant, LBGRelativePowerAccountant,
      LBGRelativePowerAccountant, LBGRelativePowerAccountant }

REQUESTORS_COUNT_THRESHOLD = { 12, 12, 12, 12 }
REQUESTORS_TIME_THRESHOLD = { 1, 1, 1, 1 }
GRANTORS_COUNT_THRESHOLD = { 12, 12, 12, 12 }
GRANTORS_TIME_THRESHOLD = { 1, 1, 1, 1 }
REQUEST_SUPPLYING_TIME_THRESHOLD = { 1, 1, 1, 1 }

JOBS_PER_PEER = { 60, 60, 60, 60 }
JOB_INIT_SHIFT = { 0, 0, 0, 0 }
JOB_INTER_TIME_LO = { 60, 60, 60, 60 }
JOB_INTER_TIME_HI = { 1200, 1200, 1200, 1200 }

TASKS_PER_JOB = { 40, 40, 40, 40 }
TASK_LEN_LO = { 60, 60, 60, 60 }
TASK_LEN_HI = { 60, 60, 60, 60 }
DATA_PER_TASK = { 1, 1, 1, 1 }

SIMULATION_SEED = 42
SEARCH_TIMEOUT = 10
CLONE_SIMULATED_TRANSFERRED_OBJECTS = true
```

B.3 Nodes Configuration Language

As there are a large number of parameters to configure Peers and Resources - even if many of them can be assigned well-known standard values - it is useful to have Grid nodes read their configuration from a file. The node configuration language is a subset of the scenario description language. This facilitates the writing of nodes

configuration files from simulation description files, and reciprocally. The availability of a simple, properties-based language also enables one to easily automate the deployment of multiple Grid nodes.

Appendix C

Peer Middleware Internals

C.1 Peer Components Dependencies

Figure C.1 presents a synoptic view of Peer components [59] (see Section 2.9.7). They manage Tasks queueing, Resource management, Task execution control, data storage/sharing/transfer, negotiation, scheduling, collection/storage of metadata, communications with Grid nodes, Peer discovery.

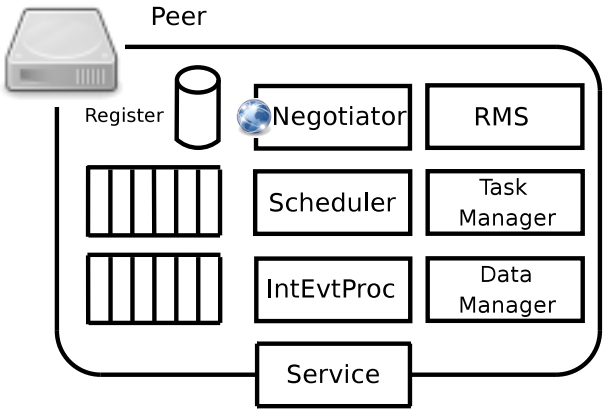


Figure C.1: Peer components.

In particular, the Task Manager is the Peer component responsible for the execution of Tasks on Resources. It is essentially a connector component used by the Scheduler (Section 2.9.4) and by the Peer Service (Section 2.8). Supported Task Manager operations include Task execution, completion, preemption/cancellation and control. It relies on the RMS (Section 2.7.6) and Queue Manager (Section 2.9.3.

Table C.1 represents the dependencies between Peer components (see Section 2.9). An arrow symbolizes a dependency of the component in the left column on the component in the top row, i.e. the former uses the latter. Components without dependency are not included.

	Data Manager	Peer Register	Negotiator	Queue Manager	Resource Manager	Search Engine Client	Scheduler	Task Manager
Peer Register		↗				↗		
Negotiator		↗	↗	↗	↗	↗		↗
Queue Manager		↗		↗				
Service	↗	↗	↗	↗	↗		↗	↗
Scheduler		↗	↗	↗	↗		↗	↗
Task Manager		↗		↗	↗			↗

Table C.1: Peer components dependencies.

C.2 Peer Events to Policies Mapping

Table C.2 shows the mapping of Peer events (see Section 2.9.2, Appendix A.2 and Section 2.9.6) to policies (see Sections 2.9.4 and 2.9.5) The numbers in each row denote the ordering in which operations are triggered for the corresponding event.

Operations corresponding to each policy are first listed:

- Local Tasks scheduling: `scheduleLocalTasks()`
- Consumption Tasks scheduling: `scheduleConsumptionTasks()`
- Supplying Tasks filtering: `filterOutSupplyingTask()`
- Supplying scheduling: `scheduleSupplyingTasks()`
- Supplying Tasks preemption: `preemptLocalResources()` (called by `scheduleLocalTasks()`)
- supplying requests evaluation: `evaluateSupplyingRequests()`
- consumption grants evaluation: `evaluateConsumptionGrants()`

	scheduleLocalTasks()	scheduleSupplyingTasks()	scheduleConsumptionTasks()	filterOutSupplyingTask()	evaluateSupplyingRequests()	evaluateConsumptionGrants()
submitJob()	1					
cancelJob()	1	2	3			
uploadLocallyCompletedTask()	1	2				
preemptLocalTaskOnLocalResource()	1	2				
preemptSupplyingTaskOnLocalResource()	1	2				
requestSupplying()					1	
grantConsumption()						1
submitSupplyingTask()		2		1		
uploadBySupplyingCompletedTask()			1			
cancelLocalTaskOnSupplyingResource()	1		2			
cancelSupplyingTaskOnLocalResource()	1	2				
addResource()	1	2				
removeResource()	1	2	3			

(a) external events to Peer operations mapping

	scheduleLocalTasks()	scheduleSupplyingTasks()	scheduleConsumptionTasks()	evaluateSupplyingRequests()	generateSupplyingRequests()
received Supplying Requests time-out				1	
received Consumption Grants time-out			1		
sent Consumption Requests time-out					1
nonempty queues after scheduling					1
Local Tasks time-out	1	2			
Consumption Tasks time-out	1		2		
Supplying Tasks time-out	1	2			

(b) internal events to Peer operations mapping

Table C.2: (a) External events to Peer operations mapping, (b) internal events to Peer operations mapping.

C.3 Peer Internal Events Processing

The internal events processor (illustrated on Figures 2.26 and C.1) is the Peer component that is responsible for the processing of the internal events. There are three classes of internal events, each with some dedicated timers:

- generation of supplying requests (1 timer), as long as some Local Tasks remain unscheduled,
- evaluation of timed-out standing negotiation events (2 timers),
- Task control (3 timers).

Thus, the internal events processor manages a total of six timers.

C.3.1 Frequency of Internal Events Processing

Internal events are triggered after a time-out has been exceeded in the relevant timer. A temporal resolution has to be defined for these timers.

We have selected a temporal resolution of one second per time step because it can be argued that finer temporal resolutions would be of little use. Indeed, a P2P Grid is a large-scale networked environment targeting long-running Tasks, very few of which are likely to be completed within one second, even on high-end computers.

C.3.2 Storage of Internal Events (Negotiation)

Three integer numbers representing timestamps are sufficient to manage the timers required for negotiation (one for the emission of additional supplying requests, one for the evaluation of received supplying requests, one for the evaluation of received consumption grants).

C.3.3 Storage of Internal Events (Task Control)

Data about internal events related to Task control are stored into three auxiliary data structures (in practice: balanced binary trees). Each structure is dedicated to one type of Task (Local, Consumption and Supplying Tasks). The data stored in these structures are (references to) Tasks. The keys are timestamps of time-out expiration and are ordered by increasing value, i.e. soonest time-out first.

Checking which Tasks have timed out in one of the three given data structures consists in walking this data structure from the first internal event until an internal

event with a timestamp in the future has been reached. The temporal cost of this operation is either constant (when there is no timed-out Task) or proportional to the number of timed-out Tasks and to the cost of tree rebalancing (which happens after timed-out Tasks have been removed from the data structure).

There is an additional overhead to maintaining these data structures: Tree rebalancing is required whenever a Task is stored or removed, which happens upon Task scheduling, completion, preemption or cancellation. Maintaining a unified list of internal events would require to merge the three data structures. Should the update of data structures related to Task control become a performance bottleneck, we could evaluate whether it is worthwhile to maintain a unified data structure instead of three separate data structures.

C.3.4 Multithreading of Internal Events Processing

Every time step, three boolean guards related to negotiation internal events must be checked, as well as the three data structures related to Task control. The internal events processor runs currently in its own separate thread (see Section 2.9.7).

Using multiple threads to run the internal events processor would bring very limited benefits. Indeed, out of the six internal events processing operations, the three related to negotiation are straightforward. This limits the number of potentially useful threads to three.

As the total number of Tasks to time-out would certainly be inferior to the number of seconds in a Peer lifetime, walking the three data structures for an extended number of data elements would be infrequent. Moreover, this operation is often limited to the inspection of the first data element, as its associated timestamp is in the future at the time of the verification. Consequently, multithreading is not used in the implementation of internal events timer management.

In practice, the Java periodic execution service [178] (ScheduledExecutorService, available in java.util.concurrent), is used to activate the internal events processor every second.

Appendix D

Virtual Organizations

D.1 Virtual Organizations

D.1.1 What is a VO?

Foster and Kesselman introduced informally the concept of *Virtual Organization* (VO) [150], to position the sharing of Resources between separate administrative domains within a socioeconomic context. A VO encompasses multiple administrative domains exchanging computing time through Grid middleware, externally appearing as a single administrative entity. It follows that VO can be recursively defined (see Figure D.1).

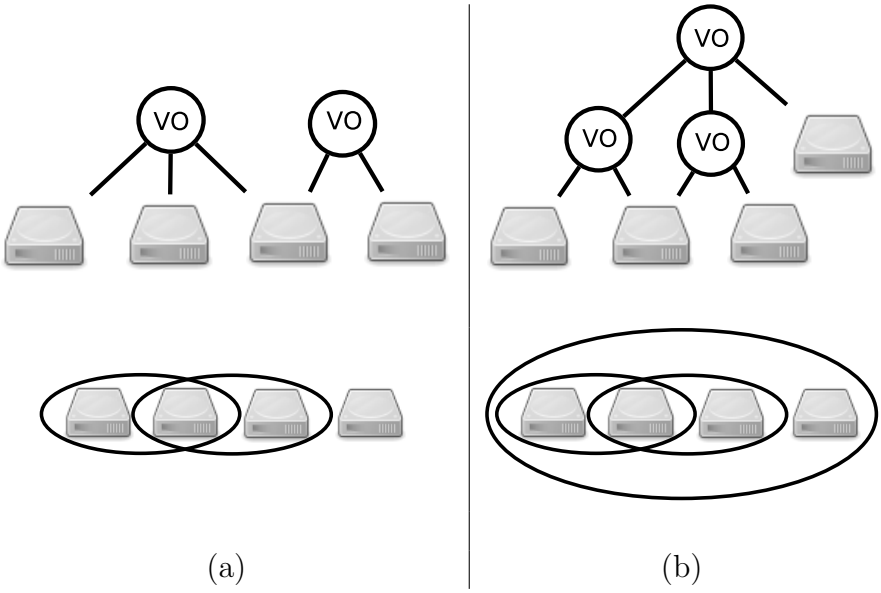


Figure D.1: Structure of Virtual Organizations: (a) Multiple VO membership, (b) Multiple VO membership, with Recursive VO.

D.1.2 VO Formation

VO formation is the process of creating relationships between the VO members, through software agents such as Peers. Mechanisms addressing the dynamic formation of a VO are thus important and constitute a subdomain [225, 148, 241, 86, 242] of Grid computing. There are basically two ways to form a VO [10]: top-down and bottom-up.

Top-down VO formation

Top-down VO formation is the formation of a VO initiated by stakeholders who want to control a Grid.

Top-down VO formation is necessarily initiated out-of-Grid because it requires a priori knowledge of the Peers forming the VO. A centrally coordinated architecture would usually be selected for top-down VO formation because it would be in the interest of rational human stakeholders to create a VO from scratch with mostly stable Resources. Most current production Grids are created top-down. The Belgian national research Grid, BEgrid [40], or the French national research Grid, Grid 5000 [72], are examples of Virtual Organizations created top-down.

Bottom-Up VO formation

Bottom-up VO formation is the formation of a VO initiated by stakeholders who want to be part of the same Grid without having prior knowledge of one another.

Bottom-up VO formation is necessarily initiated by Peers. An individually coordinated architecture would usually be selected because this bottom-up formation is the scenario of choice for Peers among which there is no or little trust and where human administrators are not related, thus implicating a possibly unstable Resource environment. P2P Grids are the canonical example of bottom-up VO formation. The consequences of the requirement of an automatic Peer discovery service in the definition of a P2P Grid (Section 1.1.4) are now fully clear: An automatic Peer discovery service, independent of the Peers, is needed to enable bottom-up VO formation. Indeed, manually giving Peers some addresses of other Peers is equivalent to forming a VO out-of-Grid, in a top-down fashion.

D.2 Peer Discovery

Peer discovery is an important feature of P2P Grids. It enables Peers to meet online without prior knowledge of one another, and to immediately start exchanging computing time. Peer discovery enables bottom-up VO formation.

It is clear that, to achieve high performance, Peers should maintain a cache of known Peers so as to limit the number of interactions with the Peer discovery service. Each time a Peer wants to consume Resources from another Peer, it should first consult its cache of known Peers to select some Peers to start negotiating with.

As the initial vision for P2P Grids is recent [147], P2P Grid research has first concentrated on decentralized scheduling, fault-tolerance, trust issues, ... with P2P Grid research focused on Peer discovery lagging behind. The reason is probably that it was assumed that Peer discovery is a pure P2P issue, and thus research work conducted in the P2P domain could be applied directly to P2P Grids.

The current Peer discovery mechanism in OurGrid [52], relies on a centralized directory, CorePeer. Peers register themselves with one CorePeer as they come online. Several CorePeers can exist simultaneously to support multiple independent Grids. However, in the future, it can be imagined that Peers may register themselves with several CorePeers, thus effectively becoming members of several VO (see Section D.1.2).

Tree-based overlay networks constitute another mechanism enabling Peer discovery. OurGrid's future Peer discovery service, NodeWiz [53], and THON [174] are examples of fault-tolerant, self-organizing tree-based overlays in P2P Grids.

DHT-based (Distributed Hash Tables) overlay networks constitute yet another common Peer discovery mechanism. JNGI [300, 183] is an example of P2P Grid using the JXTA platform [185], which is based on a loosely consistent DHT.

A centralized Peer discovery architecture is clearly not scalable. Nonetheless, the current implementation of Peer discovery [99], which is centralized, is sufficient to conduct research on the other P2P Grid topics discussed in this dissertation. It can be easily replaced with a more scalable mechanism, such as tree-based [53] or DHT-based overlays [232], or gossiping, or other mechanisms coming from research conducted in the P2P networks community.

As it is possible that some Resources will always be incompatible with some computational requests, e.g. due to limitations of available RAM, an initial filtering of unsuitable Peers during Peer discovery can be very useful. In the near future, Peer discovery will be augmented with some form of matchmaking [305, 53], also called the *"double coincidence of wants"*. This will introduce another layer of negotiation, as this does not remove the benefit to identify reliable Peers.

Matchmaking mechanisms essentially take into account static properties of Resources (such as CPU count or maximum storage space) but also some dynamic properties (such as CPU load). Early matchmaking mechanisms have followed a centralized organization. A well-known example is ClassAds [255], which frames Resource filtering

as a bilateral matching process. By relying on Distributed Hash Tables [232] (DHT), matchmaking mechanisms have gained in fault-tolerance and scalability but lost in the richness of queries (which are limited to exact matches). By replacing a DHT substrate with a tree-based P2P-structured substrate enhanced for fault-tolerance, it is possible to recuperate support for rich queries. A recent example is NodeWiz [53]. Explicit matchmaking of supplying and consumption of computing time has also been studied [237] from a theoretical graph theory perspective.

D.3 Bartering Policies

Several Bartering policies [61] (Philanthropy, Mutualism and Individualism), are classified according to their level of expectation of reciprocity.

D.3.1 Philanthropy

A *philanthropic Resource sharing policy* is defined as a policy based on a centralized coordinator that maximizes the utility of one Peer (usually itself) without giving any utility to the other Peers (which are the "philanthropists"). No accounting of Resource sharing (which is unilateral) is kept.

The main benefit of Philanthropy is the aggregation of huge amounts of Resources that allows one Peer to run large-scale applications. Philanthropic Resource sharing is the policy typically used in Volunteer Computing [51, 12, 273, 143], where most Peers are supplying computing time and one Peer consumes it.

D.3.2 Mutualism

Generally speaking, a mutualistic organization is created to provide its members with the best possible service and maximum return on investment, without keeping any benefit for itself. This kind of business may even be owned by its members.

A *mutualistic Resource sharing policy* is defined as a policy based on a centralized coordinator that globally maximizes Peers utility and Resource utilization, without keeping long-term Resource sharing accounting.

Members of mutualistic organizations are not expecting to be compensated for all their supplying of computing time, but expect instead to get some proportional compensation (i.e. schedule priority). It really is load balancing.

The main benefit of mutualism is that if one Peer suffers some trouble (e.g. Resource failure, transient request overload) and cannot supply enough Resources for

some time, it is still able to consume Resources, but less than in usual operating conditions. When a few Peers face peaks in their Resource and/or request environment, mutualism enables load balancing between all Peers. However, all the other Peers are penalized but as the burden will be equally shared, they are able to consume Resources in only a slightly smaller amount than they could in usual operating conditions. It can be seen as a form of fault-tolerance where performance penalties are shared among components in the system.

Another important aspect is that the Peer that receives help from the other Peers is not penalized because there is no long-term accounting of Resource sharing. Such a policy is therefore highly suggested when:

- There is strong trust between Peers (e.g. Peers belong to the same enterprise or association), and
- The total amount of consumed Resources within the VO is smaller than the total amount of supplied Resources jointly consumed by the Peers.

Indeed, if there is no trust, there is a high risk of free riding [149]. And if there are not enough idle Resources, the form of redundancy proposed by a mutualistic policy is not possible.

A mutualistic policy is typically used in Desktop Grids, i.e. an enterprise-level Grid where cycle stealing is performed on idle desktop PC. There is a VO with several Peers representing the various departments/units of the firm. These Peers share their Resources with a mutualistic policy.

D.3.3 Individualism

An *individualistic Resource sharing policy* is defined as a policy based on either distributed coordinators or a centralized coordinator that maximize Peers utility and maintain long-term Resource sharing accounting. With an individualistic policy, an accounting of Resource sharing is maintained independently by each Peer. Peers can then consume as many Resources as they supply and do not have to supply more than they consume. The goal of the individualistic policy is to separate the concerns of the Peers and maximize their utility independently.

The main benefit of an individualistic policy is the total avoidance of free riding, which depends upon the accuracy of Resource sharing accounting [266, 265]. This policy incites the Peers to supply Resources as they know that undue overconsumption of computing time by other Peers is limited.

A limitation of an individualistic policy is that it must be augmented with a bootstrapping policy [82] breaking the initial symmetry in the lack of trust between

Peers. Indeed, if all the Peers wait to have consumed some computing time before supplying some of their own, no exchange take places. Every Peer remains idle, ridden by fear of being free ridden.

An option to overcome an initial lack of trust consists in Peers randomly accepting a small, yet nonzero, percentage of supplying requests from Peers that do not have a good Resource sharing history.

It can be hypothesized that after stability in the Resource sharing driven by an individualistic policy has been achieved and maintained for some time, Peers could consider switching to a mutualistic policy. This would guarantee that a Peer experiencing transient abnormal conditions, precluding it from supplying computing time, would be helped by other Peers. The goal is to enable the preservation of stable Resource sharing patterns, which can be seen as a form of robustness [210].

D.3.4 Network of Favors

Another policy, adopted in P2P Grids [13, 84], consists in Peers accepting all the supplying requests as long as there are no pending requests to supply Peers with higher priority, i.e. better Resource sharing history. Resource utilization is thus promoted as idle Resources are supplied in order to build trust with other Peers.

Free riding may indeed take place, but it is limited either to a small percentage of Resource utilization or to periods when Resource utilization is low anyways.

The Network of Favors can thus be seen either as a relaxed form of individualism, or as a decentralized form of mutualism.

D.3.5 Out-of-Grid Compensations

Bartering is not always possible either because Peers not controlling any Resources want to consume from other Peers, or because Peers want to monetize the supplying of computing time. The concept of import and export of Resources are now introduced to enable out-of-Grid compensations.

Out-of-Grid compensations include real money, feel-good[1], or an external agreement between Peers administrators. They can be combined with any Resource sharing policy previously reviewed.

[1] For example, arising from the deployment of Volunteer Computing middleware to offer computing time to e-Science experiments trying to advance medical research.

Import of Resources

Import of Resources may take place when some Peers do not own any Resource or have exhausted their consumption potential, thus preventing the supplying of Resources to compensate the consumption of Resources at peak time. Import of Resources is a form of Resource sharing reduced to a classic utility computing/ASP (Application Service Provider) scenario.

Amazon Elastic Compute Cloud [9] and Sun Grid Compute Utility [290] offer computing time for sale. In another context, scientific projects of general interest [273, 143] allow home users to supply their Resources against a feeling of taking part in a project useful to mankind. In yet another context, like cryptographic grand challenges [121], human administrators of a Grid may decide to lend access to their Resources to the administrators of another Grid, and therefore transiently share some Resources with another Grid.

A Peer may also offer an out-of-Grid compensation even if it can supply its own Resources, so as to maintain a high instantaneous consumption potential.

Export of Resources

Export of Resources arises in a centralized architecture when the Grid coordinator follows objectives of its own and suppliers its Resources to an out-of-Grid entity. It can be modelled as the application of both a philanthropic and another (mutualistic or individualistic) policy. After the philanthropic policy has been applied, the other policy is applied to the Resources that were not exported.

D.3.6 Autonomous VO Management

To achieve fully autonomous Virtual Organizations, multiple objectives regarding their life cycle should be balanced [177]:

- Under what conditions is it profitable to allow a new VO member into a given VO, i.e. to start exchanging computing time? Instantaneous access to/aggregation of Resources is a main motivation to enter a VO. Thus VO members should associate with other VO members that have temporally complementary peaks of local requests.

- Under what conditions is it profitable for a VO and most of its members to keep sharing Resources with a Peer which regularly exhibits failures in the supplying of computing time? Under what conditions is it profitable for VO members to keep exchanging computing time when most VO members regularly exhibit failures in the supplying of computing time? With the Network of Favors model, the issue is not on the supplying side (which is

tolerant to free riding) but on the consumption side (if Consumption Tasks keep getting cancelled).

- Is it desirable for consumer Peers to consume from a large number of suppliers at the same time? This broadens the search for faster Peers. This also mitigates the impact of the supplying failures. For parameter sweeps applications and our proposed data transfer architecture, this leads to a large number of simultaneous BitTorrent downloaders.

- Is it desirable for supplier Peers to supply to a large number of consumers at the same time? This mitigates the impact of supplying to Peers that are not able to reciprocate proportionally.

- Is it desirable for consumer Peers to consume from a small number of suppliers at the same time? This enables to reciprocate to most suppliers, i.e. to prevent any implicit free riding resulting from the impossibility to supply to too many Peers at the same time.

- Is it desirable for supplier Peers to supply to a small number of consumers at the same time? This enables the suppliers to significantly increase their ranking with a few Peers instead of increasing it only a little with a large number of Peers. This also enables the suppliers to devote the network bandwidth of their Resources to a small number of files and to maintain small working sets of input data files.

Appendix E

Resource Negotiation

Research on autonomous Resource negotiation can be classified as follows [182]:

- negotiation protocols (how - communication perspective),
- negotiation objects (what is negotiated),
- Resource sharing mechanisms (how - processing perspective).

We argue that a fourth concern, negotiation objectives (when/why) should also be taken into account.

Resource sharing mechanisms have already been discussed in Section 2.3. Negotiation protocols, objects and objectives are now briefly discussed.

E.1 Negotiation Protocols

Negotiation protocols essentially define the structure - not the content - of the negotiation agreements. For example, the so-called Service Level Agreements (SLA) enable Grid participants to explicitly communicate the quality of service they expect [24, 307]: *"If the system is to have any type of predictable behavior, it becomes necessary to obtain commitments (contracts) about the willingness to provide a service and the characteristics, or quality, of its provision."* [148] Standardization of the protocols formatting the exchange of messages between trading partners is an important aspect of Resource negotiation. With standard protocols, negotiation data could be transformed, composed/decomposed, communicated between different Grid middlewares.

WS-Agreement is a protocol from a protocol stack proposed by the Open Grid Forum [231] to allow suppliers and consumers to negotiate Resources by means of SLA [170]. It is considered as an important step towards an automated Resource

negotiation service [148].

SNAP (Service Negotiation and Acquisition Protocol) is a protocol which provides lifetime management and at-most-once creation semantics for remote SLAs [96, 323]. It follows a classic client-server RPC pattern.

E.2 Standard Grid Protocols

Global interoperability of networks let to the emergence of Internet. Likewise, global interoperability of distributed systems is desirable. Foster's Three-Point Checklist [151] states that Grid middleware is based on standard, open, general-purpose protocols and interface. Standardization efforts are currently under way to provide standard definitions of Grid protocols and services.

The Open Grid System Architecture (OGSA) consists in *"the definition of a broadly applicable and adopted framework for distributed system integration, virtualization and management."* [229] OGSA defines a set of specifications concerning interfaces, behaviors, resource models and bindings. It provides an abstract definition of the set of requirements, which is based on many representative use cases.

In January 2004, the Web Services Resource Framework (WS-RF) has been announced in the Open Grid Forum [231] as a successor to OGSA. As implied by its name, WS-RF is the specification of a family of Web Services [26] originated by the Organization for the Advancement of Structured Information Standards (OASIS) group.

To gain in-depth insights about the history of Grid standards and software, the interested reader may read a recounting by von Laszewski [304].

As of 2008, although gaining momentum, WS-RF is not yet systematically adopted in Grid middlewares. Moreover, P2P Grid computing [84] is a recent subdomain of Grid computing [147], with interfaces yet to be mapped to Grid standards. Therefore some experimental systems exist today that exhibit all the characteristics of a Grid but do not implement standard protocols. They are thus restricted in global interoperability, despite offering Resource sharing. A relevant concept is that of lightweight Grid [286] (see Section 2.5), which either does not expose standard interfaces or does not completely support standardized features.

E.3 Negotiation Objects

A Peer seeks to stabilize its Resource environment through Resource sharing so as to exhibit a more predictable behavior. The goal of a Peer is to induce its trading partners to produce *"commitments (contracts) about the willingness to provide a service and the characteristics, or quality, of its provision."* [148] Contracts constitute an important concept and tool in Resource negotiation. In practice, a contract defining what Resources are supplied and on what terms can be detailed by a so-called Service Level Agreement [96] (SLA). Terms of an SLA can be define precisely with so-called utility functions [177, 206].

To enforce the terms of a contract resulting from Resource negotiation, there must be some form of contract monitoring [149], either centralized or autonomous. Monitoring the enforcement of contracts allows Peers to dynamically renegotiate or terminate them if they are breached or if Resource requirements of one of the trading partners change before contract completion [149].

An example of a recent architecture for Resource usage SLA specification and enforcement is GRUBER [127].

E.4 Negotiation Objectives

Not far from the concerns about negotiation objects are the concerns about negotiation objectives. However, despite being closely related, negotiations objectives should be distinguished from negotiation objects. Studying *what* Resources can be negotiated (which is the purpose of negotiation objects) is different from studying *when* and *why* these should be negotiated (which is the purpose of negotiation objectives).

The focus of a Peer can be application performance [149], system performance, user satisfaction [200, 270] or VO administrator satisfaction [270], maybe beyond what classic performance metrics such as average Resource utilization, average response time, average job completion, average job re-planning and workload completion time [126] can offer. In the long-term, whatever the focus, a multicriteria approach should prevail and take as many of them as possible into account, as the objective of a Peer is basically to automate scheduling and Resource management to *"minimize stakeholders' interventions."* [221, 270] A multicriteria approach seeks a *"compromise solution to increase the level of satisfaction of many stakeholders"* (i.e. VO members and administrators) *"and combine different points of view."* [221]

To process the incoming requests, a Peer has to produce Resource requirements. It also has to set Resource negotiation objectives, given both the produced Resources

requirements and the stakeholders-defined Peer focus. From this point of view, a Resource negotiation service can be said to be *responsive*.

There is however an additional perspective to be considered: A Peer can perform some Resource sharing without having any incoming request to service. From this perspective, the Resource negotiation service can be said to be *proactive*. The purpose of Negotiation service proactiveness is to accumulate some consumption potential for use at a later time. In other words, the Resource negotiation service can proactively acquire Resources for a period of time when it has predicted these would be needed soon (i.e. consuming Resources). It can also proactively supply Resources because it has predicted these would not be needed for some time (i.e. creating a potential of consumption of computing time) ... hoping they will be reciprocated later when they will be needed.

On one hand, while conflicting interests from multiple Peers may be hard to manage, heterogeneity of focus (i.e. different objectives) in a set of Peers has some advantage, after all. With homogeneous needs and assets and when there are tight deadlines to be met, load balancing between Peers naturally emerges but little Resource sharing takes place because each Peer keeps all its Resources committed to its own use first.

On the other hand, if there are different focuses, overall system utilization and application performance may both be simultaneously high. When a Peer has few requests to process, it can build consumption potential by supplying the computing time of its Resources, which promotes high system utilization. When a Peer has to meet strict deadlines, it can use the consumption potential it has previously built in order to reach high application performance.

Connecting the focus of a Peer to Resource requirements and negotiation objectives is a central problem. Once chosen, the Peer focus has to be translated into lower level requirements. Coupled with actual request data, the Peer focus has to be translated first into Resource requirements, then into negotiation objectives.

Related research include a study on translation of Resource requirements across abstraction layers [101] and the GrADSoft system [42] where the scheduler and negotiator are merged.

Appendix F

Future Work

The Lightweight Bartering Grid architecture presented in this dissertation enables to deploy and operate P2P Grids. In this Appendix, areas that require or that would benefit from future work are reviewed: scheduling, negotiation and Task models; data transfer and persistence; simulation; security.

F.1 Scheduling Model

F.1.1 Task Replication

Task replication consists in having each Peer schedule multiple replicas of Local Tasks. Cirne et al. [86] showed that Task replication is an efficient scheduling mechanism in P2P Grids. Task replication clearly brings fault-tolerance, but at a cost. However, Task replication is also known to considerably increase performance in endgames [107, 108], i.e. when scheduling the last few Tasks of a BoT.
Our proposed scheduling model could be extended to support Task replication, transparently to other scheduling operations.

F.1.2 Other Metrics for Consumption Tasks Scheduling

As suggested in Section 4.5, estimating the performance of suppliers and of a Peer's own Resources could be of high interest. Using the bartering reputation of other Peers when consuming (as opposed to currently using the bartering reputation of other Peers only when supplying, as proposed in the original Network of Favors model [13]), as well as estimating one Peer's own capacity of reciprocal supplying, may lead to excellent Consumption Tasks scheduling policies. In another perspective, the Network of Favors model (see Section 2.3.4) could be modified with a time window so that only the most recent interactions are accounted for, or so that the favor balance is computed as a time-discounted average of favors [29, 274].

F.1.3 Batch-Mode Scheduling

Knowledge-based scheduling policies can be classified as either online or batch-mode [215, 74, 71]. An online scheduling policy considers only one Task and tries to match it to a Resource. A batch-mode scheduling policy considers multiple Tasks at once; it can thus potentially achieve better Task-to-Resource matching.

One of our proposed Task selection algorithm for the Local and Supplying Tasks scheduling policies, Temporal Tasks Grouping (see Section 5.3.1), orders Tasks in batch-mode, but statically. It is currently not linked with the Resource selection policy. If runtime estimates were available (which is not the case in this dissertation, see Section 2.6.3), batch-mode scheduling could be fully applied to data-equal subsequences of Tasks.

F.1.4 Automatic Tuning of Task Control Parameters

The Task control operations (see Section 2.9.6) of a Peer consist in cancelling scheduled Tasks which have been running for so long that their runtime exceeds a given threshold. The purpose of Task control is to mitigate Denial of Service attacks, and to prevent Grid application bugs, i.e. infinite loops, to disrupt Peer operations.

Time-out values are currently configured by human Peer administrators. Automatic, adaptive estimation of time-out values would enable tighter Task control. There is however a trade-off to achieve, implying two risks: timing-out too late and timing-out too soon. Timing-out too late only decreases the efficiency of Task control, which might be acceptable but leads to question the usefulness of automatic estimation. Timing-out too soon inevitably leads to the loss of computing time due to the reexecution of unduly cancelled Tasks, except in the presence of checkpointing support. In this case, timed-out Supplying Tasks can be stopped, and returned to their consumer Peers. These have the possibility to resume the execution of the timed-out Tasks on their own Resources.

We could have incorporated automatic and adaptive time-out estimation in the current implementation of the LBG, but decided not to because of the current lack of checkpointing support. Finally, it must be noted that runtime estimates provided by human users might not be more efficient either, as they tend to be largely overestimated [205].

F.1.5 Classes of Priority and Urgent Computing

Support for so-called Urgent Computing [39] could be added. User Agents could tag their submitted BoTs with a priority flag, thus overriding the FIFO queueing

of Local BoTs. Such support for several classes of priority among submitted Local BoTs is very useful in practice. Moreover, it could be implemented in the Queuing Manager transparently to other Peer managers.

Support of user groups, or groups of User Agents, could be added on top of a priority mechanism. In this perspective, User Agents could also tag their submitted BoTs with a group identifier.

F.1.6 Exploration of the Grid Architecture Design Space

It can be hypothesized that the existing (OurGrid [233, 84, 286]) and proposed (Lightweight Bartering Grid) P2P Grid architectures probably do not cover the design space of possible P2P Grid architectures. It might therefore be interesting to support Grid architecture reconfiguration. A human Grid administrator could deploy some managers (like the Scheduler) to either User Agents (OurGrid), Peers (Lightweight Bartering Grid) or Resources, depending on the strategy chosen to provide robust execution, e.g. Task replication (OurGrid) or fault-avoidance (Lightweight Bartering Grid).

F.2 Negotiation Model

F.2.1 Reservations and QoS

The negotiation model could be extended to support reservations and QoS deadlines, but the unstability of P2P Grids certainly constitutes a formidable obstacle that may prevent such an extension to be ever achieved in a fully satisfying way. Moreover, reservations and QoS deadlines need an enforcement mechanism to arbitrate conflicts in case of contract breach. Fully decentralizing such a mechanism in a P2P fashion, in order to maintain scalability, is also likely to be very challenging.

Reservations and QoS deadline would both require fairly accurate runtime estimates of submitted Tasks. To incite Grid application developers to provide accurate runtime estimates of the Tasks they want to run on the P2P Grid, a metric could estimate the reliability of the provided estimates. A consumer that regularly submits Supplying Tasks completing within a certain factor of the provided estimates could then be ranked higher than other consumers when scheduling Supplying Tasks.

F.2.2 Peer Discovery

Peer discovery enable Peers to discover other Peers (see Section D.2). The current implementation of Peer discovery in LBG (see Section 2.1.4) is centralized. To enable large-scale deployments of the LBG middleware, the implementation of Peer discovery should evolve towards a fully distributed or replicated-hierarchical (DNS-like) organization. Recent research works [53, 174] in this area will certainly offer useful guidelines.

F.3 Task Model

F.3.1 Inter-Task Communications

The current Task model (see Section 2.6) is designed for applications structured as Bag of Tasks (see Section 2.4). In Chapter 6, we have shown how to support applications structured as Iterative Stencils. Supporting them would require the availability of inter-running-Task communications. Supporting applications structured as Workflow would require the availability of inter-non-running-Task communications.

To this end, support should be added for the storage of output data files on the Resources where they were generated. Likewise, the file naming mechanism should be extended to generate consistent file names for output data files. Another useful enhancement is to enable User Agents using a given Peer to obtain symbolic links to files (input, output or jar) inserted into the Grid by other User Agents using this Peer.

F.3.2 Checkpointing and Migration

The Task model could be extended to support interruptible Tasks, so that checkpointing, and then migration, could be supported. Recent advances in transparent application-level checkpointing [319] could allow to augment the LBG architecture with checkpointing support that is transparent to Grid nodes administrators and that could be made transparent to Grid application developers.

F.3.3 Grid Application Development

The LBG architecture is technology-neutral. The current implementation of the LBG Resource middleware requires Grid applications to be developed in Java. As the LBG architecture intrinsically supports Grid applications developed in

any language, the Java-based runner Virtual Machine (VM) of worker nodes (see Section 2.7.4) could substituted with an O.S.-level VM, e.g. VirtualBox [302] or Xen [315], as has been proposed for OurGrid [79].

F.4 Data Transfer

F.4.1 Scalability of the Data Transfer Architecture

Our proposed data transfer architecture is intrinsically extremely scalable. In Section 5.2.7, we have further proposed a way to remove the load of sharing data almost entirely from Peers: Each Peer could delegate its data server to some of its Resources. To scalably share files with BitTorrent without involving the Peer, one Resource can simply download a file from the Peer and start sharing it. To scalably share files with FTP without involving the Peer, it is slightly more complicated. Multiple replicas of a file should be downloaded by multiple Resources of the Peer, for example with BitTorrent. Each time the Peer has to share an input data file with FTP, it could randomly select one of its Resources to which it has replicated the file, and send the corresponding metadata with the Local or Consumption Task. This is essentially a form of Content Distribution Network [145, 242, 279] (CDN).

F.4.2 Cache Replacement Policy

Alternative cache replacement policies, for example weight-based policies, could be investigated so that the size of the cached files is also taken into account when a file is going to be ejected from the cache.

F.4.3 Estimation of the Storage Reliability of Suppliers

We proposed a data-aware supplier Peer selection algorithm (see Section 5.3.3) that is based on the expected availability of input data files in the data caches of supplier Resources.

When a Consumption Task is completed, the consumer could update the metadata in its Peer register to remember if the *a priori* estimate of data availability was correct or not. Resources downloading an input data file could be requested to communicate to the consumer Peer a hash of this file as soon as it is completely downloaded, and prior to actually starting the execution of the Task. The time to obtain this hash could be used to estimate *a posteriori* data availability.

There are however several issues with such a mechanism. First, the delay between the submission of a Consumption Task to a supplier Peer and actual scheduling of the Task to a Resource might not be immediate due to queueing delays. This introduces uncertainty on the meaning of the time taken to send a hash to the Consumer Peer. Second, such a mechanism would require to trust the Resources of other Peers. Malicious Resources could cache the hashes of input data files even if the files themselves are not cached any more, in order to mask data unavailability. Detecting such behavior might be exceedingly difficult, as a Resource downloading a file with BitTorrent might never download any piece from the consumer Peer, but only download pieces from Resources of other supplier Peers. It might be argued that the BitTorrent tracker could memorize which Resources are interested in a given file. However, if a mechanism of proactive data replication is activated (see Section 5.4), some useful information can be deduced, but not necessarily if a given file was cached by the Resource to which the Consumption Task needing it was scheduled.

F.4.4 Performance of BitTorrent Data Transfers

Optimized versions of the BitTorrent protocol [87] could also be investigated.

High-level layers of the BitTorrent [46] client could be augmented with enhanced algorithms to introduce QoS in BitTorrent [15].

As proposed by Allcock [7], low-level layers of the BitTorrent [46] client could be replaced with those of the GridFTP [165] client.

The BitTorrent protocol itself could be enhanced by adding a BitTorrent-level bartering mechanism to BitTorrent that would allow idle BitTorrent Nodes to increase their reputation by proactively downloading data files required by other BitTorrent Nodes, as proposed in the 2Fast protocol by Garbacki et al. [155, 250].

F.4.5 Performance of BitTorrent and FTP Data Transfers

An adaptive online compression mechanism, such as the one proposed by Jeannot [181], could be integrated with the Peer data manager to improve data transfer performance, orthogonally to the baseline data transfer protocol. Even if there is no redundancy between files, either within or between Bags of Tasks, adaptive compression could substantially improves the data transfer performance of files with internal redundancy.

F.4.6 Performance of GNMP Data Transfers

The raw transfer performance of the Grid Node Messaging Protocol should be compared to what can be achieved with application-level messaging protocols like the XMPP-based [136, 137] JIC [211], or ICE [167]. The compliance of GNMP with standard Grid protocols (see Appendix E.2) should also be investigated.

F.4.7 Data-Aware Negotiation

When a Peer is going to use received consumption grants, i.e. to schedule some of its Tasks as Consumption Tasks to other Peers, it does not currently take into account the number of the next subsequence of *data-equal* Tasks to schedule. Configuring a Peer with a low threshold of received consumption grants (see Section 2.9.6) may reduce the number of Consumption Tasks that can be simultaneously scheduled. Consequently, the benefit of activating Temporal Tasks Grouping (see Section 5.3.1) might be decreased. Indeed, limiting the number of Consumption Tasks simultaneously depending on a given input data file reduces the number of BitTorrent Peers simultaneously downloading this file. Therefore, it would be interesting to evaluate the impact of dynamically adapting the thresholds of consumption grants evaluation to the length of the next group of *data-equal* Tasks to schedule.

F.4.8 Proactive Data Replication

The impact of the proactive data replication algorithm discussed in Section 5.4 should be evaluated, as well as the impact of the replica selection policy under varying workloads.

F.5 Data Persistence

Data persistence consists in saving data to long term storage so that it can outlive the execution of the software by which it was saved. The goal is to have a new instance of the software that crashed to retrieve the saved data rather than generating it or downloading it again.

F.5.1 Input Data, Output Data and Jar Files Persistence

When asked to download a file, a Resource data cache first checks whether this file is cached. If not, it lists its storage directory to check if it is stored. In this case,

there is no need to download the file, and it is immediately integrated into the data cache, i.e. the data cache updates its internal metadata structures to reflect the fact that the file is present in the storage directory. This mechanism is really lightweight and easy to deploy.

Data caches (see Section 5.2.5) can be considered as persistent for Resources, but not for the Peers. Data persistence is currently not end-to-end across the Grid. A Resource booting after a crash does not notify its owner Peer that its internal data structures are empty, but will be able to reload the data from the file system when asked to download it. However, a Peer booting after a crash does not attempt to reload its internal data structures with the true state of the Resources' data caches it manages, which decreases the efficiency of data-aware scheduling. Implementing true end-to-end persistence in data caches across the Grid should be a priority in future work.

Adding support for the storage of output data files, as suggested in Section F.3.1, would be an important step to support Workflow applications and would also enable User Agents to download output data files whenever they want, even long after a BoT has been completed. Storage of output data files should also be persistent.

Adding support for the storage of jar files would prevent unuseful downloads of jar files by Resources in the case of repeated parameter sweeps. Storage of jar files should also be persistent, but a robust version control mechanism must be designed to ensure that the correct version of a Grid application is run as intended by the Grid application developer.

F.5.2 Peer Register Persistence

The Peer register (see Section 4.4.6) is currently not persistent. Adding support for persistence of the metadata stored within a Peer register should be a high priority in future work. Indeed, after a Peer crashes and reboots, it has literally lost its memory of interactions with other Peers, as well as reliability estimates of other Peers that were accumulated over time. The same issue also exists for User Agents and Resources. If Grid nodes are to run for lengthy periods of time, adding persistence support to the Grid middleware becomes a mandatory enhancement.

Furthermore, adding persistence support to the Peer register would enable a Peer to store metadata for an arbitrary high number of interactions (see Section 4.4.6).

F.5.3 Checkpointing

Application-level checkpointing is supported by the availability of a playpen (see Sections 2.7.4, 6.2.3). Middleware-level checkpointing would require persistence support to automatically save and restore the running state of Grid applications.

F.6 Simulation

F.6.1 Simulation of Data Transfers

As explained in Chapter 3, data transfers are not simulated, although data caches are fully simulated. Adding support for the simulation of data transfer times to the LBG simulator would enable more accurate and realistic simulation, especially for Data-Intensive BoTs (see Chapter 5). This is a complex issue because multiple data protocols (BitTorrent, FTP) have to be accurately simulated. Moreover, BitTorrent simulation constitutes a research domain in itself. Recent research [4, 5, 131], as well as insights from SimGrid [77, 209], General Peer Simulator [322] and P2P Lab [226] projects could provide useful insights.

The transfer of GNMP messages (i.e. control data, which is different from input data) is simulated, but its temporal impact - although certainly negligible - could be investigated. This requires a much finer temporal resolution, than the temporal resolution of one second that is currently used.

F.6.2 Simulation of Multithreading

The simulation of multithreading activities is currently restricted to what is minimally required for Peers to schedule and negotiate. The simulation of multithreading activities should definitely be extended to a more accurate and scalable model.

F.6.3 Simulation of Additional Sources of Failure

The currently implemented sources of failure include Peer-initiated and Resource-initiated Task preemption. Complete failure of a User Agent, Peer or Resource is currently not supported, although some of these failures have been simulated with temporary code which has been removed after the release of the middleware implementation. It would be very interesting to create additional sources of failures to both test the robustness of low-level code layers and systematically ensure that communications between Grid nodes are fully asynchronous.

F.6.4 Simulation Standards

There currently exists no standard of Grid deployment metadata. It is thus difficult to compare different algorithms, or the same algorithms running on different Grids. Investigating compatibility with existing Grid simulators could enable in the long term to create a repository of well-known Grid configurations and define standard interfaces for scheduling algorithms.

Moreover, support for additional statistical distributions (e.g. for time-to-Resource-failure, Task submission inter-arrival times, Peer power repartition, ...) would certainly be useful in practice.

F.7 Security

Security is of course a very important aspect of Grid middleware. Task control has been proposed to prevent Denial of Service by never-ending Tasks. Resource protection mechanisms have been proposed. However, several other issues have not been addressed in our presentation of the LBG architecture.

F.7.1 Authentication and Encryption

Authentication, and thus encryption, support should be added to GNMP, and to the data transfer architecture as well. This will enable Peers to authenticate to one another. This will also enable Resources to authenticate to their owner Peer. Existing mechanisms in widespread use, such as GridFTP security features [235], could be adapted to LBG.

F.7.2 Firewall/NAT Traversal

Firewall traversal and NAT traversal are two important issues of P2P software that are shared with other domains, like IP telephony. Grid nodes are intended to run on edge computers, which may be connected to the Internet from networks isolated by firewalling or Network Address Translation mechanisms. Such an issue applies to the LBG middleware as well as to the BitTorrent and FTP softwares. The important advances of the recent years [112, 262, 303] should thus be integrated into LBG.

F.7.3 Security of Execution

Protection of Resources against malicious Tasks is ensured by the Java sandboxing mechanism on Resources and by Task control performed by Peers.

Protection of Tasks against malicious Resources is currently not guaranteed. Security of execution [212, 213], which consists in protecting a Task against the Resource that runs it, and automatic code verification [88, 223] are still open questions, though.

Another, more pragmatic, form of protection of Tasks against malicious Resources has been proposed by Sarmenta [268]. The Spot-Checking sabotage-tolerance mechanism operates with eager (i.e. Task replication [86, 308]) scheduling to adaptively recompute a subset of the Tasks to complete. The intent is to verify that the results generated by redundant computations are identical (or at least compatible, in the case of nondeterministic Tasks). The addition of an adaptive sabotage-tolerance mechanism into the BOINC Volunteer Grid middleware has been recently proposed by Estrada et al.

Index

Colophon

This dissertation has been prepared essentially with the LaTeX document preparation system and `Dia`, a program for drawing structured diagrams.

We want to thank the Reed College [259] for the publicly available dissertation formatting rules.

Some Figures include icons from the Tango library [292] under Creative Commons Attribution Share-Alike license.

Hyperlinks are available in the electronic version (.pdf) of this dissertation. Every mention of a bibliographical reference is hyperlinked to the corresponding entry in the table of contents, and every reference to a chapter, section or figure is hyperlinked to the corresponding entity.