SLA-aware virtual resource management for cloud infrastructures

Hien Nguyen Van, Frederic Dang Tran, Jean-Marc Menaud

SLA-aware virtual resource management for cloud infrastructures[*]

Hien Nguyen Van, Frédéric Dang Tran
Orange Labs
38-40 rue du Géréral Leclerc - 92794 Issy Les Moulineaux Cedex 9, France
{hien.nguyenvan, frederic.dangtran}@orange-ftgroup.com

Jean-Marc Menaud
Ascola, École des Mines de Nantes / INRIA, LINA
4, rue Alfred Kastler - 44307 Nantes Cedex 3, France
menaud@emn.fr

Abstract

Cloud platforms host several independent applications on a shared resource pool with the ability to allocate computing power to applications on a per-demand basis. The use of server virtualization techniques for such platforms provide great flexibility with the ability to consolidate several virtual machines on the same physical server, to resize a virtual machine capacity and to migrate virtual machine across physical servers. A key challenge for cloud providers is to automate the management of virtual servers while taking into account both high-level QoS requirements of hosted applications and resource management costs. This paper proposes an autonomic resource manager to control the virtualized environment which decouples the provisioning of resources from the dynamic placement of virtual machines. This manager aims to optimize a global utility function which integrates both the degree of SLA fulfillment and the operating costs. We resort to a Constraint Programming approach to formulate and solve the optimization problem. Results obtained through simulations validate our approach.

1 Introduction

Corporate data centers are in the process of adopting a cloud computing architecture where computing resources are provisioned on a per-demand basis, notably to handle peak loads, instead of being statically allocated. Such cloud infrastructures should improve the average utilization rates of IT resources which are currently in the 15-20% range. A key enabling technology of cloud systems is server virtualization which allows to decouple applications and services from the physical server infrastructure. Server virtualization makes it possible to execute concurrently several virtual machines (VM) on top of a single physical machine (PM), each VM hosting a complete software stack (operating system, middleware, applications) and being given a partition of the underlying resource capacity (CPU power and RAM size notably). On top of that, the *live migration* capability of hypervisors allows to migrate a virtual machine from one physical host to another with no or little interruption of service.

The downside of the flexibility brought by virtualization is the added system management complexity for IT managers. Two levels of mapping must be managed (Figure 1): the provisioning stage is responsible for allocating resource capacity in the form of virtual machines to application. This stage is driven by performance goals associated with the business-level SLAs of the hosted applications (e.g. average response time, number of jobs completed per unit of time). Virtual machines must then be mapped to physical machines. This VM placement problem is driven by data center policies related to resource management costs. A typical example is to lower energy consumption by minimizing the number of active physical servers.

This paper presents an autonomic resource management system which aims at fulfilling the following requirements:

- ability to automate the dynamic provisioning and placement of VMs taking into account both application-level SLAs and resource exploitation costs with high-level handles for the administrator to specify trade-offs between the two,
- support for heterogeneous applications and workloads including both enterprise online applications with stringent QoS requirements and batch-oriented CPU-intensive applications,

[*]This research is supported by the french *Agence Nationale de la Recherche* with the ANR-08-SEGI-017

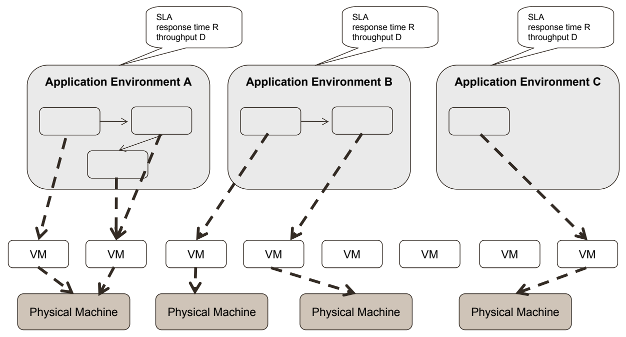

Figure 1. Dynamic resource allocation

- support for arbitrary application topology: n-tier, single cluster, monolithic and capacity to scale: either in a ”scale-up” fashion by adding more resource to a single server or in a ”scale-out” fashion by adding more servers,

Our proposed management system relies on a two-level architecture with a clear separation between application-specific functions and a generic global decision level. We resort to *utility* functions to map the current state of each application (workload, resource capacity, SLA) to a scalar value that quantify the ”satisfaction” of each application with regard to its performance goals. These utility functions are also the means of communication with the global decision layer which constructs a global utility function including resource management costs. We separate the VM provisioning stage from the VM placement stage within the global decision layer autonomic loop and formulate both problems as Constraint Satisfaction Problems (CSP). Both problems are instances of an NP-hard knapsack problem for which a *Constraint Programming* approach is a good fit. The idea of *Constraint Programming* is to solve a problem by stating relations between variables in the form of constraints which must be satisfied by the solution.

The remainder of this paper is organized as follows. Section 2 presents the architecture of the autonomic virtual resource management system. Next, we show some simulation results for different application environments in Section 3. We cover related work in Section 4. Finally, Section 5 concludes and presents future work.

2 Autonomic Virtual Resource Management

2.1 System Architecture

Our management system architecture is shown in Figure 2. The datacenter consists of a set of physical machines (PM) each hosting multiple VMs through a hypervisor. We assume that the number of physical machines is fixed and that they all belong to the same cluster with the possibility to perform a live migration of a VM between two arbitrary PMs. An *Application Environment* (AE) encapsulates an application hosted by the cloud system. An AE is associated with specific performance goals specified in a SLA contract. An AE can embed an arbitrary application topology which can span one or multiple VMs (e.g. multi-tier Web application, master-worker grid application). We assume that resource allocation is performed with a granularity of a VM. In other words a running VM is associated with one and only one AE. Applications cannot request a VM with an arbitrary resource capacity in terms of CPU power and memory size (in the remainder of this paper, we will focus primarily on CPU and RAM capacity but our system could be extended to cope with other resource dimension such as network I/O). The VMs available to the application must be chosen among a set of pre-defined VM classes. Each VM class comes with a specific CPU and memory capacity e.g. 2Ghz of CPU capacity and 1Gb of memory.

An application-specific *Local Decision Module* (LDM) is associated with each AE. Each LDM evaluates the opportunity of allocating more VMs or releasing existing VMs to/from the AE on the basis of the current workload using service-level metrics (response time, number of requests per second...) coming from application-specific monitoring probes. The main job of the LDM is to compute a utility function which gives a measure of application satisfaction with a specific resource allocation (CPU, RAM) given its current workload and SLA goal. LDMs interact with a *Global Decision Module* (GDM) which is the decision-making entity within the autonomic control loop. The GDM is responsible for arbitrating resource requirements coming from every AE and treats each LDM as a black-box without being aware of the nature of the application or the way the LDM computes its utility function. The GDM receives as input (i) the utility functions from every LDM and (ii) system-level performance metrics (e.g. CPU load) from virtual and physical servers. The output of the GDM consists of management actions directed to the server hypervisor and notifications sent to LDMs. The latter notifies the LDM that (i) a new VM with specific resource capacity has been allocated to the application, (ii) an existing VM has been upgraded or downgraded, i.e its class and resource capacity has been changed and (iii) a VM belonging to the application is being preempted and that the application should relinquish it promptly. Management actions include the lifecycle management of VM (starting, stopping VMs) and the trigger of a live migration of a running VM, the latter operation being transparent as far as the hosted applications are concerned.

We now formalize in more detail the inner workings of the local and global decision modules. Let $\mathbf{A} = (a_1, a_2, ..., a_i, ..., a_m)$ denote the set of AEs and $\mathbf{P} =$

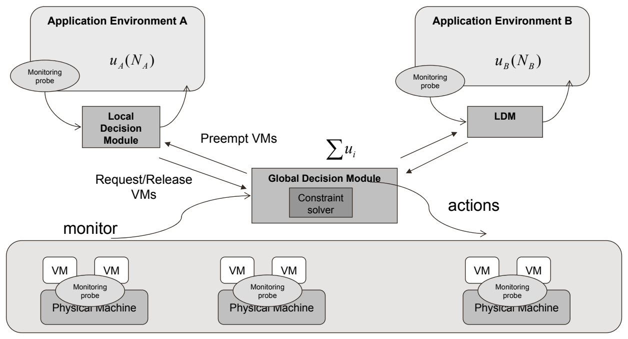

Figure 2. System architecture

$(p_1, p_2, ..., p_j, ..., p_q)$ denote the set of PMs in the datacenter. There are c classes of VM available among the set $\mathbf{S} = (s_1, s_2, ..., s_k, ..., s_c)$, where $s_k = (s_k^{cpu}, s_k^{ram})$ specifies the CPU capacity of the VM expressed in MHz and the memory capacity of the VM expressed in megabytes.

2.2 Local Decision Module

The LDM is associated with an application-specific performance model. Our architecture does not make any assumption on the nature of this model, whether it is analytical or purely empirical. A performance model is used by the LDM to assess the level of service achieved with a given capacity of resources (processing unit, memory) and the current application workload. The LDM is associated with two utility functions: a fixed service-level utility function that maps the service level to a utility value and a dynamic resource-level utility function that maps a resource capacity to a utility value. The latter which is communicated to the GDM on every iteration of the autonomic control loop. The resource-level utility function u_i for application a_i is defined as $u_i = f_i(N_i)$. N_i is the VM allocation vector of application a_i: $\mathbf{N}_i = (n_{i1}, n_{i2}, ..., n_{ik}, ..., n_{im})$ where n_{ik} is the number of VMs of class s_k attributed to application a_i. We require each application to provide upper bounds on the number of VM of each class ($\mathbf{N}_i^{max} = (n_{i1}^{max}, n_{i2}^{max}, ..., n_{ik}^{max}, ..., n_{im}^{max})$) and on the total number of VM ($\mathbf{T}_i^{max}$) that it is willing to accept. These application constraints are expressed as follows:

$$n_{ik} \leq n_{ik}^{max} \quad 1 \leq i \leq m \ \ and \ \ 1 \leq k \leq c \tag{1}$$

$$\sum_{k=1}^{c} n_{ik} \leq T_i^{max} \quad 1 \leq i \leq m \tag{2}$$

Note that theses bounds allow to specify a ”scale-up” application which is hosted by a single VM of varying capacity by setting $\mathbf{T}_i^{max}$ to 1.

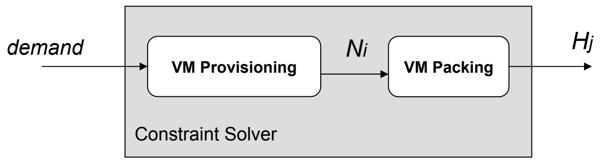

Figure 3. Constraint Solving: Provisioning-Packing

2.3 Global Decision Module

The GDM is responsible for two main tasks: determining the VM allocation vectors N_i for each application a_i (*VM Provisioning*), and placing these VMs on PMs in order to minimize the number of active PMs (*VM Packing*). These two phases are expressed as two *Constraint Satisfaction Problems* (CSP) which are handled by a *Constraint Solver* (Figure 3).

2.3.1 VM Provisioning

In this phase, we aim at finding the VM allocation vectors N_i for each application a_i while maximizing a global utility value U_{global}. The VMs allocated to all applications are constrained by the total of capacity (CPU and RAM) of the physical servers:

$$\begin{aligned} \textstyle\sum_{i=1}^{m} \sum_{k=1}^{c} n_{ik}.s_k^{cpu} &\leq \textstyle\sum_{j=1}^{q} C_j^{cpu} \\ \textstyle\sum_{i=1}^{m} \sum_{k=1}^{c} n_{ik}.s_k^{ram} &\leq \textstyle\sum_{j=1}^{q} C_j^{ram} \end{aligned} \tag{3}$$

where C_j^{cpu} and C_j^{ram} are the CPU and RAM capacity of PM p_j. The VM allocation vectors N_i need to maximize a global utility function expressed as a weighted sum of the application-provided resource-level utility functions and an operating cost function:

$$U_{global} = maximize \sum_{i=1}^{m} \Big(\alpha_i \times u_i - \epsilon.cost(N_i)\Big) \tag{4}$$

where $0 < \alpha_i < 1$, and $\sum_{i=1}^{m} \alpha_i = 1$. ϵ is a coefficient that allows the administrator to make different tradeoffs between the fulfillment of the performance goals of the hosted application and the cost of operating the required resources. $cost(N_i)$ is a function of VM allocation vectors N_i and must share the same scale as application utility functions, i.e (0,1). This cost function is not hardwired into the GDM but can be specified arbitrarily.

The output of the *VM Provisioning* phase is a set of vectors N_i satisfying constraints 1, 2, 3 and maximizing U_{global}. By comparing these allocation vectors with those computed during the previous iteration, the GDM is capable of determining which VMs must be created, destroyed or resized. The placement of the newly created VMs as well as the possible migration of existing VMs is handled by the VM packing phase described next.

2.3.2 VM Packing

The VM packing phase takes as input the VM allocation vectors N_i and collapses them into the single vector $\mathbf{V} = (vm_1, vm_2, ..., vm_l, ..., vm_v)$ which lists all VMs running at the current time. For each PM $p_j \in P$, the bit vector $\mathbf{H}_j = (h_{j1}, h_{j2}, \ldots, h_{jl}, \ldots, h_{jv})$ denotes the set of VMs assigned to p_j (i.e.: $h_{jl} = 1$ if p_j is hosting vm_l). Let $\mathbf{R} = (r_1, r_2, ..., r_l, ..., r_v)$ be the resource capacity (CPU, RAM) of all VMs, where $r_l = (r_l^{cpu}, r_l^{ram})$. We express the physical resource constraints as follows:

$$\begin{aligned} &\textstyle\sum_{l=1}^{v} r_l^{cpu}.h_{jl} \leq C_j^{cpu} \qquad 1 \leq j \leq q \\ &\textstyle\sum_{l=1}^{v} r_l^{ram}.h_{jl} \leq C_j^{ram} \qquad 1 \leq j \leq q \end{aligned} \tag{5}$$

The goal is to minimize the number of active PMs X:

$$X = \sum_{j=1}^{q} u_j, where\ u_j = \begin{cases} 1 & \exists vm_l \in V | h_{jl} = 1 \\ 0 & otherwise \end{cases} \tag{6}$$

The solving of the VM packing CSP produces the VM placement vectors H_j which are used to place VMs on PMs. Since the GDM is run on a periodic basis, the GDM computes the difference with the VM placement produced as a result of the previous iteration, determines which VM needs to be migrated. An optimal migration plan is produced as described in [2] to minimize the number of migration required to reach the new VM-to-PM assignment. Minimizing the cost of a reconfiguration provides a plan with few migrations and steps and a maximum degree of parallelism, thus reducing the duration and impact of a reconfiguration. The migration cost of a VM is approximated as proportional to the amount of memory allocated to the VM.

3 Validations & Results

To illustrate and validate our architecture and algorithms, we present some simulation results which show how the system attributes resources to multiple applications with different utility functions and how the resource arbitration process can be controlled through each application's weight α_i and the ϵ factor. Our simulator relies on the Choco [7] constraint solver to implement the VM provisioning and packing phases as separate constraint solving problems.

Our simulated environment consists a cluster of 4 PMs of capacity (4000 MHz, 4000 MB) which can host the two following applications:

Table 1. Virtual machine classes

VM class	s_1	s_2	s_3	s_4
CPU capacity (MHz)	500	1000	2000	4000
RAM capacity (MB)	500	1000	2000	4000

- Application A is a multiplayer online game where the load of player connections is spread on a cluster on servers. The SLA goal of this application is the average response time of requests. This application has stringent real-time requirements and the associated utility function is shown in Figure 4(b) with a target response time τ_{SLA}.
- Application B is a Web application implemented as a resizable cluster of Web servers. The SLA goal of the application is also the average response time of requests. The workload is measured as the number of requests per second. The utility function of this application is shown in Figure 4(c).

VM configurations are chosen among 4 pre-defined classes as shown in Table 1. We examine how the system behaves by injecting a synthetic workload measured in terms of requests per second which is distributed to applications by a round-robin algorithm. An empirical performance model based on experimental data allows to determine the average response time obtained for a given workload and a given CPU capacity as illustrated in Figure 4(a).

Considering that CPU power is more important than memory, we define the cost function as $Cost(CPU) = CPU_{demand}/CPU_{total}$, i.e. the ratio between the CPU capacity allocated to applications CPU_{demand} and the total physical CPU capacity.

The simulation proceeds according to the following steps on a periodic basis:

1. Workload values (in request/s) for each application are evaluated for the current time.
2. the *VM Provisioning module* is run. For a tentative CPU allocation, this module upcalls a function that computes the global utility as follows:
   (a) The performance model (Figure 4(a)) is used to determine the response time achieved with the CPU capacity. As the model is based on discrete experimental data, the response time for a specific CPU allocation is estimated as the average of the values provided by two nearest curves if the resource amount does not match any curve (e.g: for a demand, if the resource amount is of 3000MHz, the response time is the average of the

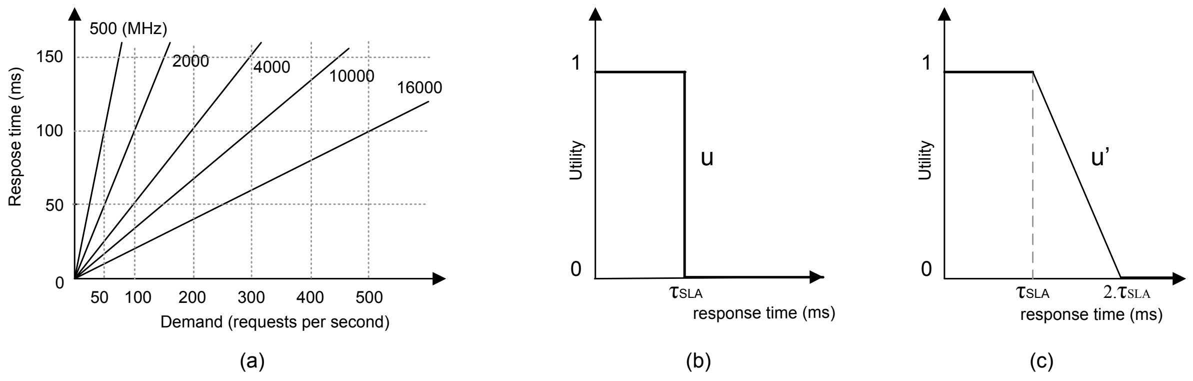

Figure 4. Performance model (a), Utility functions (b) and (c)

results obtained from two curves 2000MHz and 4000MHz).

(b) The utility function of each application (Figures 4(b), 4(c)) provides the local utility from this response time.

(c) After all local utilities have been produced, the global utility is computed as a weighted sum of the local utilities and an operating cost function.

3. The *VM Provisioning module* iteratively explores the space of all possible CPU allocation and picks the solution having the best global utility.

4. The *VM Packing* module logically places VMs on PMs for the solution while minimizing the number of active PMs and the number of required migrations.

5. the simulated time is increased to the next step.

In the first experiment, we nearly do not take into account the operating cost by testing the simulation with $\epsilon = 0.05$ and the parameters shown in Table 2. All these parameters have been presented in Equations 1, 2, 3 and 4. The time series plots are shown in Figure 5. At times t_0 and t_1, as workloads of A and B are both low, CPU demands are negligible, hence there is only one PM needed. Between times t_2 and t_3, workloads increase, so the system tends to attribute more resource (CPU) accordingly. However, the workloads are not too heavy, there are still enough resource to keep response time under the SLA goal ($\tau = 100ms$) for all applications. In this interval, almost all PMs are mobilized. The impact of the application weights α_i is also illustrated between time t_4 and t_5, where demands for application A and B are both high. As $0.8 = \alpha_A > \alpha_B = 0.2$, A has the higher resource allocating priority. Since there is not any CPU resource left to provision to application B, the response time of B exceeds the SLA goal. Note that at these intervals, the global utility has slightly decreased (to 0.85) because of the SLA violation of B. At times t_6, t_7 and t_8 a

Table 2. Experiment setting 1

AE	α	T^{max}	n_1^{max}	n_2^{max}	n_3^{max}	n_4^{max}
A	0.8	17	5	5	5	5
B	0.2	17	5	5	5	5

peak of demand for application A corresponds to a trough of demand for application B and vice-versa. CPU capacity is switched between the two applications to maintain an optimal global utility.

Now we examine how the operating cost factor ϵ affects the global utility value and the allocation result. We keep all parameters to their values defined in the first test but we now set $\epsilon = 0.3$. The result is shown in Figure 6: as the resource demands increases, the global utility value decreases more than during the previous test. The worst value is reached at t_4 and t_5 (0.55) when both applications need a maximum amount of resource. We can see at these intervals and even at t_7, t_8, B descends quicker and than only holds a small amount, as compared to Figure 5, even when nearly all resource was released by A, this behavior is due to the weight $\alpha_B = 0.2$ is negligible against $\alpha_A = 0.8$, and the local utility value of B is not enough to compensate the operating cost which is "boosted" by $\epsilon = 0.3$. Consequently, there are some SLA violations of B, notably at time t_7.

Keeping all other parameters of the first test, in order to show how weight factor α_i affects the resource allocation, we now set $\alpha_A = 0.3$, $\alpha_B = 0.7$. As shown in Figure 7, at times t_4 and t_5, the situation has turned around: thanks to the advantageous weight, B now obtains a sufficient amount of CPU and is able to meet its SLA response time goal. The increase of the response time of A (about 250ms) during high workload intervals is the result of its lower contribution the global utility value.

From a performance point of view, the average solving time for the provisioning phase is 5500ms with $T_A^{max} = T_B^{max} = 17$. The average solving time for the packing

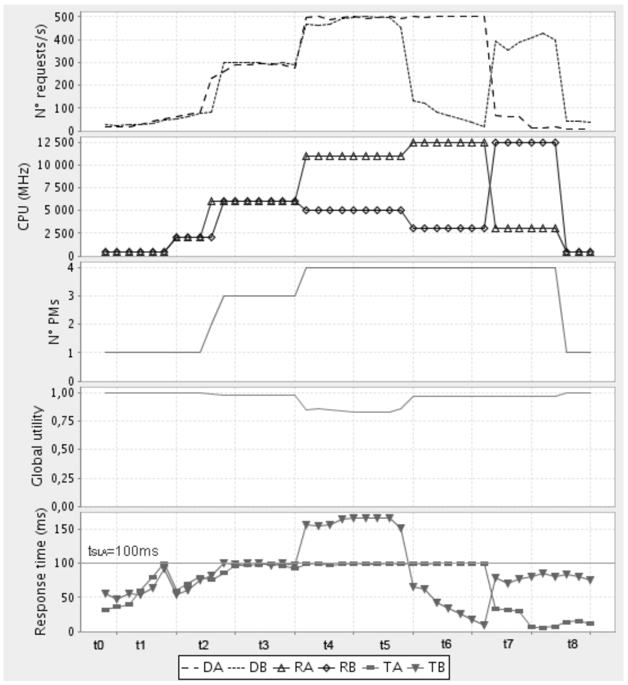

Figure 5. Time series plots of 1) Demands D_A and D_B, 2) CPUs R_A and R_B, 3) Number of active PMs, 4) Global utility, 5) Response times T_A and T_B

.

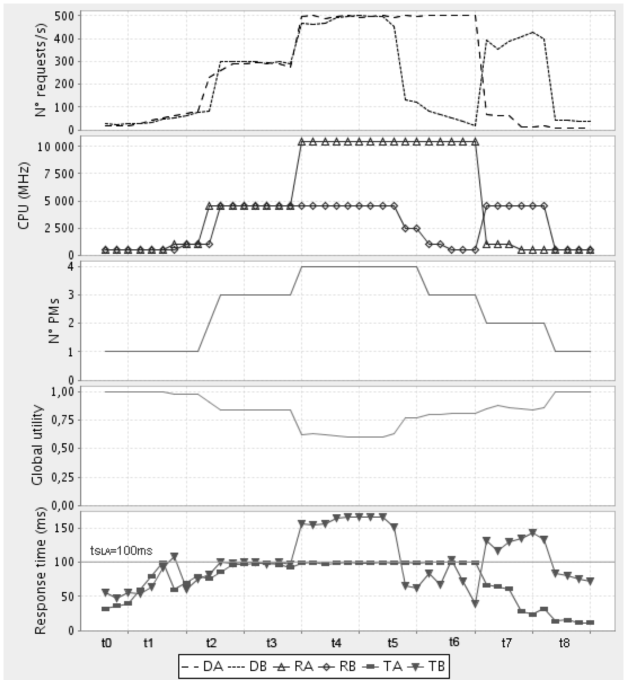

Figure 6. $\alpha_A = 0.8$, $\alpha_B = 0.2$; $\epsilon = 0.3$

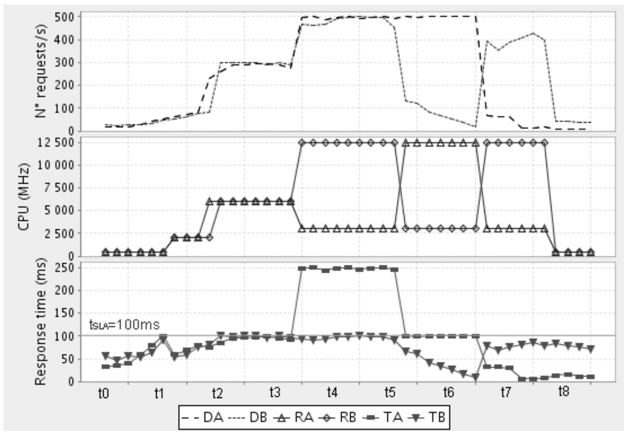

Figure 7. Time series plots of 1) Demands D_A and D_B, 2) CPUs R_A and R_B, 3) Response times T_A and T_B; $\alpha_A = 0.3$, $\alpha_B = 0.7$; $\epsilon = 0.05$

phase is 1000ms in the worst case (the Choco constraint solver was running on a dual-2.5GHz server with 4GB of RAM). A key handle to limit the space of solutions that the constraint solver must explore during the VM provisioning phase is to limit the VM configurations allowed through the T_i^{max} and n_i^{max} parameters taking account the specificities of each application. For example a clusterized application might prefer to have its load spread on a low number of high-capacity servers rather than on a high number of low-capacity servers.

As an illustration if T_A^{max} and T_B^{max} are reduced to 4, the solving time is reduced to 800ms.

The constraint solver aims to explore all possible solutions for a set of input data (demands and constraints) and to return the best solution. To avoid spending a high amount of time to get an "expired" solution which is no longer suitable to the current workload, we limit the time alloted to the solver to get an acceptable solution (which satisfies all constraints of our problem but does not necessarily maximize the global utility).

4 Related Work

Existing works on autonomic management systems for virtualized server environments tackle the allocation and placement of virtual servers from different perspectives. Many papers differ from our work insofar as they either focus on one specific type of applications or they do not consider the problem of dynamic provisioning or the possibility of resource contention between several applications with independent performance goals. For example, [3] proposes a virtual machine placement algorithm which resorts to forecasting techniques and a bin packing heuristic to allocate and place virtual machines while minimizing the number of PMs activated and providing probabilistic SLA guarantees. Sandpiper [4] proposes two approaches for dynamically map VMs on PMs: a black box approach that relies on system-level metrics only and a grey box approach that takes into account application-level metrics along with a queueing model. VM packing is performed through a heuristic which iteratively places the highest-loaded VM on the least-load PM. Some of these mechanisms, for instance prediction mechanisms, could be integrated in our architecture within application-specific local decision modules. Regarding the VM packing problem, we argue that a Constraint Programming approach has many advantages over placement heuristics. Such heuristics are brittle and must be returned with care if new criteria for VM-to-PM assignment are introduced. Moreover these heuristics cannot guarantee that an optimal solution is produced.

A CSP approach for VM packing provides an elegant and flexible approach which can easily be extended to take into account additional constraints. A constraint solving approach is used by Entropy [2] for the dynamic placement of virtual machines on physical machines while minimizing the number of active servers and the number of migrations required to reach a new configuration. Our work extends this system with a dynamic provisioning of VMs directed by high-level SLA goals.

Utility functions act as the foundation of many autonomic resource management systems as a means to quantify the satisfaction of an application with regard to its level of service beyond a binary ”SLA goal satisfied / not satisfied” metric.

[13] lays the groundwork for using utility functions in autonomic system in a generic fashion with a two-level architecture which separates application-specific managers from a resource arbitration level. We specialize this architecture to take server virtualization into account and to integrate resource management costs in the global utility computation.

[8] proposes a utility-based autonomic controller for dynamically allocating CPU power to VM. Like our work, they attempt to maximize a global utility function. But they do not consider the possibility of provisioning additional VMs to an application and restrict themselves to homogeneous application workloads modeled with a queuing network. The CPU capacity of VMs is determined through a beam-search combinatorial search procedure.

[11] uses a simulated annealing approach to determine the configuration (VM quantity & placement) that maximize a global utility. They simplify the VM packing process by assuming that all VM hosted on a PM have an equal amount of capacity. It is worth evaluating the respective performance of simulated annealing and constraint solving.

Shirako [1] proposes an autonomic VM *orchestration* for the mapping between the physical resource providers and virtualized servers. Like our work, they advocate a separation between VM provisioning and VM placement in a federated multi-resource-provider environment.

5 Conclusion & Future Work

This paper addresses the problem of autonomic virtual resource management for hosting service platforms with a two-level architecture which isolates application-specific functions from a generic decision-making layer. We take into account both high-level performance goals of the hosted applications and objectives related to the placement of virtual machines on physical machines. Self-optimization is achieved through a combination of utility functions and a constraint programming approach. The VM provisioning and packing problems are expressed as two *Constraint Satisfaction Problems*. Utility functions provide a high-level way to express and quantify application satisfaction with regard to SLA and to trade-off between multiple objectives which might conflict with one another (e.g. SLA fulfillment and energy consumption). Such an approach avoids the problems encountered by rule- and policy- based systems where conflicting objectives must be handled in an ad-hoc manner by the administrator.

Simulation experiments have been conducted to validate our architecture and algorithms. We are in the process of implementing a test-bed based on a cluster of servers fitted with the Xen hypervisor [18] along with virtual server management service handling the storage, deployment of VM images and the aggregation of performance metrics both at system-level (e.g. CPU load) and at application-level (e.g. response time).

The autonomic management system is being designed a component-based framework with a clear separation between generic mechanisms and pluggable modules.

References

[1] L. Grit, D. Irwin, A. Yumerefendi and J. Chase. Virtual Machine Hosting for Networked Clusters. Proceedings of the 2nd International Workshop on Virtualization Technology in Distributed Computing, 2006.

[2] F. Hermenier, X. Lorca, J.-M. Menaud, G. Muller and J. Lawall. Entropy: a Consolidation Manager for Cluster. In proc. of the 2009 International Conference on Virtual Execution Environments (VEE’09), Mar. 2009.

[3] N. Bobroff, A. Kochut and K. Beaty. Dynamic Placement of Virtual Machines for Managing SLA Violations. 10th IFIP/IEEE International Symposium on Integrated Network Management, May 2007.

[4] T. Wood, P. Shenoy, A. Venkataramani, and M. Yousif. Black-box and Gray-box Strategies for Virtual Machine Migration. 4th USENIX Symposium on Networked Systems Design and Implementation, 2007.

[5] K. Appleby, S. Fakhouri, L. Fong, G. Goldszmidt, M. Kalantar, S. Krishnakumar, D.P. Pazel, J. Pershing, and B. Rochwerger. Océano - SLA Based Management of a Computing Utility. IEEE/IFIP International Symposium on Integrated Network Management Proceedings, 2001.

[6] D. Irwin, J. Chase, L. Grit, A. Yumerefendi, and D. Becker. Sharing Networked Resources with Brokered Leases. Proceedings of the annual conference on USENIX '06 Annual Technical Conference, 2006.

[7] N. Jussien, G. Rochart, X. Lorca. The CHOCO constraint programming solver. CPAIOR'08 workshop on Open-Source Software for Integer and Constraint Programming (OSSICP'08), 2008.

[8] D. A. Menascé and M.N. Bennani. Autonomic Virtualized Environments. Proceedings of the International Conference on Autonomic and Autonomous Systems, 2006.

[9] J. O. Kephart, H. Chan, R. Das, D. W. Levine, G. Tesauro, F. Rawson and C. Lefurgy. Coordinating multiple autonomic managers to achieve specified power-performance tradeoffs. Proceedings of the Fourth International Conference on Autonomic Computing, 2007.

[10] R. Das, G. Tesauro and W. E. Walsh. Model-Based and Model-Free Approaches to Autonomic Resource Allocation. IBM Research Division, TR, November 2005.

[11] X. Wang, D. Lan, G. Wang, X. Fang, M. Ye, Y. Chen and Q. Wang. Appliance-based Autonomic Provisioning Framework for Virtualized Outsourcing Data Center. Autonomic Computing, 2007.

[12] S. Bouchenak, N.D. Palma and D. Hagimont. Autonomic Management of Clustered Applications. IEEE International Conference on Cluster Computing, 2006.

[13] W.E. Walsh, G. Tesauro, J.O. Kephart and R. Das, Utility Functions in Autonomic Systems. Autonomic Computing, 2004.

[14] G. Khanna, K. Beaty, G. Kar and A. Kochut, Application Performance Management in Virtualized Server Environments. Network Operations and Management Symposium. 2006.

[15] T. Kelly, Utility-Directed Allocation, First Workshop on Algorithms and Architectures for Self-Managing Systems. June 2003.

[16] R.P. Doyle, J.S. Chase, O.M. Asad, W. Jin and A.M. Vahdat. Model-based resource provisioning in a web service utility. Proceedings of the 4th conference on USENIX Symposium on Internet Technologies and Systems, 2004.

[17] F. Benhamou, N. Jussien, and B. O'Sullivan. Trends in Constraint Programming. ISTE, London, UK, 2007.

[18] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauery, I. Pratt, A. Warfield. Xen and the Art of Virtualization. Proceedings of the nineteenth ACM symposium on Operating systems principles, 2003.