Autonomic resource allocation in a J2EE cluster

A. Tchana, S. Temate, L. Broto and, D. Hagimont
IRIT-ENSEEIHT, 2 rue Camichel, BP 7122, 31071 Toulouse, France
Email: lastname@irit.fr

Abstract—One of the main services that must be assured in 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. Autonomic computing systems provide support for such a dynamic resource allocation, as they allow applications to be dynamically reconfigured in order to involve more or less resources. We implemented an autonomic management system called TUNE that we used for dynamic resource allocation in a J2EE e-commerce application. We present the results of our experiment with different management policies and we pinpoint the advantages and limitations of this approach.

Index Terms—autonomic management, hosting center, resource allocation, virtual machines

I. INTRODUCTION

In order to reduce the maintenance cost of computing environments, companies are increasingly externalizing their computing infrastructures which are therefore managed by specific companies that we will call providers. They are expected to insure quality of service requirements of their customers. Cloud computing is a recent paradigm which follows this direction.

In this context, on demand resource allocation 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 they are not used anymore. If we consider the example of an E-commerce Web application built as a multi-tier J2EE organization of servers, the allocated resource (machines) to host these servers depends on the expected load that the application will have to face. Very often, this allocation is static, i.e. once for the lifetime of the application (or it should be re-installed by a human administrator). Static allocation leads to resource overbooking and therefore resource wasting.

In this paper, we report on an experiment which consisted in ensuring dynamic resource allocation for a clustered J2EE application deployed in a hosting center. We relied on an autonomic management platform called TUNe [1] that we formerly implemented and we experimented with two solutions which are characterized by different forms of intrusivity.

In the first solution, we adapt the degree of replication of each tier in the J2EE application according to the submitted load. This solution is characterized by a significant intrusivity, either at runtime (adaptations have side effects on the application, e.g. request loss) or in the code of the application (which has to be modified to prevent such side effects).

In the second solution, we exploit virtual machines (VM) and more precisely VM migration, in order to ship J2EE tiers on more or less physical machines according to the submitted load. In this solution, migration is transparent for the application and intrusivity is only related to the performance overhead of virtual machines.

We implemented these different solutions on top of the TUNe autonomic management system and conducted a comparative performance evaluation. This evaluation relies on a E-commerce dynamic web application benchmark called RUBiS [2], which implements an auction site modeled over eBay. The results show that the virtual machine based solution represent an interesting tradeoff between the different forms of intrusivity.

The rest of the article is organized as follows. Section 2 presents the context and motivations of our work. Section 3 describes the two dynamic resource allocation solutions that we implemented. Our experimental results are reported in Section 4. After a review of related works (Section 5), we conclude the article in Section 6.

II. CONTEXT AND MOTIVATIONS

A. E-commerce J2EE hosting center and Resource allocation

The Java 2 Platform, Enterprise Edition (J2EE) defines a model for developing web applications in a multi-tiered architecture. Such applications are typically composed of a web server (e.g. Apache), an application server (e.g. Tomcat) and a database server (e.g. MySQL). In this context, the increasing number of Internet users has led to the need of highly scalable and highly available services. To face high loads and provide higher scalability of Internet services, a commonly used approach is the replication of servers in clusters. Such a replication approach relies on a master-slave scheme. It usually defines a particular software component in front of each set of replicated servers, which dynamically balances the load among the replicas. Here, different load balancing algorithms may be used, e.g. Random, Round-Robin, etc.

In the context of the hosting of a clustered J2EE application, a static allocation (once at launching without any runtime allocation) has many drawbacks. With static allocation, it is difficult for the application administrator to decide on the degree of replication of each tier. If the degree of replication of one tier is too low, this tier will saturate and may become a bottleneck for the whole application, leading to a global slowdown. If the degree of replication is too high, the allocated machines are underused, which leads to resources wasting.

Therefore, if the customer of the hosting center is billed according to the consumed resources, it is interesting for him to benefit from a dynamic adaptation of the degree of replication according to the load.
Moreover, even if the machines are statically allocated from the point of view of the customer (who will pay for the allocated machines), the provider can actually manage the application on fewer machines according to the load. The customer has the guarantee that his allocated machines will be available if required, while the provider will reuse them (for another application) or switch them off (to save energy) if the application is underloaded.

Whatever the business model of the hosting center is, dynamic resource allocation brings many advantages. Notice here that the unit of allocation is not mandatorily a machine as assumed above. A machine could be shared between several applications and the unit of allocation would be finer grained.

In the next section, we introduce the principles of autonomic management systems and depict how they can be used to manage dynamic resource allocation in a hosting center.

B. Autonomic management systems

An autonomic management system is a system which provides support for the management of applications. It generally covers the life cycle of the applications it manages: deployment, configuration, launching, and dynamic management as well. Dynamic management implies that the system should allow monitoring of the execution and reaction to events such as failures or load peaks, in order to adapt the managed application accordingly and autonomously.

Many works in this area have relied on a component model to provide such an autonomic system support. The basic idea is to encapsulate (wrap) the managed elements (legacy software) in software components and to administrate the environment as a component architecture. Then, the administrators can benefit from the essential features of the component model, encapsulation, deployment facilities and reconfiguration interfaces, in order to implement their autonomic management processes.

Autonomic management, an area in which we have been working for several years, can be obviously very helpful for managing dynamic resource allocation in a hosting center. It can be used to monitor applications in a hosting center and whenever it is required, allocates a new machine, deploys the required software component on that node and reconfigures the application in order to integrate this new component. Notice here that autonomic management has also been much used to implement repair behaviors to tolerate software or hardware failures.

The work presented in this paper relies on an autonomic management platform called TUNe [3] developed in the same research group. The contribution of this paper is to describe several experiments in which we rely on TUNe to implement dynamic resource allocation for J2EE applications in a hosting center.

III. RESOURCE ALLOCATION STRATEGIES

The main idea we are exploring in this paper is to rely on an autonomic management system in order to dynamically allocate or de-allocate resources to a J2EE application in a hosting center, according to the load submitted to that application. We identified three ways to implement dynamic resource allocation for a clustered J2EE application. Table I synthesizes the main characteristics of these solutions: the followed approach and the intrusivity.

Solution 1 is a very naive approach: we just encapsulate the J2EE tiers in components (without any modification to the encapsulated software) and we reconfigure the J2EE architecture without any precaution. This solution alters the behavior of the application when the application is reconfigured because the servers (Apache, Tomcat, MySQL) have to be restarted when their configuration changes. Some requests may be lost during the restart.

In Solution 2, we modify the code of the application tiers in order to implement a smart reconfiguration which does not require to restart the tiers. So the intrusivity of reconfigurations is not in the behavior of the application, but in the code of the application that we modified to be reconfigurable (such a smart reconfiguration may be available for some particular J2EE tiers, but we observed that it is generally not the case).

In solution 3, we exploit virtual machine technologies and more precisely virtual machine migration, in order to locate J2EE tiers on more or less physical machines according to the load. Therefore, we don’t reconfigure the J2EE application, but relocate the virtual machines on which the J2EE tiers are running. This solution is intrusive in terms of performance overhead as virtual machines have a runtime cost. Notice that if the hosting center administrator doesn’t have control over the deployed J2EE application servers, this VM based solution would be mandatory.

A. Solution 1: untouched application

This is a very straightforward approach which consists in adding or removing tiers replicas according to the monitored load of each tier. The application administrator chooses for each tier the number of initial replicas that should be launched at deployment time. Each replica is deployed and launched on a separate node. The administrator also specifies the reconfiguration policy that should be applied. A reconfiguration policy generally takes the form of a threshold that a monitored load should not exceed (or a monitored quality of service that should be maintained). Monitoring is performed by sensors which notify events whenever a constraint is violated.

The reaction to an overload event is as follows:

- Identification of the overloaded tier,
- Allocation of a new machine from a pool of free nodes,
- Deployment and launching of a new tier replica on that machine (e.g. a MySQL server),
- Reconfiguration of the load balancer associated with this tier (e.g. a restart of MySQL-Proxy) to integrate the new

<table>
<thead>
<tr>
<th>Solution 1</th>
<th>Solution 2</th>
<th>Solution 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>untouched application</td>
<td>modified application</td>
<td>virtual machine migration</td>
</tr>
<tr>
<td>application behavior alteration</td>
<td>application code modification</td>
<td>performance overhead</td>
</tr>
</tbody>
</table>

TABLE I
DIFFERENT RESOURCE ALLOCATION APPROACHES
Conversely, the reaction to an underload event is as follows:

- Identification of the underloaded tier.
- Selection of a tier replica to remove from the application.
- Reconfiguration of the load balancer to remove the replica.
- Kill and undeployment of the tier replica on that machine.
- De-allocation of the machine that was hosting the replica.

This solution is quite generic as it can be used with different applications as long as they rely on a master-slave scheme with replicated servers and load balancers. However, it has several important limitations:

- State loss. If the removal of a server is implemented by stopping the server process, requests which are under execution in the server are lost (for instance, executing servlet calls in a Tomcat server).
- Unavailability. The reconfiguration of a load balancer generally requires its restart to take into account modifications in some configuration files (for instance configuration files of Mod_JK). During this restart, the load balancer is unavailable and therefore the overall application too.
- Stateless servers. This solution requires that servers be stateless, i.e., servers must not share a state (with consistency management between replicas) because the integration of a new replicas would require the resynchronization of this replica with the others, which is not necessarily an existing feature of the server software.

These limitations are consequences from the fact that we are managing legacy software which was not supposed to be dynamically reconfigured. Such limitations can be by-passed if the software is adapted to provide (or already provides) support for smart reconfigurations, as described in the next section.

B. Solution 2: modified application

This solution requires a participation of the application. Either the application tiers already have such features or they have to be modified to integrate them. The required features are:

- Allow a smart stop (and resume) of a server. It means blocking the entering requests, waiting that the running requests are completed and then effectively stopping the server.
- Allow dynamic addition or removal of a replica in a load balancer without requiring its restart.
- Allow dynamic integration of a replica in a group with state reconciliation.

These features avoid the previously mentioned limitations, but they require significant modifications to existing software if they are not yet implemented (such features are not always available in the case of clustered J2EE applications: we implemented them for some of the J2EE tiers such as state reconciliation for MySQL replicas).

C. Solution 3: virtual machine migration

In order to provide a solution which does not require any modification of the managed software and does not alter the behavior of the application, the third solution exploits the notion of virtual machine.

Virtualization is a technology which allows running one or several operating systems (called guest OS or virtual machines) on top of an operating system (called host OS) running on the bare hardware. Several virtualization technologies have been developed but we focused on the two which are considered as the most efficient: the isolation technology provided by OpenVZ [4] and the para-virtualization technology provided by Xen [5]. The isolation technology allows running several operating system instances as different user-level processes over the host operating system. This technology is easier to manage and more efficient than other virtualization technologies. According to the OpenVZ developers, it only introduces from 1% to 3% overhead. The para-virtualization technology adds a layer between the bare hardware and the host operating system called the hypervisor. With this technology, every system calls made by the virtual machines are translated by the hypervisor. This technology is supposed to get better performance than full virtualized systems. Our experiments relied on Xen and Open VZ.

In this solution, we exploit virtualization technologies’ migration abilities. Each server to be managed runs in a virtual machine. The administrator must choose for each tier the maximum number of instances in the worst case (at the maximum load). Then each server instance is created and deployed on a virtual machine. The virtual machines are deployed on the minimal number of node able to accept the load. At runtime, instead of adding a server instance in case of increased load (as in the two previous solutions), we just migrate a virtual machine to an underloaded node (potentially a free node). In the same way, instead of removing a server instance in case of underload, we just migrate virtual machines in order to co-locate them. Thus we do not need to add or remove a software instance but just migrate it according to the infrastructure load.

According to their respective developers, Xen and OpenVZ are migration safe, e.g., there isn’t any information loss during a migration (thanks to TCP/IP remission). Thus, this approach does not lose any request, as we don’t stop any server but only migrate virtual machines, which is supposed to be transparent. Moreover, this approach is generic, i.e., it can be applied to any kind of software running on the virtual machine. Finally, as shown in the evaluation section, the overhead of virtual machines is very low. In the next section, we describe the implementation of these solutions with the TUNe autonomic management system.

IV. IMPLEMENTATION WITH THE TUNE SYSTEM

We first present the TUNe autonomic management system (see [3] for more details) and then, describe the implementation of solutions 1 (untouched application) and 3 (virtual machine migration). Solution 2 (modified application), whose
A. Design principles of TUNe

Several researches conducted in the autonomic management area follow an approach that we call component-based management (overviewed in Section II-B). The TUNe system implements this approach (relying on the Fractal component model [6]), but it evolved to provide higher level formalisms for all administration tasks (wrapping, deployment, reconfiguration).

Regarding the initial configuration of a deployed architecture, our approach is to introduce a UML profile for graphically describing configuration and deployment architecture (such a specification is called the configuration schema). The introduced formalism is more intuitive and abstract (than traditional architecture languages) as it describes the general organization of the configuration (types of components and interconnection pattern) in intension, instead of describing in extension all components instances that have to be deployed. This is particularly interesting for applications where tens or hundreds of servers have to be deployed. Thus, any component in the configuration schema is instantiated into many instances, and bindings between component instances are computed from the described interconnection pattern. Therefore, the configuration schema is used to generate a component based architecture that we call the System Representation (SR) of the administrated application\(^1\). In the described configuration schema, each component has a set of attributes that describes the configuration of the encapsulated software and a set of methods to configure and reconfigure component instances.

Similarly to software, the description of the environment in which software will be deployed is based on the same UML profile. A cluster of physical machines is represented by a Node component. This later encapsulates the characteristics of the set of machines it represents. Any Node component is instantiated in the SR into several Node component instances. A special software component attribute (host-family) indicates to TUNe the target cluster in which it has to be deployed. The deployment process instantiates an implicit binding in the SR between each software component instance and its deployment Node component instance. Figure 1(a) depicts the J2EE configuration schema used in Solution 1 (more details are given in Section IV-B).

The instantiated components in the SR are wrappers of the administrated legacy software. Regarding wrapping, TUNe introduces a Wrapping Description Language (WDL) to specify the behavior of wrapper components. WDL allows defining methods (for software and node components) that can be invoked to start, stop, configure or reconfigure the wrapped component. Each method declaration indicates a method implementation in a Java class (this implementation is specific to the encapsulated software but most of them can actually be reused for different software) and the attributes from the configuration schema that should be passed to the method upon invocation. WDL allows navigation in the configuration schema, which means that in a wrapper description, the specification of a parameter to be passed to a method can be a path (following bindings between components) to a component attribute, this path starting from the current wrapper component.

Regarding reconfiguration, our approach is to introduce a very simple specific language called Reconfiguration Description Language (RDL), to define workflows of operations that have to be executed for reconfiguring the managed environment. An event is generated either by a specific monitoring component (e.g. a probe in the configuration schema) or by a wrapped legacy software which already includes its own monitoring functions. A specific API allows a Java method associated with a wrapper to send such an event to TUNe, which will trigger the execution of a RDL program. RDL only provides basic language constructs: variable assignment, component method invocation and \(\text{if...then...}[\text{else}]\) statement. It also allows addition or removal of component instances, which is the only way to modify the architecture of the SR. One of the main advantages of RDL is that reconfigurations can only produce an (concrete) architecture which conforms with the configuration schema, thus enforcing reconfiguration correctness.

B. Solution 1: untouched application

In our experiments, we focus on resource allocation for the MySQL tier. Figure 1(a) shows the configuration schema for this J2EE application (the attributes are detailed only for the MySQL tier). The figure describes the organization of Node components, defining a different cluster of machines for each tier (ClusterApache, ClusterTomcat, ClusterMySQL). As many attributes of these Node components are similar, we use inheritance to factorize many common attributes. The top part of the figure describes the J2EE software architecture, which initially includes a single MySQL server (but reconfigurations may add new servers). A Probe component (a unique instance) is used to monitor the MySQL tier: it defines threshold attributes which indicate the minimum and the maximum accepted load on the tier (underloads and overloads generate an event).

The deployment of the overall architecture is done into two steps. In the first step, Node component instances are created in the SR and since the host-family attribute is null, TUNe doesn’t perform a real deployment (this is the bootstrap). TUNe invokes the \text{initAlloc()} method to initialize Node component instances. In this solution, the \text{initAlloc()} method (implemented in the wrapper) uses a file of physical machine DNS names, the path to this file being given by the listOfNodes attribute. At the end of this step, Node component instances are created and initialized and the allocators associated with the Node components are also initialized.

In the second step, TUNe deploys, configures and starts J2EE servers according to the configuration schema and the

\(^1\)In the rest of the paper, we make the distinction between Components which are described in the configuration schema and Component instances which are created in the SR.
starter program (Figure 1(b)) is executed. This later ensures that (1) binary code are deployed, (2) software configuration files are well generated and (3) servers are started following an adequate order (MySQL, MySQLProxy, Tomcat, Apache and Probe). Notice that some configuration operations are performed simultaneously (parallel and branch statements).

Only one MySQL server is initially deployed. Based on the MySQL tier monitored load, the Probe component can generate two types of event: fixMySQLOverload if the MySQL tier is overloaded and fixMySQLUnderload if the MySQL tier is underloaded (Figure 1(b)). We detail below the reconfiguration program associated with the fixMySQLOverload event:

- **this.stop():** this identifies the Probe component instance. This action stops the probing component to prevent the generation of multiple events.
- **newNode=ClusterMySQL.allocate()** allocates a new machine (a ClusterMySQL component instance). A reference to that instance is stored in newNode variable.
- **newMySQL=MySQL.add(newNode):** adds a new instance of the MySQL component in the SR. This instance is bound with the newNode Node instance. It is also bound with the MySQLProxy instance according to the configuration schema. A reference to the added instance is stored in the newMySQL variable.
- **$newMySQL.deploy():** invokes the deploy, configure and start methods on the new instance.
- **this.mysqlServer.select.mysqlLB.restart():** selects one MySQL component instance (monitored by the Probe) and restarts the MySQLProxy component instance it is linked with.
- **this.start():** restarts the Probe instance.

Similarly, the reconfiguration program associated with the fixMySQLUnderload event reduces the number of MySQL server instances when the MySQL tier is underloaded. The program selects one MySQL and removes it from the SR.

### C. Solution 3: virtual machine migration

An important aspect of TUNe is that nodes are themselves managed as software components. Therefore, virtual machines (VMs) can be wrapped and managed by TUNe. Therefore, this approach of Solution 3 is decomposed into three stages: (1) physical Node components are configured, (2) VMs are deployed on physical machines and (3) J2EE servers are deployed on VMs. In our experiments, only MySQL servers were deployed on VMs, the others were deployed on physical machines.

**a) Stage 1:** The Cluster Node component represents the set of physical nodes as in Solution 1 with a simple modification: its allocation strategy returns a single machine at deployment time and returns a new machine at reconfiguration time. Actually, after the deployment of VMs, the reconfiguration strategy changes from singleNode to round-robin.

**b) Stage 2:** We introduce a new Node component called ClusterVM, to manage VMs (see Figure 2(a)). Its initAlloc() method (defined in the wrapper) configures VM instance DNS names (nodeName) with a list of names stored in a file (listOfNodes attribute) as in Solution 1. ClusterVM implements a simple pool allocation strategy: round-robin. This strategy allows to deploy MySQL server instances on different VMs. Regarding ClusterVM’s wrapper (not shown in this paper), it declares methods which implement hypervisor calls to manage VM. The startVM method launches a virtual machine, the shutdown method turns off a VM and the migrate method migrates a VM from one machine to another.

**c) Stage 3:** The configuration schema of the J2EE architecture is nearly the same, with two modifications. Instead of deploying MySQL server instances on physical machines, they are deployed on ClusterVM component instances (i.e. VMs). Besides, according to Solution 3 principle, the maximum number of MySQL servers is initially estimated by the user and they are all deployed at boot time. Also, the wrapper associated with the Probe component is extended to provide advanced monitoring functions. As the Probe component monitors the load of each MySQL server and is aware of which server runs on which node (thanks to the SR), it is able to maintain a full cartography of the load of each host. Therefore, it provides the following monitoring functions: getMostLoadedNode() (the most loaded node), getMostLoadedNode(aVM) (the most loaded node which can host the aVM virtual
Fig. 2. Solution 3: autonomic VM migration with TUNe

machine, getLeastLoadedNode() (the least loaded node) and getLeastLoadedNode(aVM) (the least loaded node which can host the aVM virtual machine).

Reconfiguration programs (Figure 2(b)) in this solution essentially consist in the migration of VMs. When the MySQL tier is detected as overloaded, the fixOverload program identifies the most loaded machine on which VMs are running (stored in the node variable). It then selects a VM running on this machine (vm variable) and looks for the least loaded machine which can receive the migration of that VM (dest variable). Such a migration would better balance the load between the hosts (especially if physical machines don’t host the same number of VMs). If such a migration is not possible, a new machine is allocated to receive the migration.

In order to maintain the SR, a VM component instance is added and associated with the destination host. The nodeName attribute of this instance is updated and the VM is effectively migrated. Finally, the old VM instance (associated with the left node) is removed.

V. EXPERIMENTS

A. Experiment environment

Our environment is composed of 7 machines DELL Opti-path 755, each one with an Intel Core 2 Duo 2.66GHz with 4G RAM. We run a Linux Debian Lenny (with the 2.6.26-2-686 kernel flavor) in a single processor mode. The Xen hypervisor and the OpenVZ solution (in their 2.6.26-2-openvz-686 and 2.6.26-2-xen-686 flavors) are used as virtualization solutions. The network is connected thanks to a DELL Powerconnect 2708 switch. This network allows to share a folder (thanks to NFS) which stores the VM images. We limited our experiments to the management of the MySQL tier. The RUBiS application benchmark is used as J2EE application. RUBiS implements an eBay-like auction system and includes a workload generator which simulates clients. This simulator is configured to send mainly reading requests (95% of the requests are reading one). For each experiment, the execution time is 30 minutes. We progressively start 2000 clients, 50 clients every 20 seconds and each client runs for 15 minutes.

The software architecture that we use for the experiment is composed of: 1 machine with the RUBiS clients, 1 machine for Apache web server, 2 Tomcat application servers located on separate machines, 1 machine for MySQL-Proxy load balancer and finally 2 machines reserved for MySQL servers or VMs.

In order to evaluate the different approaches, we make some comparative tests. First of all, we evaluate the cost of the web hosting center with and without TUNe. Since the reconfiguration policy aims at freeing machines, the main comparison criterion is the energy cost. Then, we compare the resource allocation approaches in three steps: (1) the intrusion level in term of the number of lost requests (Solution 1), (2) the cost of VMs in term of request throughput (Solution 3) and (3) the energy cost of each approach (Solutions 1 and 3). Notice that Solution 2 have the same performance characteristics as Solution 1 (they only differ from the point of view of code and behavior intrusion).

B. Results

1) Benefits from TUNe:

To evaluate the benefits from using TUNe in a J2EE hosting center, we performed an evaluation under three conditions. In the first one, we do not use TUNe and the system is under-provisioned: only one MySQL server is running. In the second one, we do not use TUNe and the system is over-provisioned: two MySQL servers are started on two separate nodes. In the last situation, we use TUNe as a resource allocator (with Solution 1): initially only one MySQL server is started.

Figure 3(bottom-left) shows two (down) peaks: peak (a) when a MySQL server is added and peak (b) when a MySQL server is removed. It also shows that with one MySQL and without TUNe, the system cannot provide the full throughput. It shows that the TUNe system is fully transparent regarding the whole web hosting center: the graphs “Begin with 2 MySQL without TUNe” and “Begin with 1 MySQL with TUNe” are very similar. Therefore TUNe allows to provide the same throughput as the over-provisioned configuration. Finally, thanks to the TUNe system, the two MySQL machines are simultaneously used only 750 seconds (from the 600th second to the 1350th) instead of the whole experiment time. Therefore we save 30.55W in term of energy consumption.

2) Requests loss:

We evaluated the amount of lost requests with Solution 1 during the same experiment as in the previous
section. During the benchmark, the TUNe system adds and removes a MySQL server. While this reconfiguration occurs, some requests are lost: the executing requests in the removed MySQL server and the requests which arrive while the MySQL proxy restarts. The RUBIS benchmark logged that 163 requests were lost during the experiment.

Solution 2 prevents request loss but requires to modify the application. Solution 3, based on VM migration, does not require any modification to the application code and does not lose any request, but adds an overhead at runtime that we evaluate in the next section.

3) Virtualization overhead: The virtualization mechanisms used to address the previous problem may introduce an overhead into our system. We measured this overhead with the same architecture with Xen or OpenVZ without the TUNe system. We compare the request throughput between a MySQL server running on a native Linux system and a MySQL running in a VM. As illustrated in figure 3(top-left), the overhead is nearly nonexistent. These results confirm the results published in [7].

In Solution 3, the J2EE architecture is initially configured with a given number of VMs (say N). When the application is underloaded, all the VMs (running a MySQL server) are co-located on a single physical machine. On the other side, with Solution 1 in underload, the architecture is composed of one MySQL server running on one physical host. So we compare both configurations and we focus on the virtualization overhead introduced by several VMs on the same physical machine. In other words, we want to know if the MySQL stage could be faster overloaded with several VMs on the same machine compared to a configuration with only one MySQL and no VM at all.

We used up to 6 virtualized MySQL server on the same machine. We computed that 6 is the maximum number of VMs that we can run on one machine, based on the physical resources (CPU, RAM, ... ) of our machines. In our context, we need 512MB of RAM to run a MySQL server over a VM. We allocated 1GB of RAM for the host operating system and 3G RAM for all the 6 VMs. Figure 3(top-right) shows that there is no significant overhead running several VMs on a single machine.

4) Migration solutions: global evaluation: We measure the request throughput for Solution 3 based on VM migration. The system is deployed with 2 MySQL servers (thus 2 VMs). We only present here the results we obtained with Xen (Figure 3(bottom-right)). Similarly as the results obtained with Solution 1, we observed that: the peak (a) occurs when a VM is migrated toward a new machine and the peak (b) occurs when VMs are gathered on a single machine (the system is underloaded).

Whatever the solution we consider, our experiments show that RUBIS’ request throughput and energy consumption are nearly the same. However, in Solution 3, no request is lost and no application modification is required. We also show that TUNe is able to take into account different administrated environments with different administration policies.

VI. Related work

Many research works, rely on autonomic computing platforms and virtualization, have investigated dynamic resource allocation in hosting center environments. In this section, we present some major works classified into two categories.

The first category relies on dynamic server allocation thanks to autonomic computing mechanisms. IBM’s Oceano project [8] allows to dynamically allocate resources in a utility computing environment. The Oceano platform reacts to an increasing load by building servers on the fly (i.e. deploying the operating system and the software). This solution get the same problem than Solution 1: the newly built server does not have the same state than the others of the same kind. Urgaonkar [9] is a very close approach to the IBM one but with a predictive mechanism. Unlike our approach, the prediction mechanism takes into account the past behavior of the system. This approach addresses the problem of lost state or lost requests. Unfortunately, we can use this approach in only few cases. For instance, a server must be able to identify that it doesn’t handle sessions anymore. In his paper [10], Hady AbdelSalam presents a dynamic allocation architecture for a hosting center based on an autonomic computing system and a load balancer. The load balancer has a pool of allocated nodes and distributes the processes over this pool. The main drawbacks are the overuse of the nodes: a node without any processes is not released (unlike with our approach) and the load balancer itself can get overloaded. Also, in [10], the administration policy is implemented in the load balancer which is very intrusive and not generic. Concerning energy saving, Chase [11] permits to power on and shutdown nodes in a hosting center according to the system load. This approach get the lost state and lost requests drawbacks. Finally GreenCloud [12] aims at saving resource in an Internet Data Center. Like us, GreenCloud migrates virtual machines but with higher level policies.

The second category gathers the dynamic grid management systems. There are two kind of systems: the static one and the dynamic one, depending on the classification criteria. Indeed, VMPlants [13], Cluster-On-Demand (COD) [14] can be considered either dynamic or static systems. For instance, COD is dynamic because of its ability to detect and to take account of grid modifications (new machines arrival). But we focus on the capacity of the systems to react to the application behaviors and in this case, COD is static. Among these works, none is interested in providing support for resource management at the many different levels, e.g.: virtualization, software over virtual machines and nodes resources. Our approach takes into account these different levels.

VII. Conclusions and perspectives

In order to save exploitation costs of their infrastructures, companies are more and more interested in hosting center platforms. These platforms must provide different technical services, among which dynamic resources allocation. In this paper, we proposed to rely on an autonomic computing platform to implement dynamic resource allocation for hosted applications. We presented three approaches to address the
dynamic resource allocation problem in a hosting center and applied them to the resource management of a hosted J2EE application.

The first approach dynamically adapts the number of servers according to the application load, without any modification to the hosted application. This solution has important side effects as its behavior is altered by state and request losses.

The second approach is similar to the previous, but we modified the hosted application to prevent the side effects. This solution has the drawback to be very intrusive at the level of the code of the hosted application.

The third solution relies on virtual machines and their migration in order to increase or reduce the number of physical machines involved in the hosting of the application. This last solution is the best as it provides the best tradeoff between performance and cost. Indeed, nearly no overhead is introduced into the system, no side effect can be measured and it is versatile: we can use it with several virtualization solutions like OpenVZ or Xen and we can use it to run other software than J2EE servers.

In the near future, we plan to apply this work in the HPC research area and especially with the MPI [15] software. MPI applications communicate with message exchanges and thus our load balancer is inaccurate. We want to modify our approach to take into account MPI processes’ CPU load and to collocate processes which communicate intensively.

REFERENCES


