INSTITUTO SUPERIOR TÉCNICO
Universidade Técnica de Lisboa

Resource Usage in Grids and Peer-to-Peer Systems

Sérgio Ricardo Oliveira Esteves

Dissertação para obtenção do Grau de Mestre em

Engenharia Informática e de Computadores

Júri

Presidente:	Prof. Dr. Nuno João Neves Mamede
Orientador:	Prof. Dr. Paulo Jorge Pires Ferreira
Co-Orientador:	Prof. Dr. Luís Manuel Antunes Veiga
Vogais:	Prof. Dr. Carlos Nuno da Cruz Ribeiro

Novembro de 2009

Resumo

Durante os últimos anos tem-se assistido a um crescimento bastante acentuado na tecnologia inerente aos computadores e recursos disponíveis através da Internet. Apesar disto, máquinas comuns ainda não possuem poder de processamento suficiente para algumas aplicações vastamente utilizadas e intensivas a nível de CPU.

Através de sistemas em Grid é possível tirar partido de recursos disponíveis numa rede. No entanto, estes sistemas estão geralmente associados a organizações que impõem enumeras restrições ao seu uso. De forma a ultrapassar tais barreiras, os sistemas Peer-to-Peer podem ser usados para que a Grid fique disponível a qualquer utilizador.

A solução proposta consiste numa plataforma para a partilha de ciclos distribuída que combina os modelos Grid e Peer-to-Peer. Assim, qualquer utilizador comum poderá usar ciclos remotos e ociosos de forma a acelerar a execução de aplicações. Por outro lado, os utilizadores poderão também permitir o acesso aos seus recursos quando estes não estiverem a ser usados.

Esta solução envolve as seguintes actividades: gestão de aplicações, criação e escalonamento de jobs, descoberta de recursos, políticas de segurança, e gestão do anel da rede. A organização simples e modular deste sistema permite que os componentes sejam mudados com o mínimo de custo. Adicionalmente, o uso de políticas baseadas em história permite diversas semânticas de utilização no que respeita à gestão de recursos.

Muitos dos desafios para a distribuição de ciclos ociosos são abordados neste sistema genérico. Através da exploração da paralelização de aplicações, acreditamos que esta plataforma irá atingir comunidades de utilizadores em todo o mundo.

Palavras-chave: Recurso, Grid, Peer-to-Peer, Partilha de Ciclos, Políticas.

Abstract

The last few years have witnessed huge growth in computer technology and available resources throughout the Internet. Despite this, common machines still do not well suit some specific and widely used applications. These applications are CPU-intensive, consuming long periods of time during which one becomes bored and impatient.

Grid systems have arisen in such a way as to take advantage of available resources lying over a network. However, these systems are generally associated with organizations which impose several restrictions to their usage. In order to overcome those organizational boundaries, Peer-to-Peer systems provide open access making the Grid available to any user.

The proposed solution consists of a platform for distributed cycle sharing which attempts to combine Grid and Peer-to-Peer models. Any ordinary user is then able to use remote idle cycles in order to speedup commodity applications. On the other hand, users can also provide spare cycles of their machines when they are not using them.

Moreover, this solution encompasses the following activities: application management, job creation and scheduling, resource discovery, security policies, and overlay network management. The simple and modular organization of this system allows that components can be changed at minimum cost. In addition, the use of history-based policies provides powerful usage semantics concerning the resource management.

Many of the critical challenges that lie ahead for distributed cycle sharing are encircled on this generic system. By exploiting parallel execution of common applications, we believe that this platform will start reaching communities of Internet users across the world.

Keywords: Resource, Grid, Peer-to-Peer, Cycle Sharing, Policies.

Acknowledgements

For helping make this dissertation possible, I thank:

- Prof. Paulo Ferreira and Prof. Luís Veiga, for the valuable motivation, availability and attention they provided to me;
- Filipe Paredes and Pedro Gama, for their contribution in our understanding of the Ginger and Heimdall implementations respectively;
- Jeff Hoye and James Stewart (from the Max Planck Institute for Software Systems), for answering our questions about FreePastry and for being so supportive all the time;
- Prof. Miguel Casquilho, for the useful explanations and help about the integration of statistical application in this project;
- my colleagues at INESC-ID and IST, for additional motivation and incitements;
- (last but not least) my family and friends, with all my heartfelt gratitude.

Table of Contents

I Appendix

List of Figures

List of Acronyms

ACK	Acknowledge
API	Application Programming Interface
CPU	Central Processing Unit
DHT	Distributed Hash Table
DoS	Denial of Service
GUI	Graphical User Interface
ID	Identifier
OS	Operating System
P2P	Peer-to-Peer
QoS	Quality of Service
RTT	Round Trip Time
VM	Virtual Machine
xSPL	eXtended Security Policy Language

1 Introduction

Over the last decade, there has been significant growth in terms of the technology inherent in computers: the capabilities of these have been increasing (more or less like Moore's Law [Moo65]) in terms of computational power, memory, and persistent storage space. In parallel, also a widespread increase in Internet access by people all over the world has been witnessed. Allied to that fact, users of this global network have been involved with a spirit of seeking, and of sharing, of available resources (e.g. exchange of multimedia files) on machines around the world.

Nevertheless, the scientific community, from the most diverse areas, realized that they need great computational power in order to solve problems like: numerical weather forecasting, fluid dynamics, protein folding, simulations hard to test in practice (e.g. nuclear weapons), Monte-Carlo methods, genetic algorithms, among others [HP04, GKKG03]. Many times, not even the faster super computer of our days,[1] with a PowerXCell 8i 3200 MHz (12.8 GFlops) processor, would be able to solve those problems in a time considered useful. Overall, this need of the scientific community spurred out a new paradigm: the distributed and parallel computation, thereby giving origin to computational systems in Grid.

A Grid system is an infrastructure for distributed computing and data management, composed of a cluster of networked, loosely-coupled computers with focus on large-scale resource sharing, innovative applications, and high-performance orientation [FKT01]. Grid computing can be distinguished from typical cluster computing systems in a way that grids tend to be less coupled, heterogeneous, and geographically dispersed. Moreover, a Grid system has the purpose to serve a community of individuals and/or institutions by providing resources that have a set of rules (well defined and highly controlled) over the sharing. This is commonly referred as a virtual organization.

Such virtual organizations comprise a set of clusters where each is potentially under different administrative control, and access to computers between different clusters must be negotiated in advance. A user outside the domain of the corporation (or community) has to overcome several barriers before it can deploy its own application on the Grid. That is why the arising of Grid systems has failed to reach the common Internet user, despite its great potential and success within the scientific field.

At the same time, a distributed system known as peer-to-peer has gained a huge success across the Internet.[2] Such architectures are designed for the direct sharing of computer resources (CPU cycles, storage, content) rather than requiring the intermediation of a centralized server or authority [ATS04].

Peer-to-Peer systems are characterized by their ability to function, scale, and self-organize in the presence of highly transient population of failure-prone nodes. The great advantage of this approach over other models is the no dependence of centralized servers (possibly administrated by third parties), which suffer from problems like bottlenecks, single points of failure, amongst other. Hence, these systems have good scalability, and can handle the increasing state of populations while maintaining the performance.

However, some peer-to-peer systems make use of centralized servers for specific tasks (e.g. system bootstrapping), thus clearly stretching the definition of peer-to-peer. There is a considerable number of dif-

[1] `http://www.top500.org/system/9485` accessed on October 2008

[2] Workshop on technical and legal aspects of peer-to-peer television, Amsterdam, Netherlands, March 2006. Trends and Statistics in Peer-to-peer: `http://www.gsd.inesc-id.pt/~sesteves/p2p/CacheLogic_AmsterdamWorkshop_Presentation_v1.0.ppt` accessed on October 2008

ferent definitions of peer-to-peer, mainly distinguished by the "broadness" they attach to the term. Thus, the approach mentioned before relies on the definitions of "pure" peer-to-peer systems (i.e. fully decentralized).

More recently, it has been witnessed a convergence between the peer-to-peer computing and Grid systems as both approaches share several interests, such as the sharing of resources among multiple machines. The model of trust and security underlying grids has been leveraged together with peer-to-peer, thus leading to public-resource computing infrastructures with very transient populations. For instance, any ordinary user would have the same power to easily provide and consume resources across the Internet.

Nonetheless, machine resources still need to be controlled on peer-to-peer grid infrastructures. For example, computer owners may need to decide when their resources are available to other machines, how much resource consumption is allowed, who cannot access their machines, and so forth. There are many usage semantics that can be applied on these environments. Therefore, security policies are needed in order to deal with such constraints.

With the presented evolution some novel applications have arisen, thereby exploring these new models of computation. In respect to CPU cycle sharing, the first and most notorious project getting the user attention was the SETI@home launched in 1999 [ACK+02]. This project attempts to process a large amount of information by using the computational power of every machine that runs its software. This system is based on an asymmetric relationship where the participants are just able to donate their CPU cycles to the project, and cannot use for themselves any spare cycles of other machines connected to SETI@home. After that, some other applications emerged although only few of them were non asymmetric, therefore allowing the mutual sharing of spare cycles with a reasonable control over (discussed in further sections).

Currently, not only scientists, but also typical computer users are willing to perform intensive tasks on their computers. However, these tasks could be quite different, like: compressing a movie file, generating a complex image from a specification, compacting large files, among other. To be more precise, these tasks consume a relatively large amount of time and memory, delaying other processes that are running at the same time. Along the way, one becomes bored and impatient. From another point of view, there are many Internet-connected computers around the world whose resources are not fully utilized. Most of the time, typical users have just some low CPU-intensive processes running on their machines, therefore giving a sense of waste.

Given the current context and motivation, we intend to deploy a platform where any ordinary user can consume and provide resources, namely idle CPU cycles, over a dynamic network that could be local or wide (e.g. Internet), in order to speed up common, and widely used, applications which are CPU-intensive. In other words, we intend to exploit parallel execution in desktop applications with a fine-grained control over the shared resources. These applications (e.g. POV-ray)[3] should be kept unmodified for obvious reasons.

Moreover, there are several issues that need to be addressed. The platform: (i) needs do be scalable; (ii) needs to be portable, handling the heterogeneity of machines; (iii) needs to have a modular organization, each component should be independent from each other; (iv) needs to provide security mechanisms over computer resources, for the purpose of keeping the primacy of every user machine; (v) needs to be efficient; (vi) needs to adapt to environmental changes, like resource availability; and (vii) needs to be user-friendly. The greatest challenge is to achieve a speedup from applications closest to optimal.

[3] A ray-tracing implementation: `http://www.povray.org` accessed on October 2008

The proposed solution aims to integrate and adapt solutions for: overlay network management, resource discovery, job creation and scheduling, and resource management. Many of the solutions that will be discussed in further sections are just implemented in simulators, though they may need to be adapted to real environments. Also, the interoperability between the components of the system is taken into account.

Furthermore, it is expected at evaluation stage that the running time of the applications substantially decrease while the number of available nodes in the overlay increase. However, one has to take into account the various overheads underlying each component of the system, therefore considering if a certain job is worth its computational parallelization. Also, it is expected an efficient use of the available resources.

This dissertation is organized as follows. In the next chapter, we discuss current solutions related to the GridP2P. In Chapter 3 we present the architecture of the GridP2P, describing each of its components. Then, we focus on the implementation of our platform in Chapter 4, which is evaluated in Chapter 5. Finally, Chapter 6 presents our conclusions.

2 Related Work

In this chapter we review a representative selection of relevant solutions regarding the fundamental premises to achieve our goals. That is, job scheduling is essential to achieve efficiency in applications, security policies are important to control resource usage, and resource discovery mechanisms are needed to locate resources within a dynamic environment. In addition, different models of distributed cycle sharing systems are analyzed and compared with the GridP2P.

2.1 Job Scheduling

The job scheduling is the assignment of work to resources within a specified time frame [Ber99]. In Grid environments, scheduling is mainly used to optimize performance (e.g. execution time, throughput, fairness, and so on), providing the best quality of service. Also, the scheduling may be used for load balancing purposes.

Nevertheless, each scheduling mechanism may aim to different performance goals. For example, job schedulers attempt to maximize the number of jobs executed by a system (optimizing throughput), while resource schedulers aim to coordinate multiples requests on a specified resource by optimizing fairness criteria or resource utilization. These goals, regarding the overall system performance, may conflict with application schedulers which promote the performance of individual applications by optimizing, for instance, the speedup or minimal execution time.

Briefly, application scheduling consists of selecting resources, assigning tasks to resources, distributing computation or data (in the case of task-parallel programs), and ordering tasks and communication between them (load balancing purposes). Moreover, a scheduling model comprises a set of policies that abstract the applications to be scheduled, and a performance model which abstracts the program behavior. In addition, the scheduling model utilizes a performance measure which describes the performance activity to be optimized by the performance model.

From the accumulated experience, it is clear that the choice of a scheduling model can make a dramatic difference in the performance achieved by applications. An important case reflecting this, was the modification of successful strategies from massively parallel processor (MPP) schedulers for grid environments [Fei95]. Basically, the MPP scheduling model made incorrect assumptions about grid environments. Consequently, a new scheduling model must be developed for the Grid, reflecting the complex and dynamic interactions between applications and the underlying system.

Currently, trends on job scheduling mechanisms includes: (i) use of dynamic information; (ii) use of metainformation[1] (e.g. Autopilot [RSR01] and AppLeS [FW97]); (iii) use of realistic programs (more application code specific); (iv) restricting domain (more domain specific); (v) development of language interfaces (automating scheduling process).

Furthermore, current challenges on job scheduling mechanisms involve: (i) minimize the performance impact of architecture independence (Portability vs. Performance); (ii) Grid-Aware programming; (iii) scalability; (iv) scheduler overhead should not affect the normal application execution (efficiency); (v) repeatability (i.e. consistency, predictability); (vi) cooperate with resource schedulers (i.e. multischeduling).

[1] Assessment of the quality of the information given.

Discussion. Scheduling is the key for performance in Grid environments. Also, most advanced Grid applications are targeted to specific resources.

GridP2P aims to high-performance schedulers, as a system for speed up the execution time of applications. Currently, GridP2P is more focused on scheduler mechanisms which cover local jobs and applications (e.g. locally-controlled load balancing), though not disregarding at all the system performance as a whole.

Overall, the optimal solution concerning the scheduling problem is considered infeasible, even in the simplest environments.

2.2 Security

In Grid environments, applications are characterized by their simultaneous use of large numbers of resources, dynamic resource requirements, use of resources from multiple administrative domains, complex communication structures, and stringent performance requirements, among other [FKTT98]. Moreover, as in the context of wide-area networks, both data and resources are often administrated locally and independently. Thus, in order to achieve scalability, performance and heterogeneity, new security models are needed to face the intrinsic characteristics of computational Grids. For example, parallel computations would need to establish security relationships, not only between a client and a server, but among many processes that collectively span administrative domains. Furthermore, trust relationships between sites may not be possible to establish prior to application execution. Also, the inter-domain security solutions must interoperate with the diverse access control mechanisms concerning each individual domain.

Current solutions: The X.509 certificates are used by many systems to unequivocally identify their users [GF06].

With respect to the authorization, local mechanisms as the application of privileges over unix accounts, based on ACLs (Access Control Lists), constitute the most used option.

The communication between sites on grid systems is secured by well known standards as the TLS (Transport Layer Security)[2] protocol. Nonetheless, new webservice standards, such as WS-SecureConversation,[3] WS-SecureMessage,[4] and XML-Signature[5] are also taken into account.

In order to support computation at a large scale, the delegation of privileges is another essential point. For instance, the Globus project [WFK+04] introduced the user proxy concept which intends to avoid users from authenticating themselves each time they need to access a new resource on the grid.

With respect to policy engines, there are some prototypes and references in literature that can be found, although none of them fully cover the complexity inherent to a Grid system. On the one hand, several systems propose the use of policy engines to manage resources, but on the other hand, they do not supply a sufficiently expressive language to support a dynamic utilization model. In addition, this lack of expressiveness and flexibility obliges policy administrators to code policy rules into the resource managers themselves.

[2] `http://www.ietf.org/rfc/rfc2246.txt` accessed on October 2008

[3] `http://specs.xmlsoap.org/ws/2005/02/sc/WS-SecureConversation.pdf` accessed on October 2008

[4] `http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-soap-message-security-1.0.pdf` accessed on October 2008

[5] `http://www.w3.org/TR/xmldsig-core` accessed on October 2008

2.2.1 Heimdall Policy Engine

The Heimdall [GRF06a, GRF06b] is a history-enabled policy engine which allows the definition, enforcement, and auditing of history-based policies in Grid platforms. With history-based policies is possible to include temporal events (occurred in the past) in the policy rules. Moreover, Heimdall regards powerful usage semantics with complex constraints, in which is used the xSPL language (eXtended Security Policy Language), an extension of the SPL language [dCRdCMZFG01]. In addition, xSPL provides an high level of abstraction, clearly separating the specification and implementation of security mechanisms.

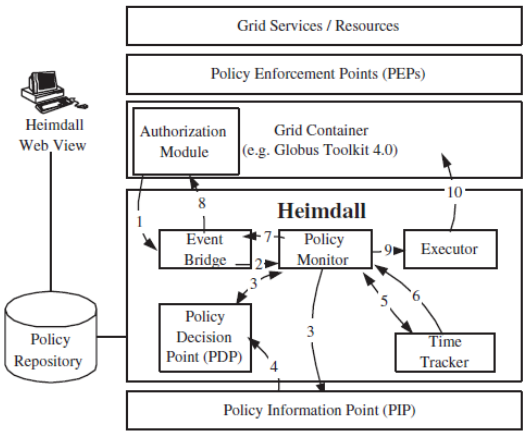

Figure 2.1: Heimdall Architecture Overview

Architecture. The Container Authorization Module sends operation information to Heimdall, which is received by the Event Bridge Module. The module converts the operation information into a more standard syntax: the xSPL event. This event is sent to be coordinated by the Policy Monitor, which exchanges information with the Policy Decision Point (PDP) to determine if the current event triggers any deployed policies. If it does, the event must be stored in the Policy Information Point (PIP) for future analysis. The PDP uses previously stored events to analyze the development of the policy. If the effects of the event cannot be determined at the time (e.g. job execution period is not know at submission time), the Time Tracker creates a timer for periodic policy analysis. However, when the policy termination condition is fulfilled (e.g. a job concludes or is not successful), the Time Tracker removes the existing timer. At periodic intervals, the Time Tracker contacts the Policy Monitor to check for policy infractions (reevaluating policies). After overall evaluation, the Policy Monitor returns the authorization decision related to the execution of the specified action. The authorization decision is translated by the Event Bridge into the Grid Container format and returns the security information to the Container Authorization Module. If an infraction is caught by the Policy Monitor, the Executor is invoked in order to execute the policy defined custom action (e.g. terminating or reducing the priority of a certain job). The Executor serves as the interface between Heimdall and the Grid Container, thereby using some protocols (e.g. webservices, HTTP, RMI, SMTP, and so forth) to provide a connector to Grid Resource Managers.

2.2.2 Deeds

The Deeds [EAC98] is a history-based access control system whose policies must be written in Java. Deeds' approach about history-based access-control is to maintain a selective history of the access requests made by individual programs, and to use this history to improve the differentiation between safe and potentially dangerous requests. Despite the focus of this system on entities that can or cannot access local systems (or local resources), Deeds is also flexible in a way that it may enforce many different history-based policies, such as limiting the utilization of resources or recognizing different patterns of access. Additionally, policies can be installed, removed, or modified during execution.

Security Events and Handlers. A Deeds security event occurs whenever a request is made to a protected resource. A security event could be a request for reading a file or to open a socket. Though, the set of security events is unlimited: programmers can associate a security event with any request they wish to keep track of or protect. Handlers are associated with security events, thus they are responsible for maintaining an event-history and check whether it satisfies user-specified constraints. Multiple handlers can be associated with each event, maintaining separate event-history. In this way, handlers use a consensus voting rule in which one negative vote is sufficient for denying a request. Otherwise, the access is granted. Finally, Deeds allows policies to be parametrized. For example, a policy that controls file creation can be parametrized by the directory which allows that file creation.

Discussion. Actually, there are a lack of policy engines targeted to new infrastructures that combine Grid and Peer-to-Peer technologies (GridP2P). Although, there are expressive languages for specification of policies, such as Ponder [DDLS01] or Lobo [LBN99], that may be incorporated in policy engines aiming at complex and dynamic environments. However, developing a policy engine from scratch is outside the scope of the GridP2P.

Due to the inherent characteristics of Heimdall, policy administrative tasks regarding the management of local resources are leveraged. Moreover, in the context of history-based semantics, several usage and security patterns can be applied on Grid environments (e.g. resource usage fairness). Nonetheless, Heimdall significantly increases response time to environmental changes. In particular, for an open access environment (like the GridP2P), this time is not so long (i.e. less rules than it would have in an institutional environment).

Deeds is hard to manage because the programmer has to individually implement all the handlers. The ad-hoc development of Policy Decision Points, and Policy Enforcement Points, is impractical in large and/or complex systems. In contrast with Heimdall, it generates Policy Decision Points automatically, and events are managed in a transparent manner by Heimdall's Policy Information Point implementation. Additionally, for practical effects of integration, Heimdall constitutes a solution that fits in the needs of the GridP2P project.

2.3 Resource Discovery

Resource discovery is a fundamental issue on resource sharing environments, such as Grids and Peer-to-Peer systems [TTP+07, MRPM08]. Available resources (e.g. CPU, memory, disk space, and so forth) lying

over a network are useless if they can not be located. Precisely, resource discovery is a service provided by network infrastructures, and is based on a list of predefined attributes. After specifying the attributes of existing resources, the system returns a list of locations where the required resources currently reside.

Common challenges in the resource discovery field are: bootstrapping, security and privacy, search result ranking, handling with the node departure and failure, efficient resource management, data representation and load-balancing.

There are currently various solutions concerning the discovery of computational resources upon an overlay network. Next, it is given an overview of current mechanisms with focus on the location of idle CPU cycles.

2.3.1 Resource Discovery on Unstructured Networks (Peer-to-Peer)

Due to the lack of an underlying structure on unstructured networks, there is no information about the location of resources, thus the prevailing resource discovery method is flooding. A peer looking for a resource broadcasts a query in the network. All positive matching responses are send to the originating peer through the reverse path. Flooding creates a large volume of unnecessary traffic in the network, thus it is not scalable. To limit the number of messages generated by flooding, each message is tagged with a Time-To-Live (TTL) field. The TTL indicates the maximum number of hops that a message can make. A small TTL can lead a query to fail (although present in the system) by reducing the network coverage. Modern mechanisms address this problem of the vast amount of messages produced by flooding without failing to cover the whole network, such as the Dynamic Querying.[6]

In dynamic querying, one peer sends a query to a subset of neighbors with small TTL. If the results of the query were not satisfactory, the originating peer may broadcast the query to a different set of its neighbors with increased TTL. This process is repeated until a satisfactory amount of results is received, or until all the neighbors are exhausted.

Many other techniques have been proposed to address the traffic/coverage trade-off [PFA+06, TR03, GM05, LCC+02, CGM02].

2.3.2 Resource Discovery on Structured and Hybrid Networks (Peer-to-Peer)

Structured networks are equipped with a distributing indexing service, known as Distributed Hash Table (DHT). Examples of this networks are the Chord and CAN (described in Section 2.4.3).

Both unstructured and structured approaches have advantages and disadvantages. Several hybrid approaches have been proposed to overcome the drawbacks of each while retaining their benefits, such as the Kademlia [MM02] and Pastry [RD01a].

Kademlia and Pastry are similarly structured. Their structure exhibits less strictness compared to Chord and CAN, in the sense that for each defined subspace, any peer belonging to that subspace can serve as a contact. In Chord and CAN, all neighbor connections are strictly defined. Kademlia is the first hybrid P2P system to achieve global-scale deployment.

[6] Dynamic Query Protocol: `http://www.ic.unicamp.br/~celio/peer2peer/gnutella-related/gnutella-dynamic-protocol.htm` accessed on October 2008

2.3.3 Iamnitchi et al.

In [IF04] queries can be forwarded using different strategies: random walk, learning-based, best-neighbor, learning-based + best-neighbor. Next, we describe the two more relevant solutions.

Learning-based Strategy. In the learning-based strategy nodes learn from experience by recording the requests answered by other nodes. A request is forwarded to the peer that answered similar requests previously. If no relevant experience exists, the request is forwarded to a randomly chosen node.

Best-neighbor Algorithm. The best-neighbor algorithm records the number of answers received from each peer, and a request is forwarded to the peer that answered the largest number of requests.

2.3.4 CCOF Resource Discovery

Cluster Computing on the Fly has tried the next four resource discovery mechanisms [ZL04].

Expanding Ring Search. When a machine node requires cycles, a request is sent to its direct neighbors. On receiving this request, the neighbors compare their profile with the request. If these neighbors are currently busy or their block of idle time is less than the requested block of time, they discard the request. Otherwise the request is accepted and an ACK message is sent to the requesting machine. If the machine requiring for cycles determine that there are not enough candidate hosts to satisfy the request, it then sends the request to nodes one hop farther. This procedure is repeated until enough candidates to start the computation are found or the search reaches the search scope limit.

Random Walk Search. When a machine node requires cycles, a request is sent to k random neighbors. On receiving this request, like the expanding ring search, neighbors try to match their current status with the request. If there is enough time to complete the task, neighbors send an ACK message to the requesting machine. Neighbors also forward the request to their k random neighbors, until the request reaches the forwarding scope limitation.

Advertisement based Search (Ads-based search). When a machine node joins the system, it sends out its profile information to neighbors in a limited scope. The neighbors then cache such profile information along with the node ID. When a machine node needs cycles, it consults the profiles it has cached locally, and selects a list of available candidates. Then, the machine requesting for cycles needs to directly contact each of the candidates to confirm their availability. If a host is not available at this time, the requesting machine then tries the next host in the list, until the list is exhausted or the request is satisfied. There are several possible selection schemes, such as choosing the nearest available hosts, choosing the hosts with longest available time or choosing hosts with shortest matching available time block. However, simulation results in CCOF showed similar performance for different selection schemes.

Rendezvous Point Search. This method uses a group of dynamically selected Rendezvous Points in the system for efficient query and information gathering. Hosts advertise their profiles to the nearest Rendezvous Points, and clients contact the nearest Rendezvous Points to locate available hosts. Rendezvous Points are placed at distributed geographic sites (the dynamic placement of these points is still an open problem). When a node is selected as a Rendezvous Point, it floods information about its new role within a limited scope. On receiving such a message, nodes add the new Rendezvous Point into a local list and deposit their profile information on this new Rendezvous Point. When a node joins the system, it acquires the list of Rendezvous Points from the nodes it contacts, or acquires the information through some out-of-band scheme. If a machine node has not already cached information about nearby Rendezvous Points and needs cycles, it queries its neighbors until it accumulates a list of known Rendezvous Points. The machine node then contacts Rendezvous Points on the list one by one and each Rendezvous Point then tries to match the request with candidate hosts. This procedure repeats until the request is satisfied or all the known Rendezvous Points have been queried.

2.3.5 Informed Walk

Each node advertises its own resources only to the peers comprising its neighborhood [Par08]. Doing so, these last nodes cache that information and, when they need to submit a task, they send it to the most appropriated node. If none of the neighborhood nodes are available, the request is sent to one of them in which the probability of forwarding to other available nodes is highest (this information is based on previous requests).

Neighborhood set. The set of neighborhood nodes is defined by Pastry and it is built with base on a proximity metric between nodes, encompassing so the *n* (number of nodes on the overlay) nodes that are geographically closest (i.e. with the lowest round-trip time values).

However, one peer may have nodes in its neighborhood set that do not see that peer as a neighbor. For example, if the number of nodes in the neighborhood set is 20, the node A could see the node B as one of its 20 closest nodes, and B could have in its neighborhood set 20 nodes that are closest than A. These situations often occur in large networks. In order to guarantee a minimum of symmetry between neighbors, when the node A announces its availability to B, this last node will add A as its neighbor and then B announces its availability to A.

For avoiding the uncontrolled growth of neighbors, one node will accept other as its neighbor only if it has a lower identifier. Thus, if a node A receives an update message by a node B with a higher identifier, A will send that update message back to B. Therefore, B excludes A from its neighborhood set.

Selection of the best available node. The best available node is the one which presents the best availability in accordance with a weighted combination of proximity level, CPU, memory, and bandwidth. This information is propagated within update messages.

Selection of the best non available node. The best non available node is the one most capable of forwarding a message to available nodes. This capability is measured based on the history of forwarded requests by a node, thus defining its reputation. The history includes the number of job failures (requiring the retrans-

mission of the job), the number of times that a request returned back to the original node, and the number of attempts on getting the results of a job. The three numbers will determine the rejection level of a node. When a node A does not know the history of other nodes, it can receive from a node B an update message which includes a field containing the reputation of the best neighbor of B.

2.3.6 Expressive Querying

An expressive query may specify a variety of required metrics for the job: the number of hosts required, the amount of free CPU required on these hosts, the minimum amount of RAM required, and so on. In [CMG05], Cheema et al propose a solution that deals with expressive queries, using structured naming to enable both (i) publishing information about computational resources, and (ii) expressive and efficient querying of such resources. This solution is based upon the Pastry.

The relevant resources of a computer can be described using either a static part, and a dynamic part. Within the static part is the operating system configuration, CPU speed, RAM, and total disk space. While at the dynamic part is the CPU Idle %, available RAM, and available disk space. The naming problem that this project tries to address is the mapping of resources to nodes on a DHT given the actual representation of resources.

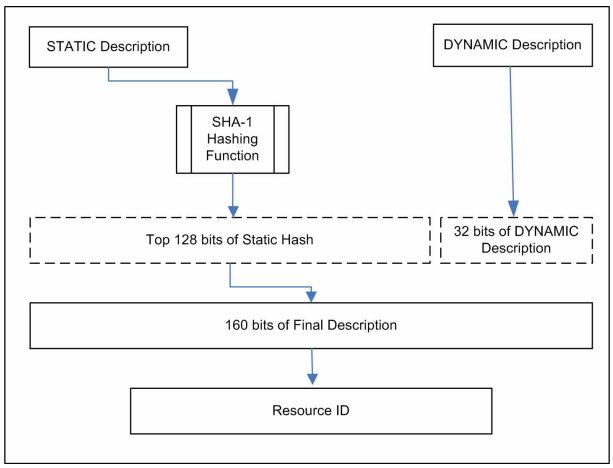

Figure 2.2: Computing the Resource Identifier

In order to overcome this problem, the static and dynamic parts are combined into a resource identifier, which serves as a key for the DHT. Moreover, the static attributes of a resource are placed at a specific point on the DHT ring. Similarly, the dynamic attributes of resources must be placed on the ring as well. Therefore, resources are represented on the ring as potentially overlapping arcs instead of individual points. Furthermore, there is an encoding mechanism (illustrated in Figure 2.2) which defines that the first 128 bits on a Pastry key are used for the hash of the full static description of a resource, while the last 32 bits are used for the dynamic description (i.e. 16 bits for the CPU Idle %, 10 bits for the RAM Free %, and 6 bits for the Disk Free %).

Also, some searching mechanisms are described. The main idea is that one node looking for a resource needs to compute the hash of a desired static description, and then (through Pastry) locate the closest node

of the generated key (the first 128 bits of the node ID and of the key/resource ID should be equal for a correct match).

Discussion. Iamnitchi et al. obtained results showing that the learning-based strategy is the best regardless of request distribution. This strategy takes advantage of similarity in requests by using a possibly large cache. The learning-based strategy starts with low performance until it builds its cache.

Overall, searching mechanisms which keep a history of past events are more efficient than the ones that do not store any information about other nodes, such as the random walk.

In CCOF, the Rendezvous Point Search obtained best global performance when compared to the other three methods (expanding ring, random walk, and ads-based search). However, this mechanism has the great disadvantage of not being able to place Rendezvous Points dynamically in a balanced way (i.e. a sufficient number of Rendezvous Points is within short distance to every peer). Thus, the Rendezvous Point Search is impratical in dynamic and/or complex systems.

The informed walk mechanism is a combination of the random walk with the ads-based search. This mechanism uses metrics to determine the reputation of a node, and therefore the best path to follow. The traffic on the network is relatively low since nodes just advertise to their direct neighbors (defined by Pastry). Also, results have shown that the use of node reputation mechanisms enhance both the routing of messages and the location of resources. In contrast with GridP2P goals, this solution achieves scalability, either in terms of time (number of hops a query propagates), as well as in terms of traffic (number of query messages required). Additionally, the informed walk also achieves efficiency (one of our goals).

Currently, DHTs only support efficient single keyword lookups which constitute a relatively small percentage of queries in real P2P systems. Moreover, efficiently supporting range queries in DHT-based systems as been posed as an open problem with no solution. Regarding CPU cycle sharing systems, the queries may be complex as resources are highly dynamic (CPU load, free memory, and so on). However, Cheema et al. have proposed an expressive querying mechanism (protocol) exploiting the single keyword DHT lookup (on Pastry) for CPU cycle sharing systems. Results have shown that this protocol is rigorous, scalable, efficient and practical. Thus, meeting the GridP2P requirements.

2.4 Cycle Sharing Systems

2.4.1 Institutional Grids

The Grid is a class of infrastructure that makes possible for geographically distributed groups to work together, by providing scalable, secure, high-performance mechanisms for discovering and negotiating access to remote resources. However, Grids have been more directed to the practice of science and engineering, forgetting, in some sense, the average computer user's trivial tasks. The GridP2P intends to enable that power, thereby making it available to any person.

2.4.1.1 Globus Toolkit

The Globus toolkit [FK97, FK99] is fundamental enabling technology for the implementation of Grid systems (as well as the applications running over), providing basic low-level mechanisms such as communication, authentication, network information, and data access. Various higher-level *metacomputing*[7] services can be constructed over these mechanisms, letting people share resources in a secure way across a corporate, institution, or geographically distributed sites. The toolkit includes software services and libraries for resource monitoring, discovery, and management, plus security and file management. Also, software for communication, fault detection, and portability is included. Globus core services, interfaces, and protocols allow users to access remote resources as if they were located within their own machine room, while simultaneously preserving local control over who can use resources and when.

The approach of 'toolkit' allows the customization of Grids and applications that can be developed in an incremental way (i.e. at each new version more resources are used throughout the Grid). For instance, the Globus can be initially used for coordinating the execution of unmodified applications, and afterward, those applications can be modified to use Grid services, such as the file transfer service.

Allocation and Resource Discovery. Resource discovery mechanisms are required in Globus because applications cannot, in general, be expected to know the exact location of required resources.

The local component for resource management is the Globus Resource Allocation Manager (GRAM). The GRAM controls a set of resources operating under the same site-specific allocation policy. As virtual organizations in Globus typically comprise sets of clusters distributed along distinct sites, grids will contain many GRAMs, each responsible for a particular local set of resources. Moreover, a GRAM is often implemented by a local resource management system, such as Condor (described bellow in this section).

Applications then can express resource allocation and process management through an application programming interface (API) regarding local resource management. Within the GRAM API, resource requests are expressed in terms of an extensible Resource Specification Language (RSL). In general, each RSL expression comprises several resource allocation requests. Doing so, *resource brokers*, implementing domain-specific resource discovery and selection policies, will transform abstract RSL expressions into progressively more specific requirements. For instance, an application might specify a computational requirement in terms of floating-point performance (FLOPs), and then be narrowed to a specific type of computer (e.g. IBM SP2). The final step in the resource allocation process is to decompose those expressions into a set of separate requests, in order to dispatch each one to the appropriate GRAM.

Additionally, due to the dynamic nature of grid environments, the toolkit provides the Globus Metacomputing Directory Service (MDS). This service is designed to allow applications to adapt their behavior in response to changes in the Grid system. Therefore MDS provides an information-rich environment in which information about system components is stored, and is accessible to applications. The data stored includes information about architecture type, operating system version, amount of memory, network bandwidth on a computer, and so forth. Thus, this service reveals itself as being fundamental to the resource discovery process.

[7] A networked virtual supercomputer, constructed dynamically from geographically distributed resources linked by high-speed networks.

Security and Authentication. One issue that complicates the use of Grids in practice is the user authentication among different administrative domains. The Globus Security Infrastructure (GSI) is the component that addresses that challenge, thereby bridging disparate security mechanisms. GSI provides basic support for delegation and policy distribution in a standard way (i.e. using the same protocol and understanding each other's identity credentials). Moreover, different local mechanisms and N-way security contexts are not supported by any other system aside GSI. Hence, GSI internal design emphasizes the important role that standards have to play in the definition of grid services and toolkits. Furthermore, GSI is based on standard PKI (Public Key Infrastructures) technologies. It uses SSL (Secure Socket Layer) or WS-Security (Web Services Security) protocols with X.509 certificates for authentication, message protection, and identity check (more details at Section 2.2). Also, it is possible to use the Kerberos protocol to authenticate a user on the Grid. Therefore, a Kerberos-to-GSI gateway is needed to translate certificates from Kerberos to GSI and vice-versa. Once a user is identified by the GSI, all of the Globus services will know if the user is who he or she claims to be.

2.4.1.2 Condor

The Condor [TTL05] project started in 1988 and was derived from the Remote-Unix [Lit87] system, which allowed the integration and use of remote workstations. Condor aims to maximize the utilization of workstations with as little interference as possible from users. This system is intended to operate in a High-Throughput Computing[8] environment. In that way, Condor provides two important functions. First, it makes available resources more efficient by putting idle machines to work. Second, it expands the resources available to users, by functioning well in an environment of distributed ownership. Additionally, Condor has proven to be an efficient system to improve the productivity of computing environments [LLM88].

Architecture. Within Condor, participant machines are organized in agglomerates forming what is called a Condor Pool. Generally, machines of an agglomerate belong to the same administrative domain, such as a company department or a local network. Nonetheless, such organization is not mandatory. In addition, as more machines join a Condor pool, the quantity of computational resources available to the users grows.

Applications can benefit from Condor without their source code being modified. However, code that can be re-linked with the Condor libraries gains two further abilities: the jobs can produce checkpoints and they can perform remote system calls. The checkpoint of a program [LTBL97] is the complete set of information of its state. In this way, a job could be continued on another machine from the point that it has stopped (known as process migration). For long-running computations, the ability to produce and use checkpoints can save days, or even weeks of accumulated computation time (e.g. machine crashes or rebooting for an administrative task). However, checkpoints have some limitations, such as not having support for process communication nor for parallel applications. Also, checkpoints are platform-dependent, which could be a problem because of machine heterogeneity. Remote system calls allow Condor to preserve the security of remote machines. If a job running remotely does a system call, for example to do an input or output function,

[8] Computing environment that delivers large amounts of computational power over a long period of time.

the operation will be executed on the machine where the job was submitted (Condor handles this process). Therefore, the data is not on the remote machine where it could be an imposition. Moreover, a user submitting jobs to Condor does not need accounts on the remote machines.

Condor allows the execution of parallel applications. Those applications could be based on MPI (Message Passing Interface)[9] or based on PVM (Parallel Virtual Machine).[10] Though the support for MPI-based applications is limited [Wri01]. For example, Condor needs dedicated scheduler to support MPI-based applications. In contrast, PVM-based applications are well supported over shared resources.

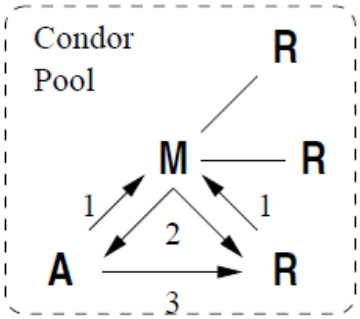

Figure 2.3: Step 1: The agent and the resource advertise themselves to the matchmaker. Step 2: The matchmaker informs the two parties that they are potentially compatible. Step 3: The agent contacts the resource and executes a job.

The Condor kernel works as follows: The user submits jobs to an agent. This agent is responsible for caching jobs in memory while finding available resources. Agents and resources (the servers which contain resource information) advertise themselves to a matchmaker, which is responsible for introducing potentially compatible agents and resources (i.e. resources satisfying agent needs). This process is illustrated in Figure 2.3. Once introduced, an agent is responsible for contacting a resource and verifying that the match is still valid. To actually execute a job, each side must start a new process. At the agent, a shadow is responsible for providing all of the details necessary to execute a job. At the resource, a sandbox is responsible for creating a safe execution environment for the job and protecting the resource from any mischief.

Agents and resources (either or both) may run in the same single machine. The unique matchmaker should be selected, and represents a community as a single entity. Each of the three parties (agents, resources, and matchmaker) are independent and individually responsible for enforcing their owner's policies.

Additionally, in 1994 the gateway flocking [ELvD+96] emerged. This project attempted to allow users to share resources across organizational boundaries. Moreover, cross-pool matches will be made if the owners of each pool agree on policies for sharing load. A very large system may be grown incrementally with administration only required between adjacent pools.

Discussion. The Globus project has made great impact on the Grid computing field: it is one of the most significant implementations based on Grid infrastructures. The user authentication and authorization mechanisms are mainly directed to institutional environments, therefore contrasting with the vision of GridP2P on

[9] Applications which communicate with other ones during their execution: `http://www-unix.mcs.anl.gov/mpi` accessed on October 2008

[10] Applications that can be executed on heterogeneous machines: `http://www.csm.ornl.gov/pvm` accessed on October 2008

which an open access environment is provided, disentangling the users' credentials.

With respect to Condor, jobs rely on executable binaries in which compatible machines are needed in order to run them (i.e. the matchmaker will provide that compatible relation). In contrast with GridP2P, the heterogeneity of machines is not a problem, since jobs are just data files, constituting the input for applications. Moreover, many features that are provided by Condor require some degree of expertise (i.e. configurations are needed), whereas GridP2P envision for a generic tool that does all the necessary work seamlessly.

2.4.2 Master-Slave Model

The most successful distributed cycle sharing platforms in reaching the average user are based on the master-slave model. In this model, typically there are servers (masters) which distribute workunits among client machines (slaves). By their turn, clients process the workunits and send the computed results back to the servers. Generally, the servers are the only resource consumers, while clients are just able to provide resources.

This model can suit a myriad of applications. For instance, from the growth of Internet in mid-90s, two projects emerged: the GIMPS[11] for searching prime numbers, and the Distributed.net[12] for brute-force decipher demonstrations.

Users' interest about distributed cycle sharing has been growing mostly driven by the success of SETI@home, which relies upon the master-slave model.

2.4.2.1 Berkeley Open Infrastructure for Network Computing (BOINC)

BOINC [And04] is a platform for volunteer and distributed computing. BOINC emerged as a generalization of SETI@home, which got a good acceptance across the world. Hence, this new platform became useful as well as for other distributed applications in areas as diverse as mathematics, medicine, molecular biology, climatology, and astrophysics. Moreover, BOINC was meant to be used mostly by scientists, not by system programmers or IT professionals. Therefore, this system provides a set of simple, and well-documented, tools for managing volunteer computing projects.

BOINC consists of a server system and client software that communicate with each other to distribute, process, and return workunits. In essence this system allows one to donate the unused CPU cycles on his or her computer to do scientific computing, contributing so to a noble collective cause.

The client application can operate in several modes: (i) as a screensaver that shows the graphics of running applications; (ii) as a Windows service, which runs even when no users are logged in and logs errors to a database; (iii) as an application that provides a tabular view of projects, work, file transfers, and disk usage; (iv) and as a UNIX command-line program that communicates through stdin, stdout and stderr, and can be run from a cron job or startup file.

BOINC also provides a system of credit and accounting for motivating users to donate their idle CPU cycles, thereby attributing a rank to each participant of a project. Like SETI@Home, these credit units are a weighted combination of computation, storage, and network transfer.

[11] `http://www.mersenne.org` accessed on October 2008

[12] `http://www.distributed.net` accessed on October 2008

Architecture. The backend server relies on a relational database that stores information related to a BOINC project, such as descriptions of applications, platforms, versions, workunits, results, accounts, teams, and so on. The scheduling server is responsible for the distribution of tasks (instructions), taking into account the specifications of each client machine. For instance, the server will not give tasks that require more RAM than a client has. Getting instructions from the scheduling server is the first step for a user. Secondly, a client machine should download input files from a data server and process them. After all the computations are done, the output files should be returned to the data server. Later, a client machine reports the completed tasks to the scheduling server, and gets new tasks. This cycle is automatically repeated indefinitely.

Security. In the predecessor project, SETI@home, some participants attempted to cheat projects to gain credits, while others submitted entirely falsified results. BOINC was designed, in part, to combat these security breaches. This platform uses redundant methods to prevent the occurrence of falsified work: each workunit is distributed to more than one client and the results are validated by checking their discrepancies (i.e. one result is only accepted if there is a majority supporting it).

In order to avoid the forged distribution of applications (i.e. the possibility for an attacker to distribute some client code as if it belonged to a project in which a client participates), BOINC uses digital signature over the application code. Hence, each project has a pair of cryptographic keys that are used to authenticate the distributed applications.

For preventing denial-of-service attacks (DoS) to the data server, where a malicious user uploads giant output files occupying the whole free space at the server, BOINC allows project administrators to specify a maximum file size. However, applications of a BOINC project could potentially cause (intentional or accidental) damage to one's computer, therefore implying a relation of trust between users and project owners. BOINC does not provide any sandbox-based mechanism to isolate the execution of its software.

2.4.2.2 BOINC Extensions for Community Cycle Sharing (nuBOINC)

This project [dOeSVF08] is a customization of the BOINC platform that makes easier for any user to create and submit jobs by allowing the use of widely available commodity applications. With BOINC, it could be difficult for an ordinary user to set up the required infrastructure, develop applications to process data (input files), and gather enough computer cycles for a project. Especially, if one does not have the required skills to efficiently use BOINC, despite the provided tools that try to facilitate that process. Besides that, projects in BOINC are usually big and must have a large visibility for attracting users to participate in. Doing so, some computer user communities are being excluded and cannot take advantage of remote idle cycles to speed their jobs. These communities in general just use highly available commodity applications that need to do a lot of computational work and/or processing a large amount of data.

Architecture. The BOINC client and server (database) were modified in order to allow the use of remote idle cycles for some commonly available software (e.g. POV-Ray). Also, an auxiliary application, Application Registrar, was developed for the integration of applications in BOINC projects. In that way, integrated applications may be then executed remotely.

In the client side there is a component, the comBOINC Project application, that is responsible for invoking commodity applications which are needed to process some input files. Contrasting with regular BOINC, its project application has all the code to process the data. In this way, workunits are checked if they need to be processed in a different way from regular cases (i.e. using the comBOINC extension). In addition, the modified client, nuBOINC, also sends the identification of the commodity applications to the server.

Firstly, the commodity applications need to be registered at the server database (implying some modifications to the regular model of BOINC) and then at the local database of registered applications. This is done throughout the Application Registrar. After logging in at the comBOINC project, a user can supply the input files and the job parameters through a web page that is displayed. Before a request can be made, the nuBOINC searches at the local database for the commodity applications concerning the jobs; if it is found, the client sends a list of those applications to the server along with a regular work request. The answer to this request is then made to all available clients connected to the project, therefore containing the input files to be processed. If a certain client does not have the required applications, it attempts to download them from the server. Finally, the applications are executed with the respectively input files. After that, the output files are sent to the server by means of PHP scripts.

Discussion. Despite the BOINC success as a system for public-resource computing, there is no communication at all between slave nodes (participants). Hence, BOINC is not as much flexible as in a way that any participant can consume cycles from other participants connected to a same project. Nevertheless, projects like BOINC claim that their experience showed not only many resources available throughout the Internet, but also many users willing to donate their idle CPU cycles. This comes to strengthen the GridP2P project goals.

The nuBOINC project attempts to overcome the lack of flexibility presented by BOINC. Thus, nuBOINC allows any ordinary user to create and submit jobs to a project, making use of commodity applications. In this way, some goals are shared along with the GridP2P. Nonetheless, nuBOINC revealed some problems with the original BOINC scheduling policies: obtained speedups (from experiences) were not optimal, because of the BOINC policies that aim at infinite workpile applications (i.e. with thousands of jobs and with continuous source of jobs); no fairness on the order of the execution of user submitted jobs, due to the BOINC policies which do not relate BOINC clients with submitted jobs. Furthermore, there are security questions in the nuBOINC that are unclear: the download and installation of BOINC applications on a client machine (i.e. after a client receives a workunit to be processed), and how to guarantee that a client is running an application inside a sandbox mechanism.

2.4.3 Peer-to-Peer Systems

The growth of the Internet, users and bandwidth, requires infrastructures that can take advantage of a diverse wealth of applications [AE07]. These infrastructures need to address three main challenges regarding Internet-based applications: scalability; security and reliability; flexibility and quality of services. The decentralized and distributed nature of Peer-to-Peer systems makes them robust over very common failures in centralized systems.

One of the application areas of Peer-to-Peer is the processor cycle sharing. The high performance com-

puting needs, as well as the computer power unused, are an incentive for using Peer-to-Peer applications to bundle that computer power. In this way it is possible to acquire computer power cheaper than a supercomputer can provide. This is achieved by forming a cluster in which a single computer is transparent and all the networked nodes are merged into a single logical computer.

Unstructured Network. In a Peer-to-Peer unstructured network, the placement of resources, like files, is not related at all with the overlay topology. Generally, the location of resources is made through searching algorithms, such as the flooding method, thereby propagating queries through the overlay nodes until the resource is found. These search mechanisms have some impact in what concerns availability, scalability, and persistence. Additionally, unstructured systems are generally more appropriated for highly-transient node populations. Examples of unstructured systems are Napster,[13] Publius [WRC00], Gnutella [Rip01], Kazaa [LKR04], Edutella [NWQ+02], FreeHaven [DFM00], as well as others.

Structured Network. In structured networks, the overlay topology is tightly controlled and resources, or pointers to them, are placed at precisely specified locations. These systems attempt to address the scalability issues showed by unstructured networks. Therefore, structured systems provide a mapping between content (e.g. file identifier or a hash to a file describing computer resources like CPU) and location (e.g. node address), in the form of a distributed routing table. In this way, queries can be efficiently routed to the node with the desired resource. However, using exact-match queries as a substrate for keyword queries remains an open research problem. A drawback of structured systems is the intrinsic complexity of maintaining the structures that comprise the routing table, especially in the presence of nodes which are joining and leaving at a high rate. Examples of structured systems include Freenet [CSWH01], Chord [SMK+01], CAN [RFH+01], Tapestry [ZKJ01], among others.

- Freenet is a loosely structured system that uses file and node identifier similarity to produce an estimate of where a file may be located, and a chain mode propagation approach to forward queries from node-to-node.
- Chord is an infrastructure whose nodes maintain a distributed routing table in the form of an identifier circle, on which all nodes are mapped. Also, to accelerate lookups, each node has an associated finger table that stores pairs (key identifier,address of node).
- CAN is a system using *n*-dimensional Cartesian coordinate space to implement the distributed location and routing table, whereby each node is responsible for a zone in the coordinate space.
- Tapestry, as well as Pastry, are based on the plaxton mesh data structure, which maintains pointers to nodes in the network whose identifiers match the elements of a tree-like structure or identifier prefixes up to a digit position.

These systems were compared and it turns out that, either in terms of routing table space cost and of performance, they have a complexity of $O(\log N)$, where *N* is the number of nodes on the network. Maintaining the routing table in the face of transient node populations is relatively costly for all systems, though perhaps more for Chord, as nodes joining or leaving induce changes to all other nodes.

[13] `http://ntrg.cs.tcd.ie/undergrad/4ba2.02-03/p4.html` accessed on October 2008

Regarding important mechanisms underlying distributed cycle sharing systems, such as resource discovery mechanisms (Section 2.3), Pastry is the predominant choice, therefore constituting a good solution for the GridP2P.

2.4.3.1 Pastry

Pastry is a generic, scalable and efficient substrate for peer-to-peer applications. Pastry nodes form a decentralized, self-organizing and fault-tolerant overlay network within the Internet. Pastry provides efficient request routing, deterministic object location, and load balancing in an application-independent manner. Furthermore, Pastry provides mechanisms that support and facilitate application-specific object replication, caching, and fault recovery. One of the Pastry properties is the sense of proximity: it looks for minimizing the traveled distance by messages in accordance with a scalar metric of proximity, such as the number of hops in the IP routing.

Architecture. Each Pastry node has a unique 128-bit identifier (nodeId), which can be generated from a hash function applied to the IP address, or public key, of the node. The nodeId indicates the node position in a circular namespace of keys from 0 to $2^{128} - 1$. Given a key, Pastry reliably routes a message to the Pastry node whose nodeId is numerically closest to the key. Routing to any node takes less than $\log_{2^b} N$ hops, where *b* is a configuration parameter, and *N* is the number of nodes.

The nodeId and the keys are sequences of digits with base 2^b. In each step of the routing, a node routes messages to other node whose nodeId shares with the key, at least, one more digit than the ones that are shared with the actual node. If no node is known, the message is routed to the node whose nodeId shares a prefix with the key, and it is numerically closer than the actual node.

For supporting this routing procedure, each node has a routing table, a set of neighborhood nodes, and a leaf set with *l* (an even integer parameter) elements. The routing table is formed by $\lceil \log_{2^b} N \rceil$ rows, each with $2^b - 1$ entries. Each table entry contains the address of potential nodes whose nodeId has an appropriate prefix (relative to the row). The neighborhood set is not normally used in routing messages, though it maintains locality properties, and is also used in the joining method. Finally, the leaf set is compounded by the set of nodes with the $l/2$ numerically closest larger nodeIds, and the $l/2$ nodes with numerically closest smaller nodeIds, relative to the actual node.

The routing procedure works as follows: Given a message, a node verifies if the respective key is in the leaf set. If so, the message is directly routed to the target node. Otherwise, the routing table is used and the message is sent to the node that shares a common prefix with the key (at least one digit).

With respect to the joining method, an arriving node can initialize its state by contacting a nearby node that, by its turn, will send a special message whose key is the nodeId of the new node. In this way, the special message is routed to the closest node of the new one. In return, the arriving node gets the leaf table of the closest node to it, and the *i*th row of the routing table from the *i*th node encountered along the route. Therefore, the new node can correctly initialize its state and notify nodes that need to know of its arrival.

For handling failures, nodes periodically exchange keep-alive messages between their neighbors. If a node is unresponsive for a period *T*, it is presumed failed. Each node, in the leaf set of the failed node, is

notified and their state updated. One node that recovers from a failure contacts the nodes of its (last known) leaf set, updating its state. After that, the node notifies every member in the leaf set about its presence.

2.4.3.2 PAST

On the top of Pastry, the PAST [RD01b] is a large-scale, peer-to-peer archival storage utility that provides scalability, availability, security and cooperative resource sharing. Also, PAST is often used for caching jobs on distributed cycle sharing systems. Files in PAST are immutable and can be shared at the discretion of their owner. There are two main operations regarding storage management: insert and lookup, allowing the insertion and retrieval of files on the system. In addition, files can either be just in memory or in hard drives as persistent storage.

A file is stored in the Pastry node whose nodeId is numerically closest to the hash generated from the file name (fileId). For replication purposes, files are copied to the neighbors that are in the leaf set. Also, the route of queries through the overlay is done by the file name hash, corresponding to the key on the Pastry overlay.

2.4.3.3 Cluster Computing on the Fly (CCOF)

The Cluster Computing on the Fly [LZLZ04] is an open peer-to-peer system that seeks to harvest idle CPU cycles from ordinary users. CCOF attempts to reach the average citizen by providing an open environment that, in contrast with Grid systems, does not require membership in any organization. Moreover, the CCOF project focus on the following research issues: Incentives and fairness; Resource discovery; Verification, trust and reputation; Application-based scheduling; Quality of service and performance monitoring; Security.

This project identifies five classes of applications that can benefit from harvesting idle cycles, described as follows.

1) Infinite workpile: applications that consume huge amounts of compute time lying on a master-slave model. For example, applications supported by BOINC (e.g. SETI).

2) Workpile with deadlines: applications similar to the infinite workpile ones, but their needs for compute cycles are more moderate. They require results to be returned before a specified deadline (regarding a business presentation, school project, among other).

3) Tree-based search: applications requiring substantial compute cycles with loose coordination among subtasks. For example: distributed branch-and-bound algorithms, alpha-beta search, and so on.

4) Point-of-presence (PoP): applications consuming minimal cycles but require placement throughout the Internet. For example, distributed monitoring applications (security monitoring, traffic analysis, and so on).

5) Tightly-coupled parallel: applications that launch new processes (e.g. using fork in C) during their computation, requiring inter-process communication. In contrast, inter-process communication is not allowed in many systems for security reasons. Moreover, this type of applications require not only great bandwidth, but also the ability to coordinate large numbers of processes that spread out over the Internet (still a challenge). Currently, CCOF does not address tightly-coupled parallel applications.

Architecture. The CCOF architecture provides a lightweight mechanism for forming community-based cycle-sharing overlay networks. Within each overlay, any node can be a client seeking idle cycles or a donor volunteering its cycles. CCOF also includes a reputation system to support fairness, quality of service, and security.

Communities in CCOF can be organized through the creation of overlay networks based on factors such as interest, geography, performance, trust, institutional affiliation, or generic willingness to share cycles. Doing so, the overlay management (of host communities) uses a central certificate-granting authority to exclude undesirable hosts.

The application scheduler is responsible for the selection of nodes in order to perform the required computation. Therefore, this scheduler also negotiates the access to the nodes, exports the application code, and gathers the results from nodes with correctness verification.

Additionally, a local scheduler looks for idle cycles and negotiates with the application scheduler the tasks to perform. This negotiation process uses local criteria with regard to trust level, fairness and performance.

Regarding the scheduling of workpile jobs (which heavily demand for free cycles), this is made through the CCOF's wave scheduler. This scheduler captures available night time cycles by following night timezones around the globe. A CAN-based overlay is used to organize hosts by timezone. Moreover, this method avoids the interruption from users reclaiming their machines; in addition there is higher guarantee of ongoing available cycles.

Concerning the resource discovery, CCOF lies on four basic search methods: rendezvous points, host advertisements, client expanding ring search, and client random walk search (described above in Section 2.3). Each one was experimented within CCOF and the rendezvous points revealed to be the best (i.e. with more global performance). This method also showed high job completion rate and low message overhead, although it favors large jobs under heavy workloads.

The correctness of returned results for workpile jobs is verified by using a quiz mechanism. Quizzes are easily verifiable computational problems whose solutions are known beforehand. There are two methods being investigated: standalone and embedded quizzes. Standalone quizzes are packaged so that a (malicious) host cannot distinguish quiz code from genuine application code. The second method embeds simple short quizzes into the application code. Additionally, the quiz results could be stored in a reputation system, such as Eigenrep [KSGM03].

With respect to PoP applications, the scheduling relies on scalable protocols for identifying selected hosts in the community (overlay) network. Therefore, each ordinary node is k-hops from C of the selected hosts (d-hop dominating set problem). This is useful for leader election, rendezvous point placement, monitor location, and so forth.

Concerning security, the local execution of an external application code is made through a virtual machine. Thus, a sandbox is created not only to protect a local computer but also to control its resources. In addition, nodes will have to deny access to clients that are untrusted, in accord with trust and reputation systems.

2.4.3.4 OurGrid

Besides the immense research for running tightly-coupled parallel applications (i.e. applications needing some sort of communication during their execution time) over Grid infrastructures, the authors of MyGrid [CPC+03] claimed that not many users utilize Grids with that type of applications. Doing so, the MyGrid has only focused in bag-of-tasks applications.[14] This way, MyGrid's solutions have come out simpler and best fitting in the uniform distribution, in the handling of machine heterogeneity, and in the Grid dynamism.

MyGrid and others, such as Globus, have solved the problem of sharing resources across multiple administrative domains. However, getting credentials to access outsider domains is still a burden on these systems: they generally rely on personal negotiations between users and domain administrators. To address this problem, MyGrid's developers presented us with the OurGrid [ACBR03]: a peer-to-peer network of sites, that shares resources equitably, forming a Grid in which any user of these sites can have access to without much burden.

Architecture. The OurGrid has three main components (as illustrated in Figure 2.4): the MyGrid which acts as

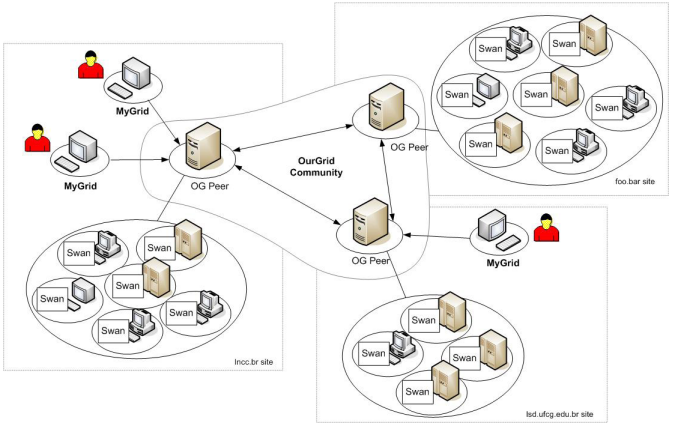

Figure 2.4: OurGrid Overview

a user broker; the OurGrid Community that encompasses diverse domains; and the SWAN which guarantees the access to resources in a secure way [ACGC05].

The MyGrid component receives from users job descriptions and an entrypoint to a peer in the OurGrid Community. In this way, the MyGrid can request to that peer available compatible resources and to schedule the user application over those resources. Additionally, scheduling heuristics can be provided by the broker in order to improve the performance of different types of applications.

A set of peers comprising the OurGrid Community, and representing each available site, are responsible for gathering a set of resources upon a request made by a local user (i.e. belonging to that site/domain). If there are not enough resources available among the machines of a site, the peer then contacts other sites, within the OurGrid Community, to obtain a satisfactory number of resources which can comply with the needs of the submitted request. These further contacts among sites/domains do not require any human negotiations between users and site administrators (nor resource's owners).

[14] Parallel applications whose tasks are independent and don't need to communicate between them during execution.

The SWAN component is a sandboxing mechanism aggregated to each machine that can provide resources to the OurGrid Community. This mechanism prevents remote users from either damaging local machines or use them for malicious purposes. Therefore, each time a machine receives a remote task, a virtual machine is created holding that task within, and isolating the execution control of that task from the operating system which is running on that machine. Moreover, these virtual machines have controlled access to resources and cannot access to the network interfaces (which works very well for bag-of-tasks applications).

Within each site, machines can access directly to the resources of each other, whereas to access resources from other sites, is used the concept of network of favors [Discouraging Free Riding in a Peer-to-Peer CPU-sharing Grid]. Basically, the network of favors is a resource allocation scheme based on the reputation of each peer. Thus, peers that donate more resources to the community are then prioritized every time they need resources. In addition, requests received from a same site are always preferred over requests from foreign sites (i.e. a machine can only perform work on a foreign site if that site is not currently busy with local requests).

Whenever machines access resources directly (within the same domain) they just, in general, rely on a traditional authentication system and they only need to provide the username and password of their users. Although, for accessing machine resources in other sites, digital certificates X.509 are used. Hence, the identity of each site/domain can be verified.

Regarding supported applications, there are two main approaches: script-based and embedded applications [BG07]. In the script-based approach, applications are written in text files using a very simple language provided by the OurGrid. These applications must have entries for all tasks that are intended to be executed. Such entries contain commands about transfer input data and executable code to the grid machine, commands to validate the results produced, commands to be executed on the grid machine, and so forth. In the other approach, embedded applications incorporate calls to the OurGrid middleware, and therefore they may access to more sophisticated features.

Essentially, the OurGrid controls agglomerates of grids in a peer-to-peer way, and therefore widening the range of potentially available resources that a user/application can access to. Furthermore, the OurGrid project revealed to be useful, efficient, and simple in terms of the deployed solutions.

2.4.3.5 Ginger - Grid Infrastructure for Non-Grid Environments

The Ginger [VRF07] is a project focused on enhancing desktop applications to run faster, by conveying the Grid and Peer-to-Peer models into a generic cycle-sharing platform. By doing so, the main idea is that any home user may take advantage of idle CPU cycles, from other computers, to speedup the execution of unmodified popular applications. In return, users may also donate their idle cycles to speedup other users' applications. Ginger attempts to leverage this process of sharing by introducing a novel application and programming model that is based on the Gridlet concept.

A Gridlet is a work unit which contains a chunk of data and the operations to be performed on that data. These operations often rely on unmodified application binaries, but not always. The Gridlet is also semantic-aware in relation to its carried data, and hence the data is not treated in a equal mechanical manner. Moreover, every Gridlet has an estimated cost associated with it: the CPU cost of processing it, and the bandwidth cost

of transferring it.

Ginger tries to face the drawback of the institutional grids in regards to the sharing and negotiating of resources across different institutions and domains. All the more so, this project aims to span boundaries of typical grid usage, enabling the grid, and its inherent powerful features, to reach the typical Internet user. This is accomplished upon a peer-to-peer model which provides large-scale deployment in a self-organized manner. Ginger also refers the necessity of integrating its model with popular peer-to-peer file-sharing tools, such as the open-source BitTorrent which is widely used in any corner of the world map.

Architecture. The Ginger presents itself as a middleware platform following an architecture composed of layers, mainly to favor portability and extensibility. These layers include the management of the overlay network, for gridlet exchange among nodes; and the management of gridlets, for task creation and reassembly of results.

Gridlet creation is based on XML-based format descriptions for the partition of the input data that will feed the desktop applications. These descriptions include headers that must be adapted in each gridlet, fragments of files that should be kept within the same gridlet, and rules for the reassembly of gridlet results.

The execution of gridlets, in peer machines, is made in the context of virtual machines, supporting both bytecode-based and native code gridlets. In this way, peers need not trust gridlet code. Data-injection into applications relies on file semantics and runtime arguments (e.g. via a pipe or as a high-level language data-structure), widely used in Grid infrastructures and desktop applications. The gridlet data payload after execution might be bigger or smaller than what it was before, depending on the type of application (e.g. cryptographic challenge versus high-resolution ray-tracing).

The computational cost of a gridlet is defined both in terms of CPU and bandwidth. This cost is always assessed against the cost of serving a standard gridlet in a standard machine. To obtain the reference gridlet, a computational challenge may be used to calculate the CPU cost, along with the assessed bandwidth to transfer the challenge generated data from one peer to another. With this, submitting peers need only to process some sample gridlets locally, and weight the cost of their execution against both current CPU-load and the average time to compute the standard gridlet on their machines. Additionally, as gridlets are computed, a peer becomes able to fine-tune the cost of similar gridlets in the future.

Furthermore, Ginger favors and eases the study and evaluation of the best remote execution strategy with regards to the trade-off between the local execution of code versus remote execution and data transfers (e.g. users can interactively instruct the gridlet managers to create gridlets with maximum CPU cost while reducing bandwidth cost).

Discussion. The use of Peer-to-Peer networks, allied to mechanisms of resource discovery and job assignment, by the GridP2P is fundamental to provide highly dynamic environments over wide networks (e.g. Internet).

The PAST appears as a plausible solution for caching results of computed jobs. In this way, bandwidth and CPU utilization are reduced. Moreover, the privacy of users (machines) is guaranteed. However, this solution has some drawbacks: the overhead for sending results to the cache (PAST), and the right time prediction to

retrieve results from the cache by the node that submitted the job. Also, load balancing could be a problem (i.e. due to statistical variation in the assignment of nodeIds and fileIds, as well as the difference between the size of files that are distributed). Furthermore, PAST has the great advantage of already being provided by the FreePastry.[15]

The CCOF project shares the GridP2P goals of deploying a generic Peer-to-Peer Grid infrastructure. CCOF is the most relevant case study (as far as we know) in the GridP2P context. Although, CCOF has many design differences from the GridP2P. The CCOF overlay gives access to joining nodes just based on trust and reputation systems. Despite that model being a good solution for global control, CCOF disregards security policies which allow a more fine-grained control over local resources.

With the OurGrid, negotiations between different organizations are leveraged through a peer-to-peer model. Nonetheless, applications need to be modified in order to run on top of this platform, and the data-based parallelism, envisioned by the GridP2P, is not exploited. In addition, tasks need to be created in a manually manner, and in the case of script-based applications, it could be very costly, as the number of tasks could be very high and each of them requires single configuration. Moreover, there are no guarantees about the quality of service obtained from the idle resources, as their idleness is not measured (i.e. machines are not distinguished by their resource availability levels).

Ginger shares our goals of deploying a scalable peer-to-peer grid infrastructure. We have no doubts that Ginger is a very important case of study, portraying the next step to the Grid evolution. Nevertheless, we think there are some important issues, such as the portability and security, that require further attention, as we describe next.

Popular desktop applications often rely on binaries that are meant to be executed as native code on peer machines. Therefore, it is not clear in Ginger how gridlet operations, consisting of those application binaries, would run on any general-purpose virtual machine, since they can be targeted to different architectures and operating systems. Plus, most of those kind of applications need to be installed, requiring root access, user presence, amongst other. Despite virtual machines being used, the control over resource usage is not taken into account, unlike the GridP2P where it is provided a flexible way to handle resource access, and consumption, through the security policies.

[15] `http://freepastry.rice.edu` accessed on October 2008

3 Architecture

In this project we propose a middleware platform, combining Grid and Peer-to-Peer technologies, that seeks to exploit parallel execution of commodity applications. Any user is then able to act as a resource consumer, using idle CPU cycles from other machines, or as a resource provider, granting access to his own idle cycles, or as both. Above all, we want to enable the Grid in large scale where any ordinary user may access without much burden. In addition, we rely on security policies to control the utilization of shared resources among different users.

Firstly, we present an overview of the architecture which composes the GridP2P platform, with various abstraction levels. We also present use case scenarios, and describe the responsibility that each component assumes. Secondly, the inter-component communication is explained when there are jobs to be computed. Finally, we present and describe relevant deployed mechanisms concerning each component.

3.1 Design Requirements

The GridP2P platform is a distributed cycle sharing platform aiming at dynamic environment. Doing so, a scalable overlay network is needed to handle the nodes and their inter-communication in an easy way. Upon this overlay, discovery mechanisms are required to locate resources within the network. Our platform should also support the creation and scheduling of jobs (where a job comprises a set of tasks) among nodes, so that applications may be parallelized. Moreover, specific mechanisms concerning each type of supported application are needed. Finally, security mechanisms are also necessary to control the resource utilization in a myriad of possible and complex usage scenarios.

3.2 Platform Architecture

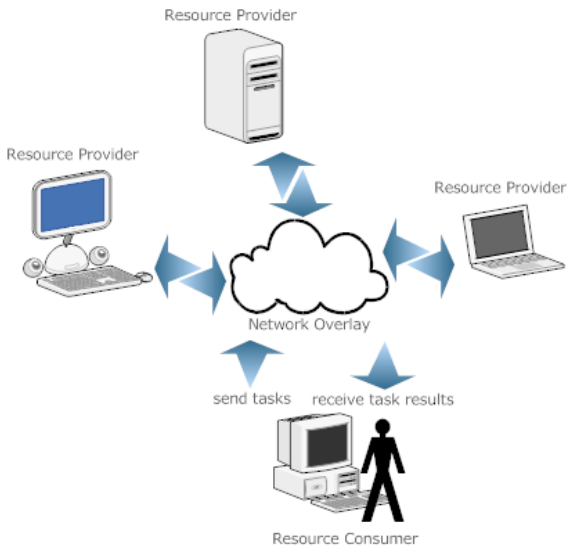

Figure 3.1: Usage Model

Figure 3.1 shows a use case where a machine, acting as a resource consumer, distributes tasks among available machines, resource providers, in order to perform a CPU-intensive job demanded by a user. Resource providers receive the tasks, compute them, and send the results back to the consumer node (the job holder). All machines are connected through an overlay network, which is built on top of another network (i.e. Internet) and provides services of routing and lookup.

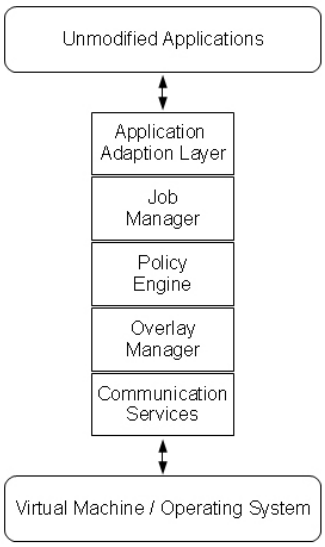

Figure 3.2: System Architecture

The proposed architecture relies on a vertical layer approach, depicted in Figure 3.2. Following, we describe the function of each layer.

Unmodified Applications. This level represents the applications that will run on top of our middleware. The application parallelism is exploited at the data-level, and thus applications do not need to be modified.

Application Adaptation Layer. The Application Adaptation Layer consists in a customization over a generic Job Manager. This layer defines which applications are supported by the local machine. Therefore, specific mechanisms for handling each of these applications are provided. For example, launching applications with the correct parameters and input files. Moreover, these mechanisms are built on loading time and are based on formal application descriptions.

Job Manager. This component is responsible for creating and scheduling tasks in accordance with available resources at the moment. The tasks, as divisions of input files, are distributed among available machines. After the computation of the tasks is completed, this module collects the results and builds the final output of the respective application. In the inverse flow, the Job Manager is also responsible for receiving and computing tasks from remote machines in accordance to the Application Adaptation Layer.

Policy Engine. The Policy Engine component is responsible for enforcing local policies that can recall on

the history of past events. Some of these policies may be defined, by the main GUI, in a way that is understandable for any ordinary user. Nonetheless, for more specific actions, policies need to be defined in XML files whose structure relies upon the xSPL language, thus requiring more expertise. Furthermore, the policy engine acts as a filter between the Overlay Manager and Job Manager layers, deciding which messages may pass.

Overlay Manager. This layer comprises four components, as depicted in Figure 3.3. It is responsible for the operations of routing and addressing on the overlay network. In addition, mechanisms of management and discovery of resources are included. Also, local resource utilization is monitored by this component. Any changes in resource availability are announced to the neighbor nodes. Furthermore, this component contemplates a distributed storage system used as a cache for storing computed tasks.

Figure 3.3: Overlay Manager Components Overview

Communication Service. The Overlay Manager uses this layer to send messages to the overlay network. Also, whenever a message coming from the network is received, the Communication Service analysis the message in the first instance, and then delivers it to the adequate handler routine in the Overlay Manager.

Operating System/Virtual Machine. The whole platform is intended to work directly upon Operating System or Virtual Machine. For improved security a Virtual Machine may be used as a sandbox mechanism. In this way, we can guarantee controlled access to machine resources, as well as prevent some malicious code from damage one's computer, in case input files consist of scripts, programming code, and so forth.

3.3 Resource Access Model

The procedure for accessing resources (illustrated in Figure 3.4) works as follows: First a user specifies an application, parameters and input files through the GUI. The GUI contacts the Job Manager (JM) - step 1 - in order to create and distribute tasks to available machines. In effect, the JM first contacts the Overlay Manager (OM) to look for available resources. The tasks are then created according to the information retrieved from the OM. Next, the JM contacts the Application Adaptation Layer (AAL) to process one task - steps 2 and 3 -, and also contacts the Policy Engine (PE) - step 4 - to apply policy-based restrictions to the remaining tasks.

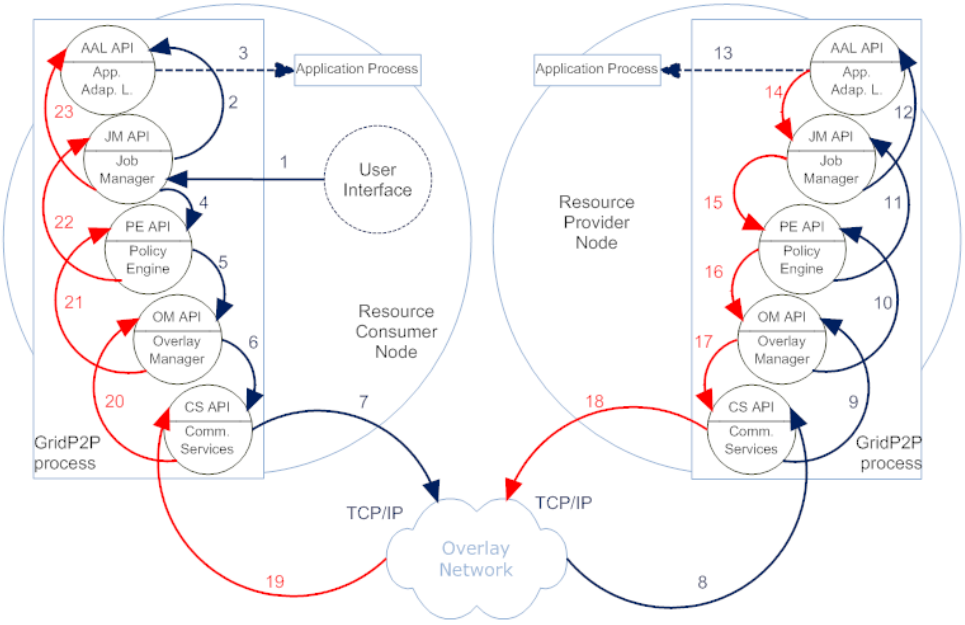

Figure 3.4: Resource Access Model

By its turn, the PE sends the tasks to the OM, which will distribute them among available nodes - steps 5, 6 and 7.

On receiving a task through the Communication Service (CS), a node sends it to the PE which will assess if the message can be delivered to the JM - steps 8, 9, and 10. If so, the JM with the AAL may compute the given task - steps 11, 12, and 13. Soon after task completion, it is sent to the PE - steps 14 and 15 - for applying of policy-based restrictions. Then, the computed task is sent back to the machine holding the job through the OM and CS layers - steps 16, 17, 18, and 19.

The OM and CS layers receive the data and send it up to the PE - steps 20 and 21 - where data may be evaluated against security policies. If no policies can be applied (i.e. no action to take), the PE lets the data pass to the JM - step 22. The JM gathers the results and, when all tasks are completed, it builds the final output, along with the AAL layer, and notifies the user through the GUI - step 23.

3.4 System Interfaces

There are two modes by which users may interact with our platform: the GUI (Graphical User Interface) and the console. The GUI (in Figure 3.5) provides a simple and minimalist interface, so that any ordinary user may utilize it without much burden. Through this graphical interface one may specify the desired application to be executed, the necessary parameters, and the input files. Also, some policies can be defined at this level, such as when other machines can use local idle CPU cycles and how much disk space machines can utilize (e.g. no more than 100 MB can be utilized). Additional configurations, such as the entrypoints to the overlay network, need to be provided in a configuration file. In contrast, the console is command-line based and provides more advanced features, such as showing the resource availability levels, the neighbor nodes and their characteristics (e.g. supported applications and idleness levels), the node's ratio, and setting some

policy parameters (e.g. number of maximum tasks that may be performed at the same time).

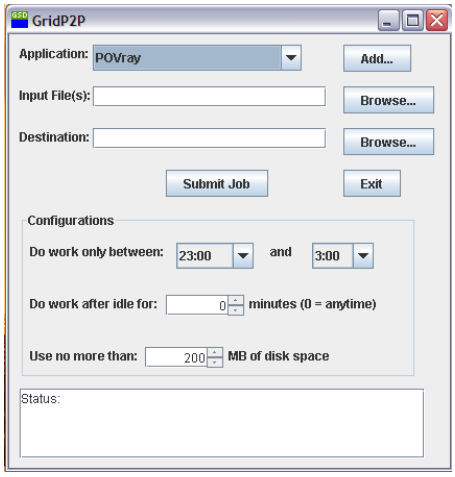

Figure 3.5: Graphical User Interface

3.5 Communities

As we have seen in Chapter 2, many systems present the concept of community. GridP2P is no exception, although in a less strictly way, i.e., there is no demanding scheme or convention for how overlays and entry-points (boot nodes' IP address and port) are configured. As an example, the INESC-ID institution can create an overlay network for their researchers and provide, through an outside mechanism (such as a webservice), a list of entrypoints, corresponding to some servers with high uptime. Hence, whoever is responsible for maintaining the overlay should provide means for obtaining entrypoints to boot nodes.

Generally, these communities would consist of groups of Internet users, with same interests, that would have the same kind of applications available and shared within the overlay community. Furthermore, these communities may scale from small groups of hobbyists to worldwide scale.

3.6 Overlay Manager and Communication Service Layers

3.6.1 Supported Applications

Currently, the kind of applications allowed by this system should either be parameterized through the command line or receive a script or configuration files as input (i.e. parameter sweep or batch file processing). They should also easily generate output files and report errors.

Moreover, it should be possible to create independent tasks from those applications. As we have seen before in Chapter 2, independent tasks are the ones that do not require communication between them during their execution time, and thus the task executing order is irrelevant, facilitating the scheduler work.

Each supported application, along with its version, has a unique and universal identifier that may be defined by external parties. For example, if an application *foo* has a version for *Windows* and another for *Linux* that are not compatible with each other, then this application could have two identifiers, *fooWin* and *fooLinux*.

It is necessary to guarantee that identifiers will be equal on every machines for the same applications (and versions). Therefore, this could be done through a external scheme contemplating webservices, through which we send the name of the application, along with its version and supported operating system, and we get back the application identifier.

Our platform needs to have specific mechanisms for handling each type of application. These application-dependent mechanisms are stored in the Application Adaptation Layer. Therefore, when users register a new application, they have to provide an XML file containing: the application identifier, the application call with the correct parameters, the number and types of input data files, the maximum and minimum task size, and the rules on how to combine task results and on how to scatter input files into pieces to create tasks. These rules are semantic-aware, and, for instance, they could tell how to split a movie file into smaller scenes by adapting headers and fragments of data. However, this automatic file partition mechanism is yet in development [dCM08], and so not fully integrated in the GridP2P.

3.6.2 Overlay Network Management

The management of the underlying network is done through the Pastry overlay. Node's identifiers are randomly generated and assigned to a precise location on the circular addressing space of Pastry. By doing so, the machines holding adjacent nodes could be completely geographically dispersed. Additionally, each node only communicates directly with its successor and predecessor nodes. Messages directed to other machines have to be routed through the adjacent nodes.

Pastry's structure provides a key set of properties for building and maintaining a network: fully-decentralized, no need of having central points of coordination; self-organized, as nodes join to or depart from the overlay, the structure has to adapt itself to those changes; scalability, the network should not lose performance as the number of nodes increase; and fault-tolerance, failing nodes should not compromise the reliability of the whole network.

The basic functions provided by this overlay network and accessible through the Communication Service Layer are: broadcast a message to all neighbors of a node, send a message to a single neighbor, send a message to a neighbor which would route it to other nodes, and receive a message and delivering it to the right handler routine in the Overlay Manager.

3.6.3 Monitoring of Resources

As a distributed cycle sharing platform, machines distribute, perform, and collect work within the overlay network. Sending and retrieving data consumes bandwidth, while running applications consumes CPU, RAM memory, and storage space in hard disk (for input, temporary and output files). Therefore, the considered resources to our platform are: CPU, bandwidth, and primary and secondary memory.

For each type of resource is assigned a value from 0 to 1, where 0 means that the resource is unavailable and 1 that the resource is powerful and has good availability. Hence, we try to measure the available power of every resource. For example, if we consider that 4 GB of RAM is very good (corresponding to 1), we may have a 2 GB RAM fully available, but we only would get a value of 0.5, considering these value functions as

being linear. In the same way, if we have a 8 GB RAM slot half or full available we would get a value of 1. Therefore, if we have more than 4 GB of RAM available, it would not matter and it is considered as being very good as well. However, the very good alternative (corresponding to 1) should be sufficiently higher in order to make good distinctions.

The global (among all types of resources) availability value of a node may be obtained through a simple additive model [GW04]. In this way, nodes may define the relative importance of each type of resource by defining weights (using methods like the swing weights). With them, it is then possible to make a weighted sum and obtain the global availability value, which would be the node rate.

Additive Value Model

$$V(a) = \sum_{j=1}^{n} k_j.v_j(a) \quad with \quad \sum_{j=1}^{n} k_j = 1 \quad and \quad k_j > 0 \quad (j = 1, ..., n) \tag{3.1}$$

$V(a)$ global value of node *a*
$v_j(a)$ partial value of node *a* in the criterion *j*
k_j weight of the criterion *j*

CPU. In order to measure the CPU power, a machine has to solve a challenge through which we can obtain useful metrics, such as the execution time or the GFlops (hence, it does not matter if machines have different CPU architectures). As an instantiation of that challenge, we rely on the Linpack benchmark [Don92] where a system of linear equations is solved.

Memory. The available memory amount is easily obtained by doing a system call to the operating system.

Bandwidth. Measuring the available bandwidth of a single node in these environments is a very hard problem. For instance, a test between two nodes, Alice and Bob, will only tell us the bandwidth between those two points. But we do not know if we are limited by Alice's upstream bandwidth or Bob's downstream bandwidth. We could attempt to determine Alice's upstream bandwidth by flooding to many nodes at the same time; and do the same thing to determine her downstream bandwidth by getting several other nodes to flood to her. Yet, we still do not know if we found the bandwidth of that node, or only the bandwidth of the other nodes. Additionally, a stress test like that will cause a lot of lag in the network.

Our approach is to check the time for a message to travel from one node to another and back again (i.e. the round-trip time, RTT). This process is done using regular messages from the system and it is done from one node to all of its neighbor nodes (but without flooding). Within a short period of time, the minimum RTT value obtained is kept and the bandwidth is calculated. In this way, we may find the bandwidth between a node and its best neighbor in terms of bandwidth. If the neighbor has more upstream and downstream bandwidth

than the node, then we have found the node's bandwidth.

Storage Space. Every node has to define the disk space available for using when receiving remote tasks or for build the output or other temporary files. This type of resource is the only one that is not rated from 0 to 1. A node has only to check if other remote nodes are suitable by having sufficient storage space to perform its tasks. Therefore, the secondary memory does not contribute to the global rate of a node (i.e. it is not a factor of preference).

Each node has to periodically monitor its own resources so that, every time changes in their utilization occur, the neighborhood gets aware of them. The resource availability is checked upon node's bootstrapping, before and after a task computation, and within a specified time frame (i.e. when an update message is about to be sent, as further described).

3.6.4 Resource Discovery Mechanisms

Our platform provides a mechanism for locating resources based on the Informed Walk described in Section 2.3.5. Nodes advertise themselves by sending update messages to their neighbors on the Pastry overlay. These messages contain the sender node's related information: its identifier, its minimum work ratio required (further described in Section 3.8.1), its supported application identifiers, the time required for this information to expire, and its resource availability (in terms of CPU, bandwidth, and primary and secondary memory). Upon receiving this information, a neighbor node calculates, with its own judgment, the global rate of the announcer node. This judgment, as described before, consists of associating weights with the measured availability of every single resource. The proximity level between the announcer node and its neighbor is also taken into account to this judgment. After that, the whole information concerning the announcer node is stored in memory for further recall. Moreover, update messages are sent every time changes in resource availability are perceived through the monitoring of resources. Additionally, these messages are also sent periodically within a specified time frame. This time frame should be long enough to not cause much network traffic. The former approach is more efficient, however, the latter cannot be discarded, since either resource availability may change due to external process activity, or message delivery may fail.

Whenever a node wants to perform a job, it looks on its neighborhood cache for the best available nodes according to their global rate. If the set of assembled nodes is not sufficiently attractive, the node may pick the neighbors with best reputation for finding more available nodes. The reputation of a node is based on previous requests that it has successfully forwarded to other nodes with availability.

3.6.5 Distributed Cache

In order to reduce resource utilization, mainly bandwidth and CPU, our system contemplates a cache for storing computed tasks. This cache is a distributed networked storage system which is maintained by the overlay nodes. As machines may fail, there is a replication factor, k, allowing equal data to reside in k different nodes (i.e. data redundancy).

For a matter of efficiency, upon task completion the result is sent both to the cache and to the node for which the task is intended. In the latter case the privacy of the user/node is lost, although we gain in efficiency, whereas a node does not have to predict the right time to retrieve a task result from the cache.

Whenever a job is submitted and the number of tasks to be created is assessed, we lookup in the cache if there is any task already computed. The lookup key is obtained by computing the digest of the task's input data. Hence, the cache maps task's input data digests into task's output data (i.e. results). If the task result is already present in cache, it is then retrieved by the Overlay Manager and sent to the Job Manager to be stored. The Job Manager may store that task result either in memory or in persistent storage space (building the partial or final output). Otherwise, if the task result is not cached, the task is then sent to an available node, which also could be its creator node, to be computed.

3.7 Job Manager and Application Adaptation Layers

3.7.1 Estimation of the Number of Available Nodes, Task Creation and Distribution

Once a job is submitted, it is necessary to create a number of tasks that will be distributed afterward among the available nodes. Hence, attending to the dynamic environment involved, our platform estimates the number of available nodes so that the resource utilization within the overlay network would be maximized. This process is completely transparent to the user, as he does not have to even know what a task is.

Through its neighborhood cache, a node assembles a set with the best available neighbor nodes (i.e. whose rate is highest). If more nodes are needed, they may be obtained through the neighbors with best reputation (as described in Section 2.3.5). During this assembling process, nodes are questioned about their absolute availability, i.e., some security policies may not allow nodes to perform tasks as they could be busy with other tasks, or they could be out of their working time, and so forth. Hence, a node's absolute availability tells us if a node is available or not for processing tasks, disregarding the levels of resource utilization (or availability) presented by that node.

Until a certain number *a*, the more available nodes we get the greater the speedup of the application would be. Once *a* is reached, assembling more nodes would not compensate over the speedup gains. The determination of *a* is application-dependent.

For applications whose tasks consist of fragments of the input data, such as a movie converter, we have:

$$a = WholeInputSize/MinimumTaskSize \tag{3.2}$$

The minimum task size is defined upon application registration, and it is the minimum size that would worth the task parallelization (i.e. assuming that all tasks have the same size). If the application input data is equal among tasks, but the input configuration is different, then we may determine *a* by defining quotients between other values than the data length. For instance, rendering a certain POVray image completely would take 100 chunks; thus, we can define that the minimum range to render is 10. Doing so, we have $a = 100/10 = 10$, and, in average, if we have more than *a* available nodes to render the image, we would not see much gains in relation to having only *a* nodes. This application dependency should be disentangled through the semantic rules provided by the XML registration file.

When we are searching for available nodes, instead of exhausting the overlay nodes with requests, we only have to search until $a + \alpha$, where α is an adjustment parameter corresponding to the number of non available nodes in the previous trial (job).

If only a few number of available nodes were found, we may have heavy tasks with large size, which is not a very good idea. For instance, a node could fail while processing a big task, and thus much of the work done is totally lost. In order to avoid this kind of situations there is also an upper bound limit for the size of a task, *b*, where *b* >= *a*. The minimum number of tasks we could have is then:

$$b = InputSize/MaximumTaskSize \tag{3.3}$$

Hence, there could be more tasks than available nodes, and consequently more than one task may be attributed to a single node. In this case, the node needs to schedule the tasks in a sequential way, rather than performing multiple tasks from the same job in parallel (i.e. only one job's task may fail within the same time frame). Analogously, *b* may be also obtained as *a*, for the applications whose tasks only differ in configuration input.

After we have estimated the number of available nodes, we may then create the tasks and distribute them among machines. Note that the process of estimating the number of available nodes, is only a way to also estimate the number of tasks that we need to create, in order to maximize the resource utilization within the overlay network.

Concerning the distribution process, tasks are sent one by one to the best nodes that are suitable for performing those given tasks. In order to verify the suitability of a node, the task minimum requirements (i.e. the task cost, described in Section 3.7.2) have to be assessed against the resource availability of that node. For example, if a task requires the *foo* application and 200 MB of storage space, then only nodes having the *foo* application and with 200 MB or more of available storage space would be selected. Additionally, if any node refuses to accept a task, due to some security policy, it should be then assigned to another node in the gathered set (hence, there would be nodes with more than one task assigned). Although, this situation is unlikely to happen, as nodes are first asked for their absolute availability.

3.7.2 Calculating Task Cost

Tasks can have different kinds of complexity, depending on the size and type of the data to be computed. So, it may be more appropriate to send computational heavy tasks only to the best nodes which comply with a set of minimum requirements. These requirements define the cost that a task would have for a node in terms of its resources (CPU, bandwidth, and primary and secondary memory). In this way, a minimum QoS is assured. Additionally, determining a task cost is a very hard problem, and thus we only provide a not too rigorous approach described as follows.

The cost of a task is calculated by computing a few parts of its job. For example, if we have a job concerning an image to be rendered with 100 chunks, then we may compute three random chunks and determine the average cost of a single chunk in terms of consumed resources. In effect, we check the elapsed time and memory taken to compute each of those chunks.

We may assess the CPU cost of a chunk by doing the quotient between the elapsed time and a parameter

value, corresponding to the maximum allowed time to compute a chunk, and then multiplying by the CPU measured availability (of the node where the chunk was computed on). In this case, as tasks contemplate a certain number of chunks, we may multiply that number by the chunk's CPU cost to obtain the task cost in terms of CPU. Thus, we have:

$$CPUcost = ElapsedTime/MaxTime \cdot CPUavailability \quad and \quad TaskCPUcost = NumberOfChunks \cdot CPUcost \tag{3.4}$$

The bandwidth cost may be assessed based on the input and expected output data size of a task. Hence, the output's final data size expected could be obtained by extrapolating the output size produced when computing a chunk (i.e. analogous to the CPU situation). After the size of the data, that would pass through the network, is estimated, we may correlate it with the node's bandwidth availability levels. For example, we may define a size limit that is acceptable for a node with *very good* bandwidth, and thus we only have to do the quotient between the assessed size and this limit to get a bandwidth cost. Additionally, estimating the output data size will give us automatically the task's secondary memory cost. This cost is represented by the equation 3.5.

$$BandwidthCost = AssessedSize/MaxSize \quad and \quad TaskBandwidthCost = NumberOfChunks \cdot BandwidthCost \tag{3.5}$$

Our approach to determine the primary memory cost consists of applying a given function over the consumed memory by a single chunk. Such function can be given upon application registration.

Overall, if this cost estimation is much different than the one obtained after the task computation, the provider node may notify the consumer about that. In this way, the consumer node may adjust the assessed cost of further tasks (concerning that same type of application). Nonetheless, one has to take into account that the tasks from a same job are assigned with the same cost, despite the fact that their complexity might be different. Therefore, that notification must only happen if the difference between the assessed and real cost is significant.

Furthermore, the task's cost is used both to determine the suitability of a node for computing that task, and to update the resource availability levels when a task is about to be computed.

3.7.3 Scheduling and Launching Applications and Computing the Final Job's Output

When a job is submitted, a node has to estimate the number of remote available nodes, create tasks and calculate their cost, distribute tasks among available and suitable nodes, gather computed tasks, and build the final output. This node may also receive a task for being computed if it is not busy with other ones.

After a task is received, a node checks if it has free slots to compute new tasks. If so, the task is immediately processed and the respective application is launched. The node's resource availability levels are also decremented in accordance to the task's cost. If there are no free slots to attend new tasks, the task is rejected and the sender node is informed. In this more efficient way, there are no waiting queues, except for the node that is distributing tasks, where it has to wait until all tasks are sent and accepted.

The Application Adaptation Layer creates a new process each time an application is launched. If the host node (or machine) has a multi-core CPU, then the operating system or virtual machine may schedule those application processes over the CPU cores. Nonetheless, the CPU availability levels of this node should not be decremented if there are still free cores (i.e. more cores than tasks currently being computed).

When the computation of a task is completely done, it is sent both to the node holding the respective job and to the computed task's distributed cache. Nodes holding jobs gather computed tasks from remote machines and store the output related data in primary or secondary memory. These task output data are slices from the final job output, and thus additional work may be required in order to join those slices. They can be joined either as soon as they are received or when all results are gathered together. Moreover, the joining mechanism is application-dependent and should be provided upon application registration.

3.8 Policy Engine Layer

3.8.1 Resource Usage Control

Across the overlay network, resources are shared among different users. Thus, there must be some rules for controlling the resource utilization, which led us to the security policies. These policies should support complex usage scenarios, and therefore they need to be specified in a sufficient expressive language. In a distributed cycle sharing platform, as the GridP2P, it may be useful for policies to consider events that have occurred in the past. For instance, we may have a policy for limiting the resource consumption, which needs to know the resource utilization history.

In order to make policies operational, our platform provides an engine which evaluates and enforce policies against generated security events. Additionally, new policies may be defined and introduced at runtime.

With respect to evaluation time, policies are evaluated when messages come both in and out of the GridP2P. More precisely, this auditing process is done when messages pass from the Overlay Manager to the Job Manager and vice versa. At those moments, security events are generated containing information about the sending or received message. After that, the engine evaluates these events, along with the policies, and decides which action to take.

Working Time Policy. In general, machines lying over a GridP2P overlay network are not dedicated servers, instead they are common machines, owned by typical computer users, whose resources may not be always available for performing outside jobs. Hence, we provide a security policy allowing nodes to only perform outside work during a specified period of time. In this way, users may specify a time range for when they expect their machines will not be in use (e.g. from 10:00 PM to 2:00 AM). In addition, users may also specify an idle time required for nodes to start accepting and performing tasks. This could be also used for processing work when a user's screensaver shows up.

Maximum Number of Concurrent Tasks Policy. In order to assure a minimum QoS, we provide a security policy, based on past events, allowing one to specify a maximum limit for the number of tasks that may be processed at the same time. For example, a machine with a quad-core CPU may limit its usage to four

tasks where each one would be assigned to each CPU core (i.e. each task corresponds to a new process). Therefore, tasks running on this machine would not be delayed by other ones.

Task Complexity Policy. Disregarding its resource capabilities, a node may want to only perform ligh-weighted tasks (i.e. tasks with a low cost). Such decision could be based on the periods of time that the node stays idle (e.g. the node could be idle for just a few moments when the screensaver is shown). Therefore, it is possible for a node to reject tasks if their cost is higher than a specified bound. Although, if this limit is much lower than the resource availability levels of the node, then it can cause tasks to be rejected, despite the fact that the node has been marked before (through an Availability Message) as available. This happens because when we are assessing the absolute availability of a node, we would not have the tasks created, and thus we do not know the cost of a task yet.

Consumption Policy. A task taking too long to be computed (i.e. beyond a defined threshold) can be interrupted. Also, if a task consumed much more resources than the ones specified by its cost, the consuming node may be marked into a black list. If a node is marked more than n times, then its access to the same provider node should be denied for a certain period of time, that could be one or several weeks, depending on user configurations. After that period of time is over, the offender node is cleared from the black list, and it may access again to that node.

Ratio Policy. Within an overlay, there could be nodes contributing more to the community, by providing more spare CPU cycles of their machines; or contributing less by not being connected much time to the overlay or by denying access to their resources. Our system incorporates the concept of resource usage fairness in which a node may specify a minimum ratio required for accepting remote tasks. In this way, each node has a ratio which is simply the quotient of the performed work in it and the performed work in other nodes. This ratio is sent along with requesting messages whenever one needs to perform work on outside nodes. Moreover, the performed work corresponds to the sum of the CPU cost of every processed task.

With this mechanism, a node only has to compare the ratio sent within a request with its minimum ratio required. The minimum ratio required may not be higher than one, i.e., a node performing work as much or more than what it asks to perform remotely cannot be rejected.

Furthermore, each node's minimum ratio reference may be adjusted by the needs of the node. For example, if a node needs to improve its ratio it may decrease its minimum ratio reference in order to potentially perform more remote tasks. Analogously, a node may also increase its minimum ratio reference if it does not need to execute jobs in the relatively short-term.

4 Implementation Details

4.1 Used Technology and Integration

In order to make our platform portable, we used the Java programming language. In this way, we may run the GridP2P upon a Java Virtual Machine, which is available for the most common operating systems and computer architectures.

The integration of the PAST (distributed cache), Informed Walk (Resource Discovery) and Heimdhall (Policy Engine) within the platform was made in an easily and seamlessly way. This happened due to the fact that each of the integrated components was also developed in Java.

For the network management, the Overlay Manager and Communication Service layers use the FreePastry tool which is a Java implementation of the Pastry overlay.

4.2 Main Data Structures

The GridP2P main data structures are:

- Node Rate: This structure is also known as the neighborhood cache. It stores all the neighbor nodes and their characteristics, namely, their resource availability levels. The global availability level (rate) of the node is also stored.
- Node Reputation: The reputation of each neighbor node is stored in this data structure.
- Resource Manifest: This structure contains a node's resource description in terms of available CPU, bandwidth, and primary and secondary memory. Apart from the secondary memory, each of these resources have an availability value from 0 to 1 (as described in Section 3.6.2).
- Policy Repository: Maintains all the loaded policy objects with their specifications.
- Event History: For history-based policies, their triggered past events are stored here.
- Job Holder: Every created job object is kept in this structure during its lifetime. These job objects contain information related to the job, such as the job identifier, the number of completed tasks, and the output data of each computed task.
- Application Repository: Contains all the supported applications: their identifiers, their calling command with the correct parameters, rules on how to reassemble task results and split data input into tasks, amongst other.

4.3 Overlay Manager and Communication Service Layers

4.3.1 Pastry Network Management

The GridP2P adopts the FreePastry as an implementation of the Pastry overlay. This particular implementation is made in Java and provides several components to create and manage a Pastry overlay network with countless nodes. These components are used by the both Overlay Manager and Communication Service layers. The main components of FreePastry are described now.

4.3.1.1 Common API

Many applications have been developed on top of structured peer-to-peer systems. In order to increase the portability of these applications over several different structured peer-to-peer systems, a common API was proposed in [DZD+03]. In this way, applications do not need to change their implementations for using different peer-to-peer solutions.

FreePastry provides the common API as a set of interfaces for common used elements on peer-to-peer systems. For instance, FreePastry provides interfaces for applications running on top of it (i.e. they must implement some functions that are specified in the Application interface), for node's identifiers (Id), for messages (MessageRaw), and for nodes (Node).

4.3.1.2 Continuations

Like a callback, or a Listener in Java, a continuation is executable code that is passed as an argument to other code. Continuations are primarily used in FreePastry for handling network latency and I/O operations that may block or take a significant amount of time. For example, when a request is sent to the network, like a PAST lookup, a continuation would avoid the requesting program to block while the request is being made, or while an answer does not arrive. In this case, a continuation object is required, so that, when an answer arrives, a function associated with that object is called, having as parameter the message with the answer. Afterward one may access to the continuation object to get the result of the function. In addition, inside that function another one may be called to handle with the received message.

4.3.1.3 Environment

The environment component was created for peer review. Peer review records all network traffic and non-determinism (e.g. clock events, random). The environment includes the following components:

- Selector Manager: a single thread to handle all the network inbound/outbound, timer events, as well as other network events. Any invoked action onto the overlay network is detected by the selector which identifies and delivers the event to the interested parts. For instance, the delivery of a message to the node application for which it is intended to.
- Processor: a dedicated thread for doing CPU intensive tasks which is not always used, and a thread for doing blocking disk I/O.
- Random Source: a random number generator
- Time Source: a virtual clock for the FreePastry. With this component nodes can be executed with a clock that is independent from the real clock of the computer. Therefore, time can move faster, which may be very useful for testing purposes (particularly important on simulation environment).
- Log Manager: allows to register relevant network events which can be used for network diagnostic and debugging.
- Parameters: this component is responsible for handling the configurations of the environment to be created. For example, the size of the leafsets in Pastry, the maximum size of a message, the sending

rate of keep-alive messages, and so forth. These configurations are stored in persistence storage space and can also be changed at execution time.

- Exception Strategy: this component defines what to do when unexpected events occur in FreePastry (not in user's application).

4.3.1.4 Application

The Application is the program that runs on top of the FreePastry. A program must implement the Application interface so that it can interact with the overlay network. Trough this interface, a program may receive and send messages to the overlay, and get notified about changes in the neighborhood.

4.3.1.5 Endpoint

The Endpoint is an object which provides an access point to applications, so that they can communicate with the Pastry overlay. Moreover, the effective routing of messages is done by this object.

Whenever a serialized message arrives, the endpoint is responsible for the desiarialization of that message, building the respective objects, and delivering the object messages to the right handler routine. Therefore, the endpoint needs to be configured with desiarialization methods in order to handle the different types of supported messages. These methods are then called automatically upon message arrival.

4.3.1.6 Serialization

In Freepastry, a message is a serializable object, i.e., an object that may be converted into a sequence of bits. Java Serialization require very little work to create a new message, but, unfortunately, it creates a lot of overhead in terms of cpu, memory, and bandwidth. Doing so, FreePastry introduced a more efficient mechanism, the RawSerialization. With this mechanism, elements (fields) of a message (class) need to be (de)serialized one by one, to and from an output and input buffer respectively. As messages may travel within the overlay quite frequently, the RawSerialization constitutes an important mechanism, thereby reducing the overhead in each endpoint.

4.3.2 Message Types

The GridP2P contemplates five types of messages listed as follows.

Update Message A node sends this message to all of its neighbors in order to announce its both presence and resources. Moreover, this message contains the node identifier, its minimum work ratio required, its supported applications' identifiers, the time required for this information to expire, and its resource availability (in terms of CPU, bandwidth, and primary and secondary memory).

Availability Message This message has the purpose of checking whether a node is available or not to perform work. It also includes a status field which says if this message means a question or a positive or negative answer about the availability of a node.

Task Message These type of messages contain the input data and required configurations for being computed by a node.

Task Result Message The output of a computed task and the identifier of the job and task are kept in these messages.

ACK Message These messages are sent for assertion purposes. A status field is included representing a positive or negative acknowledgment.

All these messages are sent through the TCP/IP protocols.

4.3.3 Resource Measurement

In order to measure resources so that they could be compared against each other in a simple additive model, we have to convert direct indicators of availability into a common scale. This common scale is rated from 0 to 1, where 0 means that the resource is unavailable and 1 is the level of resource availability that we consider as being *very good*. Therefore, we rely on the following expression to do that conversion.

$$f_r(x) = max(MAX_r, x/MAX_r) \tag{4.1}$$

In 4.1, MAX_r is the value that we consider as very good for the resource r, and x is the direct measured value.

For example, if we consider $MAX_{CPU} = 500$ and $x = 250$, we obtain $f(250) = 0.5$.

The CPU availability's direct indicator, x, is measured in terms of MFlops (i.e. Million Floating Point Operation Per Second) through the Linpack benchmark. With respect to the bandwidth, it is measured in milliseconds, corresponding to the traveled time by a message with a constant size. For the primary and secondary memory, their availability is measured directly in Megabyte (MB). However, the secondary memory indicator does not need to be converted, as this resource is not used in the calculation for the global availability value of a node.

In addition, the FreePastry provides a proximity metric (based on the RTT value) that is also converted to the common scale and used in the additive model. Therefore, the global availability value (i.e. the global node rate) is calculated through the following expression:

$$\sum k_r(a) \cdot v_r(a) \tag{4.2}$$

In 4.2, $k_r(a)$ is the weight of the resource r ($r = \{CPU,\ Bandwidth,\ PrimaryMemory,\ SecondaryMemory\}$) in the node a, and $v_r(a)$ the value of the resource r in the node a (i.e. $f_r(x)$).

Furthermore, each node is free to define the weights and the *very good* reference associated to each resource.

4.3.4 Resource Discovery

Our resource discovery mechanism consists of selecting nodes with the best available resources.

Selection of the best available node. The best available nodes are the ones whose global rate is highest. This global rate is calculated as described in the previous section. Therefore, a node may select machines in accordance to the importance it gives (through the weights) to each type of resource. For example, if a node runs applications that require much more memory than CPU, it may assign an higher weight for the memory, and thus machines would be selected mostly based on their available memory.

The proximity value should have great importance on nodes' selection, as it avoids the propagation (i.e. the number of hops) or long transmission of messages throughout the overlay network. Hence, it is always better to allocate resources on closest nodes.

Selection of the node with best reputation. The node with best reputation is the one with the lowest rejection level. This level is obtained through an weighted sum of failed and denied requests for processing data. A request failure happens when a node sends a request for being forwarded, to one of its neighbors, and no answer is received within a specified time frame (i.e. no acknowledge message). If an answer is received before the timeout, and if it is a negative acknowledgment (NACK) message, then we have a denied request. A NACK message, in this context, means that the neighbor could not found any available node to forward the node's request.

The rejection level of a node is given by the following expression:

$$f(\alpha, \beta) = \alpha \cdot 0.7 + \beta \cdot 0.3 \tag{4.3}$$

In 4.3, α is the number of failures and β is the number of NACK messages.

We consider the failures as the most important criterion, i.e., roughly two times more important than the NACK messages. In our opinion, the situation of failing messages would take much longer to stabilize than the nodes take to become available, however, these values may be changed.

In order to select the neighbor nodes with the best likelihood to forward a message, nodes have to store a rejection level for each of their neighbors. A node may also append to its update messages the identifier and rejection level of the neighbor with best reputation.

Thus, when a neighbor receives that update, and if it is not the referred node with best reputation, it checks if the rejection level referred in the message is higher than the rejection level that it has stored in the past for the referred neighbor. If so, the stored value is replaced by the new one.

The messages may be forwarded only one time (i.e. only one level is reached). Nonetheless, the number of neighbors could be parameterized, and if we have 20 of them, we may get a total of 400 reachable nodes (20 x 20). Hence, we may go through the list of neighbors, from the one with best reputation to the one with less reputation, until we get a satisfactory number of available nodes.

4.3.5 Distributed Storage of Task Results

The implementation of our distributed cache, for storing computed tasks, relies on the PAST (described in Section 2.4.3.2). The integration of PAST with our platform is easily done, since PAST is meant to work upon FreePastry (i.e. our Pastry overlay network).

PAST provides two operations, insert and lookup. The insert requires, as parameters, the output of a computed task (e.g. an image of rendered chunks in POVray) and a key, for identifying that computed task. To locate a computed task through the lookup operation, only the key is needed. This key is the generated SHA-1 hash from the task description, which includes the input data and configuration.

Furthermore, continuations are used so that processes does not have to block and wait each time an operation is executed on the cache.

4.4 Job Manager and Application Adaptation Layers

4.4.1 Estimation of the Number of Available Nodes, Task Creation and Distribution

For estimating the number of available nodes, it is necessary to assess their absolute availability. Hence, nodes are asked whether they are available or not to perform work. Beside the neighbors, if more nodes are required, then special available messages are sent to the neighbors with best reputation. These messages ask a neighbor node how many available neighbors it has. Nodes have to reply to the asking node within a given time frame. Soon after an answer arrives, the answering neighbor is flagged, within the neighborhood cache, about its absolute availability in accordance to the received answer, which could be positive or negative. In this way, a node distributing tasks may discard non-available nodes based on that flag. Moreover, the number of positive replies would correspond to our estimation of the number of available nodes .

Tasks may now be created in accordance to the estimated number of available nodes. There are two main ways to create tasks regarding different types of applications: splitting the input data file(s) or creating different configurations for each task. The former way is harder and should be semantic aware in most cases, whereas the latter is a more blind approach. Nevertheless, specific mechanisms for task splitting need to be provided for each kind of application. Currently, these mechanisms are hard-coded in the Application Adaptation Layer (as this issue is beyond the scope of this project), but in the future, they should be automatically built based on XML descriptions.

With regard to the distribution process, nodes have to be checked about their suitability to perform a given task. Therefore, a node is considered as suitable if it is not flagged as non-available, and each of its resource availability levels is equal or higher than each resource cost demanded by the task. The tasks are sent to the nodes, from the one with best rate to the one with less rate, and more than one task may be assigned to a single node.

4.4.2 Gathering Results

Once all tasks of a job are completed, it is necessary to join their output data in order to build the final output of the job. The way output data is joined is dependent from the application. For instance, chunks from a POVray's rendered image have to be joined using a script. That script removes the header of each chunk, puts them all ordered in a single file, and creates and attaches an image header for the file (that would be the job's output). In contrast, statistical applications are more likely to read the output data (like ascii text) of each task, do some calculations with that data, and then generate a completely different output file for the job (e.g. an image containing a line chart). Currently, mechanisms like these are hard-coded in the Application

Adaptation Layer, but in the future, they should be automatically built based on XML descriptions (i.e. this issue is beyond the scope of this project).

4.5 Policy Engine Layer

4.5.1 Security Policies

Our Policy Engine Layer uses the Heimdall, described in Section 2.2.1. Nevertheless, this layer may work with multiple engines in simultaneous. In that case, for an operation be authorized, all engines must authorize, otherwise, the operation is denied. The integration of engines, with our security layer, may be made through webservices or through included libraries in our source code, as in the case of the Heimdall. In this way, we use the Heimdall interfaces in order to load policies, evaluate operation events, and enforce actions in accordance to the policy evaluation.

The policies should be defined in XML files whose structure is based on the xSPL language. This high level language is sufficient expressive to support a myriad of usage scenarios. Therefore, it allows the definition of several types of variables, logical conditions, and logical and arithmetic operations. Additionally, policies may be loaded at runtime, so that system execution does not have to be interrupted each time a new policy is added.

Operations like sending or receiving a message generate security events in the GridP2P. These event information have to be converted into the xSPL syntax event and then sent to the Heimdall for evaluation.

Upon security event, the evaluation of policies usually will only tell us if an operation was authorized or not. Although, functions may be associated with policies, so that when a policy fails on evaluation, it may trigger some action to happen (i.e. by calling a function).

With Heimdall we expand our range of possibilities to control the resource usage, instead of being bound to a small defined set of security mechanisms, like we see in many other peer-to-peer resource sharing systems. Hence, we may support a myriad of use cases, within a dynamic and complex environment, without needing to change any platform implementation.

4.6 API

We provide an Application Programming Interface (API), regarding each system component, so that implementations can be changed without compromising the interaction between components. Plus, the implementation details of every component are abstracted by our API, facilitating the programmer tasks.

Hence, a new component implementation need to cope with the APIs, either for providing the functionality described by the component API, or for calling other component's functions through its interface.

Despite our API being language-dependent, components still may be built in different technologies (or programming languages) and not being bound to a given process or system. In this case, the inter-component communication would be done through remote procedure calls (RPC) where additional code is required to translate API calls into RPC and vice versa.

Furthermore, each component API can be extended or modified, although one has to take into account the impact of those modifications in all other components.

Our platform provides the following APIs: AppAdaptationLayerI, JobManagerI, PolicyEngineI, OverlayManagerI, and CommunicationServiceI. They are described in Appendix A.

5 Evaluation

In this chapter we present the evaluation of the GridP2P platform regarding its performance and viability when facing real environment. More precisely, we test the performance of the system in terms of application speedup, distributed cache, and security policies. Note that the Resource Discovery (including the selection of nodes process) and Heimdall components have already been evaluated in other dissertations. We chose to evaluate those items due to their importance among our objectives, i.e., we want to get the maximum speedup from applications by using remote resources in an efficient and controlled manner.

In order to perform all the tests we rely on 3 different jobs, listed as follows.

- The first one, job1, is a POVray image to be rendered. Each task would compute a certain number of line chunks with different complexity. Due to that complexity, some tasks can be computed faster than others.
- The second one, job2, is also a POVray image to be rendered, however, the computational cost of each task is the same (i.e. tasks would be completed at the same time). Additionally, this job is less computational heavy than the previous one, i.e., the highest number of nodes that worth the parallelization is lower.
- The third one, job3, consists of a Monte Carlo simulation for a sum of several given uniform variables.[1] An image containing a linear chart would be the outcome. Each task would generate a lot of random numbers and group them into classes. Then, the classes are summed between tasks, and a Monte Carlo curve is drawn. Additionally, all the tasks take the same time to be computed.

For all of the tests we used machines with an Intel Core 2 Quad CPU Q6600 at 2.40 GHz with 7825MB of RAM memory. The used operating system was the GNU/Linux Ubuntu with the kernel 2.6.28-14-generic. For the Java Runtime Environment and FreePastry we used the 1.6.0_14 and 2.1 versions respectively.

5.1 Application Speedup

As we want to improve application performance, it is essential to see how much overhead we have. Fonts of overhead could be network latency, cache lookups, building the final output, policy evaluation, amongst other. Therefore, we may determine the speedup and efficiency of applications as the number of available nodes on the overlay increases.

5.1.1 Procedure

To assess application speedup we tried the three jobs listed above: job1, job2 and job3. For each of these jobs we made 8 trials. In the first trial we used 1 node; in the second, 2 nodes; and so on, until we have 8 nodes in total. All the nodes were within the same LAN network and the bandwidth available was about 100

[1] `http://web.ist.utl.pt/~mcasquilho/compute/qc/Fx-sum-unifs.php` accessed on July 2009

Mbps. For each trial we obtained the execution time of the job, and therefore we also got the speedup[2] and efficiency.[3]

5.1.2 Results

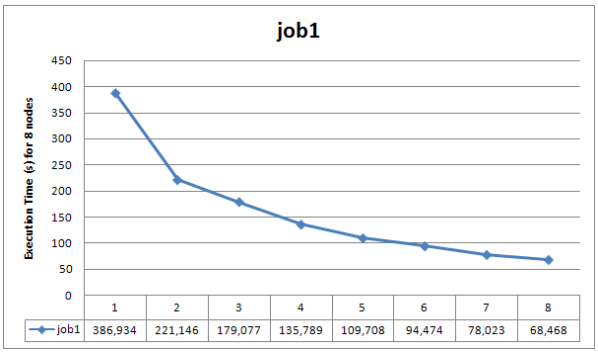

Figure 5.1: job1's Execution Time

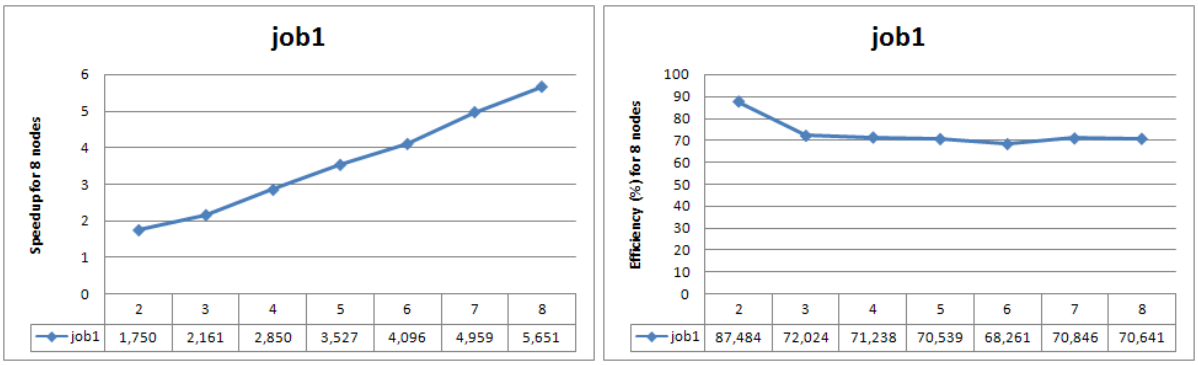

Figure 5.2: job1's Speedup and Efficiency

As we may see in Figures 5.1 and 5.2, the greatest gain happened when we passed from 1 to 2 nodes. We got a speedup of *1,750* which is very close to the ideal speedup[4] (*2*). Since then, the gains started to decrease. Moreover, the resource usage efficiency revolved around 70%, which gives the idea that some tasks were completed before others, releasing so their resources. This was expected as some tasks in this job were computational heavier than others.

In Figures 5.3 and 5.4, we see a job that does not worth its parallelization. When we passed from 1 to 2 nodes, we increased the execution time. We think this happened because of the image transference from one node to another. This overhead was not be masked by the parallelization gains. Thus, if this job was more complex to compute, and the generated output was about the same size, then we would get a better performance. Moreover, the maximum obtained speedup was $2,742$ for 5 nodes, which is a very bad performance. Regarding efficiency, most of the time it does not passed above 50%.

With job3, we almost got a linear speedup (Figures 5.5 and 5.6). The efficiency on using the resources

[2] $S_p = T_1/T_p$, where S_p is the speedup with p nodes, T_1 is the execution time with 1 node, and T_p the execution time with p nodes.
[3] $E_p = S_p/p \cdot 100$, where E_p is the efficiency with p nodes, and S_p the speedup with p nodes.
[4] Linear speedup or ideal speedup is obtained when $S_p = p$

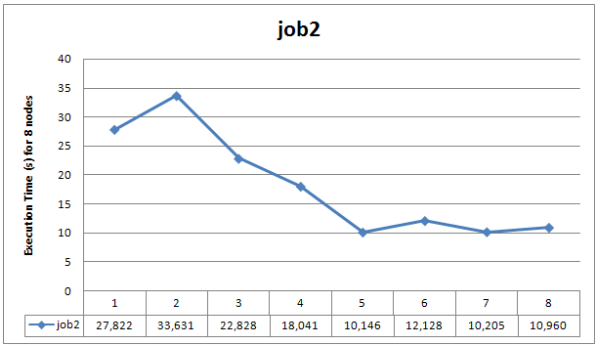

Figure 5.3: job2's Execution Time

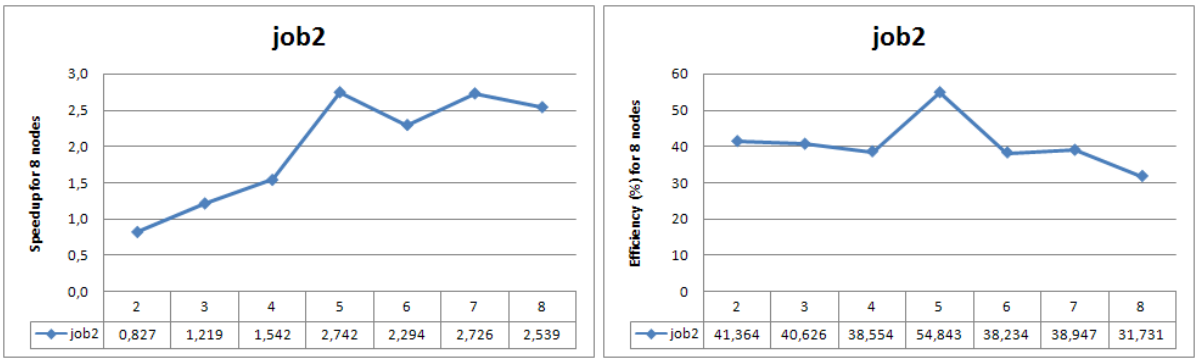

Figure 5.4: job2's Speedup and Efficiency

was very good, above 90% for each of the 8 trials. Two main reasons may explain this beahvior: all tasks had the same complexity and the overheads were minimal. Additionally, each task output was very small sized, and therefore fast to be transmitted on the network.

Following, the three jobs are compared against each other.

Through Figures 5.7 and 5.8 we may conclude that the best jobs to parallelize are the ones whose tasks have the same complexity (in this case, job3). Nevertheless, is important to see that, in job1, the gains were acceptable, and most part of the nodes become free to perform other tasks a while before the job was entirely completed.

5.2 Distributed Cache of Computed Tasks

A cache is important to avoid doing duplicated work (i.e. optimizing resource usage), and thus increase application performance.

5.2.1 Procedure

With 8 nodes within the same LAN, we try to check the application gains when the number of computed tasks, in cache, is from 1 to 8 (all tasks cached). For this purpose, we rely on the job1 and job3, as listed above.

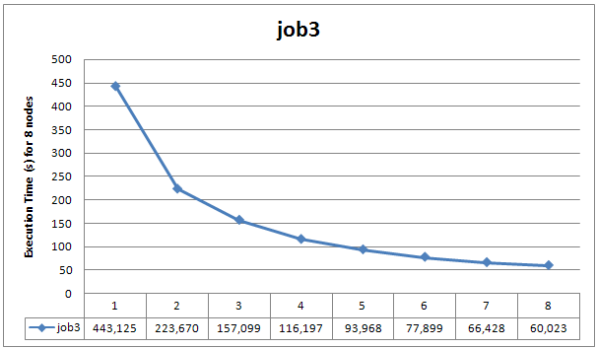

Figure 5.5: job3's Execution Time

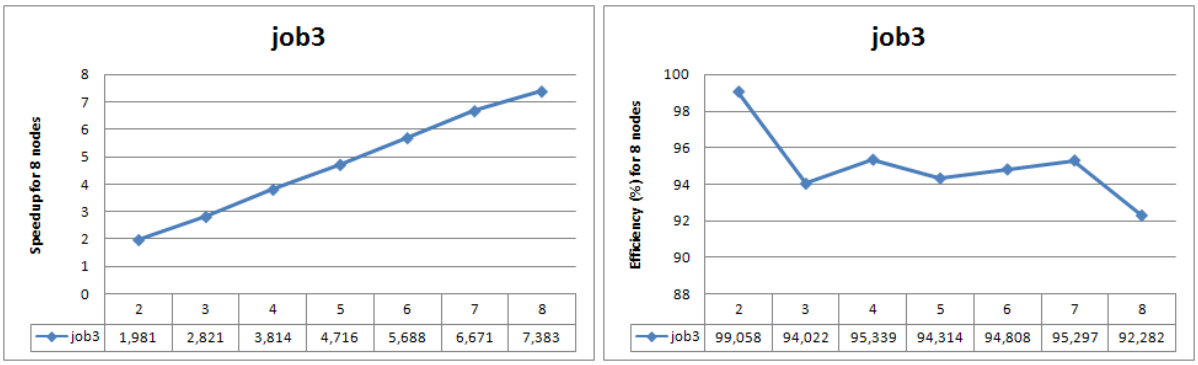

Figure 5.6: job3's Speedup and Efficiency

5.2.2 Results

As we can see in Figure 5.9 the gains are higher for tasks whose complexity differs (in this case, job1). It seems that the heaviest tasks in job1 were not cached until the number of cached tasks reached 5. Therefore, the most gain we may have, for this type of jobs, is when heaviest tasks are cached first. With respect to job3, it shows that caching tasks with the same complexity does not decrease the execution time, except if all tasks are cached. However, in any case, *n* machines will remain free if *n* tasks are present in cache, and that is the essential point.

5.3 Policy Engine

The policy engine overhead is directly proportional to the number of deployed policies. In this section we assess application performance against our deployed policies, described in Section 3.8.1.

5.3.1 Procedure

In order to see the impact of the security policies in application performance, we made 8 trials (with 1 node, 2 nodes, and so on, until 8 nodes), with the job1 (listed above), and saw the execution time taken without policies. Next, we compare the execution time of each trial with the time assessed before in Section 5.1 (i.e. with policies). That comparison is made in percentage, where 100% corresponds to the time with policies.

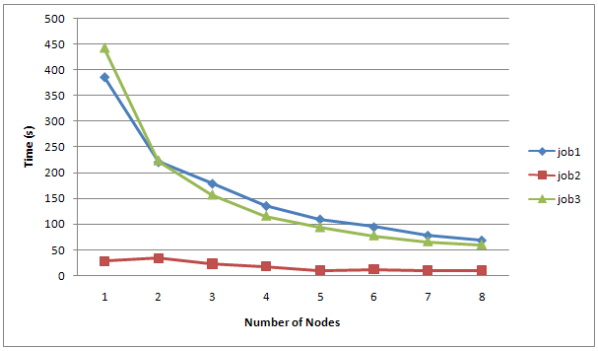

Figure 5.7: jobs' Execution Time

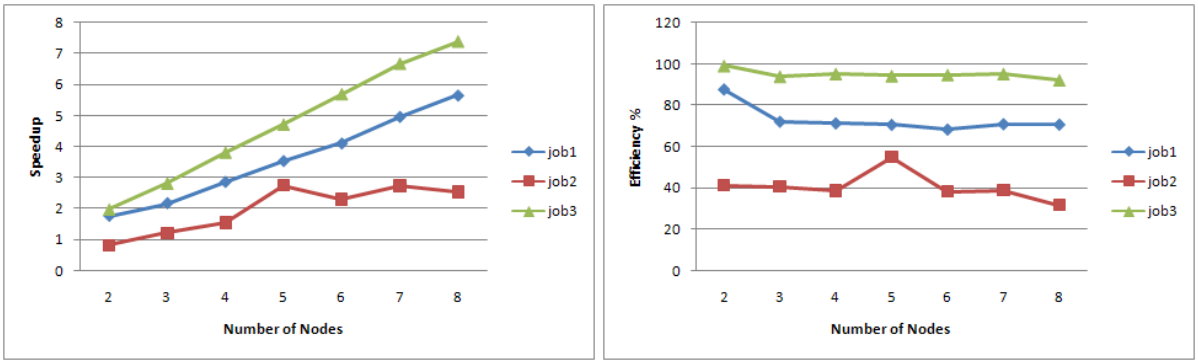

Figure 5.8: jobs' Speedup and Efficiency

5.3.2 Results

In Figure 5.10 we may see how fast an application become if the policy engine is not enabled. This obtained values are roughly equal for any job. Moreover, there is no policy evaluation when only one node is available, as would be obvious. When more nodes are available, the policy engine overhead corresponds to approximately 0,042 seconds (i.e. 0,0189% of the time taken with policies and with 2 nodes, 0,0235% with 3 nodes, and so forth).

5.4 Transmitted Messages

Supposing that we have n available nodes to process a job, then we would have the following number of messages traveling within the network:

$4n(n-1)$ number of update messages and the respective acknowledgment

$2(n-1)$ availability question and answer messages

$2(n-1)$ tasks and their result messages

$2n$ lookup and insert messages

Overall we have:

$$TotalNumberOfMessages = 4n^2 + 2n - 4 \tag{5.1}$$

Due to the update messages, we have a quadratic function, which is depicted in Figure 5.11

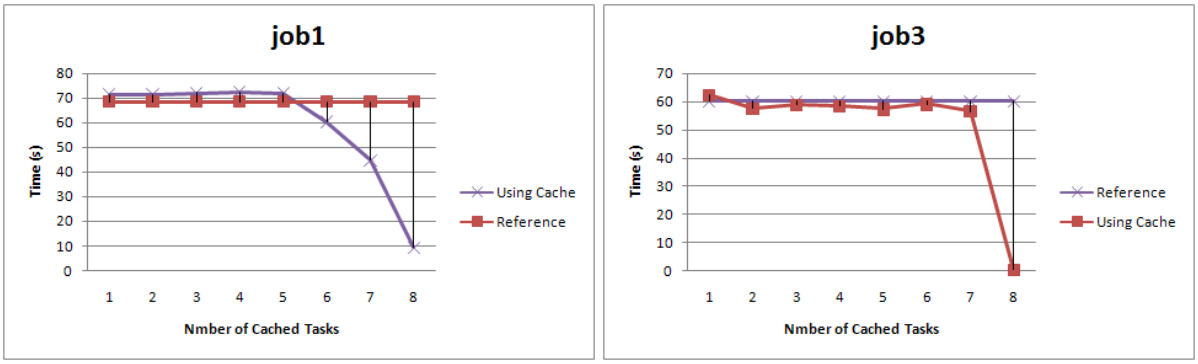

Figure 5.9: Improvement time when the cache is used. Reference is when no tasks are cached.

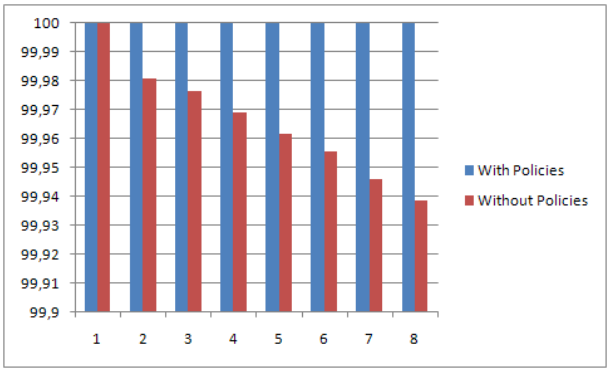

Figure 5.10: Application Performance With and Without Policies

5.5 Discussion

As we have shown in this Chapter, the GridP2P may improve the performance of applications.

Concerning application speedup, we have seen that our platform may introduce significant gains on the execution time of applications, either for equal or different task complexity. Although, the speedup is highest for tasks with the same complexity, as the work is better distributed among available nodes. For always having tasks with the same complexity, we need to predict that complexity, before the task creation process, in order to build tasks with different sizes. Such prediction represents a very hard problem and introduces more overhead in the platform.

We have also seen that the size of the output data can have a significant impact on the job performance, i.e., the time for transferring the data augments between the provider and consumer nodes. To obtain best speedups when the data size is bigger, the task complexity should be high enough to compensate the network overhead. Additionally, as tasks are independent, they do not have to block during execution time to communicate with each other, and therefore the network overhead is reduced.

In regard to the distributed cache, it is concluded that it may improve application execution time for applications whose tasks may differ in complexity. However, the heaviest task should be cached for this to work out. In addition, the use of this cache optimizes the resource usage, in a sense that machines become free to perform other jobs.

The policy engine overhead turned out to be minimal. Moreover, the time to evaluate policies against

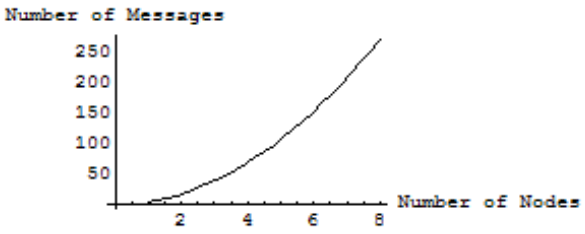

Figure 5.11: Number of Messages Growth

events is always constant. In a parallel environment, we only notice this overhead in the last task to be completed. Nevertheless, this overhead time may be increased as policies are added into the engine.

With respect to the number of transmitted messages during job's execution, it increases in a quadratic manner due to the update messages that are exchanged between all participant nodes. These messages are propagated because of the resource availability changes during the job's computation, i.e., right before and after a task being executed the resource availability levels are updated.

6 Conclusions

This dissertation presented the GridP2P platform, a solution aiming at a current reality, characterized by computers' hardware evolution, resource scarce utilization, and user's computational needs. With this platform, we apply the Grid concept into large-scale networks, namely the Internet, where a myriad of powerful machines may be lying idle for long periods of time. Moreover, we take advantage of this reality in order to exploit the parallel execution of widely used applications. Hence, we improve application performance in a transparent manner, i.e., without modifying applications. Also, the virtual organization concept, emphasized by the Grid, is thoroughly vanished by the Peer-to-Peer model, introduced in our platform. By doing so, highly computing power becomes available for any user. Additionally, and in a resource sharing environment, it is essential to maintain a fine-grained control over shared resources, which is granted by the GridP2P through the security policies.

A representative selection of relevant solutions are reviewed and it is concluded that none of them entirely cover the objectives of this project. In particular to the distributed cycle sharing systems, they fail to reach the common user due to institutional barriers, they are not portable, they require modifications in applications, they lack on security, amongst other reasons. Therefore, the GridP2P reveals itself as a compelling platform within the current state of the art. Furthermore, many challenges inherent in Grid and Peer-to-Peer systems are evidenced in this thesis, such as the scalability, efficiency, heterogeneity (of machines), security, and robustness.

With respect to the evaluation, results have shown that the GridP2P may increase the performance of applications, while maintaining a rigorous control over used resources. Hence, all of our objectives were achieved. However, the scalability is always an issue difficult to evaluate in real environment due to logistic reasons.

By introducing inexpensive computing power in the hands of any ordinary user, we believe this platform will start reaching communities of Internet users across the world. We also hope that this dissertation may contribute to the study and deployment of novel peer-to-peer grid infrastructures.

6.1 Future Work

This dissertation revealed some important issues that we intend to address in the future. They are described as follows.

- Validation of Received Results: Computed tasks may contain fake results. This could be mitigated by introducing redundancy (i.e. a task being processed by more than one node) or by sending quizzes to machines (i.e. analogous to the CCOF approach).

- Global Scheduling: To improve the overall system performance and maximize the throughput, i.e., the number of executed jobs by the whole system.

- Fault-Tolerant: If a node B is processing a task from a node A, there should be a way for the node A to check if the node B is still alive. If not, that task needs to be redistributed to another node.

- Assessment of both Task Cost and Node Available Bandwidth: As we have seen in previous chapters, this is a very hard problem that requires further attention.

- Task Partitioning: The splitting of input into tasks is application-dependent. Therefore, semantic-aware engines are required to build partitioning mechanisms (based on XML descriptions for instance) for each supported application.

- Limit Bandwidth Usage: Like many other platforms we should be able to impose downstream and upstream consumption limits (e.g. No more than 15 Kbps should be used for uploading data).

- Communities: Mechanisms for handling different communities, such as requiring users to have specific applications pre-installed.

Bibliography

[ACBR03] Nazareno Andrade, Walfredo Cirne, Francisco Brasileiro, and Paulo Roisenberg. Ourgrid: An approach to easily assemble grids with equitable resource sharing. In *Proceedings of the 9th Workshop on Job Scheduling Strategies for Parallel Processing*, Seattle, WA, USA, June 2003.

[ACGC05] Nazareno Andrade, Lauro Costa, Guilherme Germóglio, and Walfredo Cirne. Peer-to-peer grid computing with the ourgrid community. May 2005.

[ACK+02] David P. Anderson, Jeff Cobb, Eric Korpela, Matt Lebofsky, and Dan Werthimer. Seti@home: an experiment in public-resource computing. *Commun. ACM*, 45(11):56–61, November 2002.

[AE07] Jorge Ardenghi and Javier Echaiz. Peer-to-peer systems: The present and the future. *Journal of Computer Science & Technology*, 7(3):198–203, October 2007.

[And04] David P. Anderson. Boinc: A system for public-resource computing and storage. In *GRID '04: Proceedings of the 5th IEEE/ACM International Workshop on Grid Computing*, pages 4–10, Washington, DC, USA, 2004. IEEE Computer Society.

[ATS04] Stephanos Androutsellis-Theotokis and Diomidis Spinellis. A survey of peer-to-peer content distribution technologies. *ACM Computing Surveys (CSUR)*, 36(4):335–371, December 2004.

[Ber99] Francine Berman. High-performance schedulers. In *The grid: blueprint for a new computing infrastructure*, pages 279–309, San Francisco, CA, USA, 1999. Morgan Kaufmann Publishers Inc.

[BG07] Francisco Brasileiro and Campina Grande. Inexpensive and scalable high performance computing: the ourgrid approach. 2007.

[CGM02] Arturo Crespo and Hector Garcia-Molina. Routing indices for peer-to-peer systems. In *ICDCS '02: Proceedings of the 22 nd International Conference on Distributed Computing Systems (ICDCS'02)*, page 23, Washington, DC, USA, 2002. IEEE Computer Society.

[CMG05] A. S. Cheema, M. Muhammad, and I. Gupta. Peer-to-peer discovery of computational resources for grid applications. In *GRID '05: Proceedings of the 6th IEEE/ACM International Workshop on Grid Computing*, pages 179–185, Washington, DC, USA, 2005. IEEE Computer Society.

[CPC+03] Walfredo Cirne, Daniel Paranhos, Lauro Costa, Elizeu Santos-Neto, Francisco Brasileiro, Jacques Sauvé, Fabrício A. B. Silva, Carla O. Barros, and Cirano Silveira. Running bag-of-tasks applications on computational grids: The mygrid approach. *Parallel Processing, International Conference on*, 0:407, 2003.

[CSWH01] Ian Clarke, Oskar Sandberg, Brandon Wiley, and Theodore W. Hong. Freenet: a distributed anonymous information storage and retrieval system. In *International workshop on Designing privacy enhancing technologies*, pages 46–66, New York, NY, USA, 2001. Springer-Verlag New York, Inc.

[dCM08] João Manuel Rebelo da Cruz Morais. Ginger-video-3d - adaptação de uma ferramenta de rendering gráfico/codificação vídeo para execução paralela em sistema peer-to-peer de partilha de ciclos. Master's thesis, Instituto Superior Técnico, 2008.

[dCRdCMZFG01] Carlos Nuno da Cruz Ribeiro, André Ventura da Cruz Marnôto Zúquete, Paulo Ferreira, and Paulo Jorge Tavares Guedes. Spl: An access control language for security policies with complex constraints. In *Network and Distributed System Security Symposium (NDSS'01)*, pages 89–107, San Diego, California, February 2001. Internet Society.

[DDLS01] Nicodemos Damianou, Naranker Dulay, Emil Lupu, and Morris Sloman. The ponder policy specification language. In *POLICY '01: Proceedings of the International Workshop on Policies for Distributed Systems and Networks*, pages 18–38, London, UK, 2001. Springer-Verlag.

[DFM00] Roger Dingledine, Michael J. Freedman, and David Molnar. The free haven project: Distributed anonymous storage service. In H. Federrath, editor, *Proceedings of Designing Privacy Enhancing Technologies: Workshop on Design Issues in Anonymity and Unobservability*. Springer-Verlag, LNCS 2009, July 2000.

[dOeSVF08] João Nuno de Oliveira e Silva, Luis Veiga, and Paulo Ferreira. nuboinc: Boinc extensions for community cycle sharing. In *2nd IEEE International Conference on Self-Adaptive and Self-Organizing Systems (3rd IEEE SELFMAN workshop)*. IEEE, October 2008.

[Don92] Jack J. Dongarra. Performance of various computers using standard linear equations software. *SIGARCH Comput. Archit. News*, 20(3):22–44, 1992.

[DZD+03] Frank Dabek, Ben Zhao, Peter Druschel, John Kubiatowicz, and Ion Stoica. Towards a common api for structured peer-to-peer overlays. In *Proceedings of the 2nd International Workshop on Peer-to-Peer Systems (IPTPS03)*, Berkeley, CA, February 2003.

[EAC98] Guy Edjlali, Anurag Acharya, and Vipin Chaudhary. History-based access control for mobile code. In *CCS '98: Proceedings of the 5th ACM conference on Computer and communications security*, pages 38–48, New York, NY, USA, 1998. ACM.

[ELvD+96] D.H.J. Epema, M. Livny, R. van Dantzig, X. Evers, and J. Pruyne. A worldwide flock of Condors: Load sharing among workstation clusters. *Future Generation Computer Systems*, 12:53–65, 1996.

[Fei95] Dror G. Feitelson. A Survey of Scheduling in Multiprogrammed Parallel Systems. Research report rc 19790 (87657), IBM T.J. Watson Research Center, Yorktown Heights, NY, February 1995.

[FK97] Ian Foster and Carl Kesselman. Globus: A metacomputing infrastructure toolkit. *International Journal of Supercomputer Applications*, 11:115–128, 1997.

[FK99] Ian T. Foster and Carl Kesselman. The globus project: a status report. *Future Generation Comp. Syst.*, 15(5-6):607–621, 1999.

[FKT01] Ian Foster, Carl Kesselman, and Steven Tuecke. The anatomy of the grid: Enabling scalable virtual organizations. *International Journal of Supercomputer Applications*, 15(3), 2001.

[FKTT98] Ian Foster, Carl Kesselman, Gene Tsudik, and Steven Tuecke. A security architecture for computational grids. In *CCS '98: Proceedings of the 5th ACM conference on Computer and communications security*, pages 83–92, New York, NY, USA, 1998. ACM.

[FW97] F.Berman and R. Wolski. The apples project: A status report francine. In *Proceedings of the 8th NEC Research Symposium*, Berlin, Germany, 1997.

[GF06] Pedro Gama and Paulo Ferreira. Computação em grelha: Perspectivas de segurança. Technical report, INESC-ID/IST, 2006.

[GKKG03] Ananth Grama, George Karypis, Vipin Kumar, and Anshul Gupta. *Introduction to Parallel Computing (2nd Edition)*. Addison Wesley, January 2003.

[GM05] Christos Gkantsidis and Milena Mihail. Hybrid search schemes for unstructured peer-to-peer networks. In *Proceedings of IEEE INFOCOM*, pages 1526–1537, 2005.

[GRF06a] Pedro Gama, Carlos Ribeiro, and Paulo Ferreira. Heimdhal: A history-based policy engine for grids. In *CCGRID '06: Proceedings of the Sixth IEEE International Symposium on Cluster Computing and the Grid*, pages 481–488, Washington, DC, USA, 2006. IEEE Computer Society.

[GRF06b] Pedro Gama, Carlos Ribeiro, and Paulo Ferreira. A scalable history-based policy engine. In *POLICY '06: Proceedings of the Seventh IEEE International Workshop on Policies for Distributed Systems and Networks*, pages 100–112, Washington, DC, USA, 2006. IEEE Computer Society.

[GW04] Paul Goodwin and George Wright. *Decision Analysis for Management Judgment*. John Wiley & Sons, Ltd, 3 edition, 2004.

[HP04] Salim Hariri and Manish Parashar. *Tools and Environments for Parallel and Distributed Computing*. Wiley-Interscience, 2004.

[IF04] Adriana Iamnitchi and Ian Foster. A peer-to-peer approach to resource location in grid environments. In *Grid resource management: state of the art and future trends*, pages 413–429, Norwell, MA, USA, 2004. Kluwer Academic Publishers.

[KSGM03] Sepandar D. Kamvar, Mario T. Schlosser, and Hector Garcia-Molina. The eigentrust algorithm for reputation management in p2p networks. In *WWW '03: Proceedings of the 12th international conference on World Wide Web*, pages 640–651, New York, NY, USA, 2003. ACM.

[LBN99] Jorge Lobo, Randeep Bhatia, and Shamim Naqvi. A policy description language. In *AAAI '99/IAAI '99: Proceedings of the sixteenth national conference on Artificial intelligence and the eleventh Innovative applications of artificial intelligence conference innovative applications of artificial intelligence*, pages 291–298, Menlo Park, CA, USA, 1999. American Association for Artificial Intelligence.

[LCC+02] Qin Lv, Pei Cao, Edith Cohen, Kai Li, and Scott Shenker. Search and replication in unstructured peer-to-peer networks. In *ICS '02: Proceedings of the 16th international conference on Supercomputing*, pages 84–95, New York, NY, USA, 2002. ACM.

[Lit87] Michael Litzkow. Remote unix - turning idle workstations into cycle servers. In *Usenix Summer Conference*, pages 381–384, 1987.

[LKR04] Jian Liang, Rakesh Kumar, and Keith Ross. Understanding kazaa. Misc, 2004.

[LLM88] Michael Litzkow, Miron Livny, and Matthew Mutka. Condor - a hunter of idle workstations. In *Proceedings of the 8th International Conference of Distributed Computing Systems*, June 1988.

[LTBL97] M. Litzkow, T. Tannenbaum, J. Basney, and M. Livny. Checkpoint and Migration of UNIX Processes in the Condor Distributed Processing System. Technical Report 1346, University of Wisconsin-Madison Computer Sciences, Apr 1997.

[LZLZ04] Virginia Lo, Dayi Zhou, Yuhong Liu, and Shanyu Zhao. Cluster computingon the fly: P2p scheduling of idle cycles in the internet. In *the internet, 3rd International Workshop on Peer-to-Peer Systmes (IPTPS 2004)*, pages 227–236, 2004.

[MM02] Petar Maymounkov and David Mazières. Kademlia: A peer-to-peer information system based on the xor metric. In *IPTPS '01: Revised Papers from the First International Workshop on Peer-to-Peer Systems*, pages 53–65, London, UK, 2002. Springer-Verlag.

[Moo65] Gordon E. Moore. Cramming more components onto integrated circuits. *Electronics*, 38(8), 1965.

[MRPM08] Elena Meshkova, Janne Riihijärvi, Marina Petrova, and Petri Mähönen. A survey on resource discovery mechanisms, peer-to-peer and service discovery frameworks. *Comput. Netw.*, 52(11):2097–2128, 2008.

[NWQ+02] Wolfgang Nejdl, Boris Wolf, Changtao Qu, Stefan Decker, Michael Sintek, Ambjörn Naeve, Mikael Nilsson, Matthias Palmér, and Tore Risch. Edutella: a p2p networking infrastructure

based on rdf. In *WWW '02: Proceedings of the 11th international conference on World Wide Web*, pages 604–615, New York, NY, USA, 2002. ACM.

[Par08] Filipe Rocha Paredes. Topologias de overlays peer-to-peer para descoberta de recursos. Master's thesis, Instituto Superior Técnico, 2008.

[PFA+06] Harris Papadakis, Paraskevi Fragopoulou, Elias Athanasopoulos, Marios Dikaiakos, Alexandros Labrinidis, and Evangelos Markatos. A feedback-based approach to reduce duplicate messages in unstructured peer-to-peer networks. *Integrated research in Grid Computing, Gorlatch Sergei, Danelutto Morco (Eds.)*, 4, 2006.

[RD01a] Antony Rowstron and Peter Druschel. Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. *Lecture Notes in Computer Science*, 2218:329–350, 2001.

[RD01b] Antony Rowstron and Peter Druschel. Storage management and caching in past, a large-scale, persistent peer-to-peer storage utility. *SIGOPS Oper. Syst. Rev.*, 35(5):188–201, 2001.

[RFH+01] Sylvia Ratnasamy, Paul Francis, Mark Handley, Richard Karp, and Scott Schenker. A scalable content-addressable network. In *SIGCOMM '01: Proceedings of the 2001 conference on Applications, technologies, architectures, and protocols for computer communications*, pages 161–172, New York, NY, USA, 2001. ACM.

[Rip01] Matei Ripeanu. Peer-to-peer architecture case study: Gnutella network. In *Proceedings of the 1st IEEE International Conference on Peer-to-Peer Computing*, page 99, Washington, DC, USA, 8 2001. IEEE Computer Society.

[RSR01] Randy L. Ribler, Huseyin Simitci, and Daniel A. Reed. The autopilot performance-directed adaptive control system. *Future Gener. Comput. Syst.*, 18(1):175–187, 2001.

[SMK+01] Ion Stoica, Robert Morris, David Karger, M. Frans Kaashoek, and Hari Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *SIGCOMM '01: Proceedings of the 2001 conference on Applications, technologies, architectures, and protocols for computer communications*, pages 149–160, New York, NY, USA, 2001. ACM.

[TR03] Dimitrios Tsoumakos and Nick Roussopoulos. A comparison of peer-to-peer search methods. In Vassilis Christophides and Juliana Freire, editors, *WebDB*, pages 61–66, 2003.

[TTL05] Douglas Thain, Todd Tannenbaum, and Miron Livny. Distributed computing in practice: the condor experience: Research articles. *Concurr. Comput. : Pract. Exper.*, 17(2-4):323–356, 2005.

[TTP+07] P. Trunfio, D. Talia, H. Papadakis, P. Fragopoulou, M. Mordacchini, M. Pennanen, K. Popov, V. Vlassov, and S. Haridi. Peer-to-peer resource discovery in grids: Models and systems. *Future Gener. Comput. Syst.*, 23(7):864–878, 2007.

[VRF07] Luis Veiga, Rodrigo Rodrigues, and Paulo Ferreira. Gigi: An ocean of gridlets on a "grid-for-the-masses". *Cluster Computing and the Grid, IEEE International Symposium on*, 0:783–788, 2007.

[WFK+04] Von Welch, Ian Foster, Carl Kesselman, Olle Mulmo, Laura Pearlman, Jarek Gawor, Sam Meder, and Frank Siebenlist. X.509 proxy certificates for dynamic delegation. In *Proceedings of the 3rd Annual PKI R&D Workshop*, Gaithersburg MD, USA, 2004. NIST TechniPubs.

[WRC00] Marc Waldman, Aviel D. Rubin, and Lorrie Faith Cranor. Publius: a robust, tamper-evident, censorship-resistant web publishing system. In *SSYM'00: Proceedings of the 9th conference on USENIX Security Symposium*, pages 5–5, Berkeley, CA, USA, 2000. USENIX Association.

[Wri01] Derek Wright. Cheap cycles from the desktop to the dedicated cluster: combining opportunistic and dedicated scheduling with Condor. In *Proceedings of the Linux Clusters: The HPC Revolution conference*, Champaign - Urbana, IL, June 2001.

[ZKJ01] Ben Y. Zhao, John D. Kubiatowicz, and Anthony D. Joseph. Tapestry: An infrastructure for fault-tolerant wide-area location and routing. Technical Report UCB/CSD-01-1141, EECS Department, University of California, Berkeley, Apr 2001.

[ZL04] D. Zhou and V. Lo. Cluster computing on the fly: resource discovery in a cycle sharing peer-to-peer system. In *CCGRID '04: Proceedings of the 2004 IEEE International Symposium on Cluster Computing and the Grid*, pages 66–73, Washington, DC, USA, 2004. IEEE Computer Society.

Part I

Appendix

A GridP2P Application Programming Interfaces

```
public interface AppAdaptationLayerI {

    // returns the data location, within the list of input, for a given application
    public String getDataLocation(String appID, List<String> input);

    // returns the config location, within the list of input, for a given application
    public String getConfigLocation(String appID, List<String> input);

    // builds a config for the n-th task of a given application
    public byte[] buildConfig(String appID, String configLocation, int n, int nTasks);

    // builds the output of a given application by joining each task result (slices)
    public byte[] buildOutput(String appID, Vector<byte[]> slices);

    // builds the output of a given application through the stored state (index) provided
    public boolean buildOutput(String appID, int index, String configLocation, String outputLocation)

    // launches an application for access the cost of a task
    public Resources launchAppTrial(String appID, String configLocation, String dataLocation);

    // launches an application given some input data
    public String launchApp(String appID, String configLocation, String dataLocation);

    // launches an application given a configuration file
    public String launchApp(String appID, byte[] config, String outputLocation);

    // creates a state for a given application
    public void createState(String appID, int index, String configLocation);

    // each time a computed task is received the state is updated
    public void updateState(String appID, int index, byte[] data);

    // gets the config file extension of a given application
    public String getConfigExt(String appID);

    // gets the data file extension of a given application
    public String getDataExt(String appID);
```

```
    // gets the application identifiers of the current node
    public String[] getAppIDs();

    // check if an application requires to store the output files or just read them
    public boolean isStream(String appID);
}

public interface JobManagerI {

    // creates a job
    public int createJob(String appID, List<String> input, String output);

    // processes a task
    public void processTask(Id id, Task message);

    // gathers a task result from the node id
    public void gatherTaskResult(Id id, TaskResult message);

    // gets the number of current tasks in progress
    public int getNumTasksInProgress();
}

public interface PolicyEngineI {

// analyses messages from the outside
    public boolean controlInbound(Message message);

    // analyses messages to the outside
    public boolean controlOutbound(Message message);

    // checks if some node metrics are suitable (in terms of policies) to perform a job
    public boolean precontrolOutbound(NodeMetrics nm);

    // sets the lower hour limit to perform work
    public void setLboundHour(int hour);

    // sets the upper hour limit to perform work
    public void setUboundHour(int hour);
```

```
    // informs the policy engine that the current node is idle
    public void setIsIdle(boolean isIdle);
}

public interface OverlayManagerI {

// update resource levels
    public void updateResources(double cpu, double mem, double bw);

    // assesses the availability of each resource
    public void updateResources();

    // gets the number of available nodes
    public int getNumAvailableNodes(String appID, int limit);

    // distributes tasks to available nodes
    public int distributeTasks(String appID, List<Task> tasks, Resources taskCost);

    // sends a task result to the node holding the job
    public int sendResult(Id id, int jobHandler, int taskNumber, byte[] output, double workDone);

    // gets the Pastry node
    public PastryNode getNode();

    // gets the environment of the FreePastry
    public Environment getEnvironment();

    // checks if the network is initialized
    public boolean networkInitialized();

    // when a neighbor joins the network this method is called
    public void joinNode(NodeHandle handle);

    // when a neighbor leaves the network this method is called
    public void leaveNode(NodeHandle handle);

    // when a message arrives, one of these methods will be called
    public void messageReceived(Id id, UpdateMsg message);
```

```
    public void messageReceived(Id id, ACKMsg message);
    public void messageReceived(Id id, Task message);
    public void messageReceived(Id id, TaskResult message);
    public void messageReceived(Id id, AvailabilityMsg message);

    // builds an update message
    public UpdateMsg buildUpdateMsg();

    // forwards a message through the node whose identifier is id
    public Id forwardMessage(Id id, Message message);

    // caches a task result
    public void cacheTaskResult(Id key, final byte[] result);

    // lookups on cache for a task result
    public byte[] lookupCache(final Id lookupKey);

    // gets the maximum storage size used for save temporary, input and output files
    public long getMaxStorage();

    // sets the maximum storage size
    public void setMaxStorage(long size);
}

public interface CommunicationServiceI {

// routes a message to a specified node
    public void routeMyMsg(Id id, Message message);

// registers the endpoint
    public void registerEndPoint();

// gets the endpoint
    public Endpoint getEndpoint();
}
```