Introducing Virtual Execution Environments for Application Lifecycle Management and SLA-Driven Resource Distribution within Service Providers

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

***Abstract*—Resource management is a key challenge that service providers must adequately face in order to ensure their profitability. This paper describes a proof-of-concept framework for facilitating resource management in service providers, which allows reducing costs and at the same time fulfilling the quality of service agreed with the customers. This is accomplished by means of virtualization. Our approach provides application-specific virtual environments and consolidates them in order to achieve a better utilization of the providers resources. In addition, it implements self-adaptive capabilities for dynamically distributing the providers resources among these virtual environments based on Service Level Agreements. The proposed solution has been implemented as a part of the Semantically-Enhanced Resource Allocator prototype developed within the BREIN European project. The evaluation shows that our prototype is able to react in very short time under changing conditions and avoid SLA violations by rescheduling efficiently the resources.**

I. Introduction

The emergence of Cloud computing solutions has attracted many potential consumers, such as enterprises, looking for a financially attractive way to host their services. Service providers rent their resources to the enterprises on demand, which pay for the actual use of provider's resources. In return, the customers are provided with guarantees on resource availability and Quality of Service (QoS), which are typically expressed in the form of a Service Level Agreement (SLA).

In order to be profitable, service providers tend to share their resources among multiple concurrent applications owned by different customers, but at the same time they must guarantee that each application has always enough resources to meet the agreed performance goals. According to this, it would be desirable for the provider to implement a self-adaptive resource management mechanism, which could dynamically manage the provider's resources in the most cost-effective way (e.g. maximizing their utilization), while satisfying the QoS agreed with the customers.

One of the key technologies in the Cloud is virtualization, which has enabled cost reduction and easier resource management in service providers. Virtualization allows the consolidation of applications, multiplexing them onto physical resources while supporting isolation from other applications sharing the same physical resource. Virtualization has other valuable features for service providers. It offers the image of a dedicated and customized machine to each user, decoupling them from the system software of the underlying resource. In addition, virtualization allows agile and fine-grain dynamic resource provisioning by providing a mechanism for carefully controlling how and when the resources are used, and primitives for migrating a running machine from resource to resource if needed.

In this paper, we exploit the features of virtualization in a new approach for facilitating resource management in service providers, which allows reducing costs and at the same time fulfilling the QoS agreed with the customers. Our solution supports the execution of medium and long running tasks in a service provider by providing an application-specific virtual machine (VM) for each of them, granting full control to the application of its execution environment without any risks to the underlying system or the other applications. These VMs are created on demand, according to the application requirements, and then, they are consolidated in the provider's physical resources in order to optimize their utilization.

In addition, our approach supports fine-grain dynamic resource distribution among these VMs based on SLAs. Our system guarantees to each application enough resources to meet the agreed performance goals, and furthermore, it can provide the applications with supplementary resources, since free resources are also distributed among them depending on their priority and resource demand. 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.

With our solution, service providers in the Cloud can take advantage of all their resources by consolidating services. In addition, they can benefit from easier resource management, usage monitoring and fine-grain SLA enforcement, since these tasks are implemented by means of adaptive behaviors (using agents). This enables autonomic service providers that can adapt to changes in the environment conditions without any additional effort to the system administrator.

The components described in this paper are part of the Semantically-Enhanced Resource Allocator (SERA) prototype [1] developed within the BREIN European IST Project [2]. The SERA prototype enables resource allocation depending on the information given by service providers regarding the level of preference (according to business goals) of their customers and on the requirements of their tasks. The allocation process

978-0-7695-3698-9/09 $25.00 © 2009 IEEE
DOI 10.1109/NCA.2009.9

is enhanced by using agents, semantics and virtualization.

The paper is organized as follows: Section 2 presents an overview of the SERA. Section 3 introduces our proposal for managing VMs and dynamically provisioning resources. Section 4 presents our SLA management strategy, including SLA monitoring and SLA enforcement. Sections 5 and 6 describe the experimental environment and the evaluation. Section 7 presents the related work. Finally, Section 8 presents the conclusions of the paper and the future work.

II. SERA 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 reactiveness is used to implement the self-adaptive behavior of the system, that is being aware of the system performance and status variations, and coordinating the reaction to these variations (e.g. reaction to an SLA violation).

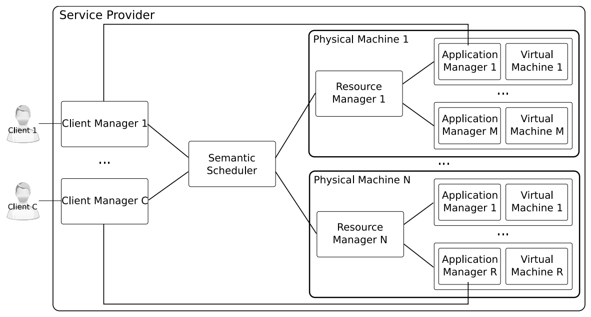

Fig. 1. Architecture of SERA prototype

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.

The Resource Manager (RM) creates VMs 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 (i.e. is it being violated?). 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 of the task and checks its SLA metrics in order to evaluate if the SLA is being violated. If this occurs, it tries to solve the SLA violation locally to the node by requesting more resources to the RM. If the RM cannot provide more resources, the AM will forward the request to the CM.

An interaction starts when a task arrives at the system and a CM is created in order to manage its execution. The CM registers the task description and requests a time slot to the SeS for executing the task. In this stage, the SeS uses the semantic metadata of the system to infer in which node the task will be executed. When the time to execute the task arrives, the SeS contacts with the RM in charge of the node where the task has been allocated and requests the creation of a VM for executing the task.

When the RM receives the SeS's request, it creates a VM and an AM. Once the VM is created, the SeS is informed 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. From this moment, the task is executed in this VM, which is being monitored by the AM in order to detect SLA violations. If this occurs, the AM requests more resources to the RM, trying to solve the SLA violation locally to the node. This request for more resources is performed as many times as needed until the SLA is not violated any more or the RM informs the AM that the requested resource increment is not possible. In the latter case, since the SLA violation cannot be solved locally, the AM informs the CM about this situation, and then, the CM asks the SeS to attempt a global solution for the SLA violation. This can include the modification of the minimum allocated resources of tasks running in the node where the SLA is being violated, or the migration of one of these tasks to another node.

This paper focuses mainly in the functionality and implementation of the RM and the AM components, which are described in the following sections. Additional description of the other components can be found in [1].

III. Resource Manager

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 the specified features and it informs the SeS about the success/failure of this operation. Virtualization is our system is supported by using Xen [3].

A. Management of VMs lifecycle

In our system, each application is executed within a dedicated VM. Any kind of application can be executed, though the system is intended for medium and long running tasks. An application can be composed of a single task or a collection of tasks which are subject to the same SLA. For creating a new VM, the following steps are required on each node: downloading the guest operating system in packaged form (a Debian Lenny through debootstrap for this prototype), creating an image with this base system installed, 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). These phases can be clearly appreciated in Figure 4, which is presented in Section VI.

From this description, one can derive that this process can have two bottlenecks: the network (for downloading the guest system; around 100MB of data) and the disk (for copying extra software needed by the client and creating all the needed images, namely base system, software, home, and swap; nearly 1.6GB of data). The network bottleneck has been solved using a caching system per node that creates a default image of the guest system with no settings when it is downloaded for the first time. Then, this image is copied for each new VM created in that node. This almost eliminates the downloading time (base system is only downloaded once per node and can be reused for each new VM in that node), but contributes to the disk bottleneck. The disk bottleneck has been solved by adding a second caching system per node that periodically copies the default base system image and the images with the most commonly used software to a cache space. When a new VM is created, the RM just needs to move these images (just an i-node change) to the final location. Using both caching systems, the complete creation of a VM has been reduced to an average time of 7 seconds. More details about the VM creation times will be shown in Section VI.

Additionally, our proposal includes a data repository that allows storing the VM images used by each customer. These images can be later reused for creating new VMs. See [4] for more details on this.

Furthermore, when a task finishes or the SeS decides that this task should be rescheduled or canceled, the VM must be destroyed. This includes killing the associated AM, and the redistribution of the resources freed by this VM among the other VMs (see Section III-B).

B. Resource distribution among VMs

The RM is also responsible of distributing the providers 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 usage and the fulfillment status of the SLAs (as shown in Figure

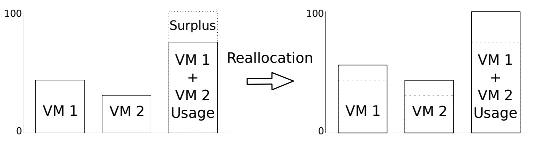

Fig. 2. Surplus resource distribution

2). In this way, the applications can be provided with better service and the provider's resources are fully exploited.

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 being 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.

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

Current implementation is able to manage CPU and memory resources. CPU management is straightforward by using the Xen Scheduler credit policy. This policy allows specifying the maximum amount of CPU assigned to a VM by defining scheduling priorities. For example, in a platform with 4 CPUs (i.e. 400% of CPU capacity) and two VMs, one with a priority of 6 and the other with a priority of 4, the first could take at most the 240% of CPU, while the other could take at most the rest 160% of CPU. The scheduling priorities can be set using the XenStat API.

On the other side, there are some limitations for dynamic memory management using VMs. In Linux systems, the mapping of physical memory is done at boot time. Once the guest system has booted, if the amount of memory allocated to the VM is reduced, the guest system adapts to this reduction automatically. Nevertheless, when assigning to the VM more memory than the initially detected by the guest system, Linux does not make it available to the user. It would be necessary to restart the guest system to make all this memory available. In order to overcome this limitation, the RM creates all the VMs with the maximum amount of memory possible and then it immediately reduces the amount of allocated memory to the value indicated by the SeS.

IV. Application Manager

The Application Manager (AM) has two main responsibilities. On one side, it enables the execution of a task into the VM created by the RM explicitly for this task. This is done by means of a Globus Toolkit 4 (GT4) [5] container 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. We use GT4 just because task management is easier when using GT4 facilities, though our solution could be generalized to dispense with it. In fact, we have an alternative implementation of the AM that uses SSH for task submission. On the other side, the AM is in charge of monitoring the resources *provided to* and *used by* the VM and ensuring the fulfillment of its SLA. There is one AM instance per application running in the service provider, thus several AM instances can exist per physical machine.

A. SLA description

Each application has its own SLA, described in XML using both WS-Agreement [6] and WSLA [7] specifications, as done in TrustCoM [8]. The SLA characterizes the agreed QoS between the customer and the provider using a set of service-level metrics (e.g. throughput, response time, availability, etc.). However, the component in charge of deriving the resource requirements needed to fulfill these service-level metrics has not been implemented within BREIN yet. For this reason, in order to test the viability of our proposal, in this prototype we use a proof-of-concept SLA that directly defines two simple resource-level metrics: the amount of memory used by an application and a performance metric that intends to compute the real CPU usage of an application. This performance metric for the CPU usage is defined as $\frac{UsedCPU}{100} \cdot CPUfrequency$. An application will have the amount of cycles/sec specified in the SLA whereas it uses all the assigned CPU. If the application is not using all the resources then the SLA will be always fulfilled. The SLA will be violated only when the application is using all the assigned resources and these resources are not enough. Notice that this metric assumes that the pool of machines is homogeneous (as occurs in our testbed). We believe that this is acceptable in a proof-of-concept prototype.

The evaluation of SLA metrics can be highly influenced by the inaccuracy when measuring and the variability of the measures. When this occurs, the evaluation of the SLA metrics can depend more on how and when they are measured than in the metrics themselves. For this reason, when defining SLA metrics, it is desirable that they can be defined through average values. This can be accomplished using the capabilities of the above-mentioned SLA specifications, which allow defining the window size of the average and the interval between two consecutive measures. We have defined a window size of 10, and an interval of 2 seconds.

B. SLA monitoring

The AM includes a subsystem for monitoring the metrics defined in the SLA, which receives the requests of the *Direct Measurer* component (see next section) in order to get the values for each metric in the specified measurement intervals. The monitoring subsystem is implemented using daemons running in the Xen Domain-0. There is one daemon per VM, which obtains all the information referent to this VM (such as CPU, memory, disk and network usage) via XML-RPC calls to the Xen API [9]. These daemons cannot run inside the VMs for two reasons. Firstly, doing this would consume part of the assigned resources to the task execution, charging to the customer the cost (in CPU and memory) of this monitoring. Secondly, the monitoring system cannot take real measures inside the VM: this can be only accomplished by measuring from the Xen Domain-0.

C. SLA enforcement

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. The agreed SLA is attached to the task execution request that arrives at the RM. This SLA is assigned to the AM that monitors the VM which is going to execute the task.

The agent within this AM executes the SLA enforcement cycle shown in Figure 3. Notice that, the needed reactiveness to SLA violations can be easily accomplished using agent behaviors. The cycle is executed every second. This forces the interval between two consecutive measures for each metric in the SLA (see Section IV-A) to be at most 1 second.

The *Direct Measurer* component gets the values from the monitoring subsystem described in the previous section, controlling the measurement intervals for each metric in the SLA and ensuring the refresh of the measures on correct time. When the new values arrive at 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 it compares the result with the agreed value specified in the SLA. If the SLA is fulfilled, the *Measurer Sched* waits until the next iteration, otherwise the SLA violation protocol starts.

The first step in the SLA violation protocol is requesting more resources to the RM (by increasing the VM dynamic priority). If the node has surplus resources that have been distributed among the VMs running in that node, the RM will redistribute them considering the new value of the dynamic priority of the VM which is violating its SLA (as described in Section III-B) and the SLA cycle will start again. If all the physical resources are already allocated, the AM will contact the CM to communicate the SLA violation. When the CM receives this notification, it asks the SeS to attempt a global solution for the SLA violation taking into account the customer's priority and the task deadline. This is out of the scope of this paper, and part of our current work, but possible actions include the modification of the minimum allocated resources of tasks running in the node where the SLA is being violated, the migration of one of these tasks to another node or the prosecution of the execution besides the SLA violation (if this is acceptable for the customer and the provider).

Notice that, since the RM always guarantees the minimum amount of resources specified by the SeS when it requests the creation of a VM, the SLA can only be violated when this minimum amount of resources are not enough to fulfill the SLA. This can occur when running an application with variable resource requirements over time (e.g. a web server), which receives an unexpected workload, or as a result of an error in the SeS inference process for estimating the

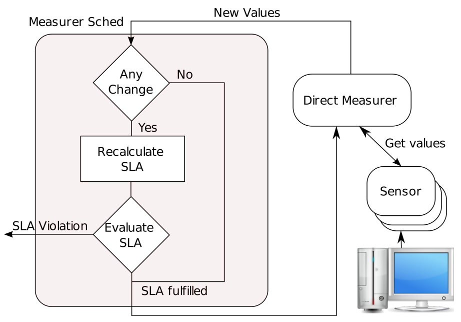

Fig. 3. SLA enforcement cycle

TABLE I
VM CREATION TIMES

Action	No cache	1 level	2 level
Download base system	88.1	-	-
Create base system image	68.4	-	-
Copy base system image (cached)	-	45.2	2.3
Copy software image (cached)	13.9		0.0
Create home & swap	13.7		0.0
Load image	4.4		
Total time for running an image	184.6	77.2	6.7

resources. The latter has been assumed in Section VI to test the functionality of the SERA.

Using the adaptation mechanism described in this section, the system is able to manage itself avoiding as much as possible the SLA violations. Of course, more intelligence could be applied in this part. For example, we are planning to incorporate economic algorithms taking in account the penalizations for violating the SLA, in order to get the best configuration in terms of profit.

V. Experimental Environment

As commented in Section II, RM and AM are part of a bigger system that enables semantic resource allocation in a virtualized environment. In order to evaluate our proposal, the whole system needs to be executed, although results presented in this paper concentrate mainly in the described components. The main technologies used in SERA include Ontokit [10], which is a Semantic OGSA implementation, for storing the semantic descriptions of the tasks and the resources; the Jena 2 framework [11] for supporting inferences and the management of the semantic metadata; and Jade [12] as the agent platform. Xen [3] is used as virtualization software.

Our experimental testbed consists of two machines. The first one is a Pentium D with two CPUs at 3.2GHz with 2GB of RAM that runs the SeS and the CMs. The second machine is a 64-bit architecture with 4 Intel Xeon CPUs at 3.0GHz and 10GB of RAM memory. It runs Xen 3.1 and a RM executes in the Domain-0. These machines are connected through a Gigabit Ethernet.

VI. Experimental Evaluation

This section presents the evaluation of our proposal, which includes the quantification of the time needed to create a VM and a proof-of-concept experiment for demonstrating the functionality of the whole SERA prototype, but focusing on the SLA-driven dynamic resource distribution. Used applications simulate scientific applications with high CPU consumption.

A. VM creation performance

This section provides some indicative measurements about the time needed to create a VM and make it usable for a customer's application, demonstrating the benefit of using our two cache systems to reduce the VM creation time compared with the default approach. These measurements are summarized in Table I.

As described in Section III, if caching systems are not used, we must download the guest system packages (around 100MB of data), which takes 88 seconds, and create a default base system image by installing these packages, which takes 68.4 seconds. When using the first caching system, this is done once per node and the image can be reused for creating each new VM in that node just by copying it. This copy takes 45 seconds. In both cases, the creation of the VM image requires also creating the software and the home & swap images, which takes 13.9 and 13.7 respectively. The second caching system pre-copies all these images (base system, software image, home and swap). This allows creating a full VM image in only 2.3 seconds. Notice that, once the image is ready, it must be loaded, which needs around 4.4 seconds. According to this, the total time needed in our system to have a full configured VM started when taking advantage of the two caching mechanisms is less than 7 seconds.

Nevertheless, the above VM creation time does not include the time that is needed by the guest operating system to boot and become available to the user. In addition, the time needed for instantiating the installed software must be also considered. All these times can be appreciated in Figure 4, which shows the execution of a short task (its run time is 75 seconds) just for exposing in the same timescale all the phases in the whole VM lifetime (from the VM creation to the VM destruction). The figure includes the CPU usage of the VM and the Xen Domain-0. In this figure (and in the rest of figures showing CPU usage measurements), the amount of allocated CPU to the 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%).

As shown in Figure 4, during phase A, the Xen Domain-0 creates the VM. This spends almost one CPU. During phase B, the guest operating system is booted (first peak in the CPU usage graph) and then GT4 is started, which includes certificates creation and deployment of the container (second peak in the CPU usage graph). At this point, the customer's task can be submitted, executing during phase C. Finally, during phase D, the Xen Domain-0 destroys the VM. Notice that the CPU consumption of the Xen Domain-0 is only noticeable during the creation and destruction of the VM.

The results in this figure confirm that the creation of a VM takes around 7 seconds, while booting the guest system and

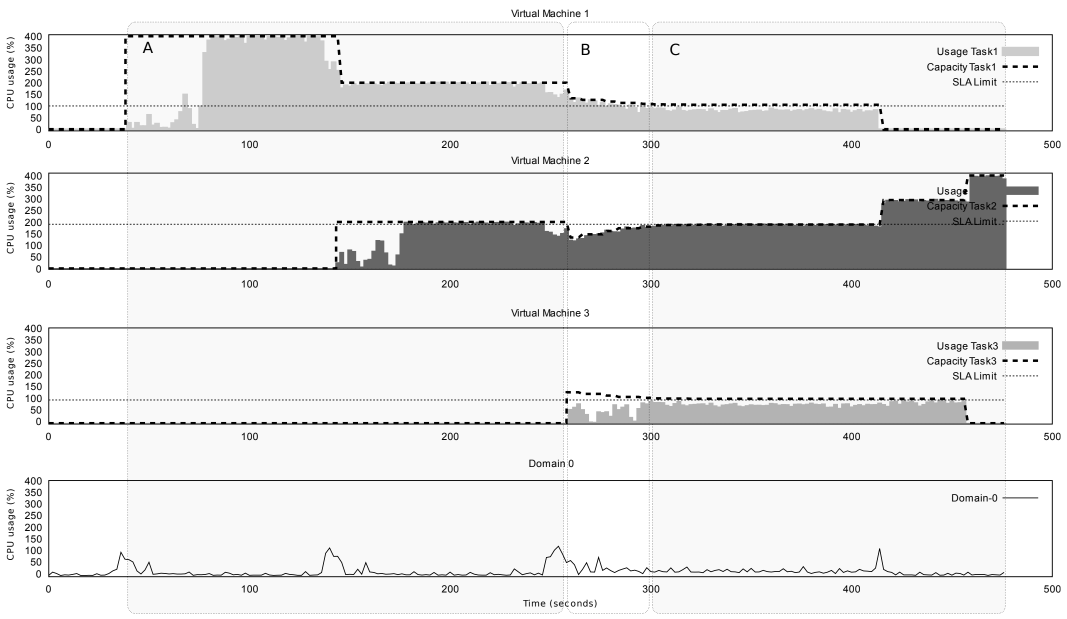

Fig. 5. CPU allocation and consumption

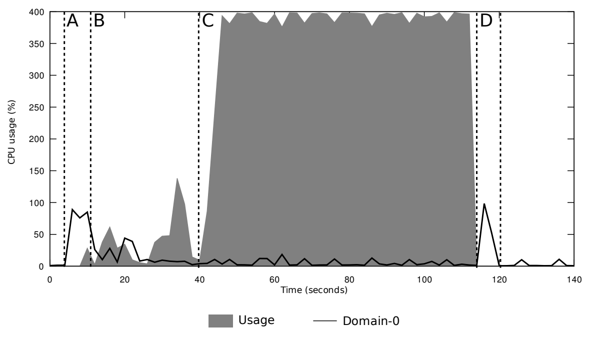

Fig. 4. VM lifecycle

starting GT4 take around 30 seconds. According to this, the full creation of the VM takes around 37 seconds (from the moment that the RM receives the request until the moment when the VM is fully functional and the customer's task can be executed). In addition, VM destruction takes 6 seconds. Notice that since our system is intended for the execution of medium and long running tasks (with running times ranging from minutes to hours or even days), the overhead incurred by using virtualization (mainly in the creation and destruction of the VM) is not significant.

B. *SLA-driven resource distribution*

This experiment demonstrates how resources are dynamically reallocated among applications and how the system is able to detect and resolve an SLA violation from one of the applications and reallocate resources (Memory and CPU) until all the SLAs are fulfilled. The experiment consists of running a total amount of three tasks with different resource requirements and different SLAs within three different VMs located in the same node. Table II describes for each task: its requested CPU (i.e. the value provided by the SeS) (*REQ*), and the agreed CPU metric in the SLA (*SLA*). For simplicity, all the measures and figures are referred only to CPU, but analogous results could be obtained with Memory.

	Requested CPU	Agreed CPU in SLA
Task1	100	100
Task2	100	190
Task3	100	100

TABLE II
DESCRIPTION OF TASKS

Figure 5 displays the execution of the three tasks executed in the test. The first three plots show the allocated CPU for this particular VM, the really consumed CPU by the VM in each moment and a horizontal line that indicates the SLA threshold. If at any moment the consumed CPU is the same as the allocated CPU and this value is lower than the SLA line, it means that the SLA is being violated. Notice that, SLA violations occur only if the task is using all the allocated resources, and despite this, the SLA threshold is not respected. Finally, the fourth plot corresponds to the consumed CPU by the Xen Domain-0, and it shows the CPU costs of creating and destroying VMs and of reallocating resources among them. In this figure, three different situations can be distinguished:

Zone A. This is the initial part of the experiment. Task1, which requests 100% of CPU (see Table II), arrives at the system. Since this is the only VM running in the node, the RM gives the whole machine (400% of CPU = 100% requested + 300% surplus) to it. Task2, which requests 100% of CPU, is sent to the system. Task2 represents a special case: we assume that there has been an error when estimating the minimum amount of resources needed by this task, so that they are not enough to fulfill its SLA. In this case, the SLA needs 190% of CPU to be fulfilled, but the CPU requested by the SeS for this VM is only 100%. At this point, the total amount of CPU needed to satisfy the requests of the two tasks is 200% (100% requested by Task1 and 100% by Task2). The remaining 200% of CPU is distributed among the two tasks as described in Section III-B (100% for each). At this moment, no SLA is being violated, as both tasks have 200% of CPU.

Zone B. In this zone, Task3, which requests 100% of CPU, is started, so the total amount of CPU needed to satisfy the requests of the three tasks is now 300% (100% for each one). In this situation, the remaining 100% is also equally shared among all the VMs, and, as we can see at the beginning of Zone B, all the tasks receive 133% of CPU. Having only 133% of CPU, Task2 violates its SLA (notice that at the center of Zone B, its assigned CPU is under the SLA threshold (i.e. 190% of CPU)). This starts the SLA violation protocol, which reallocates the CPU progressively until all the SLAs are fulfilled. We can see how the surplus CPU assigned to Task1 and Task3 is progressively moved to Task2 until its allocated CPU is over the SLA threshold.

Zone C. This is the final zone, where all the SLAs are again within their thresholds. Notice that when a task finalizes its execution, its allocated resources are freed and then redistributed among the other tasks. Obviously, if a fourth application arrived at this node and allocated all the surplus resources, the SLA for the Task2 would be violated again, and in this case, there would not be enough free resources to solve this violation locally. In this situation, the AM would communicate this situation to the CM, which would ask the SeS to attempt a global solution for the SLA violation (e.g. modify the minimum allocated resources of tasks running in the node, migrate of one them to another node, etc.).

VII. Related Work

SLA-driven resource management is still an open topic in the Cloud. In fact, current Cloud providers use SLAs that only support very simple metrics based on resource availability [13], [14]. Nevertheless, some work has been carried out for traditional eBusiness environments. For example, in [15], the authors combine the use of analytic predictive multiclass queuing network models and combinatorial search techniques to design a controller for determining the required number of servers for each application environment in order to continuously meet the SLA goals under a dynamically varying workload. Another example is Oceano [16], which is a SLA-driven prototype for server farms, which enables the dynamic moving of servers across clusters depending on the customer changing needs. The addition and removal of servers from clusters is triggered by SLA violations. Also in [17], which presents a middleware for J2EE clusters that optimizes the resource usage to allow application servers fulfilling their SLA without incurring in resource over-provisioning costs. This resource allocation is done adding or removing nodes from the cluster.

Lately, some works have exploited virtualization capabilities for building their solutions. On one hand, virtualization has been used to facilitate system administration and provide the users with dedicated and customized virtual working environments, making more comfortable their work. For example, VMShop [18] is a virtual management system which provides application execution environments for Grid Computing. It uses VMPlant to provide automated configuration to meet application needs. VMPlant also allows the creation of flexible VMs that can be efficiently deployed (by implementing a caching-based deployment) across distributed Grid resources. In addition to this, it is typical that these virtual environments can be scheduled among different nodes by using virtualization features such as pausing and migration, as occurs in Globus Virtual Workspace [19] and SoftUDC [20]. Additionally, the latter adds an efficient shared storage between nodes located in different locations. This project provides the capability of sharing resources of different organizations and solves problems such as sharing data between separated clusters.

On the other hand, while the above proposals deal only with the global scheduling of VMs between nodes, other works have also used virtualization to enable fine-grain dynamic resource distribution among VMs in a single node. For instance, [21] develops an adaptive and dynamic resource flowing manager among VMs, which uses dynamic priorities for adjusting resource assignation between VMs over a single server for optimizing global machine performance. [22] introduces an adaptive resource control system (implemented using classical control theory) that dynamically adjusts the resource shares to VMs, which contain individual components of complex, multi-tier enterprise applications in a shared hosting environment, in order to meet application-level QoS goals. [23] takes advantage of virtualization features to collocate heterogeneous workloads on any server machine, thus reducing the granularity of resource allocation. Finally, [24] has developed a new communication-aware CPU scheduling algorithm that improves the performance of a default Xen monitor by enabling the underlying scheduler being aware about the behavior of hosted applications.

Our work proposes a more general and extensive solution for managing service providers by joining in a single framework the creation of application-specific VMs on demand, global resource allocation among nodes, and SLA-driven dynamic resource redistribution at node level (based on the redistribution of surplus resources). Some other works combine some of these functionalities, albeit none of them provides all our facilities. In particular, [25] proposes a dynamic capacity management framework for virtualized hosting services, which is based on an optimization model that links a cost model based on SLA contracts with an analytical queuing-based performance model. However, this work does not support either the creation of VMs on demand or the two-level resource allocation. In addition, the evaluation does not use a working implementation of the proposed system, but a discrete event simulation. Similarly, [26] presents a two-level autonomic resource management system for virtualized data centers that enables automatic and adaptive resource provisioning in accordance with SLAs specifying dynamic tradeoffs of service quality and cost. A novelty of this approach is the use of fuzzy logic to characterize the relationship between application workload and resource demand. However, this work does not support the creation of VMs on demand either.

VIII. Conclusions and Future Work

This paper has described a working prototype of a framework for facilitating resource management in service providers, which is part of the Semantically-Enhanced Resource Allocator developed within the BREIN European project. Described solution exploits well-known features of virtualization for providing application-specific virtual environments, granting in this way full control to the applications of their execution environment without any risks to the underlying system or the other applications. These virtual environments are created on demand and consolidated in the provider's physical resources, allowing him to optimize resources usage and reduce costs. In addition, our approach supports fine-grain dynamic resource distribution among these virtual environments based on SLAs (encoded using a real SLA specification). The system implements a self-adaptive behavior: each application receives enough resources to meet the agreed QoS, and free resources can be dynamically redistributed among applications when SLA violations are detected.

We have presented experimental results demonstrating the effectiveness of our approach. These experiments show that application-specific VMs can be created in around 7 seconds and be ready to be used in around 37 seconds, which is a reasonable period of time when executing medium and long running tasks (which are the target of our work). In addition, the evaluation demonstrates that our system is able to adapt the resource allocations in very short time under changing conditions while fulfilling the agreed performance metrics and solve SLA violations by rescheduling efficiently the resources.

Although current prototype has a pretty good functionality, we have started the work for translating high level SLA metrics into resource requirements. In addition, we are planning to add more complex policies to the RM based on economic parameters (e.g. rewards, penalties), and to consider the Xen Domain-0 CPU usage in the resource allocation decisions.

Acknowledgment

This work is supported by the Ministry of Science and Technology of Spain and the European Union (FEDER funds) under contract TIN2007-60625, by the Generalitat de Catalunya under grant 2009FI_B 00249, and the European Commission under FP6 IST contract 034556 (BREIN).

References

[1] J. Ejarque, M. de Palol, I. Goiri, F. Julia, J. Guitart, R. Badia, and J. Torres, "SLA-Driven Semantically-Enhanced Dynamic Resource Allocator for Virtualized Service Providers," in *4th IEEE International Conference on e-Science, December 7–12*, Indianapolis, USA, 2008, pp. 8–15.

[2] "EU BREIN project," http://www.eu-brein.com.

[3] "Xen Hypervisor," http://www.xen.org.

[4] I. Goiri, F. Julia, and J. Guitart, "Efficient Data Management Support for Virtualized Service Providers," in *17th Euromicro Conference on Parallel, Distributed and Network-based Processing (PDP'09)*, Weimar, Germany, February 18–20, 2009.

[5] "Globus Toolkit," http://globus.org/toolkit/.

[6] GRAAP Working Group, "Web Services Agreement Specification (WS-Agreement), Version 2005/09, Global Grid Forum," Tech. Rep., 2005.

[7] H. Ludwig, A. Keller, A. Dan, R. King, and R. Franck, "Web Service Level Agreement (WSLA) Language Specification, Version 1.0, IBM Corporation," Tech. Rep., 2003.

[8] "EU TrustCoM project," http://www.eu-trustcom.com.

[9] E. Mellor, R. Sharp, and D. Scott, "Xen Management API," revision 1.0.6. July 2008.

[10] "EU OntoGrid project," http://www.ontogrid.net.

[11] "Jena Semantic Web Framework," http://jena.sourceforge.net/.

[12] "Java Agent DEvelopment Framework," http://jade.tilab.com/.

[13] "Amazon EC2 Service Level Agreement," http://aws.amazon.com/ec2-sla/.

[14] "GoGrid Service Level Agreement," http://www.gogrid.com/legal/sla.php.

[15] M. Bennani and D. Menasce, "Resource Allocation for Autonomic Data Centers using Analytic Performance Models," in *2nd International Conference on Autonomic Computing (ICAC'05), Seattle, USA*, June 13–16 2005, pp. 229–240.

[16] K. Appleby, S. Fakhouri, L. Fong, G. Goldszmidt, S. Krishnakumar, D. Pazel, J. Pershing, and B. Rochwerger, "Oceano - SLA-based Management of a Computing Utility," in *Symposium on Integrated Network Management (IM'01), Seattle, WA, USA*, May 14–18 2001, pp. 855–868.

[17] G. Lodi, F. Panzieri, D. Rossi, and E. Turrini, "SLA-Driven Clustering of QoS-Aware Application Servers," *IEEE Transactions on Software Engineering*, vol. 33, no. 3, pp. 186–197, 2007.

[18] I. Krsul, A. Ganguly, J. Zhang, J. Fortes, and R. Figueiredo, "VMPlants: Providing and Managing Virtual Machine Execution Environments for Grid Computing," in *2004 ACM/IEEE conference on Supercomputing (SC'04)*. Washington, DC, USA: IEEE Computer Society, 2004, p. 7.

[19] K. Keahey, I. Foster, T. Freeman, X. Zhang, and D. Galron, "Virtual Workspaces in the Grid," in *11th Europar Conference, Lisbon, Portugal*, September 2005, pp. 421–431.

[20] M. Kallahalla, M. Uysal, R. Swaminathan, D. Lowell, M. Wray, T. Christian, N. Edwards, C. Dalton, and F. Gittler, "SoftUDC: a Software-based Data Center for Utility Computing," *Computer*, vol. 37, no. 11, pp. 38–46, 2004.

[21] Y. Song, Y. Sun, H. Wang, and X. Song, "An Adaptive Resource Flowing Scheme amongst VMs in a VM-Based Utility Computing," in *7th IEEE International Conference on Computer and Information Technology*, Fukushima, Japan, October 16–19 2007, pp. 1053–1058.

[22] P. Padala, K. Shin, X. Zhu, M. Uysal, Z. Wang, S. Singhal, A. Merchant, and K. Salem, "Adaptive Control of Virtualized Resources in Utility Computing Environments," *ACM SIGOPS Operating Systems Review*, vol. 41, no. 3, pp. 289–302, 2007.

[23] M. Steinder, I. Whalley, D. Carrera, I. Gaweda, and D. Chess, "Server Virtualization in Autonomic Management of Heterogeneous Workloads," in *10th International Symposium on Integrated Network Management (IM'07), Munich, Germany*, May 21–25 2007, pp. 139–148.

[24] S. Govindan, A. Nath, A. Das, B. Urgaonkar, and A. Sivasubramaniam, "Xen and Co.: Communication-aware CPU Scheduling for Consolidated Xen-based Hosting Platforms," in *3rd International ACM SIGPLAN/SIGOPS Conference on Virtual Execution Environments (VEE'07), San Diego, California, USA*, June 13–15 2007, pp. 126–136.

[25] B. Abrahao, V. Almeida, J. Almeida, A. Zhang, D. Beyer, and F. Safai, "Self-Adaptive SLA-Driven Capacity Management for Internet Services," in *10th IEEE/IFIP Network Operations and Management Symposium (NOMS 2006), Vancouver, Canada*, April 3–7 2006, pp. 557–568.

[26] J. Xu, M. Zhao, J. Fortes, R. Carpenter, and M. Yousif, "On the Use of Fuzzy Modeling in Virtualized Data Center Management," in *4th International Conference on Autonomic Computing (ICAC'07), Jacksonville, USA*, June 11–15 2007.