

Using Semantics for Enhancing Resource Allocation in Service Providers

Jorge Ejarque, Marc de Palol, Ferran Julià, Íñigo Goiri, Jordi Guitart, Rosa M. Badia, Jordi Torres
Barcelona Supercomputing Center

Jordi Girona 31, 08034 Barcelona, Spain

{jorge.ejarque, marc.depalol, ferran.julia, inigo.goiri, jordi.guitart, rosa.m.badia, jordi.torres}@bsc.es

Abstract

Current situation in the service provisioning is that agreements and management are done manually, with lots of human interaction. In project BREIN, a full framework to enable service providers to reduce costs whilst maximizing profit is under design. This paper focuses in a prototype developed during the first year of the project, the Semantically-Enhanced Resource Allocator that focuses on the service provider level. The Semantically-Enhanced Resource Allocator provides a framework to assign resources depending on the information given by service providers with regard to the level of preference (according to business goals) of their clients and on the requirements of the clients, and evaluates how agent and semantic technologies can help to improve this process. Besides, virtualization is used to better exploit the utilization of the resources by designing fine-grain resource allocation policies and to guarantee full isolation and customization of the applications running in the service provider. The paper presents the design, implementation and evaluation of this prototype.

1 Introduction and Motivation

BREIN (Business objective driven REliable and Intelligent grids for real busiNess)[1] is an Integrated Project (IP) funded by the European Commission, with the objective of bringing the e-business concept developed in recent Grid research projects [13], namely the concept of so-called *dynamic virtual organizations* toward a more business-centric model, by enhancing the system with methods from artificial intelligence, intelligent systems, semantic web, etc. Thus, the BREIN project will enable business participants to easily and effectively use Grid technologies for their respective business needs.

Of remarkable importance in the first 12 months of BREIN has been the activity named *Rapid Prototyping*, which aimed to provide experimental implementations, to perform experimental tests about design approaches of tech-

nologies integration and middleware interoperation. Four prototypes have been developed in the framework of this activity during this first year of life of the project. The team at the Barcelona Supercomputing Center (BSC), composed by researchers from the BSC research lines *Autonomic Systems and e-Business Platforms* and *Grid Computing and Clusters*, has been leading one of the prototypes, developed together with ATOS Origin. This prototype is at the Service Provider level and has been named *Semantically-Enhanced Resource Allocator*.

The *Semantically-Enhanced Resource Allocator* provides a framework to assign resources depending on the information given by service providers with regard to the level of preference (according to business goals) of their clients and on the requirements of their tasks. The allocation process is enhanced by using challenging technologies such as agents, semantics and virtualization.

Virtualization technique is a key issue in this system. It improves resource utilization and enables easy and fast resource management, maintaining the applications isolation. Nowadays, there are many alternatives for virtualizing machines [12], such as, VMWare or QEMU. Nevertheless, these solutions provide full virtualization which implies an overhead that cannot be assumed by the system. In addition, most of these solutions do not provide tools or interfaces for easy resource managing and monitoring. A technology called *paravirtualization* achieves high performance levels [3, 2, 6] by presenting a software interface to virtual machines that is similar but not identical to the underlying hardware instead of offering the whole system. An example of virtualization system using paravirtualization is Xen Hypervisor [14]. This open-source project also provides many ways to easily access to its internal resource management, such as XenStat or XenAPI.

Regarding the semantic technologies, there has been a significant interest of enriching grid resources and services with semantic description in order to make them machine understandable[9]. One of the models proposed to support semantics in Grids is the Semantic OGSA [5]. It was proposed by the EU FP6 OntoGrid project [7] and consists of

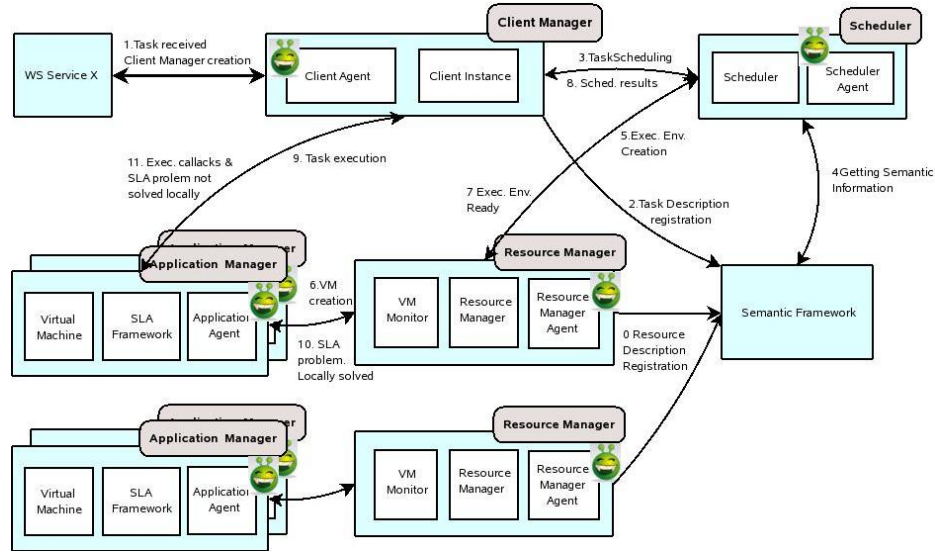


Figure 1. Semantically-Enhanced Resource Allocator prototype architecture

a set of OGSA compliant services in charge of managing semantic descriptions bound to a grid resource, ontologies, rules and reasoning engines. Previous work about semantic support for scheduling and ontologies for resource access coordination has been presented in [8] and [10]. We have partially used their results to build our resource allocator.

The paper is organized as follows: Section 2 presents an overview of the Semantically-Enhanced Resource Allocator; Section 3 describes how semantics and ontologies are used to take scheduling decisions; Section 4 describes the dynamic resource management implemented in the system through paravirtualization; Section 5 presents the SLA management of our prototype, including SLA monitoring and SLA enforcement; Section 6 describes the experimental environment and tests we performed to evaluate the prototype. Finally, Section 7 presents the lessons learned in the implementation of the prototype and the future work.

2 Architecture of the Semantically-Enhanced Resource Allocator

The aim of this section is to give an overview of the overall architecture of the system presenting the main components and how they interact with each other. The Agent-based and Semantically-Enhanced Resource Allocator consists of four active components (Client, Scheduler, Resource and Application Managers) and a semantic framework which is responsible of managing the metadata related to each system entity. Figure 1 shows the main components involved in this prototype and gives an overview of how they interact with each other. The following paragraphs ex-

plain briefly the components functionalities, the tasks life-cycle in the system and the technological frameworks used to implement the prototype.

Client Manager (CM): It is in charge of managing the client job execution, requesting the resources needed, executing jobs using the assigned resources and taking decisions about what can be done when an unexpected event arises (failure or SLA violation).

Semantic Scheduler: It assigns the amount of required resources per client's task depending on the client's priority, load of the system and the state of resources, in such a way that the clients with more priority are favored.

Resource Manager (RM): It creates a virtual machine (VM) to execute a client's task according to the minimum resource assignment (CPU, memory, disk space...) done by the scheduler and the task requirements (e.g. software required). Additionally, it dynamically redistributes the remaining resources among the different client tasks depending on the resource usage of each running task. If a temporal increment of the resources usage demand for one task appears, the system can solve it with the resources that are not used by the other ones.

Application Manager (AM): It is in charge of monitoring the resource usage and SLA parameters. It also evaluates if a SLA could be violated. If a SLA problem occurs, the AM will contact the RM to try to solve the

problem locally. If it cannot be solved locally, the AM will forward the message to the CM.

Semantic Framework: It consists of a set of services which are in charge of registering, storing and publishing all semantic descriptions associated to all service provider level entities.

2.1 Task Lifecycle

In figure 1, we can follow the task lifecycle in the prototype. An initial step performed once the system has been initialized is the registration of the resources descriptions in the Semantic Framework (0). The task lifecycle starts when a task arrives to the system (1) and a CM is created to manage its execution in the system. It registers the task description into the Semantic Metadata Repository (2) and requests to the Scheduler a time slot and resources to execute the task (3). The Scheduler infers where the task can be executed with the metadata stored in the Semantic Metadata Repository (4), informs the CM if its task has been successfully scheduled or canceled and contacts the RM in charge of the node where the task has been allocated requesting for the creation of a VM to execute the task (5).

When the RM receives the request, it creates a VM and an AM to monitor the SLA fulfillment for this task (6). Once the VM is ready, the access information to the VM is sent back to the Scheduler (7) and the CM (8). Then, the CM can submit the task to the recently created VM (9).

When the AM detects a SLA violation (the SLA cannot be accomplished), it requests for more resources to the RM, trying to solve the problem locally (10). If the RM cannot provide more resources to the VM violating the SLA, the AM forwards the request to the CM (11), who will try to increase the resource assignment of the task by requesting more resources to the Scheduler. The CM also receives the execution callback and it notifies the client or resubmits the task if it fails.

2.2 Technological Frameworks used in the Implementation

Each component contains an agent and a core component. The agent is in charge of the communication between components, task execution monitoring, detecting undesired events (SLA violations, failures,...) and coordinating the reaction of a component to these events. The agent wraps the core component functionalities by means of a set of agent behaviors which basically call methods of the core components. The reason of this wrapping is because the inter-component communication is easier with agents, and some of the required features are provided by a specific behavior, for instance the SLA monitoring functionality can

be implemented as a cyclic behavior or other behaviors are executed when a special message is received.

Other technologies used in the prototype implementation are shown in figure 2. We want to highlight the use of Globus Toolkit 4 (GT4) for task execution, data transfers, and access to the Semantic Metadata Repository services; Jena 2 framework as inference support and semantic metadata treatment; Jade as agent platform and Xen Hypervisor to manage virtual machines.

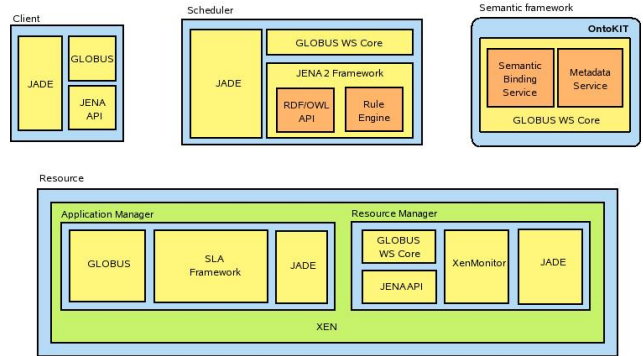


Figure 2. Technological frameworks used in each component of the prototype

3 Using Semantics for Scheduling: Client Manager and Semantic Scheduler

3.1 Semantic Model

The ontology of our prototype is based in the work presented in [10] that describes tasks and resources in general. We have modified this ontology in order to adapt it to our system. Basically, we have created subclasses of resource which are controlled by a Resource Manager (CPU, Memory, Disk, Software,...). Moreover we have changed the task description to include the task requirements which did not exist in the original ontology. These requirements include:

- Hardware requirements
- Software requirements
- Task dependency requirements

Finally, we have also extended the client description, which is the owner of tasks, to reflect its priority within the system. It allows to take advantage of the reasoning engines and, for example, assign priorities to tasks depending of its owner.

3.2 Client Manager (CM)

When a new task request enters the system, a new CM is created to manage the task execution into the system. The CM agent and their associated tasks are semantically described according to our ontology and registered into the Semantic Framework.

Once the task and the CM agent have been described, the agent performs the first step of the scheduling process. It creates a SPARQL semantic query [11], which selects from the Metadata Service all the machines that match with the hardware requirements of the task. The query results are also inserted in the task description. Performing this step in the CM, we unload the Scheduler and it is also a first step toward a fully distributed scheduling system via agents.

After these steps are completed, the CM sends a scheduling request to the Scheduler that will later notify if the task has been successfully scheduled or not. If the answer is positive, the CM remains waiting until a new notification is received notifying that the virtual machine is ready and that the task can be executed in a given machine. This task execution is performed by means of GT4 job submission interface and the WS-GRAM installed in the created VM.

In case that the Scheduler is not able to find a machine/time slot to execute the task the CM must decide what to do. This is one of the aspects of the system where we take more advantage of the intelligent capabilities of agents. A decision must be taken, and several factors must be taken into account, such as the task owner and the task deadline. This decision could be a modification in the task requirements, for example requiring less CPU power, or trying to discover new suitable machines. The CM also reacts when a SLA violation appears and it has not been solved locally. This behavior is explained in Section 4.

3.3 Semantic Scheduler

The Scheduler is a proof of concept implementation of decision taking using reasoning, rules with builtins and agents. The reason for a heavy use of semantics in the scheduling process is to be able to efficiently manage resources as well as for compatibility with other Job Schedulers.

Once the Scheduler receives the job description from the CM, it performs the following tasks:

1. Get the needed metadata: As the Scheduler features semantic properties, it needs semantic information, as RDF data, about the data involved in the scheduling process. All this data is maintained united and updated in the semantic repository, so all necessary data for the scheduling process is fetched from there via WSRF Web Services. This data is used in the inference, so there must be:

- (a) Semantic Data describing each machine: The process needs data describing the different machines that will take part in the scheduling process. Not just the proposed machines in the task description, also the proposed machines in all scheduled tasks in the system, because this process can reschedule a task into another machine.
 - (b) Semantic Data describing already scheduled/running tasks: The inference must know all tasks that are in the system, including the ones that are queued and the ones that are executing, also must know their priorities, that is, their owner.
 - (c) Semantic Data describing the clients that are executing tasks in the system: As stated in the previous bullet, the clients hold, among other metadata the priority, taken as a business data.
2. Inference: Once the Scheduler has retrieved the data, it prepares the data to the inference creating a Jena 2 Model with the data, the ontology and rule engine attached. We have implemented three Jena 2 rules which use a set of auxiliary builtins (rule extensions written as application code). These rules are loaded to the rule engine and are fired during the inference process.

- **Rule 1: First Come First Serve.** This rule is fired for all the tasks that are in the system which are in a requested state (recently created). This rule tries to find a time slot and a machine using the FCFS algorithm, taking into account the hardware requirements and the space left in each computer to allocate a new virtual machine.
- **Rule 2: Task reallocation.** If the Rule 1 is not capable to find a time slot and a machine for executing the task, the Rule 2 tries to move scheduled tasks (not already running) from one machine to another one in order to find a time slot for the new task (taking into account hardware requirements and space left as in Rule 1).
- **Rule 3: Less priority task(s) cancellation.** If Rule 2 is not able to find a solution, then the Rule 3 tries to cancel the tasks with less priority than the new one.

3. Collection and interpretation of the results. When the inference process is over, the Jena 2 gives back a deductions' graph. Each deduction in this graph means a change in the original graph, so it means that there has been some changes in the state of the system, or in other words that a task has been scheduled, rescheduled or canceled. So, the scheduler gets all necessary information about the updated tasks. After that,

all the affected data gets updated in the semantic repository. Task cancellations are notified to their CM and the Scheduler update its scheduled task queue.

At the same time, the Scheduler monitors the execution time of the queued tasks. When the execution time is close, it request the RM a VM with the task requirements. When the RM has completed the VM creation, the Scheduler notifies to the CM that its task can be executed there.

4 Management of Virtualized Resources: Resource Manager

This section discusses the architecture and the implementation issues of the Resource Manager (RM), which manages the Virtual Machine (VM) life cycle and the distribution of the physical resources.

4.1. Virtual Machines Lifecycle

Once the RM is created and fully started, its agent waits for requests from the Scheduler and sends required information to its core component. The RM checks if it is possible to create a new VM with the required features and informs the Scheduler about the success of the VM creation.

Each VM is created using the configuration parameters provided by the Scheduler. Configuration parameters specified by the Scheduler includes VM requirements and SLA. The RM guarantees that these minimum requirements are met during the whole VM lifetime. As a consequence of this resource allocation decision, is possible that some surplus resources are not really allocated to any VM. In order to maximize resource utilization and provide better service to the applications, the RM distributes these surplus resources among the VMs (see next section).

Creating a new VM implies: downloading and installing the operating system, copying extra needed software, creating home directories and swap space, setting up the whole environment, packing it in an image and starting it up. From this description, we can derive two bottlenecks: network (system download stage) and disk (applications copy and system image creation, approximately 1GB of data).

First bottleneck is solved by creating a default image with no settings that is copied by each new VM. Nevertheless, it increases second bottleneck, disk. Second one has been solved by adding a caching system that periodically copies disk images to a cache space in memory. Finally, when a new VM is created, RM moves these images (just an i-node change) to the final location.

Thanks to these caching techniques, downloading time has been eliminated and the whole machine creation has been reduced from up to 40 seconds to an average time of 10 seconds.

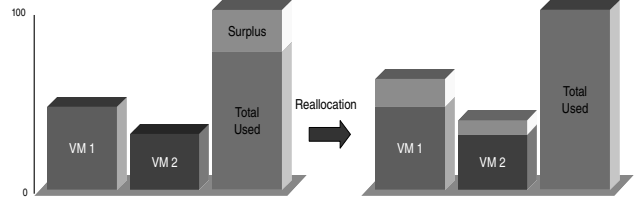


Figure 3. Surplus resource distribution

Extra software needed by the client is located in the applications image that is automatically mounted in the VM. One of the mandatory software on the system is the Globus Toolkit, which enables the client executing and monitoring tasks. The certificates are regenerated for each VM and it is started up when the VM system boots. This image can contain any Linux application from a transactional one, like Tomcat, to a computational intensive.

When the client has finished the task or the Scheduler decides that this task should be rescheduled or canceled, the VM needs to be destroyed. VMs are always monitored by its Application Manager, hence, its agent must be stopped and killed. Once the VM is destroyed, resources become underused again, hence, the surplus resources has to be re-distributed again.

4.1.1 Resource Distribution among VM

The RM is also the responsible of distributing the physical resources among the VMs. The goal is to maximize physical resources utilization, while respecting the minimum assignments decided by the Scheduler. This is done at two different levels: static and dynamic. Static allocation ensures the initial assignment requested by the Scheduler, and the second one shares the surplus resources among the VMs according to its resource utilization and its SLA (see figure 3).

Each user has a given priority, e.g. Gold, Silver, etc. which is taken as the initial priority of its VM by the RM. This priority is dynamically modified by the Application Manager in order to request for more resources (by increasing the VM priority) if the SLA is violated (see next section). Every priority modification or VM change implies a recalculation of resource assignment and a resource binding to each VM according to the following formula:

$$R_{assigned}(i) = R_{requested}(i) + \frac{p_i}{\sum_{j=0}^N p_j} \cdot R_{surplus}$$

where p_i is the priority of client i

Current implementation is able to manage CPU and memory resources. CPU management is straight-forward,

but dynamic memory management has limitations because the guest OS (Linux) can not manage more memory than the initially allocated. In order to overcome this limitation, when a new VM is created, RM reduces all VMs memory to the minimum requested allowing the new machine to take as much as is possible.

5 SLA Management: Application Manager

This section describes the AM (AM), which is the component that monitors the execution of tasks in order to detect SLA violations. As we have mention in the introduction, this prototype is focused on the internal part of a service provider, so we do not deal with the SLA negotiation between the customer and the provider. We assume that an SLA has been negotiated and our prototype must guarantee the fulfillment of the SLA parameters when a task is executed.

The previously negotiated SLA is attached to the task execution request that arrives to the RM. This SLA is assigned to the SLA monitor that monitors the VM which is going to execute the task. The AM guarantees that the SLA is not violated, and in the opposite case, the RM is asked for more resources that are dynamically reallocated. Nevertheless, the physical machine has finite capacity, so it can be possible that the RM cannot give more resources to the VM which violates its SLA. In this case the AM contacts the CM to communicate the violation. When the CM receives this notification, it must decide what to do with the task that is violating its SLA, taking into account the client priority and task deadline. The possible solutions are: To continue the execution besides the SLA violation, if this is not important for the client; modify task requirements and re-schedule the task, if the client is important and task deadline permits the rescheduling; or cancel the task if the client is not important.

In order to monitor resource usages, the AM must be executed out of the VM. This is mainly for two reasons. First, if the monitoring system was inside the VM, it would consume part of the assigned resources to the task execution. This would mean that the costumer would assume the monitoring cost. Second, the monitoring system cannot take real measures inside the VM, all the data collected would be wrong and the only way to get correct measures is asking the Xen Hypervisor. We tried to use other monitoring options (see Ganglia [4]), but they do not have explicit support for Xen and the measures obtained where not correct. In our case, measures are taken using the XenStat and XenAPI.

The other AM functionality is to allow the task execution. This is done by a Globus container deployed in each VM. It is configured during the VM creation and started at the VM boot time, in such a way that the CM can easily submit the task to the VM and check its state using the GT4

interface.

6 Experimental Environment and Evaluation

We have created a simple testbed to test our prototype. This is a real demonstration we made for testing purposes and for measuring some performance metrics.

The experimental environment is composed by two Resource Managers in two different machines, RManager1 and RManager2, which can create virtual machines with preinstalled software (such as GT4 or Tomcat) on a de-bootstrapped Debian Lenny. The Client Managers, the Scheduler and the Ontokit are installed in another machine. All the messaging and communication is done through the agents.

In order to quantify the amount of CPU allocated to each virtual machine, typical Linux CPU usage metric is used. If the machine has 4 CPUs, the amount of CPU will be 400%.

Unifying metrics between machines to define the SLA is a key issue. For instance, if a machine has two cores at 1,33 GHz it will be different to a quad core at 2GHz each one. We have used a simple approach for quantifying the needed CPU for a given task: the product of desired percentage and the CPU frequency.

This experiment focuses on dynamic reallocation of CPU between different virtual machines and on the Scheduler behavior. We have designed a test focused in this particular case, nevertheless, these concepts could be applied to other resources like memory, network bandwidth, etc.

In this experiment we send a total of seven tasks, (table 1), to the system (table 2). Table 1 describes for each task the requested CPU (*Req.*), the actual consumption of the task (*Consum.*), the task duration (*Durat.*) and the deadline (if specified in the requirements, *Deadline*). Deadlines are specified assuming that the use case initial time is 00:00. Table 2 describes for each RM, the amount of available CPU and Memory in MBytes.

Id	Task	Req.	Consum.	Durat.	Deadline
1	tomcat	105	105	20 '	-
2	consumer	105	150	20 '	-
3	consumer	105	70	20 '	-
4	consumer	100	100	20 '	-
5	true	100	100	20 '	-
6	true	400	400	20 '	00:40
7	true	100	100	20 '	00:40

Table 1. Tasks description

The first task sent to the system is an application server requiring 105% of CPU. It can only be executed on RManager1 because RManager2 can only execute jobs of less

Name	Id	CPU	Memory
RManager1	RM1	400	6144
RManager2	RM2	100	2048

Table 2. Resources description

than 100% of CPU. It is immediately scheduled and sent to RManager1, which creates the virtual machine allocating the whole 400% of CPU to this task (105% required and 295% as surplus resources). The second task sent requires also 105%, so it is also scheduled on RManager1. This simulates the case when the requirements specified in the semantic description of the job does not match with the ones specified in the SLA. In this case, the task is a CPU consumer (simulating and HPC scientific application) that really consumes 150%, while the SLA indicates a requirement of 105%. This task is started with 200% of CPU (105% required and 95% surplus) while the first task surplus is downgraded to 95%. The third task is another CPU consumer, which requires 105% and consumes just 105%. Again, the surplus of the machine is distributed between the three tasks executing. This results in an assignment of 133% for each running job. Since the second task requires 150%, an SLA violation arises and its Application Manager is forced to ask for more resources to the RManager1. Figure 4, shows the graphical evolution of the CPU assignment to each VM until all tasks fulfill their SLA. After the CPU reallocation, the second task has an assignment of 190% and the first and the third of 105% each.

The fourth task, requires 100% of CPU and really consumes 100% of CPU, is scheduled in RManager2. At this moment, only a task with a requirement of 85% of CPU can be allocated in the system. Tasks demanding more resources will be, then, queued in the Scheduler job queue until the resources are available.

Next three tasks exemplify how the Scheduler reschedules tasks between different Resource Managers. The fifth task is demanding more than the available 85% of CPU, and it is initially scheduled into the RManager1. In this case, the task is not immediately executed, and remains queued in the Scheduler (see Table 3.a). The sixth task has a deadline that implies that at least has to be scheduled for execution right after the current running tasks. (note that we do not interrupt already running jobs). Since this task requires 400% of CPU, it cannot be scheduled into RManager2. But RManager1 cannot accomplish the tasks deadline unless the Scheduler reschedules the fifth task to RManager2 (see Table 3.b). The seventh task has also a sharp deadline, the same reasoning is done, as the deadline cannot be met. The fifth task is again rescheduled after the sixth and the seventh queued in the RManager2 queue (see Table 3.c).

The experiment exposed here was designed to show the special qualities of the prototype, and it can also execute

RManager1	RManager2
Task5	

(a)

RManager1	RManager2
Task6	Task5

(b)

RManager1	RManager2
Task6 Task5	Task7

(c)

Table 3. Task rescheduling

the different tasks in a different order. We implemented the test case and build a running demonstrator that fully accomplishes the expected behavior of the system.

7 Conclusions and Future Work

We have presented a prototype to assign resources depending on the information given by service providers. During the implementation of the prototype we have evaluated the suitability of using semantics, agents and virtualization to enhance this process and we have reached the following conclusions.

Some agent behaviors are useful for monitoring and reacting to possible undesired events, such as SLA violations or failures, as well as increase the flexibility of the system. The semantic description of the system components with metadata improves the possibilities to reason and use business factors. The reasoning by means of rule engines, produce a slight performance slowdown, but it is compensated with the enrichment of data involved in the process and the flexibility to change policies only changing the rules into the rule engine.

Regarding paravirtualization, it has enabled a new way to create virtual machines that achieve high performance levels, and it is a promising technology due to the virtualization support included in the new processor architectures. However, we have found some problems which are not still solved. Xen works fine in x86 (32 and 64 bits), but has problems in others and there are limitations on dynamic memory reallocation on default Linux kernels. The disk creation is a bottleneck during the VM creation so we have reduced its impact using caching techniques. Moreover, typical monitoring tools do not work properly with virtualized environments, as a result we have implemented a monitoring tool taking into account the quantity of resources assigned to a

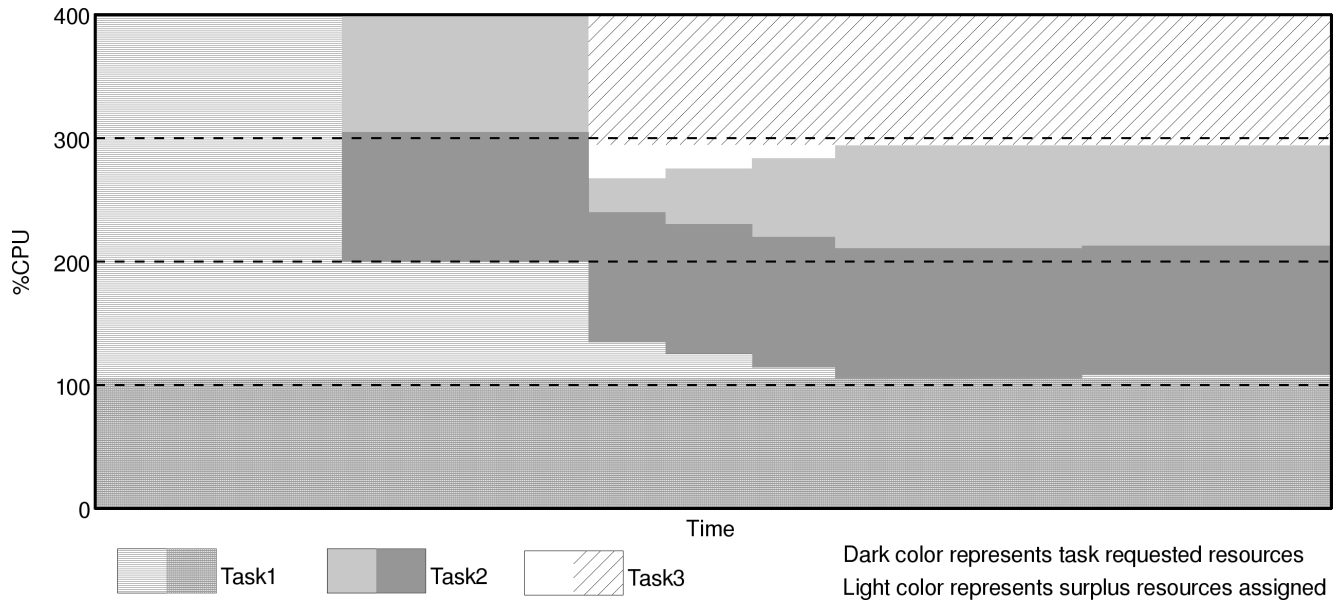


Figure 4. Tasks CPU allocation

VM.

As a short term work, we are going to make more measurements in order to evaluate the overhead introduced by the technologies and other performance analysis. As result of a preliminary analysis, we have found that sometimes the inference process can have a big computational load. It can produce scalability problems, for this reason as a longer term work, we will study the possibility of a fully distributed approach for the scheduler component taking advantage of the autonomy of the agents.

8 Acknowledgments

This work is supported by the Ministry of Science and Technology of Spain and the European Union (FEDER funds) under contracts TIN2004-07739-C02-01 and TIN2007-60625 and the European Commission under FP6 IST contract 034556 (BREIN).

References

- [1] EU BREIN project. <http://www.eu-brein.com>.
- [2] B. Clark, T. Deshane, E. Dow, S. Evanchik, M. Finlayson, J. Herne, and J. N. Matthews. Xen and the art of repeated research. In *Proceedings of the Usenix annual technical conference*, 2004.
- [3] B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, I. Pratt, A. Warfield, P. Barham, and R. Neugebauer. Xen and the Art of Virtualization. In *Proceedings of the ACM Symposium on Operating Systems Principles*, 2003.

- [4] Ganglia Monitoring System. <http://ganglia.sourceforge.net/>.
- [5] C. Goble, I. Kotsiopoulos, O. Corcho, P. Missier, P. Alper, and S. Bechhofer. Overview of S-OGSA: a Reference Architecture for the Semantic Grid. *Journal of Web Semantics*, 2006.
- [6] R. G. L. Cherkasova. Measuring CPU Overhead for I/O Processing in the Xen Virtual Machine Monitor. In *Proc. of USENIX Annual Technical Conference*, 2005.
- [7] EU OntoGrid project. <http://www.ontogrid.net>.
- [8] W. Z. Paolo Missier and P. Wieder. Semantic Support for Meta-Scheduling in Grids. Technical report, 2006.
- [9] OGF Semantic Grid Research Group (SEM-RG). http://www.ogf.org/gf/group_info/view.php?group=semrg.
- [10] B. L. Smith, C. van Aart, M. Wooldridge, S. Paurobally, T. Moyaux, and V. Tamma. An Ontological Framework for Dynamic Coordination. In *Proc. of the Fourth International Semantic Web Conference*, 2005.
- [11] SPARQL query language for RDF. <http://www.w3.org/TR/rdf-sparql-query/>.
- [12] N. Susanta and C. Tzi-Cker. A Survey on Virtualization Technologies. Technical report, 2005.
- [13] EU TrustCoM project. <http://www.eu-trustcom.com>.
- [14] Xen Hypervisor. <http://www.xen.org>.