GridP2P: Resource Usage in Grids and Peer-to-Peer Systems

Sérgio Esteves
INESC-ID/IST
Distributed Systems Group
Rua Alves Redol, 9, 1000-029 Lisboa, Portugal
sesteves@gsd.inesc-id.pt

***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.

I. Introduction

Over the last decade, there has been significant growth in terms of the technology inherent in computers. Their capabilities have been increasing in terms of computational power, memory, and persistent storage space. We have also witnessed a widespread increase in Internet access by people all over the world.

With this outlook, each time becomes more advantageous to use computational models that can exploit the utilization of shared resources, such as CPU and bandwidth, for applications which demand great computational power. One of those models, achieving great success between the scientific community, was the Grid.

A Grid system is an infrastructure for distributed computing and data management, comprising a cluster of networked, loosely-coupled computers with focus on large-scale resource sharing, innovative applications, and high-performance orientation [20]. 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 commonly-referred virtual organization, 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 organization 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.

At the same time, a model known as Peer-to-Peer has been gaining a huge success across the Internet.[1] 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 [3].

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, which suffer from problems like bottlenecks, single points of failure, amongst other.

More recently, it has been witnessed a convergence between the Grid and Peer-to-Peer computing 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

[1] 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

environments.

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. More precisely, 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, we intend to deploy a platform where any ordinary user may 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)[2] should be kept unmodified.

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.

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 overhead 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 paper is organized as follows. In the next section we present the architecture of the GridP2P, describing each of its components. Following, implementation details take place in Section III. Next, in Section IV, we describe the evaluation of our middleware. We then discuss current solutions related to the GridP2P, in Section V. Finally, Section VI presents our conclusions.

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

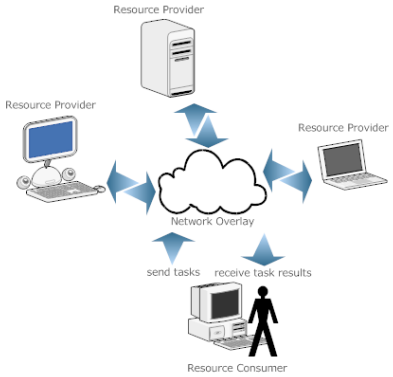

Fig. 1. Usage Model

II. Architecture

In this paper we propose a middleware platform, combining Grid and Peer-to-Peer models, that seeks to exploit parallel execution of commonly used 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.

Figure 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.

The proposed architecture relies on a vertical layer approach, depicted in Figure 2. We describe the function of each layer next.

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

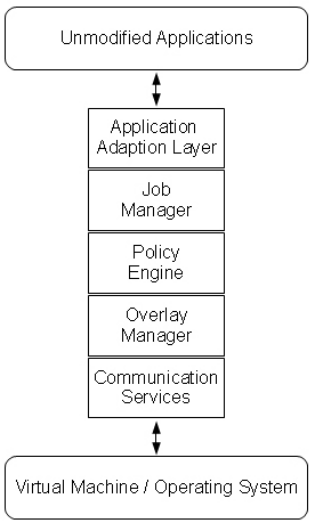

Fig. 2. System Architecture

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. 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 archival storage used as a cache for storing computed tasks.

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, for the

Fig. 3. Overlay Manager Overview

case where input files consist of scripts, programming code, and so forth.

Briefly, the procedure for **accessing resources** works as follows: First a user specifies the application, parameters and input files through the GUI. The GUI contacts the Job Manager (JM) 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 again the OM to distribute the tasks among available machines. When the computation related to each task is done the results are sent back to the machine that holds the job. The OM receives the data and send it up to the Policy Engine (PE) where data will be evaluated. If no policies can be applied (i.e. no action to take), the PE lets the data pass to the JM. The JM gathers results and, when all tasks are completed, it builds the final output and notifies the user through the GUI.

A. 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). It should be possible to create independent tasks from those applications, i.e., tasks that do not require communication between them during execution time.

Our platform has specific mechanisms for handling each type of application. Such application-dependent mechanisms are built, at boot time, by the Application Adaptation Layer, and they are driven by XML format descriptions. 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, and the rules on how to combine task results and on how to scatter input files into pieces to create tasks.

B. Overlay Network Management

The management of the underlying network is done through the Pastry overlay [14]. On top of it, our platform provides a

mechanism for locating resources based on the Informed Walk, described in Section V-B. Nodes advertise themselves by sending update messages to their neighbors across the Pastry overlay. These messages contain the sender node related information, such as its identifier, supported application identifiers, and resource availability levels (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 consists of associating weights with the measured availability of every single resource. Soon after, 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 by the resource manager. 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.

C. Distributed Storage of Task Results

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.

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 input data. Hence, the cache maps task input data digests into task output data (i.e. results). If the task result is cached, 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.

D. Task Creation

Once a job is submitted, our platform estimates the number of available nodes so that the resource utilization within the overlay would be maximized. This process is completely transparent to the user.

Through its neighborhood cache, a node assembles a set with the best available neighbor nodes (i.e. whose rate is highest). 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, the absolute availability tells us if a node is available or not for processing tasks, disregarding the load of resource utilization.

Concerning the distribution process, tasks are sent one by one to the best nodes that are suitable for performing those given tasks. Nodes that have been marked before as non available are disregarded in this process. Besides that, a node is suitable if its resource availability levels are equal or greater than the cost of the task.

The cost of a task defines the minimum requirements that a node machine has to comply with to compute the respective task. 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 more powerful and available machines in terms of their resources. Furthermore, the task 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.

E. 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.

W.r.t. 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. Soon after, 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 lighweighted 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.

III. Implementation

A. 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[3] which is a Java implementation of the Pastry overlay.

B. Main Data Structures

- 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.
- 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.
- 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.

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

C. Message Types

- Update Message: A node sends this message to all of its neighbors in order to announce its both presence and resources.
- 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.

D. Distributed Storage of Task Results

The implementation of our distributed cache, for storing computed tasks, relies on the PAST [15]. The integration of PAST with our platform is easily done, since PAST is meant to work upon FreePastry.

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.

E. Security Policies

Our Policy Engine Layer uses the Heimdall [9]. 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.

F. 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.

IV. Evaluation

In this section we present the evaluation of the GridP2P platform regarding its performance and viability when facing real environment. Due to their importance among our goals, we test the performance of the system in terms of application speedup, distributed cache, and security policies.

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.

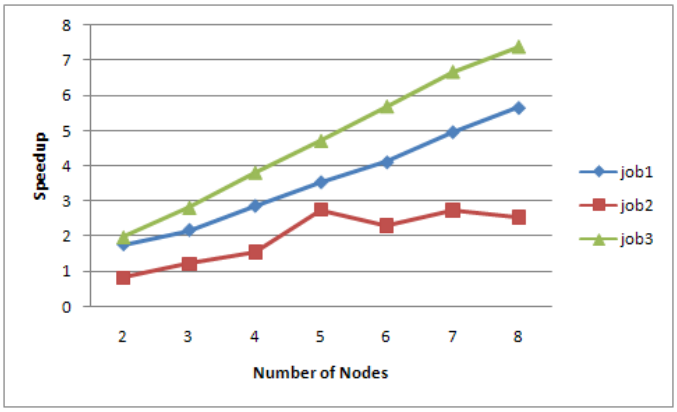

Fig. 4. Jobs' Speedup

- The third one, job3, consists of a Monte Carlo simulation for a sum of several given uniform variables.[4] 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.

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 Mbps. For each trial we obtained the execution time of the job, and therefore we also got the speedup.[5]

In Figure 4 we may conclude that the best jobs to parallelize are the ones whose tasks have the same complexity, in this case, job3. With this job, we almost got a linear speedup.[6] 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. Besides that, the job3 output was very small sized in comparison with the job1 output, i.e., we also found out that the network overhead has a significant impact on the overall system performance.

Regarding the second stage, a cache is important to avoid doing duplicated work (i.e. optimizing resource usage), and thus increase application performance.

The gains are higher for tasks whose complexity differs, in this case, job1 - Figure 5. 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. W.r.t. job3, it showed that caching tasks with the same complexity does not decrease the execution time, except if all tasks are cached (obviously). However, in any case, *n* machines will remain free if *n* tasks are present in cache, and that is the essential point.

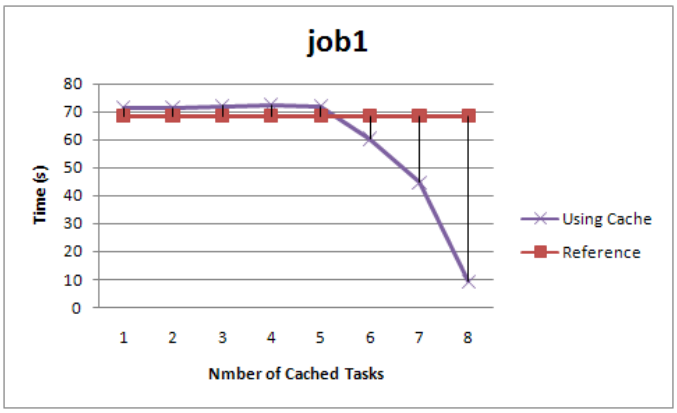

Fig. 5. Improvement time when the cache is used. Reference is when no tasks are cached.

Finally, the policy engine overhead turned out to be minimal. It is directly proportional to the number of deployed policies. Now, we assess application performance against our deployed policies. That comparison is made in percentage, where 100% corresponds to the time with policies.

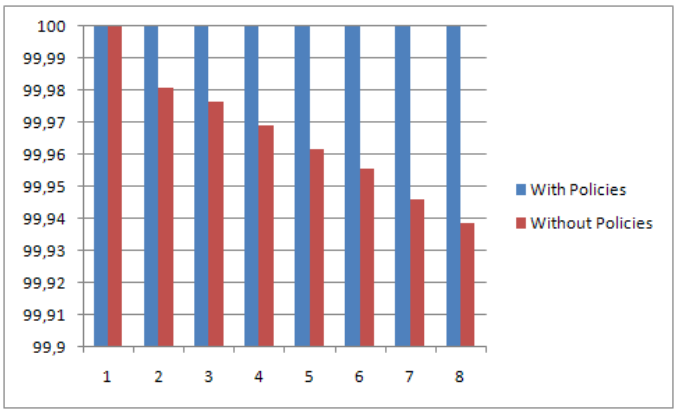

Fig. 6. Application Performance With and Without Policies

In Figure 6 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 1 node is available, as would be obvious. When more nodes are available, the policy engine overhead corresponds to approximately 0,042 seconds. In the case of job1, where we got *221,146* and *179,077* seconds of execution time with 2 and 3 nodes respectively, the overhead corresponds to: 0,0189% of the time taken with policies and with 2 nodes, 0,0235% with 3 nodes, and so forth.

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

[5]$S_p = T_1/T_p$, 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.

[6]Linear speedup or ideal speedup is obtained when $S_p = p$

A. Discussion

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.

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 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.

V. Related Work

In this section 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.

A. Job Scheduling

The job scheduling is the assignment of work to resources within a specified time frame [4]. 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. Scheduling may also be used for load balancing purposes. 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.

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.

Currently, GridP2P is more focused on scheduler mechanisms covering local jobs and applications, although not disregarding at all the system performance as a whole (as security policies may control multiple requests over resources).

B. Security

Deeds [7] is a history-based access control system whose policies must be written in Java. The security events and enforcement mechanisms (handlers) should be individually implemented for every resource to be protected. This ad-hoc manner of managing policies becomes hard and impractical, especially in dynamic and complex environments.

Heimdall is a history-enabled policy engine targeted to Grid environments. This system allows the definition, auditing, and enforcement of history-based policies. Through a language with an high level of abstraction and expressiveness, several usage semantics and security patterns may be applied within a complex environment. With this manner of defining policies, the specification and implementation of security mechanisms are separated, and thus leveraging the policy administrative tasks. By doing so, Heimdall constitutes then a solution that fits in the needs of the GridP2P.

C. Resource Discovery

Iamnitchi et al [10] have compared different searching methods and it turned out that a learning-based strategy achieves more performance. Such strategy consists of forwarding a request to the node that answered similar requests previously (i.e. using a possibly large cache). Moreover, results have shown that 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.

CCOF [17] has tried several approaches, and the one obtaining more global performance was based on a partially centralized peer-to-peer overlay. Within the best search approach, some nodes may acquire a special role in the network and provide a service of lookup for nodes nearby. This way, nodes advertise their profiles and address requests to those supernodes. Whenever a request is made, supernodes attempt to match the query with cached profiles and return a set of candidate nodes. Nevertheless, the dynamic placement of these supernodes is still an open problem.

Paredes [13] presents a solution through which queries are forwarded to the neighbor nodes with best availability and reputation. The best availability concerns the idleness level of a node in terms of its resources, and the reputation consists in the capability of a node to forward a query to other nodes with availability (i.e. this reputation is based on previous requests). Results have shown that this approach is efficient and scalable, and thus matching the GridP2P requirements.

Cheema et al [5] proposed a solution for exploiting the single keyword DHT lookup for CPU cycle sharing systems. This solution consists in encoding resource identifiers based on

static and dynamic resource descriptions. The static ones could be, for instance, the OS configuration, RAM, or CPU speed. While the dynamic descriptions are related to the availability levels of resources, such as the percentage of idle CPU. With this encoding mechanism, it is possible to create a mapping between resource and node identifiers in structured peer-to-peer networks, like the Pastry, and take advantage of the efficient routing of queries.

D. Cycle Sharing Systems

1) Institutional Grids: Globus [8] is an enabling technology for grid deployment. It provides mechanisms for communication, authentication, network information, data access, amongst other. The authentication and authorization models are directed to institutions, making difficult for ordinary users to deploy applications on top of the Grid. In contrast, the GridP2P envisions for an open access environment whereby the complexity of getting credentials to use the Grid is reduced.

Condor [11] allows the integration and use of remote workstations. It maximizes the utilization of workstations and expands the resources available to users, by functioning well in an environment of distributed ownership. Although, Condor's jobs rely on executable binary code in which compatible machines are needed in order to run them. Contrastingly, machine heterogeneity is not a problem in GridP2P, since jobs consist of data files that can be easily read by any computer (i.e. the kind of architecture and operating system are not relevant). Plus, many Condor features require some degree of expertise (i.e. advanced configurations are needed), whereas our platform keeps the vision of simplicity and tries to do all the necessary work with almost no interference from users.

2) Master-Slave Model: BOINC [1] is a platform for volunteer distributed cycle sharing based on the client-server model. It relies on an asymmetric relationship where users, acting as clients, may donate their idle CPU cycles to a server, but can not use spare cycles, from other clients, for themselves. Besides that, setting up the required infrastructure, developing applications, and gathering enough cycles could be difficult for an ordinary user. On one hand, users need to have the required skills to create BOINC projects, and on the other hand projects should have a high profile to attract users to participate in. In comparison, GridP2P is more flexible as users have the same power to both provide and consume idle cycles to and from other machines. Moreover, it is possible to use common and widely used applications with our system.

nuBOINC [6] is a project that attempts to overcome the drawbacks presented by BOINC. It allows one to use idle cycles from other users, through servers, and making use of commodity applications. However, GridP2P relies on a more scalable model that does not require the intermediation of servers. Also, when work units are being distributed, our platform takes into account the idleness levels of user machines, which is disregarded by the nuBOINC.

3) Peer-to-Peer: CCOF [12] is an open peer-to-peer system seeking to harvest idle CPU cycles from its connected users. It shares our goals of reaching the average user by not requiring any kind of membership or negotiations in any organization (i.e. in contrast with institutional grids). The access of joining nodes to the CCOF is only 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.

OurGrid [2] is a peer-to-peer network of sites which tries to facilitate the inter-domain access to resources in a equitably manner. Each of these sites comprises grid clusters possibly belonging to different domains. The sharing of resources is made in a way that makes those who contribute more to get more when they need. 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. Besides that, machines are not distinguished by their idleness levels, whereas our platform always attempts to select the best available nodes for a job.

Ginger [16] 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. It introduces the Gridlet concept: a semantics-aware work unit containing a chunk of data and the operations to be performed on that data. Home users exchange gridlets across the peer-to-peer overlay in order to compute chunks of data. Ginger shares our goals of deploying a scalable peer-to-peer grid infrastructure. Although, it is not clear how Gridlet operations, consisting of application binaries, would run on remote machines. For instance, popular desktop applications often need to be installed (i.e. root access is needed); and their binaries are dependent from machine architectures/operating systems. Additionally, the control over resource usage is not taken into account, unlike the GridP2P.

VI. Conclusions

This paper 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 project, 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 this work may contribute to the study and deployment of novel peer-to-peer grid infrastructures.

A. Future Work

This paper 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.

REFERENCES

[1] D. 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.

[2] N. Andrade, W. Cirne, F. Brasileiro, and P. 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.

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

[4] F. 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.

[5] 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.

[6] J. N. de Oliveira e Silva, L. Veiga, and P. 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.

[7] G. Edjlali, A. Acharya, and V. 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.

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

[9] P. Gama, C. Ribeiro, and P. 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.

[10] A. Iamnitchi and I. 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.

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

[12] V. Lo, D. Zhou, Y. Liu, and S. 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.

[13] F. R. Paredes. Topologias de overlays peer-to-peer para descoberta de recursos. Master's thesis, Instituto Superior Técnico, 2008.

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

[15] A. Rowstron and P. 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.

[16] L. Veiga, R. Rodrigues, and P. 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.

[17] 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.