Resource management in software as a service using the knapsack problem model

Fotis Aisopos*, Konstantinos Tserpes, Theodora Varvarigou

Department of Electrical and Computer Engineering, National Technical University of Athens, 9, Heroon Polytechniou Str, 15773 Athens, Greece

ARTICLE INFO

Article history:
Received 14 January 2011
Accepted 6 December 2011

Keywords:
Software as a Service
Resource allocation
Service Level Agreement
Quality of Service
Fractional Knapsack problem
Service oriented architecture

ABSTRACT

This paper proposes a resource allocation model for “Software as a Service” systems that maximizes the service provider’s revenues and the resource utilization under a heavy load. Employing the elasticity of virtualized infrastructures, the proposed model dictates that system resources must be fully exploited by incoming jobs, even if they do not satisfy their requirements completely. This yields a higher Service Level Agreement violation probability, which is mitigated by the assignment of more resources when these become available. The problem is deduced to the Fractional Knapsack problem and the heuristic solution is implemented in the frame of a SOA environment.

© 2011 Elsevier B.V. All rights reserved.

1. Introduction

A distributed environment is a system comprised of a set of interconnected resources, with a virtualization layer applied on top of them for interoperability and scalability, delivering a single point of interaction to the customer. A feverish discussion is taking place in the research community about the various distributed computing environments utilities, especially after the emergence of Cloud Computing as a business solution on the web, when Grid computing had only started addressing the same space adequately. Cloud Computing enables the distinction between the application development framework and the underlying resources. It is therefore possible to provide to the end user a platform for application development and/or deployment, an infrastructure or even the application itself, all through web services and usable front ends. The fact that computing or storage resources are virtualized as web resources broadens the customer target group making Cloud services available to various stakeholders. Grid computing on the other hand seems to be addressing the needs of larger organizations. The Grid services’ consumer agrees to use Grids without having much control over the application, which is solely provided and known to a service provider and/or the application developer.

Although these prominent distributed system paradigms present differences in the way they propagate the control of their resources to the service customers and the level of usability that each one exposes towards the customers, the principle remains the same: they are both service provisioning systems which, when it comes to resource management, they both attempt to abstract/virtualize the underlying resources providing a seamless way to access and manage them. Thus, solutions such as Software as a Service (SaaS) are basically common approaches in distributed computing.

Generally, existing distributed SaaS environments support the provision of dynamically scalable and virtualized resources as a service over the Internet. In such systems, the infrastructure resources can be provided externally as a “pool” of resources, like a seamless, single infrastructure with aggregated capabilities. Service customers run various jobs by invoking the respective services with a set of specific requirements for a certain tariff. The resources assigned to each job are elastic, i.e. the provider can dynamically assign the amount of memory, CPU and disk space to a specific job and therefore their performance and capabilities can vary based on the set up. Taking advantage of this property (elasticity), providers are able to maintain high resource utilization while trying to deliver services at the requested quality. An important issue in this supply chain is the establishment of appropriate performance measures, ensuring the supply chain effectiveness and efficiency. These can be categorized as either qualitative (e.g. customer satisfaction with the service received) or quantitative (e.g. objectives depended directly on cost or profit defined in the service tariff) (Beamon, 1998).

Service quality and performance is evaluated in a form of electronic contract, called Service Level Agreement (SLA). SLA

is a contracting tool keyed to a client’s service performance expectations, for identifying the responsibilities of both the customer and provider (Manthou et al., 2004), by defining the Quality of Service (QoS) level promised to be delivered, using quantifying metrics. These metrics specify – explicitly or implicitly – the resources that are to be allocated by the provider to the consumer. Among others, SLAs define the period that this agreement will be active, the cost for the resources as well as clauses that safeguard both the sides from potential SLA term violations (e.g. execution deadline). The latter is usually expressed into monetary compensations, thus, their avoidance is a priority for providers that wish to preserve a good reputation.

With a few exceptions such as Gallizzo et al. (2009), SLAs are long-term and resource-oriented (Vs job-oriented) which means that the provider guarantees the allocation of a certain amount of resources to the consumer who in turn can use them at anytime and any extend during the contract’s lifetime. This implies, however, that it is not necessary for all resources to be reserved on behalf of the consumer during the whole SLA lifetime. This observation is critical because providers tend to employ inflexible allocation schemes such as resource reservation in fear of conducing SLA violations. However, at any given instance, if the jobs submitted by customers demand in total more resources than those available based on their SLAs, the provider comes across a problem that needs to be solved. If the resource pool does not suffice to cover the demand, providers resort to outsourcing jobs or grant extra resources, given that they are keeping valuable resources as a backup. Although this is not a significant problem for large organizations like Amazon (Amazon Web Services LLC, 2010) with practically unlimited resources available, for Small Medium Enterprises (SMEs) this means leasing extra resources or simply denying jobs (violating the SLA). All these alternatives cost money and indicate a lack of flexibility when it comes to assess the risk of undertaking a job using uncommitted resources, even if they do not suffice to cover demanded quality, at the particular time.

The implementation of a mechanism to effectively allocate resources on run-time comes under a set of NP-hard optimization problems with numerous parameters being required to be counted in. This paper presents a solution for maintaining the maximum resource utilization at the cost of risking potential SLA violations, on the pending jobs that will yield the smaller profit for the provider. It does however make two basic assumptions that may affect the feasibility of the proposal: (a) all submitted jobs are preemptive, i.e. they can be interrupted and resumed at a later stage; (b) the pool of resources is homogeneous (or treated as such).

The proposed solution considers the resource infrastructure as a pool of elastic resources using any technology for resource virtualization. Therefore the resources are divided in small chunks and allocated for incoming jobs. The amount of given resources can vary based on the workload generated by the job requirements. The term “workload” refers to the amount of resources that need to be allocated in order to fully satisfy the QoS requirements of a customer when he submits a job. A more specific term could be the “demanded workload”, which is calculated by the provider when mapping high level requirements to low level parameters. It is thus depended on the customer requirements but also on time, because the demanded workload is reduced when part of the job has been executed and can be increased when a job remains in the queue for execution for a long time. The problem now is to maximize the resource utilization while at the same time maximizing the profit. For solving this we resort to a well-known problem with which our own presents analogies: a variant of the knapsack problem: “Given a set of items (jobs to be submitted), each with a weight (workload) and a value (profit), determine the number of each item to include in a collection (set of jobs to be executed/resources to be committed) so that the total weight (total workload) is less than or equal to a given limit (infrastructure limitations) and the total value (total profit) is as large as possible”.

Given the above analogy, we investigate the possible solutions and formulate our allocation as a knapsack problem. In order to conclude to the most well-fitted solution, we also model the profit from job execution. Profit in the examined case, is a multi-parametric function, depended on the probability of an SLA violation given the amount of committed resources, the compensation cost and the infrastructure operating cost yielding from a job execution.

To evaluate the proposed solution, we implement a resource allocation scheme at the level of a distributed system’s meta-scheduler. The implementation approach takes into consideration the cost of a job with specific requirements, the cost of an SLA violation and the probability of an SLA violation, given the job’s workload in order to calculate the profit at any time. In the evaluation phase, various pricing strategies are employed so as to test the model against the potential profit it yields.

The document is organized as follows: Section 2 presents all the related work in the specific area. Section 3 describes our resource allocation model deducting it to the Fractional Knapsack problem (the variant of knapsack) and presenting a heuristic greedy solution for the current multi-dimensional problem. Section 4 describes the architecture of the system that the evaluation of the provided solution took place, while Section 5 presents the experiment performed and analyses the results. Finally, Section 6 presents the conclusions of the current work.

2. Related work

So far, there has been a great deal of research on resource allocation and notable progress in maximizing resource utilization in systems as the ones described above.

An implementation of a distributed, market-based resource allocation system was provided by Lai et al. (2004), where distributed clusters like the Grid and PlanetLab were studied. The purpose of the platform developed (Tycoon) was to allocate computing resources like CPU cycles, memory, network bandwidth, etc. to users in an economically efficient way. Tycoon system introduced three new allocation components, while using Linux VServer for virtualization, focusing on the maximization of the revenues of the specific system. Another business-focused resource managing model for service oriented architectures (SOA) was introduced by Pueschel et al. (2009). This work involves Cloud Computing, proposing decision support mechanisms for the Cloud provider, in order to get a revenue optimization in the respective business model. Resource management for Clouds was also studied by Nguyen Van et al. (2009), who presented an autonomic virtual resource management mechanism for service hosting platforms. Zhao and Sakellariou (2007) studied advance resource reservation in Grids, while Chase et al. (2003) presented new mechanisms for dynamic resource management in a cluster manager, allocating servers from a common pool to multiple virtual clusters.

The papers presented above tackle the resource allocation problem focusing on the available resources and the demands of specific applications. Our approach involves both the low-level parameters of the reservation process, as well as the high-level Quality of Service provisioning details, which are a result of the agreement between the service provider and the customer. As mentioned in Section 1, these details are usually encompassed in the Service Level Agreement in the form of contractual terms and define the level of the provided quality. Job Scheduling based on
SLAs was also studied by Yarmolenko et al. (2005), but in terms of negotiation, as well as “A Metascheduler for the Grid” by Vadhiviar and Dongarra (2002), presenting a high-level scheduling mechanism in the context of GridADS (Berman et al., 2001). Another meta-scheduler is presented by Liu et al. (2009), extending a Grid design to achieve an adaptive usage of Cloud resources, without, however, taking into consideration the SLA with the service provider and the Quality of Service parameters required. QoS-driven resource management algorithms for network computing were studied by Maheswaran (1999), related to the QoS-Based Scheduling Problem analyzed by Doğan and Özgür (2006), for scheduling a set of independent tasks with multiple QoS needs in global heterogeneous computing (HC) systems. More relevant to this work is the “SLA-based Allocation of Cluster Resources” studied by Yeo and Buyya (2005) and the “SLA Framework for QoS Provisioning and Dynamic Capacity Allocation” introduced by Garg et al. (2002). The latter presents an interesting work, encompassing QoS requirements into SLAs and reserving the resources with respect to calculating the probability of future requests. It does not, however, actually use the historicity elements to the point that it is possible, such as the SLA violations history that is taken into account in our meta-scheduler. Job arrival history is used by Chandra et al. (2003), relating application resource requirements to their dynamically changing workload characteristics, but does not make any correlation with SLAs and client requirements, analyzing the low-level problem in the server side. Menychtas et al. (2009) present a useful real-time reconfiguration mechanism for guaranteeing QoS provisioning levels, using experience from previous executions (e.g. SLA violations), as well as Tserpes et al. (2007), the QoS History Service of the type of each task as well as some input parameters (deadline, files that are going to be processed on a remote machine. Based on or only executables submitted for remote execution, or just input files that can be executables together with their input files and parameters, or only executables submitted for remote execution, or just input files that are going to be processed on a remote machine. Based on the type of each task as well as some input parameters (deadline, quality), the provider side estimates the job computational requirements. In this case, in order to find a feasible and effective solution to the resource management problem described, it was decided to reduce it into a Fractional Knapsack, since various efficient algorithms can solve it, some of which are presented at Section 4. The problem deduction and two basic variations of the knapsack problem (the 0/1 and the Fractional) are described in the following sections.

3. Resource allocation model

3.1. Reduction to the Fractional Knapsack for one dimension

3.1.1. Problem general model

As discussed, a basic issue arising for architectures as the ones described above, is the resource management problem: since the system resource capacity, say $M$, is limited and numerous service clients require resources based on specific electronic contracts (SLAs) guaranteeing the availability, heuristic algorithms and “intelligent” techniques are required to dispense available resources, satisfying the requestors as much as possible. In this model, we assume that specific QoS requests for slowdown, reflecting the requirement that all job requests should be treated equally regardless of their resource requirements do not exist, to ensure the maximum flexibility of the resource management. If the total demanded workload, say $W$, exceeds the system capacity $M$, resource reservation and QoS terms safeguarding to the customers come into conflict. Some of the client requirements cannot be satisfied, or at least be fully satisfied, due to the lack of sufficient resources, resulting in loss of provider profits through SLA violations that will eventually occur. It is then obvious that appropriate dynamic resource allocation mechanisms are needed, using risk management techniques in order to minimize this loss, while maximizing resource utilization and service provider profits. Resource allocation and scheduling problems with financial constraints are usually solved through a deduction to well known NP-complete problems, such as the knapsack problem (Basso and Peccati, 2001; Golenko-Ginzburg and Gonik, 1998). In our case, in order to find a feasible and effective solution to the resource management problem described, it was decided to reduce it into a Fractional Knapsack, since various efficient algorithms can solve it, some of which are presented at Section 4. The problem deduction and two basic variations of the knapsack problem (the 0/1 and the Fractional) are described in the following sections.

3.1.2. Problem parameters simplified

In the environments described in the previous sections, the seamless system resource pool can be comprised of several available resources (memory, CPUs, disk, etc.). Initially, we will consider only one “dimension” of this pool, as for example system memory. The total memory (system capacity) of such a system can be denoted as $M$. Assume now that there are currently $R$ incoming preemptable tasks submitted by service clients that require resources, each one denoted as $j, i \in \{L2, \ldots, R\}$. These tasks can be executables together with their input files and parameters, or only executables submitted for remote execution, or just input files that are going to be processed on a remote machine. Based on the type of each task as well as some input parameters (deadline, quality), the provider side estimates the job computational requirements, as described in Section 1.

So assume that each pending job requires a certain amount of memory $w_i$ as dictated from the respective customer’s SLA parameters. The total required memory (demanded workload) can be denoted as a set of job workloads: $W = \{w_1, w_2, \ldots, w_R\}$. Moreover, each job successful execution yields a certain profit, based on the job cost also defined in the SLA, so a set of job profits can be denoted as $P = \{p_1, p_2, \ldots, p_R\}$, indicating the profit that yields if each job runs without an SLA violation. If the demanded workload is greater than the system capacity ($\sum w_i > M$), it must be decided which of the incoming jobs run in a given time period. The actual goals are to achieve the maximum utilization of the available memory within the infrastructure, as well as to gain the maximum profit out of them, according to each task cost. The problem parameters indicate similarity with the NP-complete knapsack problem and, as will be discussed later on, with the Fractional Knapsack variation that they both are presented below.

3.1.3. Definition of knapsack problem

3.1.3.1. The 0/1 knapsack problem. The classic 0/1 knapsack problem is a problem in combinatorial optimization: we are given two sets of $R$ positive integers, $P = \{p_1, p_2, \ldots, p_R\}$ and $W = \{w_1, w_2, \ldots, w_R\}$, and an integer $M$. The $w_i$’s may be interpreted as
3.1.3.2. The Fractional Knapsack problem. The Fractional Knapsack problem is a variant of the classic knapsack problem: given objects of different values per unit volume and maximum amounts and being able to take pieces (fractions) of objects of different values per unit volume and maximum amounts, the problem is to fill the knapsack with many objects in a way that the total profit earned is maximized. Mathematically the 0/1 knapsack problem may be stated as

maximize: \( \sum_{i} p_i \delta_i \)
subject to: \( \sum_{i} w_i \delta_i \leq M \) and \( \delta_i = 0,1 \)

3.1.4. Problem deduction

The resource management problem described in Section 3.1.2 can be easily deducted to the knapsack problem: coming down to knapsack’s logic, the required memory set \( W = \{w_1, w_2, \ldots, w_k\} \) is regarded as the set of weights and \( M \) as the size of the knapsack. The set of job profits \( P = \{p_1, p_2, \ldots, p_k\} \) for the incoming jobs is modeled as the set of knapsack profits, leading to a classic knapsack problem (Fig. 1).

In our case, however, resource allocation for the incoming jobs is dynamic and can be fractional. For example, assume that jobs \( j_1 \) to \( j_{j-1} \) are already running and a new incoming job \( j_i \) arrives requesting resources. If \( j_i \) requires memory \( w_i \) and the available memory \( w_i \) is less than the amount required \( (w_i < w_i) \), then only a fraction \( w_i \) of the desired amount \( w_i \) can be allocated for \( j_i \), taking into consideration a severe probability that \( j_i \) may fail or its output may not fulfill the QoS requirements (Fig. 2).

This leads us to modeling the problem as a Fractional Knapsack problem, as described above. Amongst others, a known feasible solution of the Fractional Knapsack is the greedy solution (Ishii et al., 1977) (solution alternatives are provided in Section 3.1.6). The idea is to calculate for each object the profit density (ratio of profit/weight), and sort them according to this ratio. For typical sorting algorithms good behavior has complexity \( O(n \log n) \) and bad behavior \( O(n^2) \). The objects with the highest ratios should be chosen and added to the knapsack until you cannot add the next object as whole. Finally add as much as you can of the next most valuable object. Continue until the knapsack is full. Mathematically this can be stated as

maximize: \( \sum_{i} p_i \delta_i \)
subject to: \( \sum_{i} w_i \delta_i \leq M \) and \( \delta_i = 0,1 \),
Given the above, a simple measure of job $j_i$’s potential profit for the provider $p_i$ is

$$p_{i,j} = PR_i - VP_i \cdot VC_i - EC_i$$

Taking into account the limited capacity provided when various jobs wait in the queue requiring resources, we set $SP_i$ as the job success probability, that depends on the resources remaining (satisfying only a fraction of the job demands): $SP_i = 1 - VP_i$. Then, another measure indicating the profit $p_i$ is the violation cost that the provider will save, in case the job runs successfully:

$$p_{i,2} = SP_i \cdot VC_i$$

If free resources are larger than the ones required by incoming jobs, the above measure gets close to $VC_i$, otherwise it depends on the amount of resources available: the greater percentage of the desired resource amount available, the more profit (hence priority using the greedy solution) this job will yield. Adding these two measures of the profit, total profit $p_i$ function for job $j_i$ is calculated as:

$$p_{i,1} = PR_i + p_{i,2} = PR_i - VP_i \cdot VC_i - EC_i + SP_i \cdot VC_i = PR_i - 2 \cdot VP_i \cdot VC_i - EC_i + VC_i$$

Therefore, step 1 of the greedy algorithm of the previous section is now updated to the following (Alg. II):

1. Sort the array

$$\begin{align*}
&[p_1, p_2, \ldots, p_R] \\
&[wR_1, wR_2, \ldots, wR_k],
\end{align*}$$

where

$$p_i = PR_i - 2 \cdot VP_i \cdot VC_i - EC_i + VC_i, i \in \{1, \ldots, R\}$$

In step 4 the recalculation of the $p_i$’s is essential (complexity $O(n^2)$) again as in Alg. I, since $p_i$ is dependent on the violation probability $VP_i$, as depicted above, and when the demanded resources are not fully available this probability obviously changes.

### 3.1.6. Knapsack problem solution alternatives

The knapsack problem, as mentioned above, belongs to the family of NP-complete problems. Therefore, to find the optimal solution a basic research is needed. Litke et al. (2007) studied a similar deduction to the knapsack problem with the current one, examining several solution techniques, such as Backtracking (BT), Branch and Bound (BB), Dynamic Programming (DP) and a heuristic greedy algorithm (GR). In that work, the greedy technique was considered the best and was finally employed in order to find a “near-optimal” solution, as will be done in the present paper also. The performance of the algorithm mainly depends on the heuristic functions selected, as well as on the degree of correlation between the profit and the weight (see Fig. 6). In our case, it is reasonable to assume that we deal with weakly correlated data, since not only resource consumption from each incoming job, but other parameters as well affect pricing policy (client relationships management, large demands on system workload, emergency situations, etc.) illustrated in the SLAs.

### 3.2. Multi-dimensional concept and solution

For a more realistic implementation of the solution in a virtualized SaaS system, it must be taken into account that resource management in a computer cluster involves multiple

![Fig. 4. An incoming job $j_{k+1}$ arrives, with a higher priority than job $j_{k-1}$ that is already running in the pool.](image)

![Fig. 5. Resources are allocated for incoming job $j_{k+1}$, job $j_k$ is sent to the queue.](image)

![Fig. 6. Uncorrelated, weakly correlated and strongly correlated data examples.](image)

PC resources (CPUs, disk, memory, etc.). Thus, the weight \( w_j \) that represented the demanded memory from job \( j \) in the previous sections, will actually be a vector of \( k \) resource demands: \( w_j = (w_{j1}, w_{j2}, \ldots, w_{jk}) \) and in turn the system capacity will be a vector of the maximum available resource values: \( M = (w_{\text{max}1}, w_{\text{max}2}, \ldots, w_{\text{max}k}) \). To the best of our knowledge, all related works tackle the problem of resource allocation as a mono-dimensional problem, with the exception of (Campegiani and Lo Presti, 2009), which uses a different variant of the knapsack at the virtualization level that cannot be applied to the high level “one sink” problem analyzed here.

Works of Bertsimas and Demir (2002) and Chu and Beasley (1998) present several heuristic algorithms and solutions for the multi-dimensional knapsack problem. This paper follows the approach of Pirkul (1987) using a heuristic algorithm as well as the original Fractional Knapsack’s greedy solution presented above with multiple weights (dimensions). According to that, the knapsack is packed in the decreasing order of profit/weight ratios, where “weight” here is a number illustrating the size of the \( w_j \) vector. The ratio in Pirkul’s approach is defined as: \( p_j = \frac{\text{profit}}{\sum_{i=1}^{k} w_{ji}} \), where \( k \) is the number of the dimensions and the multiplier \( a_j \) introduced is the surrogate multiplier for each dimension \( j \), representing the importance of each weight. The \( a_j \) multipliers in our case were chosen to divide each weight to the maximum resource amount provided by the system: \( a_j = 1/w_{\text{max}j} \).

This was decided in order to get equivalent percentages for each dimension, as satisfying each resource demand is of the same importance in the calculation of the priorities. So the above ratio gets equal to: \( p_j = \frac{\text{profit}}{\sum_{i=1}^{k} (w_{ji}/w_{\text{max}i})} \). Given the above alteration for the greedy solution if multiple dimensions exist, the multi-dimensional problem can be now extended as such.

Considering a set of \( k \) available resources \( M = (w_{\text{max}1}, w_{\text{max}2}, \ldots, w_{\text{max}k}) \), each incoming job \( J \) will have an adequate set of requirements \( \{w_{j1}, w_{j2}, \ldots, w_{jk}\} \) for the resources provided and a profit:

\[
p_j = PR_j - 2 \cdot VP_j \cdot EC_j - EC_{\text{other}} \cdot V_{\text{other}} \in \{1 \ldots R\}
\]

The resource allocation problem reduced to a multi-dimensional Fractional Knapsack is then solved by sorting the following array \( A \), containing the profit/weight ratios:

\[
A = \left[ \begin{array}{c} p_1 \\ \frac{p_1}{\sum_{j=1}^{k} a_j w_{j1}} \\ \frac{p_2}{\sum_{j=1}^{k} a_j w_{j2}} \\ \vdots \\ \frac{p_k}{\sum_{j=1}^{k} a_j w_{jk}} \end{array} \right] = \left[ \begin{array}{c} PR_1 - 2 \cdot VP_1 \cdot VC_1 \cdot EC_1 + VC_{\text{other}} \\ \frac{PR_1 - 2 \cdot VP_1 \cdot VC_1 \cdot EC_1 + VC_{\text{other}}}{\sum_{j=1}^{k} (w_{j1}/w_{\text{max}1})} \\ \frac{PR_2 - 2 \cdot VP_2 \cdot VC_2 \cdot EC_2 + VC_{\text{other}} - EC_2}{\sum_{j=1}^{k} (w_{j2}/w_{\text{max}2})} \vdots \frac{PR_k - 2 \cdot VP_k \cdot VC_k \cdot EC_k - EC_{\text{other}} + VC_k}{\sum_{j=1}^{k} (w_{jk}/w_{\text{max}k})} \end{array} \right]
\]

Moreover, after step 4 of the generic algorithm defined in the previous sections, the job that is chosen after recalculating the profits to run, may not “fill” the knapsack, although being the most profitable at the moment. Hence, resources may still be available for another job to run, thus this step must be repeated. The resulting greedy algorithm (Alg. III) that is the complete solution of the problem is defined as follows:

For jobs \( j = \{j_1, j_2, \ldots, j_k\} \):
1. Sort the array \( \langle A \rangle \) given above.
2. Allocate resources for the first job in the array and remove it.
3. Repeat 2 until next job’s resource requirements are not available.
4. For the remaining, resort the array after recalculating \( p_i \)'s and choose the one \( j_l \) with the highest ratio \( p_l / \sum_{j=1}^{k} a_j w_{jl} \).
5. Repeat 4 until at least one of the remaining resources runs out, so as to fully exploit them.

After step 5, one of the available resources has ran out, so the ideal utilization of resources is taking place at the time and no other job is capable of running, until one of the running jobs ends and allocates resources, or a new incoming job appears with a potential higher priority.

4. Evaluation system architecture

4.1. Method of evaluation

For the evaluation of the presented method, we implemented the proposed resource management algorithm as a meta-scheduling solution in a cluster, on top of which, a GRIA middleware (The GRIA project, Grid Resources for Industrial Applications, 2009) is operating, which is really suitable for managing client accounts and SLAs. However, since a fully virtualized environment was desired at the resource management level, being able to pause a job running at any time (see pool “pause” conditions described above), it was decided to extend the existing resource managers in GRIA to use Xen Virtual Machines (Barham et al., 2003), in order to host the job runtime environments. Xen Virtual Machines (VMs) are quite elastic, since the resources (memory, CPUs, etc.) they use can be dynamically altered at runtime. Furthermore, it is possible to pause and resume the virtual machines whenever required, while saving the environment images. Time needed to pause and/resume Xen images constitute scheduling-related costs and can potentially affect the algorithm performance. However, as will be also presented in Section 5.2, the overall effect of dynamic allocation in Xen VMs is not critical for the system performance. Therefore, a meta-scheduler system was built that wraps GRIA middleware with the Xen Virtual Machines cluster. This meta-scheduler implements the greedy multi-dimensional Fractional Knapsack solution, so as to prioritize the jobs to run.

In general, existing mechanisms completely reserve the requested resources, otherwise enqueuing the requesting jobs. In order to achieve the maximum resource utilization possible, our meta-scheduler allocates resources according to the algorithm Alg. III presented above, each time a new job arrives to the meta-scheduler queue or a finished job disengages resources. Utilization of the available resources is intuitively near-optimal using this method, since the “knapsack” is constantly as full as possible: in step 4 of the algorithm, more jobs from the queue are selected to run until the “knapsack” gets full, with a respective risk of failure or violating their SLAs. Hence, available resources are fully utilized even if they do not completely satisfy the workload request. This way, when resource demands exceed the available ones, one or more of the resource “dimensions” are fully exploited.

Moreover, the provider’s revenues are maximized using a risk-management technique, instead of simple scheduling and resource management methods such as “First Come First Served” for example. This risk-based allocation schema is based on the probability of a violation in the SLA, generated either intentionally by not providing the necessary resources, or unintentionally e.g. due to a system failure. Violation probabilities are extracted by the QoS History Service for GRIA produced by Tserpes et al. (2007), which predicates the probability of a provider to deliver a specific QoS level to the consumer via its QoS Indices Estimator. The calculation of the probability is based on the available resources, as well as on the history of the SLA violations that have been observed in the past when running jobs of similar characteristics. Such probabilities are delivered from the components taken from the work mentioned above, the implementation details, however, are considered out of the scope of this paper. In the first steps of using the service, the violation history will be
incomplete, while after quite some period in the service lifetime, it will be more helpful and of greater importance.

To validate the current solution, the performance of GRIA using the Fractional Knapsack algorithm to schedule jobs to the Xen VMs must be compared to existing schemes. Several scheduling policies have been studied (Harchol-Balter et al., 2002), such as the Shortest-Remaining-Processing-Time-First (SRPT), the Longest-Remaining-Processing-Time-First (LRPT) or the Least-Attained-Service (LAS) policy, which actually implements the “First-Come, First-Served” scenario mentioned. Our meta-scheduler is tested against such alternative scenarios to highlight its effectiveness.

4.2. System architecture

4.2.1. Evaluation in the GRIA middleware

The GRIA resource allocation service supports the establishment of Service Level Agreements, whose principal is trusted to set these up under an account by its budget holder (Surridge et al., 2005). GRIA SLA Service interface enables the provider to set resource usage constraints along with tariffs for defining specific pricing terms (e.g. application cost) to a new SLA Template (see Appendix 1), as well as the cost of an SLA Violation. SLA Templates act as offers: each customer can propose the one with the desired terms, to create an SLA instance between him and the provider. For a new consumer's request, the allocation service connects to a capacity model representing the physical network bandwidth, data storage and processing nodes, to determine the specific level of service to offer, with regards to the terms of the customer SLA. GRIA Job Service periodically generates usage reports using QoS criteria which may be qualitative (e.g. error conditions) or quantitative (e.g. processing time, data transferred). The SLA Service uses these reports to monitor customer usage and the level of commitments from existing agreements compared with available capacity.

In order to manage the cluster resources, GRIA Job Service, which is the one submitting incoming jobs to the cluster, allows multiple Resource Managers (RMs) to be used, such as TorquePBS (Cluster Resources Inc., 2010) and Condor (The Condor Project, 2010), via a simple RM selection mechanism. The Job Service does not actually access Resource Managers directly, to submit and check jobs’ statuses. Instead, it introduces an extra layer of manager-dependent RM Connector scripts, written in python: for each manager, GRIA requires a separate RM Connector Plugin. The default RM connector script is tailored for local execution, not using any cluster. In the original model, a job can be accepted for full submission or rejected. No intermediate mechanism or queue exists in GRIA to keep jobs while resources are not available.

Therefore, if the Resource Manager cannot accept any more jobs, job submission will simply fail.

4.2.2. System components

In order to implement the solution presented in this paper, we added a Xen Connector Plugin and created a meta-scheduling and queuing algorithm, based on the Fractional Knapsack model described above. The new Plugin was written in python and implemented four basic functions needed for every RM Connector Plugin in GRIA:

- canRunJob
- submitJob
- checkJob
- jobUsage
- killJob

The extra development work at this stage involved mainly the “submitJob” function, which the one that implements the knapsack solution, checking the available cluster resources and prioritizing jobs according to the algorithm provided in Section 3.2. This algorithm runs each time a new job is submitted to GRIA Job Service or whenever resources are allocated by a job termination. The new Plugin connects to a Xen Manager component, which is actually consisted of a set of linux shell scripts that controls all virtual machines available in the cluster.

When a new job i is submitted, Xen Manager checks the available cluster resources via the “getAvailableResources” script and returns the result to the meta-scheduler Plugin (Fig. 7). The meta-scheduler, checks the requirements R (r1 CPUs, r2 MB of memory, r3 MB of disk) of job i from the Job Description documents provided (Appendix 2), as well as the costs from the client’s SLA to prioritize incoming jobs. To calculate the knapsack “profit”, it must also estimate an SLA Violation Probability. As mentioned, to find this Violation Probability, it invokes the QoS History Service for GRIA produced by Tserpes et al. (2007), which can store the violation history and come up with an estimation for the violation probability VP of the SLA of each job i. After running the algorithm, jobs are prioritized and sent to the Xen Manager to run accordingly. Xen Manager defines the dimensions R’ (r1’ CPUs, r2’ MB of memory, r3’ MB of disk) of the Virtual Machines running, allocating this way resources for each job through the Xen VM that hosts the application to run. Fractional (R not equal to R’) or complete resource allocation can be imposed by setting resource constrains R’ to the respective Virtual Machine, during the job submission. Each of the Xen Virtual Machines used has an instance of the application wrapper, which is a script running the required application, while the input and output data are...
automatically transferred from and to the Manager environment through a Secure Shell (SSH) connection. Fig. 8 presents an overview of the resulting system’s components.

A rudimentary job queue was also created, as in the original GRIA middleware there is not any capability as such.

5. Experiment and evaluation

5.1. Testbed and dataset used

For the evaluation process of the scenario presented above, a Xen Virtual Machine cluster was created, providing 12 CPUs with 6 GB RAM and just 10 GB of disk space, to allow stress testing under limited resources. Each generated VM operated in Ubuntu 9.04. Sets of 30 incoming resource consuming rendering jobs were created, to be submitted in a 3D Rendering Service deployed in GRIA. This service was based on a distribution of the 3Delight® renderer (3Delight RenderMan—compliant renderer, 2008). 3Delight is a known commercial application for rendering with an open license. To give an idea of the application load, we ran one of the simple test examples available over the Internet for 3Delight, giving as inputs the file “cornell.rib” and default resolution (320 × 240) for only one frame in our cluster. Using a Xen VM with 1 processor (2.8 GHz) and 200 kB Memory, 10 s were needed to complete the specific rendering. This kind of rendering is achieved by using descriptors expressed by the RenderMan Interface Bytestream (RIB) (RenderMan Interface Specification v3.2., 2005) format which is a complete specification of the required interface between modelers and renderers. In detail, the rendering procedure is extended by adding an intermediate step, where a RIB file is produced describing the scene that is to be rendered. Files in RIB format are significantly smaller in size and therefore easier to be exchanged through the network. On the downside, they are more difficult to be managed since they are numerous and they have a great number of interconnections. Specified renderers are then able to reconstruct and render the image using the intermediate RIB files. The output of this process is the full rendered image.

The GRIA Supplier offers the rendering service by federating a number of intermediate services, such as collecting the data, checking the validity, invoking the rendering service and delivering the result. The submitted sets of jobs were selected according to their complexity, which is mainly affected by the following four (4) factors:

- Number of objects.
- Lights (spot, omni, direct).
- Materials (standard, image, animated).
- Modifiers (lattice, melt).

Thus, specific sets of scenes were created in order to be rendered, which were extending to a wide range of all the possible selections, in order to stretch the performance of the service provider in the aforementioned aspects.

In the context of the experiment each job submitted to the renderer was comprised of 50 RIB files, extracted by 30 respective animations lasting some seconds, and had a different configuration, or – in other words – a different combination of number of objects, type of lights, materials and modifiers. The time between two job arrivals was chosen to follow a simple Gaussian distribution and thus the system queue can be modeled as a Markov chain, since the Markov property is in effect (the possibility of every potential future state of jobs waiting in the queue is only dependent on the current state): $Pr(X_{n+1} = x|X_1 = x_1, X_2 = x_2, ..., X_n = x_n) = Pr(X_{n+1} = x|X_n = x_n)$, which is realistic. To test the performance of the algorithm in contrast to existing meta-scheduling tactics, various and realistic datasets were also needed, concerning pricing as well as the input requirements (job costs, violation costs, deadlines). For job and violation costs the normal distribution was chosen so as to generate input datasets. Three categories of costs were defined (Table 1).

<table>
<thead>
<tr>
<th>Table 1 Violation cost categories.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Violation cost (VC) category</td>
</tr>
<tr>
<td>Mean value in the VC distribution/ mean value in the job cost distribution</td>
</tr>
</tbody>
</table>

To validate the mechanism under different pricing circumstances, three iterations of the validation process were attempted. During the first iteration incoming jobs fell into the SMALL category, while in the second into the LARGE category. In the third iteration, the SLAs of the incoming jobs were divided into 3 groups having SMALL, MEDIUM and LARGE violation cost value (mixed case).

Finally, there were 3 resource requirements initially taken into consideration in the knapsack algorithm: requirements concerning
disk space, memory and the number of CPUs. A fourth dimension indicating the CPU time left for the application to be finished was added, to integrate a SRPT policy in the meta-scheduler: given that incoming jobs have specific deadlines, the CPU time left for the job to finish is another measure of the job demanded workload (jobs close to termination should be of high priority as well, in an effort to decongest the queue). The workload, as mentioned in the beginning, is changeable: when the time left to the job’s deadline, for example, is less than the remaining CPU time requested by the job, the number of CPUs the job requests must increase in order to catch the deadline, given of course that the render application can be parallelized. Hence, the required workload for each job in the queue must be recalculated each time the knapsack algorithm runs.

5.2. Experiment results

As mentioned, the experiment consisted of 30 render jobs running in the Xen VM cluster described above for the three iterations. In the diagrams presented (Figs. 9–17), the cluster resource utilization and the total profit gained by the provider, measured in currency units (calculated by the prices of the jobs that ran successfully and the violation costs of the SLAs violated), are illustrated. In each iteration, the same set of jobs was submitted to the system implemented using the Fractional Knapsack algorithm solution, as well as to the original GRIA system connected to the Xen VM Cluster and to a GRIA implementing the LAS scheduling policy in the Resource Manager Plugin level of the cluster. Of course, the two alternative schemes do not use our risk management technique and reserve only resources fully satisfying the incoming job requirements.

Note that if the resources available are significantly smaller than the amount desired they are not actually worth allocating, as the possibility of job failure or SLA violation is extremely high. For that reason a lower threshold of 25% was considered. Moreover, for applications like 3Delight, there are strict minimum requirements for memory which, in case they are not satisfied, they will lead to definite failure and thus resources are not reserved in such a case. These resources can be allocated for other jobs waiting in
the queue or just be kept uncommitted for backup and system integrity and safety purposes for a while.

Analyzing the experiment results (Table 2) and resultant diagrams, certain outcomes can be extracted about the meta-scheduling method applied. Firstly, the total payoffs earned with the knapsack meta-Scheduler are significantly greater than the ones of GRIA or the LAS scheduler, as seen above. The same happens for the mean success rate of jobs terminations without SLA violations having taken place, as well as the utilization of cluster resources, a fact also indicated by the acuteness of the curve, frequently reallocating resources each time there are “pause” conditions in the pool, as explained above. Furthermore, as can be seen in Table 2 and Section 5.2, the utilization of some resources may be occasionally greater with the LAS scheduler, mainly due to early job terminations with use of our algorithm and late arrivals. However, under the same workload, the knapsack meta-Scheduler’s utilization is almost ideal (this can be observed at the point when the first job requesting resources more than the available ones arrives), and also the stimulating measure of total payoff is always better.

Finally, the shape of the curves denotes that there is a small delay brought in by the run of the meta-scheduling algorithm in the allocation of resources for the incoming jobs, which, however, does not substantially affect the performance overall. The effects in performance overhead due to dynamic allocation were also studied by Wang et al. (2007), in conjunction with the reallocation frequency. In this work, having Transactional Workloads in Xen Virtual Machine Containers the total CPU capacity (as regards CPUs offered over a period of time) loss is up to 5%, as a result of dynamic allocation.

6. Conclusions

Exploiting the properties of virtualized infrastructures for elastic resource allocation is one of the latest issues that concern the scientific community more and more lately. This paper
investigated this problem under a more complex light, that of profit maximization. In a business environment profit is tightly coupled to resource utilization and quality provision towards the customer. The experiment that was set up aimed at showing how a service provisioning business infrastructure, following a SaaS principle, can maximize the provider’s profit using a combination of service oriented architectures and virtualization technologies. We generated a utility function that assesses the priority of each job in run-time based on resource availability, risk for job failure and SLA violation, cost of violation and total payoff upon successful execution.

Given the dynamicity with which the priority for job execution is changing, jobs can instantly fall into one of these categories: no resources allocated, some resources allocated, maximum required resources allocated. The transition from one category to another is not always simple as jobs may be non-preemptable. For that reason, the allocated resources are virtualized which means that under the frame of a virtual machine hypervisor they can be suspended, stored and re-enacted at any point. The proposed solution exploits this fact in order to utilize the part of the resource pool that remains free, even if it does not completely satisfy the demanded workload of the most “important” tasks remaining in the queue. Furthermore, the virtualization of this infrastructure allows for the greater segmentation of available resources, increasing the elasticity factor of the whole pool.

Even though the latter observation leads to a very flexible technological solution it also leads to the conclusion that in practice the optimization of the utility function mentioned above is intuitively an NP-hard problem and presents similarities to the Fractional Knapsack problem. As such, the presented problem is modeled as a Fractional Knapsack problem and a greedy algorithm for solving it from the literature is employed.

It is noteworthy that the following assumption is made: even though we are not interested in the parallelization of the submitted jobs, in fact this is a crucial parameter if we want to cover all the cases, where virtualization actually creates a seemingly singular resource. For instance, if the computing Resource Pool is comprised of a number of PCs and a job requires more resources than those that the best PC can provide, then how the virtualization layer will handle that without actually breaking the job to smaller units? However, this is an issue that touches the resource allocation layer itself and is considered to be out of scope of this paper which deals with the problem at the meta-scheduler layer. Still it remains a plan for this issue to be studied in the future.

Furthermore, this paper proposes an architecture for implementing the model, combining a middleware, typically built as a SOA solution, and a virtualized infrastructure. Based on that, the authors have set up a testbed and used evaluation datasets (typically SLAs with variable costs and resource requirements) to conduct experiments. The evaluation and the comparison against traditional resource allocation mechanisms showed promising results regarding the maximization of the profit and the utilization of resources.

The results indicate that if the utility function that calculates the profit can actually model the one deriving from the business models of service providers, then this suggestion can become immediately exploitable. However, there are still things that need to be identified such as the cost of keeping the resource utilization so high, even though the uptime in the long run may be reduced. The latter is especially true if batches of task arrive in bursts, however, in the case of a more smooth distribution of the incoming flow this is not true and the proposal of this paper needs to be further investigated.

Finally, it remains as a future work to also model the overhead that is generated from the suspension and re-enactment of virtual machines. This overhead has been considered in the calculations as they were incorporated in the measurements, though, a model predicting or estimating them has not been devised.

Appendix 1: a sample GRIA SLA template

```
<?xml version="1.0"?>
<slaTemplate>
  <label>Sample SLA Template</label>
  <description>A template limiting instantaneous use to 2 CPUs</description>
  <billingPeriod>
    <year>2010</year>
    <month>6</month>
    <dayOfMonth>1</dayOfMonth>
  </billingPeriod>
  <startTime>
    <year>2011</year>
    <month>6</month>
    <dayOfMonth>1</dayOfMonth>
  </startTime>
  <endTime>
    <year>2011</year>
    <month>6</month>
    <dayOfMonth>1</dayOfMonth>
  </endTime>
  <constraints>
    <constraint type="INSTANTANEOUS">
      <metric type="RESOURCE">
        <uri>http://www.gria.org sla metric resource cpu</uri>
        <description></description>
        <units type="DECIMAL">
          <instantaneous>CPU</instantaneous>
        </units>
      </metric>
      <bound>LE</bound>
      <private>false</private>
      <limit>2.0</limit>
      <contention>1.0</contention>
      <repeating>false</repeating>
    </constraint>
  </constraints>
  <pricingTerms>
    <pricingTerm type="INSTANTANEOUS_INCREASE">
      <description>creation charge</description>
      <lowerBound>0</lowerBound>
      <upperBound>-1</upperBound>
      <price>100</price>
      <metric type="ACTIVITY">
        <uri>http://www.gria.org sla metric activity job</uri>
        <description></description>
        <job></job>
        <units type="DECIMAL">
          <instantaneous>CPU</instantaneous>
        </units>
      </metric>
    </pricingTerm>
  </pricingTerms>
</slaTemplate>
```

This term specifies a charge of 100 EUR every time a job is created.
Appendix 2: a simple JSDL document job.jsdl

<?xml version="1.0" encoding="UTF-8"?>
  
  <JobDescription>
    <JobName>render 1</JobName>
  </JobDescription>
  
  <Application>
    <ApplicationName>render</ApplicationName>
  </Application>
  
  <Resources>
    <TotalCPUCount/>
    <Range>
      <LowerBound>0.0000000000</LowerBound>
      <UpperBound>2.0000000000</UpperBound>
    </Range>
    <TotalVirtualMemory/>
    <Range>
      <LowerBound>0.0000000000</LowerBound>
      <UpperBound>2000000000.0000000000</UpperBound>
    </Range>
    <TotalDiskSpace/>
    <Range>
      <LowerBound>0.0000000000</LowerBound>
      <UpperBound>20000000.0000000000</UpperBound>
    </Range>
    <DataStaging name="inputImage">
      <FileName>inputImage</FileName>
      <CreationFlag>overwrite</CreationFlag>
      <DeleteOnTermination>true</DeleteOnTermination>
    </DataStaging>
    
    <DataStaging name="outputImage">
      <FileName>outputImage</FileName>
      <CreationFlag>overwrite</CreationFlag>
      <DeleteOnTermination>true</DeleteOnTermination>
    </DataStaging>
  </Resources>

</JobDefinition>