A Self-Management Framework for Virtual Machine Environments

Dan Marinescu and Markus Schmid

Wiesbaden University of Applied Sciences
Distributed Systems Lab
Kurt-Schumacher-Ring 18, D-65197 Wiesbaden, Germany
{dan.marinescu,schmid}@informatik.fh-wiesbaden.de

Abstract. In this paper, we present our design for a modular framework for autonomic QoS management of virtual machine environments. The framework separates the management APIs provided by individual virtual machine technologies from the high-level control algorithms, which makes the design of generic controllers possible. In addition, control algorithms can be replaced without effort, which provides an easy way for comparing the performance of different control algorithms. We exemplarily discuss a selection of control algorithms that were designed for the use with our framework. In addition, we present the prototypical implementation of the framework and show a number of test results we gained during the validation of the approach. The paper closes with a conclusion and an overview on future work.

1 Motivation

System virtualisation is a technique that was first developed in the mid 1960's. It introduces a layer of indirection between hardware and operating system, called *virtual machine monitor* (VMM). The VMM provides support for creating and running multiple virtual machines (VMs) in parallel that share the underlying hardware. Originally, virtualisation techniques were developed to increase the overall system reliability by isolating individual applications from each other [1]. As a result, a fault in one application cannot corrupt others. A flourishing technology in the 1960's and 1970's, virtualisation almost disappeared in the 1980's and 1990's due to dropping hardware prices.

Over the past years, virtualisation has emerged again, nowadays being used on both, servers and desktop systems [2–4]. The adoption of virtualisation in data centers is taking place at a high pace, as it allows to reduce hardware and maintenance costs by consolidating numerous, mostly under-utilised servers. Another advantage of virtualisation is the introduction of a homogeneous hardware environment (on the VM layer) that results in a significant simplification of the management of individual systems. In addition, virtualisation may reduce system downtimes caused by hardware defects, as VMs can be seamlessly migrated to different physical hosts [5, 6].

However, in general, virtualisation significantly increases the overall complexity of computing systems as administrators have to deal with a potentially much bigger number of (virtual) computer systems that still require adequate monitoring and management. With VMs not depending on the investment in physical hardware, the previously rather static structure of data centers can gain more dynamics. In addition, in a virtualised data center, the overall number of inter-component dependencies increases and wrong decisions can potentially cause bigger damage. At the the same time, virtualised data centers offer a huge potential for optimisation, both in the reduction of physical resource allocation (resulting in reduced costs / "green" data centers) and the possibility to offer a great range of Quality of Service (QoS) classes for provided services.

This indicates, that in the long run, it will not be possible to efficiently handle the management of virtualised data centers without significant automation and the development of self-management approaches [7] that are capable of handling standard management tasks without human interaction. Currently, VMM interfaces for monitoring and reconfiguration of VMs that allow dynamic allocation of resources (memory, CPU shares and others) at run-time, and often even live migration of VMs, provide ideal preconditions for management automation and the introduction of intelligent self-management controllers for QoS enforcement.

This paper presents a modular framework for autonomic QoS management of VM-based application services. The framework separates the management APIs provided by individual VMM technologies from the high-level controllers used for autonomic management, which makes the design of VMM-independent controllers possible.

Traditionally, Service Level Management (SLM) is the discipline concerned with the monitoring and management of processes and applications according to agreed-upon QoS criteria [8–10]. In service provisioning relationships, provider and customer agree on QoS criteria and failure penalties in formal contracts, called Service Level Agreements (SLAs). SLAs contain SLA Parameters that define QoS aspects to consider and Service Level Objectives (SLOs) to be met regarding these parameters.

The paper is structured as follows: Section 2 describes current approaches to the automation of management in VM environments and briefly shows the limitations of these approaches. In section 3 we discuss our modular self-management architecture for VMs, while section 4 shows a number of control algorithms that rely on our management architecture. Section 5 presents the prototypical implementation of this approach in combination with measurements and first practical experiences. Section 6 concludes the paper and gives an overview on future work.

2 Related work

A number of approaches exist that aim at automating the management of virtual environments. In [11], the authors present *VIOLIN*, a policy-based system, which uses the Xen hypervisor for virtualisation. The system consists of one monitor daemon per physical machine and a central adaptation manager. The adaptation manager uses the data gathered by the monitor daemons to request changes in VM resource allocation.

In [12] Grit et al. present a second policy-based management approach. The authors use *Shirako*, a Java-based toolkit for dynamic ressource sharing, to explore algorithmic challenges with regard to policy usage for adaptive VM hosting. Zhang et al. [13] describe a control-theoretical model for VM adaptation, based on the idea that each VM is responsible for adjusting its demand for resources, with respect to efficiency and fairness.

In [14], the authors use a feedback-control strategy to address dynamic resource allocation problems. They employ an infrastructure based on Xen, RUBiS [15] and TPC-W [16]. Here, time series analysis is used to forecast the behaviour of a virtualisation-based system. In [17] Bobroff et al. present a mechanism for dynamic migration of VMs based on a load forecast. Menascè et al. use utility functions for dynamic CPU allocation to virtual machines [18]. The authors test their approach by means of simulations that are based on historical data.

Currently, no commercial solutions exist that are capable of managing VM-based environments in a fully autonomic way. Mainly, existing commercial solutions aim at supporting the work of system administrators, e.g. by providing user-friendly management interfaces.

The previously presented approaches represent first steps taken by the research community to develop strategies for highly specialised, intelligent controllers. It is however hard to objectively evaluate and compare the presented strategies as the authors use different architectures, perform incommensurable tests and some even rely on simulations with historical data to test the performance of their strategies.

We argue that a common, modular framework for the development and evaluation of self-management strategies for virtualisation-based environments is a prerequisite to be able evaluate and compare the efficiency of different controller design approaches. One advantage of using a modular architecture is that control approaches like the ones in [11, 12, 14, 17, 18] can be easily adapted and integrated into the framework. This way, certain aspects of the approaches presented above can be reused. In the following section our approach for such a modular management framework is presented.

3 The Management Framework

3.1 Requirements

A framework for developing and evaluating self-management strategies for virtualisation-based distributed environments must fulfil the following requirements:

(1) Separation of control algorithms from the management framework
(2) Support for different virtualisation technologies
(3) Support for a common evaluation mechanism

(1) basically assures that the management framework is responsible for dealing with aspects like monitoring and execution of tasks, while a separate intelligent controller defines management tasks based on gathered monitoring data. As such, a framework should support different types of intelligent controllers by providing a generic controller interface.

(2) requires the framework to support different virtualisation technologies (e.g. VMware ESX or Xen), transparently for the controller. Thus, from a controller's perspective, the framework is acting as an abstraction layer on top of the virtualisation technology.

(3) means that the framework has to support a consistent mechanism for tracing and comparing of management decisions in order to evaluate different self-management approaches against each other.

3.2 Overall Architecture

We have designed a management framework for VM environments that fulfils the requirements discussed above. The framework is used to manage a cluster of n physical machines, each hosting between 0 and m VMs. Fig. 1 depicts the overall architecture of the framework. It comprises three types of management components: a *VM Manager* is responsible for a VM, one *Physical Manager* is assigned to each physical machine and a *Cluster Manager* takes care of the entire cluster. These management components are described in detail in the following paragraphs.

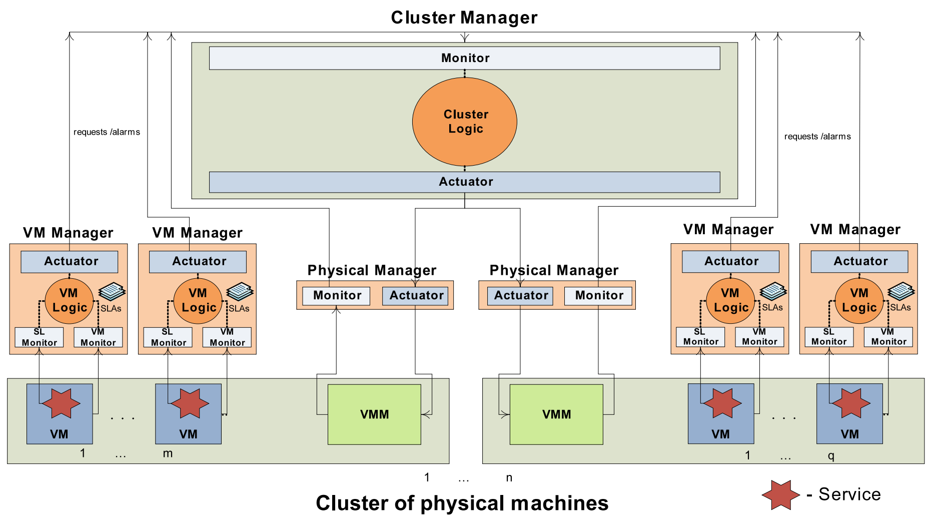

Fig. 1. Overall architecture of the management framework

The VM Manager component monitors and controls both, the OS and the applications of a VM. It monitors OS parameters like CPU utilisation and used/available memory trough its *VM Monitor* module. In addition, a *SL Monitor* module monitors specific quality of service parameters regarding the service hosted by the VM. We assume that each VM hosts only a single service, e.g. Web server, mail server or DBMS. We argue that this is common practise in a server consolidation scenario. Furthermore, we define service level objectives (SLOs) for the services provided by the VMs. The *VM Logic* module uses the data obtained from the SL Monitor to detect SLO violations. In case a SLO violation occurs, the VM Logic module uses the data gathered from the VM Monitor to determine the resource bottleneck (e.g. CPU or memory) that causes the problem. After the bottleneck has been identified, the VM Manager informs the responsible Cluster Manager about the resource bottleneck through its *Actuator* module.

The Physical Manager component controls a physical machine, basically by executing two different tasks: it uses a *Monitor* module to observe the resource utilisation of the machine and forwards the collected data to the Cluster Manager. In addition, the Physical Manager executes management actions that are requested by the Cluster Manager through its Actuator module, e.g. resource reallocations or migration of VMs to a different physical machine.

The Cluster Manager component comprises a Monitor module which receives monitoring data from the Physical Managers within the cluster. It uses the aggregated data to create a global view of resource usage and availability in the managed cluster. This global view is used by the *Cluster Logic* module to fulfil (VM Manager) requests for additional resources. Having determined a way to reallocate resources for VMs in the cluster, the Cluster Manager uses its Actuator component to request Physical Managers to perform the required resource allocation changes.

3.3 Internal Architecture of a Manager Component

In compliance with the IBM reference architecture, we have developed a modular *Self-Manager* that provides a customisable basis for the VM Managers, the Physical Managers and the Cluster Manager (see fig. 2 for details).

Internally, the Self-Manager consists of four main components: The central element of the Self-Manager is the `management kernel`, which provides `adapters` to connect extension modules. Extension modules are instantiated by the central `module manager.` The modules implement sensors, effectors, and the internal logic of the self-management controller. A `messaging subsystem`, which is part of the kernel, is responsible for message handling within the Self-Manager.

The Self-Manager supports three kinds of extension modules: `event modules`, `action modules`, and `control modules`. `Event modules` create their own threads and thus are able to react actively to changes within the environment, e.g. by creating internal messages. `Action modules` are passive; they act – triggered by internal messages – by analysing application-specific sensors, or performing management tasks. Sensors can be realised using either `event modules` (push model) or `action modules` (pull model). Application-specific actuators are realised through `action modules`. The modules implement application-class specific interfaces, e.g. for accessing the control API of a VMM.

`Control modules` form the "brain" of the Self-Manager as they host the management knowledge and implement the control algorithms. `Control modules` act periodically or are triggered by incoming messages. Management decisions are communicated to other modules using the internal messaging capabilities.

Internally, VM Managers and the Cluster Manager comprise a control module and several action and event modules, while the Physical Managers solely consist of action and event modules. A number of controller modules are presented in the following section. The architecture uses Java RMI-based event modules for inter-manager communication.

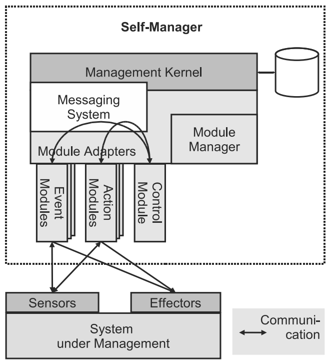

Fig. 2. Modular Self-Manager architecture

The management framework can be easily adapted to different virtualisation technologies by simply replacing the technology-dependent action and event modules in the VM Managers and the Physical Managers.

3.4 Evaluation of the Framework

It can be easily observed that this framework fulfils the previously defined requirements. Various Cluster Logic components that use different self-management strategies can be plugged into the framework. Furthermore, different virtualisation technologies can be addressed by simply providing necessary adaptor modules for each virtualisation technology. Last, it is possible to use standard benchmarking tools to generate load for application services provisioned in the VMs and then utilise the VM Monitor component to evaluate their behaviour. This way, different intelligent controller approaches can be evaluated with respect to service performance under defined loads.

For our management framework, we look at an architecture with three types of entities/components: VMs, physical machines and the cluster itself. Since in our management framework each of these entities is equipped with an autonomic manager, our system is composed only of autonomic components. Compared to other architectural approaches for self-managing systems, our architecture resembles the approach proposed by White et. al in [19]. Furthermore, the autonomic manager of each component continuously executes a MAPE (Monitor, Analyse, Plan, Execute) loop. This conforms to the IBM vision [7] of how autonomic computing systems should operate.

4 Autonomic Controllers for the Management Framework

To demonstrate the flexibility and adaptability of our management framework we implemented a number of self-management control approaches for both, the VM

Logic and Cluster Logic components. All approaches presented in the following use *service response time* as example SLA parameter for service monitoring. The use of other SLA parameters (e.g. throughput) is however possible.

4.1 Autonomic Management for a VM Logic Component

The purpose of a VM Logic component is to monitor and analyse the resource utilisation of a VM and the response times of the service hosted by the VM. Once service response times exceed a given threshold (the SLO), the VM Logic component uses the data representing the resource utilisation of the VM to determine the root cause of this increase. Thus, the VM Logic component performs a bottleneck determination process, just like a human system administrator would do. The controllers' knowledge about the managed system is represented as a number of rules, such as:

facts : SLO violation occurred, the OS is swapping
action : allocate additional memory

The bottleneck-determination process is implemented as a rule-based expert system. At runtime, the rule engine uses input data like service response times and VM resource utilisation to try to match the facts of a rule. If all facts of a rule match, the corresponding action is performed. For the given example, the VM Logic Component sends a request for additional memory to the Cluster Manager, which then evaluates all incoming requests and takes appropriate actions from its global perspective.

4.2 Autonomic Management for the Cluster Logic component

Due to the abstraction layer provided by the VM Managers and the Physical Managers, the Cluster Manager solely deals with the problem of satisfying the resource requests of the different VM Managers. Thus, at this abstraction level, we see resource consumers (the VMs), and resource providers (the physical machines). We thus aim to satisfy the needs of the resource consumers using the resources supplied by the resource providers.

Various approaches can be used to solve this resource allocation problem. We have previously argued that an evaluation of different self-management approaches can be easily achieved using our framework. To back this statement, we present two different heuristic-based algorithms that try to solve our resource allocation problem.

Algorithm A first checks whether the additional resource share required by a VM is available on the physical machine that currently provisions the VM. If this is the case, the algorithm gives instructions to allocate the additional resource share to the VM and terminates. If the requested resource share is not available on the current physical machine, the algorithm looks for a different physical machine in the cluster that provides enough spare resources for hosting the requesting

VM. This means that not only the requested resource share must be available, but in addition all the resources currently assigned to the VM. If such a remote physical machine is found, the VM is migrated to its new destination and the algorithm terminates. The selected destination machine can be either online or in an offline state. However, destinations that are currently offline cause higher migration costs, due to the time span needed for powering up and booting. In case no suitable destination is found, the algorithm fails to find a solution.

This algorithm uses a "first fit" strategy. The disadvantages of Algorithm A are obvious: besides not always being capable of finding a solution, the solutions found by the algorithm can sometimes be quite far from optimal. On the other hand, the algorithm has a linear complexity. The solutions found the by Algorithm A, although not always optimal, represent states that can be achieved by performing a single migration from the initial state. This ensures that no costly solution will ever be suggested by the algorithm.

Algorithm B uses a local search approach based on the K-Best-First Search (KBFS) algorithm, first introduced by Felner et al. [20]. Our algorithm encodes the mapping of VMs to physical machines in the cluster in form of states. In addition, the encoding includes allocated and available resources of each physical machine. Each state has its global profit, representing a function of the sum of the local profits of each physical machine and the validity of the state.

The local profit of a physical machine represents a function of its resource utilisation. Thus, the higher the resource utilisation of a machine, the higher the local profit. An offline physical machine has the maximum local profit possible. This assures that the global profit of a state increases when some machines are kept at a high resource utilisation while the others are kept offline.

Global profit is calculated by multiplying the sum of the local profits with the validity of the state, which can be either `0` or `1`. Thus, the global profit can either be the sum of the local profits if the state is valid, or `0` if the state is not valid. A valid state is a state where the free capacity (e.g. in terms of memory, CPU) of any physical machine is bigger or equal to the sum of requested capacities of the VMs provisioned on that physical machine. All other states are invalid. As a result, Algorithm B defines a valid gobal state as a state with a global profit higher than `0`. However, valid states can also be ranked, depending on their global profit.

Besides the global profit of a state, Algorithm B also computes the total cost of moving from the initial state to any given state. The total cost is a function of the number of migrations necessary for the translation from the initial state to that given state. The total cost is then used in combination with the global profit to determine the utility of a translation between an initial state and a given state. This is the ultimate quality factor that is used to select the final state.

Using this encoding and criteria, Algorithm B performs its local search. For this, two lists are used: the *open list* of states (nodes) that have been evaluated with respect to their utility but not expanded, and the *closed list* which contains the nodes that have already been expanded. At each iteration, the best k nodes from the open list are expanded, their children are evaluated and added to the open list. After a limited number of iterations, the algorithm terminates and the node with the highest utility is selected. This represents the final state, and the system moves to that state by means of successive migrations.

Algorithm B is always capable to find a valid solution if valid solutions exist and the translation costs to at least one of these solutions does not exceed a certain, predefined value. However, in finding a solution the algorithm determines the local optimum which is not necessarily the global optimal solution.

5 Implementation and Experiences

We have successfully implemented a Java-based prototype of the management framework. The prototype is able to autonomically manage VMs based on the Xen hypervisor. The implementation comprises sensors and effectors to access the Xen API [21] for VM reconfiguration and migration. VM parameters and application response time metrics are monitored using a JMX [22] interface.

The Physical Managers use the Xen-API to manage a physical machine that hosts an instance of the *Xen 3.1* hypervisor. The Xen-API is used for monitoring, dynamic resource allocation and to perform live migration. Both the VM Manager and the Physical Manager communicate with the Cluster Manager using Java RMI. The Cluster Manager provides support for plugging-in different Cluster Logic components through a messaging interface.

Examples for VM Logic and Cluster Logic components have been implemented as presented in section 4. We used the rule engine *Jess* in the implementation of the VM Logic approach.

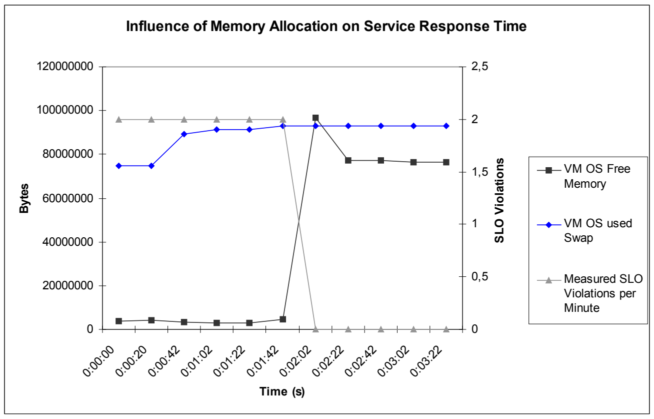

Fig. 3. Influence of available memory on the number of SLO violations

We have tested the framework using a Java implementation of the TPC-W benchmark.The TPC-W e-commerce application suite is provisioned by a

Tomcat server that runs in a Xen-based VM. The associated VM Manager enforces an SLO, which defines a maximum response time of 3000ms, tolerating a maximum of three exceptions within a two minute period. As a TPC-W load generator is started on a client system, the VM Manager observes a dramatic increase in response times, resulting in an SLO violation. The manager determines the amount of memory assigned to the VM being the bottleneck. Thus, the VM Manager requests additional memory from the Cluster Manager. As this request is fulfilled, the VM Manager observes a significant decrease in response times down to an acceptable level. This behaviour can be observed in fig. 3. The decrease in SLO violations per minute shows that the VM Manager is able to successfully determine the resource that caused the bottleneck.

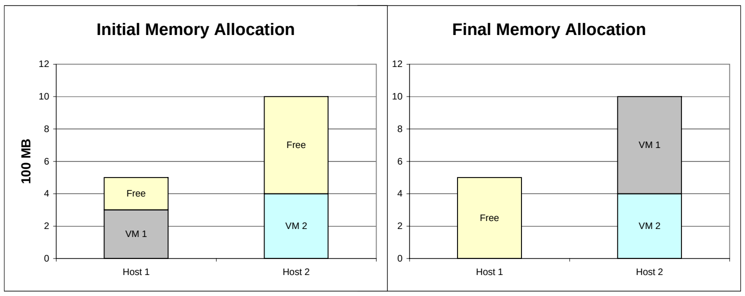

Fig. 4. A management scenario comprising two physical machines

We evaluated the two Cluster Logic algorithms described in section 4 live and by means of simulation. Since in the previous test the VM Manager has determined memory to be the bottleneck resource, we concentrated on memory allocation. In the following we describe our experiences in two scenarios.

In the first scenario, two physical machines named Host 1 and Host 2 each provision one VM, VM 1 respectively VM 2 (see fig. 4). Host 1 possesses a total of 500 MB RAM for VMs, while Host 2 provides 1000 MB. At the beginning, VM 1 uses 300 MB, while VM 2 uses 400 MB. As the TPC-W load generator stresses VM 1, the VM Manager responsible for VM 1 requests additional memory (in this case an extra 300 MB) from the Cluster Manager. In this scenario, both Cluster Logic algorithms deliver the same solution, namely to migrate VM 1 from Host 1 to Host 2 as Host 1 is not able to provide extra 300 MB of memory.

The second scenario comprises three physical machines: Host 1, Host 2 and Host 3 (see fig. 5, left chart). At the beginning, each of them provisions one VM: VM 1, VM 2 and VM 3 respectively. In this scenario, Host 1 possesses a total of 500 MB RAM for VMs, Host 2 600 MB and Host 3 1100 MB, while VM 1 uses 300 MB, VM 2 400 MB and VM 3 600 MB. Again, the VM Manager responsible for VM 1 requests 300 MB of additional memory. Unlike in the first scenario, the two algorithms perform differently. In fact, Algorithm A is not even capable of finding a valid solution as neither Host 2 nor Host 3 have 600 MB of free memory

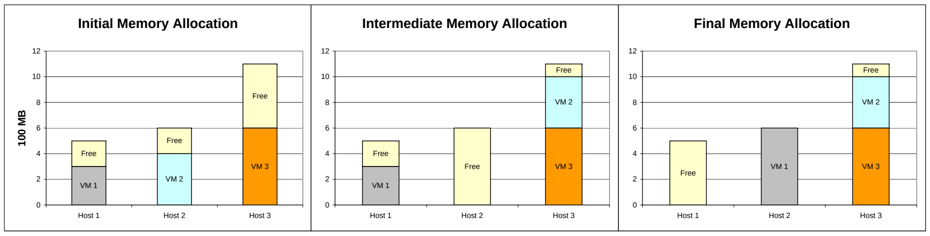

Fig. 5. A management scenario comprising three physical machines

available. This scenario clearly shown the limitations of this simple algorithm. The more sophisticated Algorithm B is able to find an optimal solution: VM 2 is migrated to Host 3 before VM 1 is migrated to Host 2. As a side effect, this solution results in a high resource utilisation on both, Host 2 and Host 3, while Host 1 provisions no VM and thus can be set offline. The evolution of Algorithm B from the initial state to this final state can be observed in fig. 5.

6 Conclusions and Future Work

We designed a modular framework for autonomic QoS management of VM-based application services. The framework separates the management APIs provided by individual VMM technologies from the high-level controllers used for autonomic management, which makes the design of VMM-independent controllers possible. In addition, control algorithms can be easily replaced, which provides an easy way for comparing the performance of different control algorithms.

The framework has been prototypically implemented and has been successfully evaluated using different approaches for autonomic control algorithms. Experiences gained from initial tests and simulations prove the feasibility of our approach. Results also show that our self-management framework can be used to successfully test and evaluate different self-management control approaches that can be implemented independent from the underlying virtualisation technology. This also provides the basis for further optimisation of controllers to support additional strategies, e.g. to try to accumulate existing VMs only on a subset of the physical machines available in the cluster in order to support "green datacenter" strategies by minimising the number of hosts to be kept online.

Future work will concentrate on two main tasks: further research in appropriate control algorithms is needed as we determined that in general our Cluster Logic components are faced with a multi-dimensional multiple knapsack problem, which is NP-hard. We will work on the definition of constraints that will allow us to simplify this problem and will hopefully lead to control approaches that at least grant a good (if not optimal) solution. In addition we will work on the integration of the management approach for VMs with an existing architecture for decentralised SLM of SOA workflows and services, which is currently developed in our lab.

References

1. Rosenblum, M., Garfinkel, T.: Virtual machine monitors: current technology and future trends. Computer **38**(5) (2005) 39–47
2. VMware ESX. `http://www.vmware.com/products/vi/esx/` (l. visited 12/2007).
3. Xen Source. `http://www.xensource.com/` (l. visited 12/2007).
4. Virtual Box. `http://www.virtualbox.org/` (l. visited 12/2007).
5. Bressoud, T.C., Schneider, F.B.: Hypervisor-based fault tolerance. In: SOSP '95: Proceedings of the 15th ACM symposium on Operating systems principles. (1995)
6. Cox, A.L., Mohanram, K., Rixner, S.: Dependable unaffordable. In: ASID '06: Proceedings of the 1st workshop on Architectural and system support for improving software dependability, New York, NY, USA, ACM Press (2006) 58–62
7. Kephart, J.O., Chess, D.M.: The vision of autonomic computing. Computer **36** (2003) 41–50
8. Sturm, R., Morris, W., Jander, M.: Foundations of Service Level Management. SAMS Publishing (April 2000)
9. Lewis, L.: Service Level Management for Enterprise Networks. Artech House Publishers (1999)
10. Verma, D.: Supporting Service Level Agreements on IP Networks. Macmillan Technical Publishing (1999)
11. Ruth, P., Rhee, J., Xu, D., Kennell, R., Goasguen, S.: Autonomic live adaptation of virtual computational environments in a multi-domain infrastructure. In: IEEE International Conference on Autonomic Computing. (2006)
12. Grit, L., Irwin, D., Yumerefendi, A., Chase, J.: Virtual machine hosting for networked clusters: Building the foundations for autonomic orchestration. In: Proceedings of the 2nd International Workshop on Virtualization Technology in Distributed Computing, IEEE (2006)
13. Zhang, Y., Bestavros, A., Guirguis, M., Matta, I., West, R.: Friendly virtual machines: leveraging a feedback-control model for application adaptation. In: VEE '05: Proceedings of the 1st ACM/USENIX international conference on Virtual execution environments. (2005)
14. Padala, P., Shin, K.G., Zhu, X., Uysal, M., Wang, Z., Singhal, S., Merchant, A., Salem, K.: Adaptive control of virtualized resources in utility computing environments. In: EuroSys '07: Proceedings of the 2007 conference on EuroSys. (2007)
15. RUBiS. `http://rubis.objectweb.org/` (l. visited 12/2007).
16. TPC-W. `http://www.tpc.org/tpcw/` (l. visited 12/2007).
17. Bobroff, N., Kochut, A., Beaty, K.: Dynamic placement of virtual machines for managing sla violations. In: Integrated Network Management, 2007. IM '07. 10th IFIP/IEEE International Symposium on. (2007) 119–128
18. Menasce, D.A., Bennani, M.N.: Autonomic virtualized environments. In: ICAS '06: Proceedings of the International Conference on Autonomic and Autonomous Systems, Washington, DC, USA, IEEE Computer Society (2006) 28
19. White, S.R., Hanson, J.E., Whalley, I., Chess, D.M., Kephart, J.O.: An architectural approach to autonomic computing. In: Autonomic Computing, 2004. Proceedings. International Conference on. (2004) 2–9
20. Felner, A., Kraus, S., Korf, R.E.: Kbfs: K-best-first search. Annals of Mathematics and Artificial Intelligence **39**(1) (2003) 19–39
21. The Xen API. `http://wiki.xensource.com/xenwiki/XenApi` (l. visited 12/2007).
22. Java Management Extensions. `http://java.sun.com/javase/technologies/core/mntr-mgmt/javamanagement/` (l. visited 12/2007).