Two levels autonomic resource management in virtualized IaaS

Alain-Bouzaïde Tchana, Giang Son Tran, Laurent Broto, Noel Depalma, Daniel Hagimont

To cite this version:

Alain-Bouzaïde Tchana, Giang Son Tran, Laurent Broto, Noel Depalma, Daniel Hagimont. Two levels autonomic resource management in virtualized IaaS. Future Generation Computer Systems, Elsevier, 2013, vol. 29 (n 6), pp. 1319-1332. <10.1016/j.future.2013.02.002>. <hal-01131180>

HAL Id: hal-01131180
https://hal.archives-ouvertes.fr/hal-01131180
Submitted on 13 Mar 2015

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L’archive ouverte pluridisciplinaire HAL, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d’enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.
Open Archive TOULOUSE Archive Ouverte (OATAO)

OATAO is an open access repository that collects the work of Toulouse researchers and makes it freely available over the web where possible.

This is an author-deposited version published in: http://oatao.univ-toulouse.fr

Eprints ID: 12507

To link to this article: DOI: 10.1016/j.future.2013.02.002
URL: http://dx.doi.org/10.1016/j.future.2013.02.002

To cite this version: Tchana, Alain-Bouzaïde and Tran, Giang Son and Broto, Laurent and Depalma, Noel and Hagimont, Daniel Two levels autonomic resource management in virtualized IaaS. (2013) Future Generation Computer Systems, vol. 29 (n° 6). pp. 1319-1332. ISSN 0167-739X

Any correspondence concerning this service should be sent to the repository administrator: staff-oatao@listes-diff.inp-toulouse.fr
Two levels autonomic resource management in virtualized IaaS

Alain Tchana, Giang Son Tran, Laurent Broto, Noel DePalma, Daniel Hagimont

A B S T R A C T

Virtualized cloud infrastructures are very popular as they allow resource mutualization and therefore cost reduction. For cloud providers, minimizing the number of used resources is one of the main services that such environments must ensure. Cloud customers are also concerned with the minimization of used resources in the cloud since they want to reduce their invoice. Thus, resource management in the cloud should be considered by the cloud provider at the virtualization level and by the cloud customers at the application level. Many research works investigate resource management strategies in these two levels. Most of them study virtual machine consolidation (according to the virtualized infrastructure utilization rate) at the virtualized level and dynamic application sizing (according to its workload) at the application level. However, these strategies are studied separately. In this article, we show that virtual machine consolidation and dynamic application sizing are complementary. We show the efficiency of the combination of these two strategies, in reducing resource usage and keeping an application’s Quality of Service. Our demonstration is done by comparing the evaluation of three resource management strategies (implemented at the virtualization level only, at the application level only, or complementary at both levels) in a private cloud infrastructure, hosting typical JEE web applications (evaluated with the RUBiS benchmark).

1. Introduction

In order to reduce the maintenance cost of computing environments, companies are increasingly externalizing their computing infrastructures to specific companies called providers. These later are expected to ensure quality of service (QoS) for their customers while minimizing hosting costs. Cloud computing is a recent paradigm which follows this direction.

In this context, on demand resource management is one of the main services that such an environment must ensure. It must allow the allocation of resource as needed and resource deallocation when unused, while reducing the number of used machines and therefore energy consumption. Resource usage reduction also concerns cloud customers since they want an efficient service while using the minimum number of resources in the cloud (which impacts their invoice).

Many research and industry works ([1,2], or [3]) have investigated resource management strategies in the cloud. These strategies can be grouped into two categories: those which are implemented at the customer applications level through dynamic application sizing (according to its workload) and others that are implemented at the virtualized level through virtual machine consolidation (according to the cloud infrastructure utilization rate). However, (1) these strategies have been studied and experimented separately, and (2) they do not consider each level particularity regarding resource management.

In this article, we investigate the implementation of resource management strategies at both levels simultaneously in the case of an IaaS (Infrastructure-as-a-Service) cloud model hosting master–slave applications. We show that virtual machine consolidation and dynamic application sizing can be beneficently combined in order to reduce resource usage and to keep application’s Quality of Service (QoS). Our study is done by comparing the evaluation of three resource management strategies (implemented at the virtualization level only, at the application level only, and complementary at both levels) in a private IaaS, hosting typical JEE web applications (evaluated with the RUBiS [4] benchmark). Our private IaaS cloud infrastructure automates the execution of these strategies with an autonomic management system called TUNE [5].

The rest of this paper is organized as follows. Section 2 presents the context of this work. Section 3 presents our motivations for two-level resource management in a cloud infrastructure. Section 4 presents the environment in which we conducted our experiments. Sections 5–8 detail, evaluate and synthesize the resource
management strategies we study in this article. Section 9 presents the various related works. Finally, we conclude and present future works in Section 10.

2. Context

In this section, we introduce the context of cloud environments in which our experiments take place and the type of applications we are considering.

2.1. Cloud computing

Using a cloud computing environment, an organization (customer) can greatly reduce maintenance costs by relying on an external institution, called provider. Resources in a cloud computing environment are dynamically provisioned and must satisfy service-level agreements between the provider and its customers [6].

In the literature, cloud infrastructures are generally classified into three models:
- Infrastructure-as-a-Service (IaaS): a virtualized infrastructure managed by a provider, in which external customers deploy and execute their applications;
- Platform-as-a-Service (PaaS): a virtualized infrastructure managed by a provider, in which customers develop, deploy and execute their applications using the application development environment provided by the provider;
- Software-as-a-Service (SaaS): the provider gives to customers the application they want.

In this paper, we consider the cloud as an IaaS. The actors involved in such a cloud platform are grouped into three categories: cloud providers, cloud customers and end users. A cloud provider is responsible for the administration of the cloud resources (e.g. hardware, network, virtual machine) and services. Cloud customers use the provided cloud resources to deploy and execute their applications (e.g. JEE applications in our example). Cloud customers are provided with VMs, without having a global view or direct access to the cloud physical infrastructure. These VMs host the customer’s applications and represent a confined portion of physical resource. End users use customer applications deployed in the cloud.

Fig. 1 summarizes these cloud layers and the scope of each cloud participant: the cloud provider has access to physical resources and VMs; cloud customers have access to VMs and their applications; and end users have access to the customer applications.

2.2. Master–slave applications: example of JEE applications

Typical web applications in Java 2 Platform Enterprise Edition (JEE) are designed as master–slave architectures. This type of application represents the commonly hosted applications in cloud platforms. Its design consists of a web server tier (e.g. Apache), an application server tier (e.g. Tomcat) and a database server tier (e.g. MySQL). When an HTTP request is received, it refers either to a static web document (e.g. HTML, CSS), in which case the web server directly returns the requested document to the client; or to a dynamically generated document, in which case the web server forwards the request to the application server. In turn, the application server executes requested application components (e.g. Servlets, EJBs), creating queries to a database through a JDBC driver (Java DataBase Connection driver). Finally, queried data from the database is processed by the application server to generate a web document which is returned to the client. Fig. 2 summarizes this architecture of an JEE application.

In this context, the increasing number of Internet users has led to the need for highly available services. To face high loads of Internet services, a commonly used approach is the replication of servers. This kind of approach usually defines a particular software component in front of each set of replicated servers, which dynamically balances the load among the replicas using different algorithms (e.g. Random choice, Round Robin). Server replication is a way to implement elastic configuration through dynamic sizing of the number of replicas.

Here, an important remark is that these JEE applications are mostly CPU bound, i.e. the goal of dynamic sizing is the balance the CPU load between a variable number of replicas.

3. Motivations

As any business activity, the main objective of a cloud provider is to make a profit while meeting customers requirements (which essentially means to guarantee that all their VMs will receive the contracted resources). This objective can be achieved in two ways.

Firstly, the provider can strictly keep allocated all resources the customer has contracted (for the entire VM lifetime). Consequently, each VM will keep its resources even if it does nothing.
Since this way reduces the cloud mutualization capacity (it is not possible to group idle VMs on fewer machines because their resources are booked), VM allocation in the cloud will be expensive.

In the second way, the provider ensures to customers that their VMs will received all resources they need (in the limit of the contract) only when the need is effective. Thus, the provider can increase his hosting capacity by collocating VMs (using VM migration) when they are underused, and relocating (spreading) them when they are effectively used. Consequently, VM allocation can be cheaper since the provider makes profit by increasing his hosting capacity (which implies hosting more customer applications). VM collocation and relocation are known as VM consolidation strategies.

In short, cloud providers can provide two VM reservation modes:

- **Hard VM reservation.** Resources are kept allocated.
- **Soft VM reservation.** Resources are only allocated when effectively used.

At the application level, customers can either implement a static or a dynamic configuration strategy.

With a static configuration strategy, a customer will start the number of server replicas needed to face expected peak loads in order to ensure application QoS. Each server replica is created in a separate VM. Then, the allocated VMs are not always intensively active and the customer can therefore rely on Soft VMs. The IaaS consolidation system will dynamically allocate resource to these VMs according to their load.

With a dynamic configuration strategy, a customer monitors VMs load and adapts the degree of replication according to this load, adding a new replica (on a new VM) when the application workload exceeds a threshold; and removing a replica when the load decreases. In this case, this dynamic policy ensures that VMs are effectively used and the customer can therefore rely on Hard VMs.

In summary, two resources management strategies can be implemented in a cloud running master–slave applications:

- Static application configuration with Soft VMs.
- Dynamic application configuration with Hard VMs.

These two strategies are analyzed below.

### 3.1. Static application configuration with Soft VMs

Let us consider a situation where the cloud runs two multi-tier applications (an Apache web server linked to one or many Tomcat application servers), owned by different customers. Each tier is run on a separate VM which corresponds to a fraction of a physical machine. Applications are statically dimensioned with the maximum number of replicas needed to maintain their QoS under peak loads: three Tomcat instances for each application are initially allocated to serve peak loads. Fig. 3(a) shows an overloaded state of the applications. As applications become underloaded, VM consolidation progressively gathers VMs on a minimal number of physical machines (Fig. 3(b)). However, as argued in [7], an important limitation of VM consolidation is memory capacity: any VM, even idle, consumes physical memory, which limits the number of VMs that can be hosted on a machine. Therefore, VM
consolidation can reduce, but cannot minimize the number of active machines in the hosting center.

Dynamic application configuration (sizing) would allow to significantly reduce the number of physical machines in use (Fig. 3(c)).

3.2. Dynamic application configuration with Hard VMs

We consider an initial situation where each application is configured with one Tomcat server (Fig. 4(a)). The customers execute two sizing operations: size up of the two applications (each has three Tomcats) facing a peak load (Fig. 4(b)), followed by a size down of the second application when it becomes underloaded (Fig. 4(c)). This last operation leaves two unused spaces in the last two physical machines. These holes can be easily optimized if VM consolidation is implemented (Fig. 4(d)), resulting in a physical machine being freed for energy saving.

Dynamic application sizing ensures that VM resources are effectively used (else VM would be deallocated), so that this solution can rely on Hard VM allocation. Since VMs are effectively used, the VM consolidation algorithm does not have to monitor CPU. It simply packs VMs on the minimal number of physical machines according to the resource allocated to these Hard VMs. We observe here that this solution combines both application sizing and VM consolidation, but the consolidation system is in this case much simpler.

This analysis shows the theoretical advantage of two-level resource management. In the following, we describe an implementation of the above strategies and an evaluation in a private cloud infrastructure.

4. Evaluation context

This section describes the IaaS environment used to evaluate the resource management strategies presented above. Our experimental context is a private virtualized infrastructure, equipped with an autonomic administration system (called TUNE) which provides self-* capabilities.
4.1. TUNe: an adaptable autonomic administration system

The TUNE [5] system aims at bringing a solution to the increasing complexity in distributed software management. It is based on the concept of autonomic computing. An autonomic management system automates the management of applications without human intervention. It covers applications’ life cycle: deployment, configuration, launching, and dynamic management. TUNE allows the definition of managers which monitor the execution conditions and react to events such as failures or peak loads, in order to adapt the managed application accordingly and autonomously.

TUNE has been experimented in various application domains: web and middleware architectures [8], grid computing systems [9] and energy saving in replicated systems [10].

This adaptability to different domains made possible thanks to the component-based approach used to design TUNE. Each legacy software is wrapped inside a software component, and is managed by its component’s management interface. TUNE supports the definition of new description languages to define software and hardware architectures as well as system management policies. For example, TUNE’s deployment policies are highly adaptable, from binary file transfer to NFS remote directory mounting. The definition of reconfiguration policies is also flexible, allowing to implement autonomic management behaviors for application software as well as operating systems (virtual machines). In summary, TUNE is designed and developed with pluggable components, even administrative services are implemented as components. Thus, in the context of this article, we were able to adapt TUNE to manage both customer applications (with dynamic sizing) and the virtualized cloud environments (with consolidation).

4.2. Experimental JEE application

The RUBiS[4] benchmark (version 1.4.2) is used as our customer application throughout this article. RUBiS is an implementation of an eBay-like auction system, including a workload generator to simulate web clients. RUBiS is an implementation of a JEE benchmark. In our experiments, our RUBiS application is composed of an Apache web server, a Tomcat servlet container, a MySQL-Proxy to load balance SQL requests among a set of MySQL database servers. We limit our evaluation to the MySQL tier due to paper length restriction. Moreover, as identified by [11], the database tier is the primary bottleneck tier in such applications. The workload submitted to the RUBiS applications is database bound and our sizing policies are applied only on the MySQL tier. However, we treated in a previous work [12] the dynamic sizing of each JEE tier.

For these experiments, we consider that our IaaS infrastructure hosts two RUBiS applications which belong to different customers.

4.3. Experimental environment

Our experiments were carried out using the Grid’5000 [13]1 experimental testbed (the French national grid), using Xen [14] 3.2 as the virtualization platform. All nodes use Linux CentOS distribution, running on a Dell PowerEdge R410 Intel Xeon ES620 2.4 GHz, and are connected through a gigabyte Ethernet LAN connection from a cluster. VM disk images shared by all the nodes are located on a NFS server. To summarize, the use of TUNE running over a Grid’5000’s cluster is considered as our IaaS. The number of nodes during these experiments varies according to the dynamic changes of the workload submitted to the RUBiS application. For the two experimented applications, we use 3 nodes to run VMs (a total of 6 VMs) which host frontend servers (Apache, Tomcat and MySQL-Proxy), and up to 3 nodes (when the two applications receive their maximum load) to run VMs which host backend servers (MySQL, up to 6 VMs).

4.4. Experimental procedure

The goal of our experiments is to demonstrate the benefit of combining an application level policy (dynamic sizing) and an IaaS level policy (consolidation). These experiments are done in three phases.

• In a first step, we execute a scenario (two RUBiS applications with a given workload) with static application configurations and Soft VMs. This means that here, resources are only managed by the consolidation system in the IaaS.

• In a second step, we execute the same scenario with dynamic application configurations and Hard VMs. This means that here, the degree of replication of application tiers is adapted according to the received load. In this step, we observe that holes may appear in the IaaS (as depicted in Section 3.2) and that a form of consolidation is required at the IaaS level.

• In a last step, we add the required consolidation system to the previous experiment, thus combining the two resource management policies (dynamic sizing at the application level and consolidation at the IaaS level).

5. Resource management at virtualized level only

5.1. Resource management policy

In this case, applications are deployed in a static configuration (without runtime reconfiguration) with the maximum number of replicas to prevent peak loads. All the replicas are deployed on Soft VMs which are created on a minimal number of physical nodes. A resource management policy is only implemented at the IaaS level. The IaaS administrator specifies the consolidation policy based on sensors (for VMs and physical nodes) as follows:

• If a physical machine saturates, a VM is migrated to another physical machine with enough resources to accept it. If necessary, a new physical machine is switched on. We call this VM relocation.

• If a physical machine is underloaded and its hosted VMs can be accepted by other physical machines, these VMs are migrated. This physical machine can be freed and switched off. We call this VM collocation.

Therefore, VM consolidation implies the implementation of two algorithms: VM relocation (when overloaded) and VM collocation (when underloaded). Designing an efficient VM consolidation policy has to take into account live migration cost. Even if several research efforts are made to minimize migration cost [14], the multiplication of this operation in a short time significantly affects the execution of applications.

Algorithm 1 presents the VM relocation algorithm (when the CPU load of a machine is over a predefined maximum CPU threshold). This algorithm is summarized as follows. Firstly, it identifies the most loaded node. Then it evaluates for each VM on this node the slope of its CPU variation (VM CPU load are logged over a period). These slopes allow us to identify the VM responsible for the node overload. The principle of our VM relocation algorithm consists in giving CPU power to the overloaded VM as much as possible. This objective can be reached either by relocating this

---

1 Grid’5000 is an initiative from the French Ministry of Research through the ACI GRID incentive action, INRIA, CNRS and RENATER and other contributing partners.
VM or by relocating another VM (so that the overloaded VM will stay on the initial node but with more CPU); we choose the relocation which gives the maximum CPU to the overloaded machine (without allocating a new physical machine). If no such relocation is found, a new machine is allocated and the VM is relocated to this machine.

Regarding VM collocation (presented in Algorithm 2), for each VM on the most underloaded node, we look for the Best-Fit node which can welcome this VM without reaching its maximum CPU threshold. If this was the last VM on the left node, the machine can be freed.

5.2. Evaluation

At the application level, two customers’ applications are statically dimensioned and deployed with the maximum number of MySQL server (three per application). This management of the IaaS is ensured by the TUNe autonomic system. At creation time, TUNe maximizes the number of VMs per physical machine. This policy explains the fact that all VMs are hosted on node N1 (six VMs, Fig. 5) at the beginning of our experiment. A Scheduler component (implemented in TUNe) periodically receives and processes all VM and physical machine loads from remote probes and migrates VMs when necessary. It implements our consolidation policy presented in the previous subsection.

Fig. 5 shows the generated workload submitted to the two RUBiS applications and the IaaS node allocation status during the experiment. The representation “3A/40” at situation (o) means that this node hosts three VMs belonging to application A and the current CPU load of all three VMs is 40% of the capacity of the processor. Fig. 6 presents the results we obtained regarding the evolution of the number of VMs per node and the nodes’ CPU loads (N1–N3). We discuss these results and compare them with those predicted in Fig. 5:

- Time (o): all VMs are hosted on N1.
- Time (a): the load increase for application A caused a migration of a VMA (VM of application A) on a new node (N2).
- Time (b): the load increase for application B caused a migration of a VMB, since it gives more capacity to the saturating VMB (on N2) than if we had migrated another VM from N1.
- Time (c): the load increase for application A caused a migration of a VMA from N1 to N2 (migrating a VMB from N1 would not do better).
- Time (d): the load increase for application B caused a migration of a VMA from N2 to N3 (migrating a VMB from N2 to N1 would saturate N1).
- Time (e): the load decrease for application B caused migrations of VMBs to N2.
- Time (f): a VMA is migrated from N1 to N3 (the most loaded node).
- Times (g and h): two VMAs are migrated to N2.
Algorithm 1 VMs Relocation (Virtualized Level Only)

Symbols:
- $\text{MaxThreshold}$: The maximum acceptable CPU load on a IaaS node
- $\text{CurrentCPULoadOn}(N_i)$: The current CPU load on node $N_i$
- $\epsilon$: VM live migration overhead (in term of CPU)
- $\text{NbOfVMsOn}(N_i)$: A function which returns the number of VMs running on node $N_i$

Begin
1: $N_o \leftarrow \text{The Most Loaded Node}$
2: $V_{M_o} \leftarrow \text{The VM with the greatest CPU slope on } N_o$

# The objective of this algorithm is to give move CPU to $V_{M_o}$.
# How much CPU will remain on a node $N_i$ if $V_{M_o}$ is migrated from $N_o$ to $N_i$?
3: for Each IaaS’s node $N_i$ with $\text{NbOfVMsOn}(N_i) > 0$ and $N_i \neq N_o$ do
4: \begin{align*}
\text{AvailableCPUOn}(N_i) & \leftarrow \text{MaxThreshold} - \text{CurrentCPULoadOn}(N_i) \\
\text{RemainCPUOn}(N_i, V_{M_o}) & \leftarrow -1
\end{align*}
5: if AvailableCPUOn$(N_i, V_{M_o}) > \text{CPU}(V_{M_o}) + \epsilon$ then
6: \begin{align*}
\text{RemainCPUOn}(N_i, V_{M_o}) & \leftarrow \text{AvailableCPUOn}(N_i) - \text{CPU}(V_{M_o})
\end{align*}
7: end if
8: end for
9: end for

# How much CPU will remain on $N_i$ if another VM is migrated to another node?
10: for Each VM on $N_o$ and $V_i \neq V_{M_o}$ do
11: for Each IaaS’s node $N_i$ with $\text{NbOfVMsOn}(N_i) > 0$ and $N_i \neq N_o$ do
12: \begin{align*}
\text{RemainCPUOn}(N_i, V_i) & \leftarrow -1
\end{align*}
13: if AvailableCPUOn$(N_i, V_i) > \text{CPU}(V_i) + \epsilon$ then
14: \begin{align*}
\text{RemainCPUOn}(N_i, V_i) & \leftarrow \text{AvailableCPUOn}(N_i) + \text{CPU}(V_i)
\end{align*}
15: end if
16: end for
17: end for

# We try to relocate either $V_{M_o}$ or another VM.
# The destination node can be a new IaaS node (which was not previously on).
18: $(\text{MigratedVM}, \text{DestinationNode}) \leftarrow V_i$ and $N_i$ so that $\text{RemainCPUOn}(N_i, V_i)$ is the highest
19: if RemainCPUOn(DestinationNode, MigratedVM) = $-1$ then
20: \begin{align*}
\text{DestinationNode} & \leftarrow \text{A new machine} \\
\text{DestinationNode} & \leftarrow V_{M_o}
\end{align*}
22: end if
23: Migrate MigratedVM from $N_o$ to DestinationNode

End

All CPU peaks on Fig. 6(b) correspond to VM migrations and are due to live migration costs. Since our collocation algorithm considers that all nodes are identical in terms of hardware configuration, our experiment ends with all VMs on N2 instead of N1 (as at startup).

Despite the benefits in terms of the number of used machines, VM consolidation has important limitations when implemented without dynamic sizing. First, the number of VMs colocated on the same node is limited by node size (in terms of memory). In our experiment, a node can support up to six VMs at the same time. Secondly, VM overhead increases with the number of VMs on the node. We evaluated this overhead in a previous work [8]. Finally, there are many situations where several VMs hosting replicas of the same application tier (e.g. MySQL for application A) were hosted on the same node. For example, at time (a), node N1 hosts 2 MySQL instances of application A and 3 instances of application B. Dynamic sizing would avoid such situations.

6. Resource management at the application-level

6.1. Resource management policy

Dynamic sizing at application level (also known as dynamic server provisioning) consists of adding or removing replicas according to the monitored load of each tier. In that way, new resources are requested only when necessary, while ensuring the application QoS. Each replica is deployed and launched on a separate VM. The customer chooses an initial number of replicas for each tier and defines a dynamic sizing policy which generally takes the form of a threshold that a monitored load should not exceed. Sensors monitor each tier and generate an event whenever a constraint is violated. The reaction to an overload event is presented in Algorithm 3. On the other side, the reaction of an underload event is presented in Algorithm 4. As we previously mentioned, this policy only makes sense when applications allocate Hard VMs. Obviously, it may leave holes (unused resource portions) on IaaS machines if no consolidation mechanism is implemented at this level, as illustrated in the next section.

6.2. Evaluation

The objective of this evaluation is twofold: (1) to confirm the results of many other research works that show the benefit of dynamic sizing; and (2) to show the limits of this policy as identified in Section 3.

We based our self sizing policy on the average CPU load of MySQL servers. TUNe is used at the application level to implement this policy. Therefore, each MySQL VM is equipped with a monitoring agent, informing its TUNe manager about the variation of the
Algorithm 2 VMs Co-location (Virtualized Level Only)

Symbols:
- This algorithm uses the same symbols as Algorithm 1

Begin
1: \( N_u \leftarrow \text{The most underloaded node} \)  # The most underloaded node.
# We try to free the most underloaded node \( N_u \).
# All its VMs will be relocated (if possible) on the other underloaded nodes.
2: \( \text{AvailableCPUOn}(N_u) \leftarrow \text{MaxThreshold} - \text{CurrentCPULoadOn}(N_u) \)
3: for Each VM \( M_i \) on \( N_u \), sorted in a decreasing order do
4: \( \text{Rest}_{\text{min}} \leftarrow 100 \) # What is the Best-Fit node for the relocation of \( M_i \)?
5: \( \text{DestinationNode}(M_i) \leftarrow \text{NULL} \)
6: for Each IaaS's node \( N_j \) with \( N_j \neq N_u \) do
7: \( \text{Rest} \leftarrow \text{AvailableCPUOn}(N_i) - \text{CPU}(VM_i) - \text{epsilon} \)
8: if \( \text{Rest}_{\text{min}} < \text{Rest} \) then
9: \( \text{Rest}_{\text{min}} \leftarrow \text{Rest} \)
10: \( \text{DestinationNode}(M_i) \leftarrow N_j \)
11: end if
12: end for
13: if \( \text{DestinationNode}(M_i) \neq \text{NULL} \) then
14: \( \text{AvailableCPUOn}(\text{DestinationNode}(M_i)) \leftarrow \text{AvailableCPUOn}(\text{DestinationNode}(M_i)) - \text{CPU}(VM_i) \)
15: Migrate VM \( M_i \) from \( N_u \) to \( \text{DestinationNode}(M_i) \)
16: end if
17: end for
18: if \( \text{NbOfVMsOn}(N_u) = 0 \) then
19: Turn off \( N_i \)
20: end if
End

Algorithm 3 Size Up (Application Level)

Begin
1: for Each tier \( T_i \) of the application do
2: if \( T_i \) is saturated then
3: \( \text{NewVM} \leftarrow \text{Allocation a new VM from the IaaS} \)
4: Deploy and launch a new instance of replica of \( T_i \) on \( \text{NewVM} \)
5: Reconfigure the loadbalancer in front of the \( T_i \) tier
6: end if
7: end for
End

Algorithm 4 Size Down (Application Level)

Symbols:
- \( \text{NbOfReplicaOn}(T_i) \): The number of server replica on tier \( T_i \)

Begin
1: for Each tier \( T_i \) of the application do
2: if \( T_i \) is underloaded and \( \text{NbOfReplicaOn}(T_i) > 1 \) then
3: \( (\text{ReplicaToRemove}, \text{VMToRemove}) \leftarrow \text{A server replica and its VM} \)
4: Reconfigure the loadbalancer in front of the \( T_i \) tier
5: Stop \( \text{ReplicaToRemove} \) and terminate \( \text{VMToRemove} \) from the IaaS
6: end if
7: end for
End

CPU load. The TUNe manager computes the average CPU load and when this metric reaches 100%, it requests a VM allocation from the IaaS, deploys and starts a new MySQL server instance on this VM. Notice that we choose this threshold in order to show the impact of overload of the MySQL tier on the application QoS (response time). Indeed, the application will still be overloaded during the addition time (startup of a new VM in the IaaS) of the MySQL server. To avoid this impact, one could either reduce this threshold or always keep a pool of unused VM in the running state. Likewise TUNe removes one MySQL server when its CPU load can be distributed on other replicas without provoking a MySQL tiers overload.

The overall management architecture for this scenario is as follows: two instances of TUNe are used for managing separately the two RUBiS applications (A and B) and implementing dynamic server provisioning. The IaaS does not implement any VM consolidation.
We submit the same workload as in the previous experiment. Fig. 8 shows CPU loads on the nodes hosting MySQL servers and the RUBiS response time. These results show the reactivity of the autonomic system (TUNe) in order to ensure application QoS through dynamic resource allocation. These figures are interpreted as follows:

- In the first half of the workload (time (o)–time (d) in Fig. 7), the CPU load increases for both applications. As a result of TUNe's reconfigurations, MySQL servers are gradually added (times (a)–(d)) to keep the response time at its original level (less than 0.5 s). At the end of this phase (time (d)), each RUBiS application has 3 MySQL replicas. These size up operations are observed in Fig. 8(a) at times $T_1 = 600$ s and $T_2 = 2000$ s: the measured CPU load of the VMs reaches the maximum threshold and the response time also increases (about 3 s). The addition of a replica reduces the CPU load and also reduces the response time (at its minimal value). Notice that there is only one VM before $T_1$, two VMs between $T_1$ and $T_2$, and three VMs after $T_2$.
- After time (d) in Fig. 7, application B receives fewer requests, therefore, the number of replicas is reduced down to a single (on machine N1 at time (e) and (f)). This situation produces holes in the infrastructure (on nodes N2 and N3).
- A similar situation happens to application A: its load decreases. Its TUNe instance undeploys its MySQL instances and deallocates VMs on machines N2 and N3 (application A).
- At the end of this experiment (time (g) and (h) in Fig. 7), only machine N1 is running with two VMs, one for each application, similarly to the beginning of our scenario at time (o).

As can be seen in Fig. 8, the resource management policy at the application level effectively maintains QoS (i.e., keeps the response time low): MySQL instances are dynamically added and removed, according to the CPU load of this tier. This behavior ensures not only application response time but also a minimal number of VMs (and therefore cost reduction for the customer).

7. Resource management at both levels

This section describes a combination of the two strategies described in the previous sections to eliminate each solution's drawbacks.

7.1. Resource management policy

With a multi-level policy, we will both have dynamic sizing (the same algorithms as in the previous section) at the application level and a new form of consolidation at the IaaS level.

Since we have dynamic sizing, VM usage is optimized: VMs are effectively used or they will be removed by the TUNe instance at the application level. Therefore, it is not necessary at the virtualized level to reuse the consolidation policy based on CPU loads, and we can allocate Hard VMs. But as observed in Fig. 7 at time (e) and (f),...
the application level policy can produce holes in the infrastructure. Thus, we implement a consolidation policy which migrates VMs to remove these holes. However, this consolidation policy is not based on the CPU usage of VMs, but on a resource quota allocated to these VMs.

Algorithm 5 describes this consolidation policy. In this algorithm and in our experiments, we allocate only one type of VM, i.e., all VMs have the same memory size and CPU quotas. It implies that we can host the same number of (Hard) VM on any machine in the IaaS. The Scheduler component of TUNe at the virtualized level tries to migrate the VMs from the machine with the minimum number of VMs. MaxVM is the number of VMs that a physical node may host and $\text{NbOfVMsOn}(N_i)$ the number of VM on each node. Such a consolidation can be performed as soon as $\sum \text{NbOfVMsOn}(N_i) \leq \text{MaxVM} \times (\text{MaxVM} - 1)$. This condition is checked each time a VM is deallocated.

7.2. Evaluation

We use in this scenario three TUNe instances for managing the IaaS level and the two RUBiS applications. To distinguish between these TUNe instances, let TUNeApp be the TUNe instances running at the application level, and TUNeVM be the one assigned to manage the IaaS. These TUNe instances work independently, each handling the management operations at its level. TUNeApp’s behavior (in terms of resource management) is the same as presented in Section 6: based on the average CPU load of the MySQL tiers, TUNeApp asks for the addition/removal of a MySQL server (running on a VM). TUNeVM’s goal was described in the previous section: server consolidation based on VM allocated resources.

This two-level management policy provides benefits for resource saving and power reduction, as shown in the experiment below.

The workload submitted to the two applications is the same as in the previous sections. Both RUBiS applications are started with one MySQL server. Each TUNeApp instance can allocate MySQL servers when necessary. Each MySQL VM requires half (in terms of CPU and memory quota) of an IaaS node during the experiment.

Fig. 9 shows the generated workload and predicted VM placements on physical nodes, according to our consolidation algorithm. Response times and VM loads in this scenario are similar to those of Section 6 (Fig. 8). The VM allocation status (per node) over time is presented in Fig. 10.

- Until time (d) ($T_1 = 2800$ s), our experiment is in the allocation phase since the load increases for both applications. MySQL servers are gradually added.
- At time (e) and (f) ($T_2 = 3300$ s), we observe the removal of two servers for application B on nodes N2 and N3, caused by the load reduction for application B.
- At time (g) ($T_3 = 3400$ s), the TUNeVM Scheduler analyzes the situation, migrates a VMA from N3 to N2 and frees N3.
- At time (d) ($T_4 = 4200–4400$ s), the load reduction for application A provokes the removal of two MySQL servers.

This experiment shows that both-levels management cumulates the advantages of the two previous management strategies: it reduces the number of VMs in the IaaS and it consolidates these VMs on fewer machines (removing holes).

However, we can still have the problem identified in Section 5: the possibility to have on the same node several VMs hosting replicas from the same application tier. For example, at time (g)},
The number of VMs that an IaaS node may host

The number of IaaS nodes which are running

A function which returns the number of VMs running on node $N_i$

Algorithm 5 VMs Consolidation (Both Level)

Symbols:

- $\text{MaxVM}$ The number of VMs that a IaaS node may host
- $\text{NbNode}$ The number of IaaS node which are running
- $\text{NbVMsOn}(N_i)$ A function which returns the number of VMs running on node $N_i$

Begin

1: $N_{\text{min}} \leftarrow N_i$ so that $\text{NbVMsOn}(N_i)$ is the smallest.
   # We try to free the node $N_{\text{min}}$.
2: for Each VM, on $N_{\text{min}}$ do
3:  for Each IaaS's node $N_i$ with $N_i \neq N_{\text{min}}$ do
4:   if $\text{NbVMsOn}(N_i) < \text{MaxVM}$ then
5:     Migrate VM, from $N_{\text{min}}$ to $N_i$
6:   Break # Goto 2 (continue with another VM on $N_{\text{min}}$).
7: end if
8: end for
9: end for
End

9. Related work

Following our classification, we present in this section an overview of research works that have been conducted around resource management in the cloud computing area.

9.1. IaaS level management

Regarding resource management at the IaaS level, consolidation systems such as GreenCloud [16], Entropy [17,18] aim at saving energy in a hosting center using powering on/off nodes and VM migration. In the same vein, the Aneka [19] platform implements dynamic resource allocation in the context of hybrid clouds: it allocates new resources from a public cloud when those in a private desktop grid are overbooked. A deep thinking about VM consolidation problematic and solutions was done in [20]. This later formalizes the problem by defining two classes of solutions: deterministic algorithms and non deterministic algorithms. Regarding their formalism, our proposed algorithms are deterministic. In addition, [20] formalizes the cost of VM live migration in the case of a consolidation algorithm. The main purpose of [20] is not to propose a consolidation algorithm (as we have done), but to formalize the problematic. [21] studies VMs interference (the impact of co-locating several VMs which uses the same type of resources, on the same node) when applying a VM consolidation algorithm. It proposes a predictive model to avoid this impact. We do not take this into consideration. [22] presents a work in the same vein as [21], focusing on n-tier applications. [23] identifies and evaluates fourteen provisioning and allocation policies in the IaaS. It presents an overview of existing resource management policies and also evaluates their efficiency.

All these works address IaaS level consolidation policies. These VM consolidation algorithms could bring more benefits to cloud provider (in term of resource utilization) when they are combined with application sizing in the context of master–slave applications.

9.2. Application level management

Many research works have investigated dynamic resource allocation for application running in a virtualized hosting center environment. The major part of these works targets n-tier applications. [24] presents a dynamic allocation system for n-tier
applications, based on an autonomic computing system and a load balancer. This latter has a pool of allocated nodes and distributes the processes over this pool. The main drawback of [24] is that it relies on a single load balancer for all application, which can be a bottleneck. Also, its management policy is implemented within the load balancer which is very intrusive and not generic (our autonomic management system is not tied to a load balancing pattern and is therefore more generic). [25] proposed many strategies in a similar context as [24]: jobs distribution to a pool of VMs in a cloud infrastructure. These strategies are based on dynamic VM allocation/deallocation. New VMs are deployed when others are overloaded and they can be released when idle (our application level policy for J2EE applications is very similar). IBM’s Oceano platform [26], as well as SmartScale [27], describes an automatic framework to scale applications using algorithms which meet those we have presented in this paper. [28] studies resource management at application level and focuses on the question of when to increase or decrease application replicas. To answer this question, it relies on trace analysis to model the behavior of the application in order to anticipate sizing actions. This method is well known in the literature under the term “feedback control LRU algorithm”. Even if we have presented a proactive algorithm (based on actual traces), it is possible to implement [28]’s algorithm in our platform.

In summary, these research works about resource management in the cloud area are done regardless of each cloud actor. Moreover they work at the application level and manage only one application, while we address two cloud levels and the hosting of multiple customers applications.

9.3. Multi-level management

To the best of our knowledge, very few works have addressed dynamic resource management at both levels (application and IaaS) as presented in this paper. [29,30] have proposed a two-level resource management policy, but their mechanism of resource provisioning at the IaaS level is only based on the allocation of additional resources to VMs (vertical scalability). [31] proposes a solution to the coordination problem between VMS and the hosted applications when the resource availability have changed (more CPU or allocated memory for example). The [31]’s approach uses an hybrid solution (a feedback learning solution combine with a proactive solution) to prevent the reconfiguration of VMs and the applications they run. The latter are configured in a coordinated order: VMs before applications or vice versa. [32] presents and analyzes different resource management level in the cloud. The two levels we explore in this paper are mentioned in few words without a detailed solution. [33] proposes a model to coordinate different resource management policies involving the cloud customers and the provider. It defines a set of API and constraints allowing the customer to specialize resource allocation and services placement policies within the cloud. The limits of this work are: (1) the collaboration is done in one direction (from the customer to the provider); (2) it focuses on the customer’s application elasticity policy (nothing about VM consolidation for instance). [34] describes a resource management framework for virtualized cloud environments. It implements both vertical and horizontal VM scalability in order to face workload evolution. The described framework needs to monitor specific application metrics (response time, throughput, etc.). Therefore, it is restricted to cloud platforms in which the cloud provider has access to the hosting applications. It has no interest in the complementarity of running several resource management policies at different cloud levels as we have presented in this paper. One of the first research work which investigates multi-level resource management in a virtualized cloud is [35]. As we claim in this paper, it proposes an autonomic system which automates resource management at two levels: application and IaaS. It also identifies two other levels: a local-level (VM on the same node), and a global-level (all VM running within the IaaS). Unlike our work, the different identified levels are all managed by the single resource management policy. This implies that the cloud customer and the provider are the same entity.

10. Conclusions and perspectives

Cloud computing is a recent trend where companies are externalizing their computing infrastructures in hosting centers, in order to reduce the cost of their IT. One of the main services that must be ensured in such a hosting center is resource allocation. For economic reasons, resources should be allocated (and deallocated) dynamically to the hosted applications according to their runtime load.

In this paper, we have investigated resource management strategies in the cloud computing context, with master–slave applications. We showed that consolidation of virtual machines at the IaaS level only with static application configuration incurs the overhead of needless collocated virtual machines. We then showed that dynamic application sizing only requires an IaaS consolidation in order to avoid holes (unused resources) on the IaaS physical machines, thus leading to a two-level resource management strategy.

Relying on a common application scenario, we justified such a two-level resource management policy that we prototyped and experienced in a private cloud infrastructure.

A perspective to this work is to improve this two-level resource management to further reduce collocated application replicas when the IaaS performs VMs consolidation. As described in this article, our algorithm relies on fixed size VM allocation and several VMs for the same tier may be required (according to the load) and collocated on a node. They could advantageously be replaced by a single bigger VM on that node. It becomes possible if the cloud provider has the knowledge of the architecture of customers’ applications. In other words, it implies a collaboration between the two cloud actors (provider and customers), which also implies the upgrade of APIs commonly used in IaaS platforms.

We also plan to investigate the impact of VMs interferences in our resource management algorithms. Interferences occur when collocating on the same node several VMs which highly use shared resources whose reservation is not possible. These resources are disk and network IO.

Acknowledgments

This work is supported by two projects: (1) the French Fonds National pour la Societe Numerique (FSN) and Poles Minalogic, Systematic and SCS, through the FSN Open Cloudware project; and (2) the ANR INFRA (ANR-11-INFRA 1305) under a grant for the project ctrl-Green.

References

Noel DePalma received his Ph.D. in computer science in 2001. Since 2002 he was Associate Professor in computer science at University of Grenoble (ENSIMAG/INP). Since 2010 he has been a professor at Joseph Fourier University. He is a member of the SARDES research group at LIG laboratory (UJF/CNRS/Grenoble INP/INRIA), where he leads researches on Autonomic Computing, Cloud Computing and Green Computing.

Daniel Hagimont is a Professor at the Polytechnic National Institute of Toulouse, France and a member of the IRIT laboratory, where he leads a group working on operating systems, distributed systems and middleware. He received a Ph.D. from the Polytechnic National Institute of Grenoble, France in 1993. After a postdoctorate at the University of British Columbia, Vancouver, Canada in 1994, he joined INRIA Grenoble in 1995. He took his position as a Professor at Toulouse in 2005.