SLA-Driven Semantically-Enhanced Dynamic Resource Allocator for Virtualized Service Providers

Jorge Ejarque, Marc de Palol, Íñigo Goiri, Ferran Julià, Jordi Guitart, Rosa M. Badia, and Jordi Torres
Barcelona Supercomputing Center and Universitat Politecnica de Catalunya
Jordi Girona 31, 08034 Barcelona, Spain
{jorge.ejarque, marc.depalol, inigo.goiri, ferran.julia, jordi.guitart, rosa.m.badia, jordi.torres}@bsc.es

Abstract

In order to be profitable, service providers must be able to undertake complex management tasks such as provisioning, deployment, execution and adaptation in an autonomic way. This paper introduces a framework, the Semantically-Enhanced Resource Allocator (SERA), aimed to facilitate service provider management, reducing costs and at the same time fulfilling the QoS agreed with the customers. The SERA assigns resources depending on the information given by service providers according to its business goals and on the resource requirements of the tasks. Tasks and resources are semantically described and these descriptions are used to infer the resource assignments. Virtualization is used to provide a full-customized and isolated virtual environment for each task. In addition, the system supports fine-grain dynamic resource distribution among these virtual environments based on SLAs. The required adaptation is implemented using agents, guaranteeing to each task enough resources to meet the agreed performance goals.

1 Introduction and Motivation

BREIN [3] is a European Commission project, with the objective of bringing the e-business concept developed in recent Grid research projects [27], the 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.

Of remarkable importance in the first 12 months of BREIN was the Rapid Prototyping activity, which aimed to provide experimental implementations based on existing technologies. This paper presents one of those prototypes, placed at the Service Provider Level and called Semantically-Enhanced Resource Allocator (SERA).

SERA provides a framework for supporting the execution of medium and long running tasks in a service provider, which facilitates the provider management (thus reducing costs) while fulfilling the Quality of Service (QoS) agreed with the customers. Resources are assigned 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. Semantic descriptions of the tasks and the resources are constructed and used to infer the resource assignments, empowering the inferences and the use of business factors. Each task is provided with a full-customized (i.e. task specific) and isolated virtual environment, granting full control to the task of its execution environment without any risks to the underlying system or the other tasks. These virtual environments are consolidated to achieve a better utilization of the provider resources. In addition, SERA performs fine-grain dynamic resource distribution among these virtual environments based on Service Level Agreements (SLA) in order to adapt to changing resource needs of the tasks.

SERA implements an adaptive behavior by means of agents which guarantees to each application enough resources to meet the agreed QoS goals. Furthermore, it can provide the virtual environments with supplementary resources, since free resources are also distributed among applications depending on their priority and their resource requirements. The system continuously monitors if the SLAs of the applications running in the provider are being fulfilled. If any SLA violation is detected, an adaptation process for requesting more resources to the provider is started.

The rest of the paper is organized as follows: Section 2 describes the overall architecture of the SERA; Section 3 describes how semantics is used to take scheduling decisions; Section 4 describes the virtual machines management and the dynamic resource provisioning performed by the SERA; Section 5 presents our SLA fulfillment approach; Sections 6 and 7 describe the experimental environment and the evaluation. Section 8 presents the related work. Finally, Section 9 presents the conclusions and the future work.
2 Overall Architecture

This section gives an overview of the architecture of the SERA, describing the main components and their interactions. Each component contains an agent and a core. The agent wraps the core functionalities by means of a set of behaviors which basically call methods from this core. The agents are in charge of the communication between components. In addition, their implicit reactivity is used to be aware of the system performance and status variations, and for coordinating the reaction to these variations (e.g. reaction to a SLA violation).

![Figure 1. Architecture of SERA prototype](image)

Figure 1 shows the main components of the SERA, whose functionality is herewith described. The Client Manager (CM) manages the client’s task execution by requesting the required resources and by running jobs. In addition, it makes decisions about what must be done when unexpected events such as SLA violations happen.

The Semantic Scheduler (SeS) allocates resources to each task according to its requirements, its priority and the system status, in such a way that the clients with more priority are favored. Allocation decisions are derived with a rule engine using semantic descriptions of tasks and physical resources. These resource descriptions are automatically generated from the system properties and stored in the Semantic Metadata Repository (SMR) when the machine boots.

The Resource Manager (RM) creates virtual machines (VM) to execute clients’ tasks according to the minimum resource allocation (CPU, memory, disk space...) given by the SeS and the task requirements (e.g. needed software). Once the VM is created, the RM dynamically redistributes the remaining resources among the different tasks depending on the resource usage of each task, its priority and its SLA status. This resource redistribution mechanism allows increasing the allocated resources to a task by reducing the assignment to other tasks that are not using them. Finally, the Application Manager (AM) monitors the resource usage in order to evaluate if an SLA is being violated.

Figure 2 shows the task lifecycle in the SERA and the interaction among the different components. Initially, at boot time, every component stores its semantic description in the SMR. An interaction starts when a task arrives at the system and a CM is created in order to manage its execution. The CM preselects potential nodes for running the task querying the SMR and registers the task description (1). Then, it requests a time slot to the SeS for the task (2). In this stage, the SeS uses the metadata stored in the SMR (3) to infer in which node the task will be executed. At this point the SeS informs the CM whether the task has been successfully scheduled or canceled (4). When the time to execute the task arrives, the SeS contacts with the selected RM and requests the creation of a VM for executing the task (5).

![Figure 2. Task lifecycle](image)

When the RM receives the SeS request, it creates a VM and an AM that monitors the SLA fulfillment for this task (6). Once the VM is created, the SeS is informed (7) and it forwards the message to the CM indicating the access information to that VM. At this point, the CM can submit the task to the newly created VM (8). From this moment, the task is executed in a VM which is being monitored by the AM in order to detect SLA violations (9). If this occurs, the AM requests for more resources to the RM (10), trying to solve the SLA violation locally to the node (11). However, if the SLA violation cannot be solved locally, the AM informs the CM about this situation (12). In this case, the CM should decide between asking the SeS to reschedule the task with higher resource requirements or canceling the task and notifying the violation to the customer.

3 Using Semantics for Scheduling

3.1 Semantic model

The ontology of the SERA is based on the work presented in [23]. It describes agents (as requesters or providers), activities, resources and relations between them to describe how the usage of resources can be coordinated. However, some changes and extensions have been required
to convert this coordination ontology into a suitable ontology for resource allocation which is depicted in Figure 3.

![Figure 3. Ontology for resource allocation](image)

The first required extension was the addition of a set of Grid resources which can be contained in a Host. Currently, they are simple resource subclasses for prototyping purposes, which will be later extended using other resource ontologies such as [9]. We have basically defined a set of HardwareResources, Software which is used in Images and Files, which are stored in Disks. We have also extended the Agent class to support business parameters (such as the customer’s priority property). Additionally, we have defined the ResourceManager and the ClientAgent agent as Provider and Requester agents.

The most important change in the original ontology is the Task class description. The original class requires a single instance of Resource. However, this resource instance is unknown because our resource allocator tries to find the best resources for each client task. Therefore, the requires property in the resource allocation ontology contains a set of TaskRequirements which describes the required resources of a client task. Hardware and SoftwareRequirements are used to look for suitable resources to execute the task while DataRequirements can be used for detecting data dependencies and exploiting data locality getting the host which already contains the required files.

### 3.2 Client Manager (CM)

When a new task is submitted to the SERA, a new CM is created to manage the task execution. The CM agent and their associated tasks are automatically semantically annotated according to our ontology. Once the task and the CM agent have been described, the agent creates a SPARQL semantic query [25], which selects all the machines that match with the hardware requirements of the task from the SMR. The query results are also inserted in the task description and then, registered into the SMR. By performing this step in the CM, we unload the SeS.

After this, the CM sends a scheduling request to the SeS that will later notify if the task has been successfully scheduled. If the answer is positive, the CM remains waiting until a new notification is received notifying that the VM is ready and that the task can be executed in a given machine. This task execution is performed by means of the Globus Toolkit 4 (GT4) [6] deployed in each VM. GT4 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.

If the SeS is not able to find a machine/time slot to execute the task, the CM must decide what to do, considering factors such as the task owner and the task deadline. For instance, it could decide to resubmit the task with different resource requirements. The CM must also react when an SLA violation occurs and it cannot be solved locally. This is explained in Section 5.

### 3.3 Semantic Scheduler (SeS)

The SeS is a proof of concept implementation of decision taking using semantics and reasoning with Horn rules, providing an extensible way to schedule tasks (only changing or adding rules). The scheduling process consists of the following steps:

**Get the required metadata:** All the data involved in the scheduling process, previously annotated by the different components (CM, RM), is maintained and updated in the SMR following the ontology described above. This data, which includes the semantic data describing each host, the already scheduled/running tasks and the clients that are executing tasks in the system (from where the business information will be taken), is fetched from the SMR via Web Services for being used in the inference.

**Inference:** Using the retrieved data and the ontology, the SeS creates a Jena 2 [11] model. Some Jena 2 rules which use a set of auxiliary built-ins (rule extensions written as application code) are also attached to the rule engine and will be fired during the inference process. These rules implement the scheduling policies. For the current prototype we have developed the following rules:

**Rule 1: First Come First Serve.** This rule is fired when a time slot in a machine for a requested task is found using the FCFS algorithm, taking into account the hardware and software requirements.

**Rule 2: Task reallocation.** If Rule 1 cannot find a time slot in a machine, 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.

**Rule 3: Less priority task(s) cancellation.** If Rule 2 is not able to find a solution, then Rule 3 tries to cancel the tasks with less priority than the new one.

**Interpretation of the results:** When the inference process is finished, Jena 2 gives back a deductions graph, which
is compared with the original one by the SeS. Finding differences means that some event has occurred, that is, a task has been scheduled, rescheduled or cancelled. Then, the SeS updates this information in the SMR, notifies task cancellations to their respective CM, and stores the new scheduled or rescheduled tasks in a task queue. Periodically, the SeS inspects this queue checking the execution time of the tasks. When a task execution time is close, it requests the RM for a VM to execute that task. When the VM is ready, the SeS informs the CM.

4 Managing Virtualized Resources: RM

4.1 Management of VM lifecycle

The Resource Manager (RM) is composed by its corresponding agent and core. There is one RM instance per physical machine in the service provider. Once the RM is created and fully started, it waits for requests from the SeS. When a new request arrives, the RM checks if it is possible to create a new VM with specified features and informs the SeS about the success/failure of this operation.

The creation of a new VM requires the following steps: downloading and creating the guest operating system (a Debian Lenny through debootstrap for this prototype), copying extra software needed by the client in an image that will be automatically mounted in the VM, creating home directories and swap space, setting up the whole environment, packing it in an image, and starting the VM. Once the VM has completely started, the guest operating system is booted. After this, the additional software needed by the client needs to be instantiated (if applicable).

From this description, one can derive that this process can have two bottlenecks: the network (for downloading the whole system) and the disk (for copying applications and creating system images, approx. 1GB of data). The network bottleneck has been solved by creating a default image of the system with no settings, and copying it for each new VM. This almost eliminates the downloading time, but contributes to the disk bottleneck. The disk bottleneck has been solved by adding a second caching system that periodically copies the default image and the images with the most common used software to a cache space. Finally, the RM has only to move these images (just an i-node change) to the final location when a new VM is created. Using these caching techniques, the complete creation of a VM has been reduced from up to 40 seconds to an average time of 10 seconds.

4.2 Resource distribution among VMs

The RM is also responsible of distributing the physical resources among the VMs. The goal is to maximize physical resources utilization, while fulfilling the SLAs. In order to accomplish this, the SeS provides the RM with two parameters for each VM, namely the minimum resource requirements of the VM and the initial priority of this VM, which corresponds to the priority for the service provider of the customer executing in this VM (e.g., Gold, Silver, etc.).

For each VM, the RM guarantees that its minimum resource requirements are met during the whole VM lifetime. Surplus resources that are not allocated to any VM are dynamically redistributed among VMs according to their resource utilization and the fulfillment status of the SLAs (as shown in Figure 4). In this way resource wasting is avoided and the applications are provided with better service.

The surplus resources are redistributed among the VMs according to their dynamic priority. This priority initially corresponds to the priority set by the SeS and can be dynamically increased by the AM to apply for more resources if the SLA is violated. Any dynamic priority change induces the RM to recalculate the resource assignment of the VMs according to the following formula (where \( p_i \) is the priority of client \( i \)) and bind the resources to the VMs.

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

5 SLA Fulfillment: AM

Described proposal assumes that the SLA negotiation between the customer and the provider has been carried out previously, being our prototype responsible of guarantying the fulfillment of the agreed SLA by means of adequate resource allocation. According to this, the Application Manager (AM) has two main responsibilities. On one side, it enables the execution of the task into the VM created by the RM explicitly for this task. This is done using the GT4 container deployed in each VM. On the other side, the AM is in charge of monitoring the resources allocated to and used by the VM in order to detect SLA violations. There is one AM instance per application running in the service provider, thus several AM instances can exist per physical machine.

Each application has its own SLA, described in XML using both WS-Agreement and WSLA specifications, as done in Trustcom [27]. The SLA includes two simple metrics: the amount of memory used and a performance metric which intends to compute the real usage of the CPU (see the definition below). The guarantee for this metric en-
Figure 5. SLA enforcement cycle

ensures that the application will have the amount of cycles/sec specified in the SLA whereas it uses all the CPU assigned. The performance metric for the CPU usage is defined as $\frac{\text{Used CPU}}{100} \cdot \text{CPU freq}$.

The SLA enforcement cycle, which is shown in Figure 5, is implemented by the AM agent. Notice that, the needed reactivity to SLA violations can be easily accomplished using agent behaviors. The Direct Measurer component gets the values from a monitoring subsystem, which is implemented using daemons running in the Xen Domain-0. There is one daemon per VM, which obtains all the resource usage information of this VM via XML-RPC calls to the Xen API. The Direct Measurer controls the measurement intervals for each metric in the SLA and ensures the refresh of the measures on correct time. When these values arrive to the Measurer Sched component, it checks if any metric has updated its value. In this case, the Measurer Sched recalculates the top level metric defined in the SLA and compares the result with the agreed value specified in the guarantee terms of the SLA. If the SLA is fulfilled, the Measurer Sched waits till the next iteration, otherwise the SLA violation protocol starts.

The first step in the SLA violation protocol is requesting to the RM for more resources (by increasing the VM dynamic priority). If node has surplus resources, the RM will redistribute them as described in 4.2 and the SLA cycle will start again. If all the physical resources are already allocated, the AM will contact 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 customer’s priority and the task deadline. This is currently not implemented, but possible actions include continuing the execution besides the SLA violation (if this is acceptable for the customer), modifying the resource requirements of the task and ask the SeS to do a rescheduling, or canceling the task and notifying the violation to the customer.

6 Experimental Environment

Main technologies used in the prototype include Ontokit [18], which is a Semantic OGSA implementation, for implementing the SMR; the Jena 2 framework [11] for supporting inferences and the management of the semantic metadata; and Jade [10] as the agent platform. Xen [28] is used as virtualization software.

Our experimental testbed consists on three computers. Two of them are used as RMs, namely a 64-bit architecture with 4 Intel Xeon CPUs at 3.0GHz and 10Gb of RAM memory and a 64-bit architecture with an Intel Xeon CPU at 2.6GHz and 4Gb of RAM memory. Both systems run Xen 3.1. The other computer hosts the SeS, the CMs and the SMR, and it is a Pentium D with two CPUs at 2.6GHz with 2Gb of RAM. All the machines are connected through a Gigabit Ethernet.

Most part of the software is written in Java and runs under a JRE 1.5, except the scripts that manage the creation of the virtual machines, which are written in Bash script, and some libraries used for accessing Xen, which are in C.

7 Evaluation

This section describes a real demonstrator that shows the functionality of our prototype. A total amount of seven tasks are sent to a provider composed of two Resource Managers (RM1 and RM2). Table 1 describes for each task: its type (a Tomcat server or a CPU-consuming HPC application), its requested CPU (i.e. the value indicated in the task descriptor) (REQ), its real CPU consumption (CON), its duration (DUR) and its deadline (if specified in the requirements, DLN). Deadlines are specified assuming that the initial time of the experiment is 00:00. The amount of CPU allocated to each VM is quantified using the typical Linux CPU usage metric (i.e. for a computer with 4 CPUs, the maximum amount of CPU will be 400%). For simplicity, figures focus on the CPU.

<table>
<thead>
<tr>
<th>ID</th>
<th>TASK</th>
<th>REQ</th>
<th>CON</th>
<th>DUR</th>
<th>DLN</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Tomcat server</td>
<td>105%</td>
<td>min¹</td>
<td>20 '</td>
<td>-</td>
</tr>
<tr>
<td>2</td>
<td>HPC application</td>
<td>105%</td>
<td>190%</td>
<td>20 '</td>
<td>-</td>
</tr>
<tr>
<td>3</td>
<td>HPC application</td>
<td>105%</td>
<td>105%</td>
<td>20 '</td>
<td>-</td>
</tr>
<tr>
<td>4</td>
<td>HPC application</td>
<td>100%</td>
<td>50%</td>
<td>20 '</td>
<td>-</td>
</tr>
<tr>
<td>5</td>
<td>HPC application</td>
<td>100%</td>
<td>100%</td>
<td>20</td>
<td>-</td>
</tr>
<tr>
<td>6</td>
<td>HPC application</td>
<td>400%</td>
<td>400%</td>
<td>20 '</td>
<td>00:40</td>
</tr>
<tr>
<td>7</td>
<td>HPC application</td>
<td>100%</td>
<td>100%</td>
<td>20 '</td>
<td>00:40</td>
</tr>
</tbody>
</table>

Table 1. Description of tasks

¹min represents the minimum CPU consumption needed to maintain the Tomcat server running without receiving requests.
The top four plots in Figure 6 show the allocated CPU, the real consumed CPU and the SLA threshold (indicated using a horizontal line) for the first four tasks executed in the test as a function of the time (in seconds). The SLA is being violated. The SERA will guarantee the resources on or beyond the SLA threshold, if the task uses all its allocated resources, otherwise the system may not give more resources than the used by the application. Finally, the fifth plot corresponds to the consumed CPU by the Xen Domain-0 for both RM1 and RM2, and it shows the CPU costs of creating and destroying VMs and of real-locating resources among them.

Task1, which is a Tomcat server requesting 105% of CPU, arrives at the beginning of Zone A in Figure 6. It must be executed on RM1 because RM2 can only execute jobs requesting less than 100% of CPU. Task1 is immediately scheduled and sent to RM1, which creates a VM allocating the whole 400% of CPU to this task (105% requested and 295% as surplus resources). Task2 requests 105% of CPU, and it is also scheduled on RM1. This second task represents a special case: we have forced an error when estimating the minimum amount of resources needed to fulfill the SLA, in such a way that those are not enough to fulfill the SLA. In this case, the task is an HPC application that consumes 190% of CPU, while the task descriptor indicates a request of only 105% of CPU. Task2 is started with 200% of CPU (105% requested and 95% surplus) while Task1 surplus is downgraded to 95% of CPU. This fact can be appreciated in the middle of Zone A in Figure 6.

At the end of Zone A, Task3, another HPC application which requests 105% of CPU and consumes 105%, arrives at the system. Again, the surplus of the host is distributed between the three tasks executing. This results in an assignment of 133% of CPU for each running job. Since Task2 really needs 190% of CPU, an SLA violation arises (the CPU allocated to Task2 is lower than the SLA threshold) and its AM is forced to ask for more resources to the RM1. The steps in the allocated CPU which can be appreciated at Zone B correspond to the evolution of the CPU assignment to each VM until all tasks fulfill their SLAs. After the CPU reallocation, Task2 has an assignment of 190% of CPU and Task1 and Task3 of 105% each.

At the beginning of Zone C, Task4, which requests 100% of CPU and really consumes 50% of CPU, arrives at the system, being scheduled in RM2. From this moment, only a task requesting less than 85% of CPU can be executed in the system. Tasks demanding more resources will be queued in the SeS queues until the resources become available.

Next three tasks (Task5 to Task7) are used to exemplify how the SeS reschedules tasks between different RMs. These tasks arrive at the system at Zone C, but we cannot see any CPU consumption from them in Figure 6 because they are initially enqueued and will be executed in a time-space out of the figure scope. When Task5 arrives, since it
requests more than the available 85% of CPU, it is scheduled into RM1 queue (see Table 2.a). Task6 deadline forces this task to be scheduled for execution at least right after the current running tasks (note that we do not interrupt already running tasks). Since Task6 requests 400% of CPU, it cannot be scheduled into RM2. But RM1 cannot accomplish Task6 deadline unless the SeS reschedules Task5 to RM2 (see Table 2.b). Task7 has also a sharp deadline, and the same reasoning is done. Task5 is rescheduled after Task6, and Task7 is scheduled into RM2 queue (see Table 2.c).

(a) | RM1 | RM2 | (b) | RM1 | RM2 | (c) | RM1 | RM2
---|---|---|---|---|---|---|---|---
Task5 | Task6 | Task5 | Task6 | Task5 | Task7

Table 2. Task rescheduling

Notice that the use of semantics involves some overhead, mainly spent in the queries to the SMR and the inference process, which take together between 200 and 1800 msec [5]. The same occurs with virtualization, since having a fully functional VM for running a task requires around 37 seconds [7]. Although these are significant values, they are acceptable when executing medium and long running tasks (which are the target of the SERA).

8 Related Work

Primary technologies used in our proposal have been independently used in the literature to tackle the resource allocation problem. In particular, SLAs have been used to drive approaches that enable the movement of servers across clusters in order to continuously meet the SLA goals under dynamically varying workloads [2, 15].

Similarly, semantic technologies have been lately of significant interest for enriching Grid resources and services, making them machine understandable and easily interoperable [22, 21]. Traditionally, Grid organizations published their resource properties and application requirements using their own language and meanings, making interoperability more expensive. Recently the use of semantics in Grids has been somehow standardized in the EU FP6 OntoGrid project [18], which has proposed a reference model (i.e. Semantic OGSA [4]) consisting of a set of OGSA compliant services for managing semantic descriptions in Grids.

Semantic models have been also used to propose new resource allocation solutions. For example, [26, 17] present two different ontology-based resource matchmaking algorithms implemented by a set of rules which identify the resources to fulfill the requirements, while in the Phosphorus project [20], semantics is used to automatically select resources. However, the resource matchmaking is only part of the resource allocation and scheduling problem. Our approach uses semantics in the whole resource allocation process, that is, resource matchmaking and allocation of tasks in the selected resources. We also consider not only the task requirements, but also business indicators such as the customer’s priority. Besides, results of previous works about semantic support for scheduling and ontologies for resource access coordination [16, 23] have been partially used to build our resource allocator.

Regarding virtualization, some works use this technology for cost reduction and easier management in service providers. [13, 14] allow the creation of customized virtual environments to their users, and in addition, they provide new job scheduling techniques between nodes by supporting VM pausing and migration. Furthermore, [12] extends these functionalities by adding an efficient shared storage between nodes located in different locations. While the above proposals focus on the global scheduling of VMs between nodes, other works [8, 19, 24] focus on the resource distribution among VMs in a single node.

Our proposal exploits virtualization facilities for both provider management and resource provisioning, joining in a single system the creation of customized virtual execution environments, the global resource allocation among nodes and the SLA-driven dynamic resource redistribution at node level. Other works [1, 29] use a similar point of view, proposing also SLA-driven resource allocation using virtualization, but none of them semantically enhances the resources descriptions as done in our approach.

9 Conclusions and Future Work

This paper describes a working implementation of a framework for facilitating service provider management, which allows reducing costs and at the same time fulfilling the QoS agreed with the customers. Our approach supports fine-grain dynamic resource distribution among customers' applications based on SLAs. It guarantees to each application enough resources to meet the agreed performance goals. The proposed architecture has been prototyped in the scope of the BREIN European project.

In our approach, the resource allocation process has been enhanced using emerging technologies such as semantics, agents and virtualization. The use of semantics improves the reasoning and the addition of business factors. Regarding agents, their behaviors are useful for monitoring and reacting to possible undesired events, such as SLA violations or failures. Using virtualization we can provide specific execution environments to the tasks and grant full control of them without risks to the underlying system or the other tasks. The consolidation of VMs allows better utilization of the provider physical resources and reduces costs. The involved overhead when using semantics and virtualization is
acceptable when executing medium and long running tasks.

We have presented a simple demonstrator showing that the SERA is able to react almost immediately under changing conditions and avoid SLA violations by rescheduling efficiently the resources. We have used a real SLA specification and we have introduced a novel way to use free (unallocated) resources that helps to maintain all the SLAs within their thresholds.

Although current prototype has a pretty good functionality, we are planning some improvements including the implementation of more complex policies based on economic parameters (rewards, penalties), and the migration of tasks when there is an SLA violation and there are not enough resources on the local machine to avoid it.

Acknowledgment

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

References


