Int. J. Communications, Network and System Sciences, **2011, 4, 790-802**
doi:10.4236/ijcns.2011.432097 Published Online December 2011 (http://www.SciRP.org/journal/ijcns)

Towards Cloud Management by Autonomic Manager Collaboration

Omid Mola, Michael A. Bauer
Department of Computer Science, University of Western Ontario, London, Canada
E-mail: *omola@csd.uwo.ca, bauer@csd.uwo.ca*
Received September 29, 2011; *revised November* 8, 2011; *accepted November* 24, 2011

Abstract

The management of clouds comprised of hundreds of hosts and virtual machines present challenging problems to administrators in ensuring that performance agreements are met and that resources are efficiently utilized. Automated approaches can help in managing such environments. Autonomic managers using policy-based management can provide a useful approach to such automation. We outline how collections of collaborating autonomic managers in cloud can be a step towards better management of clouds. We describe how a hierarchy of policy-based autonomic managers can collaborate using messages. The messages and when to communicate is inferred automatically from the policies given to the managers. We evaluate the approach via a prototype inspired by a cloud virtualized infrastructure and show how collaboration between managers in a hierarchy can improve the response time of a web server and avoid service level agreement violations. Results of three different scenarios shows the importance of collaboration between managers at different authority levels and how this collaboration can help to increase efficiency of current infrastructures.

Keywords: Autonomic Management, Collaboration, Policy-Based Management, Cloud Management

1. Introduction

Cloud computing environments often depend on virtualization technology where client applications can run on separate operating virtual machines (VMs), particularly for providers of Infrastructure as a Service (IaaS). Such environments can consist of many different host computers each of which might run multiple VMs. As the number of hosts, virtual machines and client applications grow, management of the environment becomes much more complicated. The cloud provider must worry about ensuring that client service level agreements (SLA) are met, must be concerned about minimizing the hosts involved, and minimizing power consumption. Our focus is on how to better manage the virtual machine and system infrastructure of the cloud provider.

In recent years there has been a lot of research into "Autonomic Computing" [1], especially about how to build autonomic elements and managers [2]. Autonomic managers try to monitor and manage resources in real time by building systems that are self-configuring, self-optimizing, self-healing and self-protecting. In the broader vision of autonomic computing, large complex systems will consist of numerous autonomic managers handling systems, applications and collections of services [3]. Some of the systems and applications will come bundled with their own autonomic mangers, designed to ensure the self-properties of particular components. Other mangers will be part of the general management of the computing environment. The complexity of managing a large system will entail a number of different autonomic managers which must cooperate in order to achieve the overall objectives set for the computing environment and its constituents. However, the relationships between these managers and how they cooperate introduce new challenges that need to be addressed.

We consider the use of policy-based managers in addressing this problem and with an initial focus on a hierarchy of autonomic managers where policies are used at each level to help managers decide when and how to communicate with each other as well as using polices to provide operational requirements. The ultimate goal is to automatically monitor and manage a larger system by a collective of collaborating local autonomic managers (AMs). In such an environment we assume that each local AM has its own set of policies and is trying to optimize the behavior of its local elements by responding to the changes in the behavior of those elements. We as-

sume some managers will also be expected to monitor multiple systems and directly or indirectly to monitor other local AMs. We also assume that one of the roles of a "higher level" manager is to aid other AMs when their own actions are not satisfactory.

The focus of this paper is on collaboration and communication between different managers at different levels of the hierarchy based on the active policies. The core issue addressed is how these local managers should communicate with each other and what information they have to exchange to achieve global performance goals. Finally, we will show how to automate the collaboration process itself.

2. Related Work

Some researchers have already begun to study how the collaboration or cooperation among local autonomic managers can be done in order to achieve a global goal. This work has looked at hierarchical organization of managers for cooperation, agent-based approaches and registry-based techniques [4-7].

A hierarchical communication model for autonomic managers has been used by some researchers. Famaey and Latre [4] used a policy based hierarchical model to show how it can be mapped to the physical infrastructure of an organization and how this hierarchy can dynamically change by splitting and/or combining nodes to preserve scalability. They also introduced the notion of context that needs to be accessible in the hierarchy, but do not describe in detail what this context should be and how it should be communicated. In this paper, we focus on what this context should be, how it can be transferred from one manager to the other and when this should happen.

Aldinucci, *et al*. [8] described a hierarchy of managers dealing with a single concern (QoS). They introduce three types of relationship between components but do not explore the details of how and when such components should interact in actual systems. They used a simulator to evaluate the framework and their main focus was on the concept of a "behavioral skeleton" where they used autonomic management for skeleton-based parallel programs.

Mukherjee, *et al*. [9] used a flat coordination of three managers working on three different parts of a system (Power Management, Job Management, Cooling Management) to prevent a data center from going to the critical state. They showed how the three managers can cooperate with each other to keep the data center temperature within a certain limit that is suitable for serving the current workload and at the same time not using more power than required. They showed how these three managers should cooperate based on different business policies. However, these three managers are fixed and adding new managers to this system will be challenging both in terms of collaboration and scalability.

The same approach as in [9] is used in [10,11] to show the collaboration between a power and a performance manager (only two managers) to minimize the power usage as well as maximizing the performance. This method however does not seem to be generalizable to a larger environment with more autonomic managers involved because of the complexity introduced in terms of interacttions between managers.

Salehi and Tahvildari [12] proposed a policy-based orchestration approach for resource allocation to different autonomic elements. They proposed that the orchestrator get the requests from autonomic elements and coordinates elements using some global policies. The same kind of approach is used in [13] by having a coordinating agent that tries to coordinate power and performance agents. This approach could be used as part of the hierarchical approach that will be presented in this paper but it does not seem to be applicable to a larger system just by itself, because of scalability issues. This is actually a special case of the hierarchical approach discussed in this paper, but with only one level of hierarchy.

Schaeffer-Filho, *et al*. [14,15] have introduced the interaction between Self-Managed Cells (SMCs) that was used in building pervasive health care systems. They proposed "Role" based interactions with a "Mission" that needs to be accomplished during an interaction based on predefined customized interfaces for each role. This approach is very general and does not address the details of the interactions. In the work presented in this paper, we will address what the policies look like and what specific information needs to be exchanged.

3. Cloud Management

3.1. Architecture & Virtual Machines

The infrastructure of IaaS providers, such as Amazon EC2, is typically composed of data centers with thousands of physical machines organized in multiple groups or clusters. Each physical machine runs several virtual machines and the resources of that server are shared among the hosted virtual machines. Therefore, there are a large number of virtual machines that are executing the applications and services of different customers with different service level requirements (via Service Level Agreement (SLA) parameters).

It is also possible for a customer and the service provider to mutually agree upon a set of SLAs with different performance and cost structure rather than a single SLA. The customer has the flexibility to choose any of the

agreed SLAs from the available offerings. At runtime, the customer can switch between the different SLAs.

Depending on the load and resource usage of a virtual machine, the cloud provider should allocate enough resources to ensure that the service level requirements are met. On the other hand, if there is a load drop and no need for allocated resources, it is advantageous for the service provider to remove some of the extra resources and to reallocate them to other virtual machines in need.

To have a better understanding of cloud provider environment and architecture, we take a closer look at Eucalypstus [16] (an open-source infrastructure for the implementation of cloud computing on computer clusters). In Eucalyptus, there are five elements that form the cloud infrastructure (see **Figure 1**):

- Cloud Controller (CLC)
- Walrus Storage Controller (WS3)
- Elastic Block Storage Controller (EBS)
- Cluster Controller (CC)
- Node Controller (NC)

These elements can physically locate on one single machine to form a small cloud but each one has a different role in forming the cloud infrastructure.

The CLC is the top level component for interacting with users and getting the requests. The CLC then talks with the Cluster Controllers (CC) and makes the top level choices for allocating new instances of virtual machines.

The Cluster Controller receives requests from the Cloud Controller and in turn decides which Node Controller will run the VM instance. This decision is based upon status reports which the Cluster Controller receives from each of the Node Controllers. ESB and WS3 are used for storage control of the images inside cloud.

A Node Controller (NC) runs on the physical machine responsible for running VMs and the main role of the NC is to interact with the OS and hypervisor running on the node to start, stop, deploy and destroy the VM instances.

3.2. Challenges

All of the specified elements in the cloud architecture are needed for instantiation of new images or destroying currently deployed VMs and they have some minimal management capabilities. However, the main challenges in managing the cloud environment occur after the VMs start working and receiving loads:

- How should the system respond to the load changes inside one or more virtual machines?
- What should happen to maximize the performance of a specific virtual machine (or an application inside it) according to the agreed SLA?
- How can we scale the system up and down on the fly (change VM parameters)?
- How can one enforce specific operational policies for the entire system?
- How can one make sure that minimum resources are used to perform a task (e.g. minimizing the power usage)?

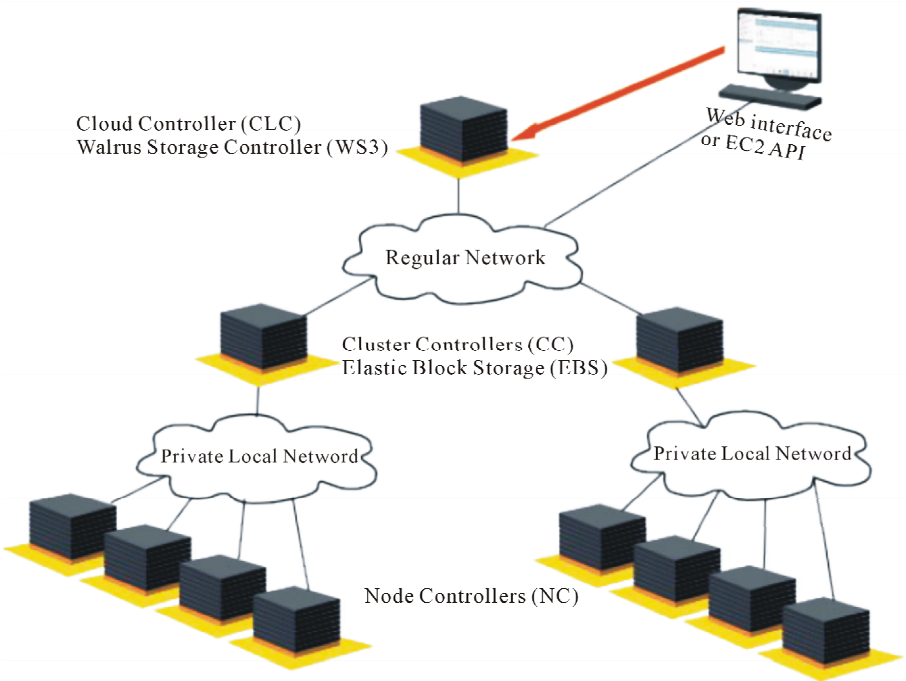

Figure 1. Eucalyptus cloud architecture (from [17]). It shows the hierarchy of cloud architecture and how controllers are interacting with each other in this hierarchy.

A deeper look at the cloud architecture and the management needs suggest that providing all these capabilities through a single centralized manager is almost impossible, because the cloud is composed of hierarchical layered elements with different responsibilities at each layer. Any management system needs to deal with all these layers to be able to provide the management capabilities. Therefore, a hierarchical approach towards cloud management would be a more efficient way to achieve all of the goals. At the same time, each element in the management hierarchy should act autonomously and manage part of the hierarchy on its own.

4. Approach

Based on the previous discussions, we propose to use a number of different autonomic managers. We need one AM for each VM and its applications, though we could also consider AMs for applications in the VM as well. We will also assume AMs at each layer in the cloud architecture. By using this approach, the problem of managing a large system entails a number of autonomic managers where each one is dealing with smaller or more localized components, and then each manager's job is to focus on managing that component (or small set of components) efficiently.

For example, an AM for an Apache web server should only focus on the behavior of the web server and not the relationship that it might have with a database server and the Node Controller (NC) AM should only focus on the behavior of the VMs inside that node and the general performance of that node.

The hierarchy of autonomic managers might appear as in **Figure 2**. In the lowest level the AMs are managing the applications inside the VMs. The AMs at the node controller (NC) level monitor and manage the VMs. Then the AMs at cluster controller (CC) level are responsible for all physical nodes inside that cluster. Similarly the AM at cloud controller (CLC) level monitors and manages all of the clusters.

Note that this is a logical organization of autonomic managers and does not necessarily reflect the physical allocation of the AMs, *i.e.*, they do not need to be located on different physical machines. In a large cloud computing provider they could be located on separate machines or some may be located on the same machines. These AMs should then collectively work together to preserve a set of policies for optimizing performance, minimizing resource usage, avoiding SLA violations, etc.

The hierarchy of managers can be expanded dynamically into more levels as required. A good example of splitting and combining elements in the hierarchy is illustrated in the work of Famaey, *et al.* [4] to improve the scalability of the hierarchical approach; we have not considered this in this paper.

For management to happen, the big or more complex tasks should be divided into smaller tasks and delivered to different responsible managers at lower levels. For example, the AM at the Cloud Controller (CLC) level should take care of the balancing the load between different clusters and the AM at the Cluster Controller (CC) level should look after balancing the load between different nodes inside that cluster. Similarly, the AM at the node level should optimize the resource usage of that physical machine among different VMs and while the AM inside a VM should work on optimizing the applications' performance.

Assuming that the management "tasks" are specified in terms of policies, this means that we need policies with different granularity deployed at different levels of the infrastructure and we need to ensure that AMs can communicate properly with each other to enforce those policies.

4.1. AM Requirements

In this section we explain the assumptions we have for a general autonomic manager to work in collaboration with other managers. Although AMs are heterogeneous and can belong to different vendors, they should all follow some specifications to make the collaboration possible.

We assume that inside each AM there is an event handling mechanism for processing, generating events and notifying the interested parties inside the AM. For example, there could be an event bus and different subscribers

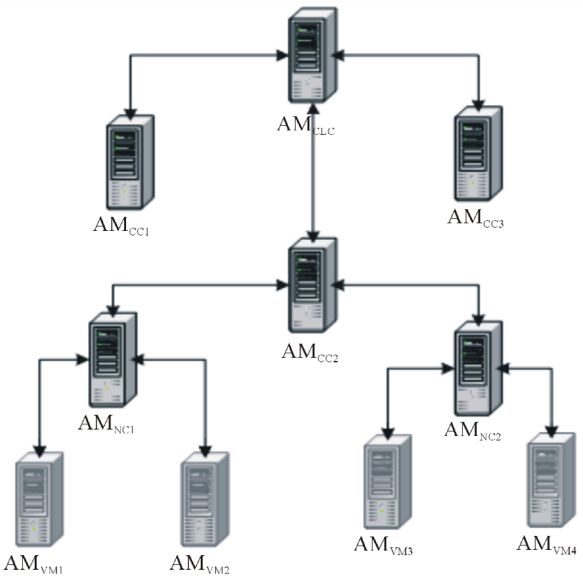

Figure 2. Hierarchy of autonomic managers based on the cloud architecture.

to certain events (within the AM) and upon raising those events any subscribers will get notified. This event handling mechanism is useful for handling event, condition, action policies (explained in the next section) and also for communication between managers (explained below).

Each manger should also provide an interface for receiving messages from other managers. This interface should be able to receive different message types (e.g. M1, M2, M3 explained below), parse them and do the proper actions according to the specification of that message. It is important, therefore, that there be a small number of different types of messages; this is presented in this paper (Section 4.3).

We also assume that managers can interact with the applications, VMs, etc. (*i.e*., "objects") in their managed environment through some interfaces called "ManagedObjects". For example, if a manager is responsible for managing some VM, then properties and metrics of the VM is encapsulated in the VMManagedObject. Other possible managed objects are: ApacheManagedObject, VMManagedObject, NodeManagedObject, ClusterManagedObject.

Each managed object has a set of properties, metrics and actions associated with it. Properties of a managed object are defined in the definition of the managed object and are set upon instantiating a new managed object. The metrics associated with a managed object are those properties that change more often and therefore must include actions specifying how it is to be defined and how they can be updated (refreshed). Actions are operations that can be done on that managed object. For example, all VMManagedObjects have a common set of properties: VM name, VM allocated memory and VM operating system type. They would likely have metrics like CPU utilization and memory utilization and therefore would also include an action to get the new values for each of these metrics (e.g. by connecting to the AM inside the VM and sending a message to get the updated values). Other actions for a VMManagedObject could be StartVM, StopVM, GetVMIP, etc. These would be defined by the system administrator/designer inside these managed objects.

The actions, metrics and properties defined inside managed objects can later be used in policies to evaluate a specific condition or to perform an action on that managed object.

4.2. Policy-Based Management

We assume each AM operates based on a set of policies provided to it. An overview of policy-based management along with relevant standards and implementation techniques can be found in Boutaba, *et al*. [18].

An AM can have different types of policies which can be useful for certain purposes. For example, an AM might rely on *configuration* policies for self-configuretion of managed elements, or might utilize *expectation* policies for optimization of the system or for ensuring that service level agreements (SLAs) are met.

In this work, we use expectation policies expressed as event, condition, action (ECA) policies. In general, all of our policies are of the form:

On event: *E*
if (*Set of Conditions*) *then*
{*Set of Actions*}

Upon raising an event inside the autonomic manager, then any policy which matches the event will get evaluated. If the conditions in the policy are met, then the policy actions get triggered. We provide examples of policies in the following sections.

At AM startup there are configuration policies that set up the AM environment, identify the appropriate managed objects and configure them. A sample configuration policy would look like:

On event: *StartupEvent*
if (*true*) *then* {
setFatherIP: "192.168.31.1"
VMManagedObject create: *vm*1.
*vm*1 *setIP*: "192.168.31.3".
}

This policy happens on AM startup and configures the parent's IP of this AM in the hierarchy and also adds one ManagedObject for managing *vm*1. This AM will be responsible for managing *vm*1 and will communicate with the manager inside *vm*1 if necessary. The AM hierarchy can be built this way upon system startup but it can change dynamically throughout their lifetime (e.g. by migration of a VM to another machine). We will show an example of this dynamic change in Section 5.1.3.

4.3. Communication Model

In previous work [19], we suggested the use a message-based type of communication between AMs. Several different types of messages were proposed as sufficient for communication between managers:

Msg = <*Type*, *Info*>
Type = *NOTIFY*|*UPDATE_REQ*|*INFO*
Info = *Metrics*|*Details*
Metrics = {<*m*,*v*>|*m is the metric name, v is the metric value*}
Details = <*T*, *Metrics*>
T = <*HelpReq*, *SLAViolation*, ...>

By using a message "Type", we introduce the possibility of different types of relationships between managers (e.g. request, response) and based on the type of

message, one manager can expect the kind of information that would be available in the *Info* section of the message. The *Info* can be the latest metrics of elements managed by a particular local manager or could be details on some event that has happened. Having a small set of different types of messages also makes it easy to define the operation of each AM.

Since we are dealing with a hierarchy of managers then each manager needs to communicate with either its father or its children. However, it is also possible for an AM to send NOTIFY messages to another AM in some other part of the hierarchy based on a request (we will provide an example of this in Scenario 3 of the experiments). So, each AM knows the address of its father and children for a point to point communication (e.g. sending messages) at the time of startup but it will get the address of other AMs as part of NOTIFY messages, if there is a need for it.

The form of each of these types of messages is as follows:

- ***M*1 = <*NOTIFY, Details*>:** When one manager wants to raise an event in another manager it can be encapsulated inside a notify message. The type and content (*Metrics*) of the event is very system specific and can both be defined in the *Details* portion of the message. Possible events would be a "help request event", "policy violation event", "system restart event", "value update event", etc. We illustrate this type of message in the next section. When a manager receives a notify message from another manager, it will raise an event inside the event bus and deliver it to interested subscribers (e.g. evaluate proper policies).
- ***M*2 = <*UPDATE_REQ, Metrics*>:** This is a message asking for the status of the metrics declared in *Metrics*. Another manger can respond to this message by sending an INFO message back. These *Metrics* can be specifically declared in policies that used for a communication or it can be inferred automatically from policies. The *Metrics* are very dependent on the nature of the system and can be different from one system or application to another. Examples of such information include CPU utilization, memory utilization, number of requests/second, number of transactions, available buffer space, packets per second, etc.
- ***M*3 = <*INFO, Metrics*>:** This is a message that provides information about metric values, which can help the process of decision making in the higher level manager. This message is usually sent in response to the UPDATE_REQ message from a higher level manager (e.g. M2 explained before).

The UPDATE_REQ message is sent from higher level managers to lower level ones. INFO messages are sent in response to the UPDATE_REQ message and NOTIFY messages are sent from one manager to another based on the need. We will explain in more details how we can use policies to generate these messages for communication among AMs based on demand.

4.4. Inferring Messages from Policies

In order to better illustrate the problem and approach, we will show several examples of policies that can be used at different levels of a hierarchy and how these policies can influence the relationship between managers.

Assume that on each VM there is a LAMP (Linux-Apache-Mysql-PHP) stack that hosts the web application and that one AM is managing the applications inside that VM. We use event, condition, action (ECA) policies to specify operational requirements, including requirements from SLAs, and we also use policies to identify and react to important events.

Assume that the following policy is being utilized by AM_{vm1} and is a policy specifying the requirements needed to meet an SLA. The policy indicates that the Apache response time should not go above 500 ms. This policy gets evaluated once a "ManagementIntervalEvent" event happens and there is a configurable timer that triggers this event at certain intervals (e.g. every 1500 ms).

On event: *ManagementIntervalEvent*
if (*apache*::*responseTime* > 500)
    apache::*increaseMaxClients*: +25 *max*: 200

This policy specifies that if the response time of the Apache server goes beyond 500 ms, then the manager should increase the MaxClients configuration parameter by 25. The policy also indicates, however, that this cannot be done indefinitely, but that the limit for MaxClients is 200, which means that the manager should not increase it to more than 200.

Another sample policy for an AM at the node controller level (e.g. AM_{nc1}) is:

On Event: *HelpRequestEvent*
if (*vm*1:: *memoryUtil* > 85 & *vm*1::*cpuUtil* > 95)
    *vm*1:: *increaseMem*: +50 *max*: 500

Upon receipt of a "HelpReq" notify message from another AM (e.g. AM_{vm1}), a HelpRequestEvent gets triggered inside the receiving manager and those policies that match that event evaluated by the manager.

This policy specifies that when a HelpRequestEvent happens, if the memory utilization of the VM in need is more than 85% and its CPU utilization is more than 95%, then the manager should increase its memory by 50 MB. Again, this can only be done to some limit. In this case, the maximum limit is 500 MB.

One of the challenges in collaboration between managers is to determine when they need to send/receive a message from another AM in the hierarchy. Since we are

using a policy-based approach, one way to specify when a message should be sent is to have a specific policy that determines when an AM is to communicate. For example, one could include a policy explicitly identifying a communication action to send a help request event from a lower level to a higher level manager; such as:

On event: *ManagementIntervalEvent*
if (*apache*:: *responseTime* > 1000)
sendHelpRequestEvent.

This approach requires a substantial work by the administrators in order to define all the policies needed.

An alternative would be to automatically infer from policies the right time for sending a message and the content of the message; this is the focus of this paper. In the remainder of this section, we explain how autonomic managers can infer the right message type and the right time for sending a message to another AM. Based on the Message Types proposed, if a manager has detected a policy violation and has tried to take an action, but has reached some limit in changing a parameter and can no longer do a local adjustment, then it will create a help request message and send it to the higher level manager. That is, as long as there is something that can be done locally there is no need for further communication unless it is an UPDATE_REQ message.

Therefore, by defining a "maximum" limit in a policy, an AM can determine exactly when it has reached the local limits and create the right message to be sent to the higher level manager. Thus, the AM can infer automatically from the policy when to send this type of message at run time. This can happen when the manager has reached the limit specified in trying to take actions specified in a policy.

For example, if AM_{vm1} increased MaxClients to 200, it has reached the local limit and can no longer increase it more. Then, upon detecting a policy violation for the Apache response time, AM_{vm1} cannot increase MaxClients further, and so will create a HelpReq message and send it to the higher level manager. A general form of this message is like this:

***M*1 = <*NOTIFY, Details*>**
***Details* = < *HelpReq*, {<*m*1, *v*1>, <*m*2, *v*2>, ...} >**

Based on this technique we can build a system with different AMs working autonomously at different levels and interacting with each other based on demand but the important point is that all these AMs are collectively trying to adhere to a set of policies that minimize the number of SLA violations (or maximize performance based on SLA parameters), and minimize resource usage at the same time. In this work, we assume that these policies are defined by system administrators and are given to different AMs for enforcement. This happens while each manager has a local view of the system and is trying to solve problems locally but when no further local adjustment is possible it asks the higher level manager for help.

An UPDATE_REQ message is of the general form:

***M*2 = <*UPDATE_REQ, Metrics*>**
***Metrics* = {<*m*1, *null*>, <*m*2, *null*>, ...}**

This type of message can also automatically be inferred from policies and sent to the lower level manager to get the updated results. For example, a policy at AM_{nc1} for refreshing the metrics related to VMs could be:

On Event: *RefreshIntervalEvent*
if (*true*)
managedSystem:: *refresh.*

The AM automatically contacts all of its children via the UPDATE_REQ message to get the latest metrics related to each managed object under its control (e.g. VMs).

At AM startup, each managed object will get configured with proper values for properties and then at run time the AM can automatically figure out metrics that need to be refreshed and generate the proper message for updating them. This message will look like:

Msg = <UPDATE_REQ, {<cpuUtilization, null>,
<memoryUtilization, null>}>

Then, upon receipt of this message by the AM_{vm1} and AM_{vm2}, a reply INFO message is automatically generated to be sent back. If for any reason, these AMs cannot calculate these values then they can send an INFO message back with null values which shows that there was a problem in getting values for the requested metrics. The general form of the INFO message is:

***M*3 = <*INFO, Metrics*>**
***Metrics* = {<*m*1, *v*1>, <*m*2, *v*2>, ...}**

and an example of the message to be returned would look like:

Msg = <INFO, {<cpuUtilization, 60>, <memoryUtilization, 75>}>

There can be separate policies that specify the need for sending these messages but the important point is that it is not necessary. Rather, these messages can be automatically inferred from what is defined in the policies.

4.5. Management Policies

One more aspect that needs to be considered in the management of this cloud environment is that AMs can dynamically join or leave the hierarchy and that at each level of the hierarchy we need to enforce some similar policies. How we can make sure that right policies get created and put in place automatically on the fly? The answer to this question is through a set of defined general management policies that are responsible for creating and/or removing other concrete policies.

For example, if the manager at the node controller (e.g. AM_{nc1}) is responsible for helping the VMs in need, then a policy can be created on the fly for each VM that joins the hierarchy and also gets removed once a VM has been removed or migrated from this node to another. A example of this kind of policy is:

```
On event: NewManagedObjectEvent
if (newManagedObject is a VM) {
   create a new policy {
      On Event: HelpRequestEvent
      If (vmName == HelpRequestEvent.name &&
            vmName::memoryUtil > 85)
   vmName:: increaseMem: +50 max: 500.
   }
}
```

This policy says that upon detecting a new managed object event (which might happen at AM startup for configuring the system or after one VM is migrated to another machine), if that managed object is a VM then our manager should create another policy and activate it. We will explain an example of VM migration and how this event can be triggered in the scenario 3 of experiments.

This new policy says that upon receiving a HelpRequestEvent, if the name in the event matches the name of this VM (the event is coming from the manager inside this VM) and this VM's memory utilization is above 85% then increase its memory by 50 MB up to maximum of 500 MB.

In fact, this is a policy for creating policies and should be defined by administrators at each level of the hierarchy. This form of a policy is very useful for creating and applying general rules to every instance at a certain level of hierarchy. It will help administrators define what is expect to happen in a dynamic environment and the right policies will be created based on demand for each instance. The use of these management policies also simplifies the work of the administrator in that for specific policies may not have to be created, e.g. for a certain type of VM.

A similar approach can be used for deactivating or removing other policies related to a VM instance that has been migrated to another host in the hierarchy and the current manager does not need to take care of that particular VM any longer.

5. Prototype

In order to evaluate our approach, we used two VMs running on a single server with LAMP installed on them and a two tier web application based on an online store was configured to run on the VMs. There was also a privileged autonomic manager running in the physical server and its job is to manage (optimize based on policies) the behavior of that server by collaborating with the managers running inside VMs.

We used KVM virtualization [20] with an Ubuntu distribution to build the guest VMs. "Domain 0" is the first guest operating system that boots automatically and has special management privileges with direct access to all physical hardware by default. The manager running inside Domain 0 has the authority to change the configuration of other VMs such as allocated memory, allocated CPU cores, etc. **Figure 3** shows the physical structure of the system.

We implemented the autonomic manager using the Ponder2 [21] system and used Ponder Talk for communication between managers.

Each of the AMs has its own set of policies and tries to optimize the performance of the local system. Manager AM2 (see **Figure 3**) manages physical server "1", trying to optimize its performance and behavior based on the policies given to it. This includes monitoring the other VMs (VM1 and VM2) in order to help them when they are in need. Because AM2 is running in domain0, which is a privileged domain, it can change/resize VMs.

Although we have implemented this system for only two levels of hierarchy, the architecture and concepts used are generalizable to the larger systems such as an entire organization, a data center, etc. **Figure 4** shows the hierarchy and relationship between AMs in our system.

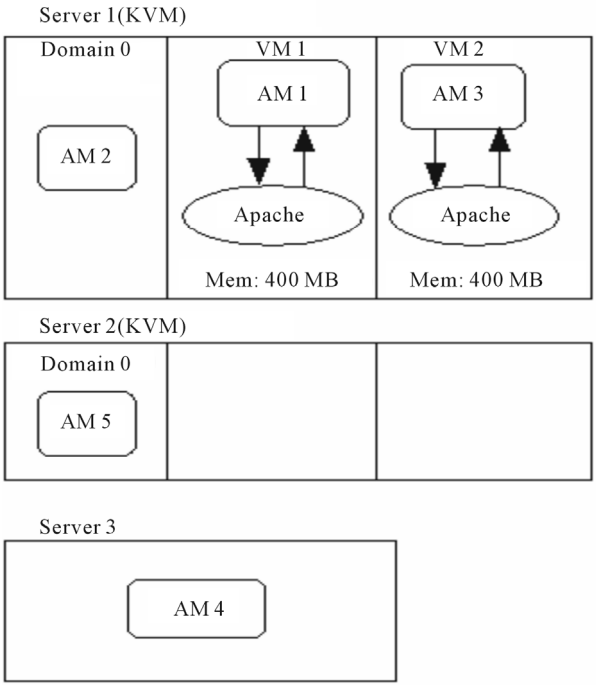

Figure 3. Physical layout of the experiments which consist of three physical servers. Two of them are using KVM virtualization for running VMs; server1 hosts two VMs each running a web app that receive loads.

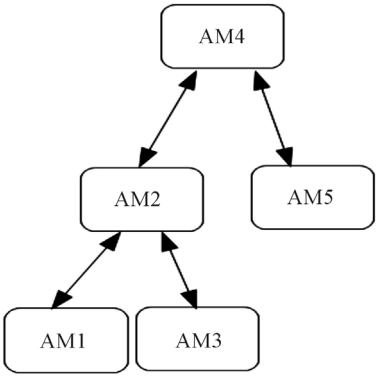

Figure 4. Hierarchy of managers based on physical layout.

5.1. Experiments and Results

We used an open source online store called "Virtuemart" [22] to measure the response time of Apache web server running on VM1. We used JMeter [23] to generate loads to this virtual store and measured the response time of Apache in three scenarios. The ultimate goal of the whole system is to keep the response time under a certain threshold (e.g. 500 ms) that we assumed was defined in an SLA.

5.1.1. Scenario 1: No Collaboration

In the first scenario we disabled all communications between managers. In this case, only the local managers tried to optimize the system based on policies that they had. **Figure 5** shows the response time of the Apache web server in this case.

In this case, when the load increases the local manager tries to adjust the web server by allocating more resources. For example at points A, B, C and D in **Figure 5** an SLA violation was detected by the manager. In response to the SLA violation at points A, B and C and based on the policies explained before, the autonomic manager (AM1) increased the MaxClients property of the Apache server that it was managing by 25. At point D it also detects an SLA violation, but cannot increase MaxClients since it has already reached the maximum value for the MaxClients property (*i.e.*, 200).

As a result, the system will face more SLA violations and the response will get worse, as can be seen by the graph of the response time in **Figure 5**. Thus, the load is more than what this system can handle alone. This also causes a long term violation of the SLA which could mean more penalties for the service provider.

We calculate two measures of the performance of the system and managers in this case: the total time that the system could not meet the SLA (T) and the percentage of time that the system spent in a "violation" (V). For these experiments each time interval was 1 second. Therefore, the results for Scenario 1 are:

T_1 = 18 seconds

S_1 = Total time = 25 seconds

$V_1 = T_1/S_1 = 0.72 = 72\%$

5.1.2. Scenario 2: One Level Collaboration

In the second scenario, we consider the situation when the

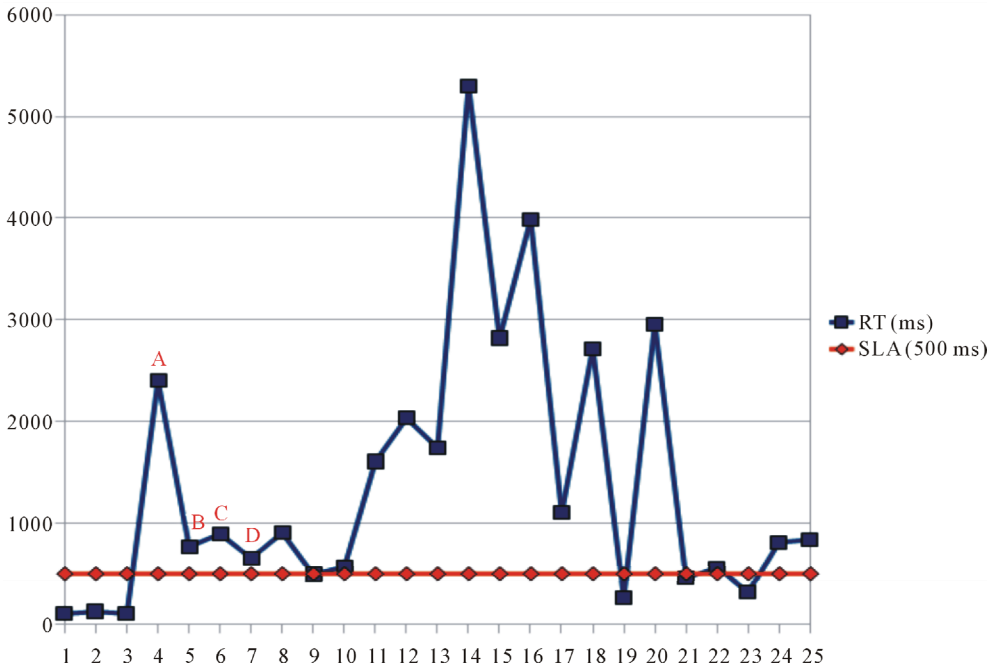

Figure 5. Apache response time with no manager collaborations.

local manager can request help. When the local manager can no longer make adjustments to the system, it requests help from the higher level manager. This is specified in the policies of AM1 and AM2, as mentioned in the previous section with the exception that in this case memory limits can change. For example, a policy of AM2 would be:

limit = 500.
On Event: *HelpRequestEvent*
if (*vm*1::*memoryUtil* > 85 & *vm*1::*cpuUtil* > 95)
*vm*1:: *increaseMem*:+50 *max*: ***limit***

The current limit for increasing memory is set to a default value (e.g. 500MB) but it can change over time based on the changes in the system. We will see an example of this in Scenario 3. **Figure 6** shows the Apache response time in this Scenario.

As in the previous scenario, the local manager (AM1) tries to adjust the web server to handle the increasing load at points A, B, C and D. Eventually, there are no more local adjustments possible (after D) and so the local manager does nothing. In this case, however, when the next SLA violation happens (point E), AM1 generates a "help-request" message and sends it to AM2. In response, AM2 allocates more memory to VM1 (according to the "VM-Mem" policy). At this point, the response time starts decreasing, but since the load is still high, AM1 detects another SLA violation at point F and asks for help again, and AM2 allocates 50 more megabytes of memory to VM1.

After the adjustment of memory at point F, there is a sharp spike in the response time as the VM is adjusted to accommodate the increase in memory allocated to it. Once this is completed, the response time decreases. There are still subsequent instances where there are occurrences of heavy load and occasional SLA violations still happen. In these cases, AM1 still sends the help request to AM2, but since AM2 has allocated all available memory to VM1 (as per its policy), it cannot do more and simply ignores these requests. To solve this problem, we add another level of management to the system.

Based on the output for this scenario, we calculated the same measures of performance:

T_2 = 10.5 seconds
S_2 = 25 seconds
$V_2 = T_2/S_2 = 0.42 = 42\%$

As is evident is the graph (**Figure 6**), the time that the system spends in "violation" of the SLA is much less.

5.1.3. Scenario 3: Two Level Collaboration

In the third and final scenario, we use another level of management to help reduce the occasional SLA violations happened in Scenario 2. **Figure 7** shows the Apache response time in this case.

Like the previous scenarios, the local manager (AM1) tries to adjust the web server at points A, B, C and D. At points E and F, AM2 assigns 50 more megabytes to VM1 to solve the stress. At point G there is another SLA violation. At this point, AM1 asks for help from AM2 but since AM2 already assigned all the available memory as per its policy, it cannot provide more help and automatically creates a help request which it sends to its parent (AM4; see **Figures 3** and **4**).

AM4, running at the cluster control level, has a global view of all physical servers and finds the least busy server. It then tells the AM2 to migrate one of the VMs to that server. This happens according to the following policy in

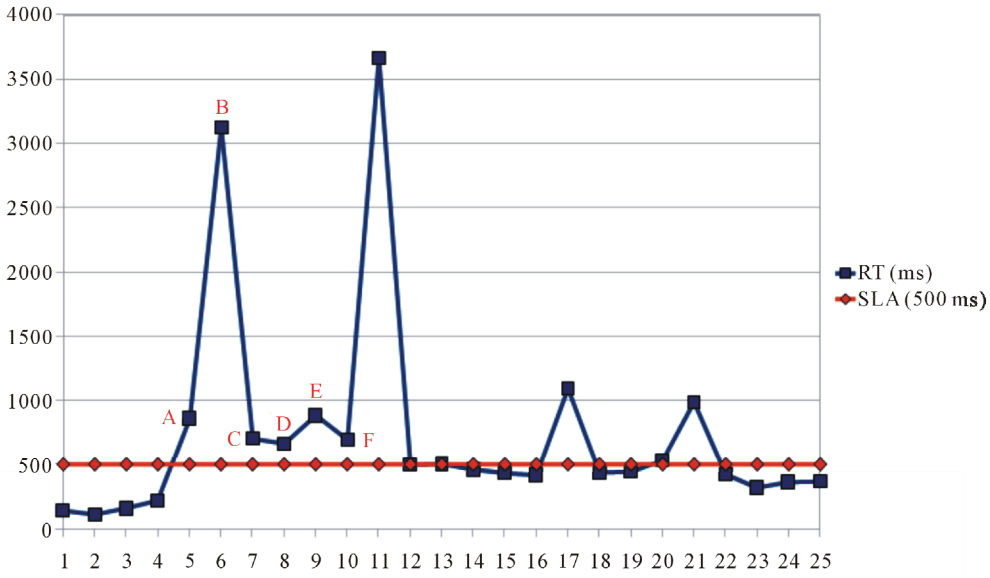

Figure 6. Apache response time with one level of collaboration.

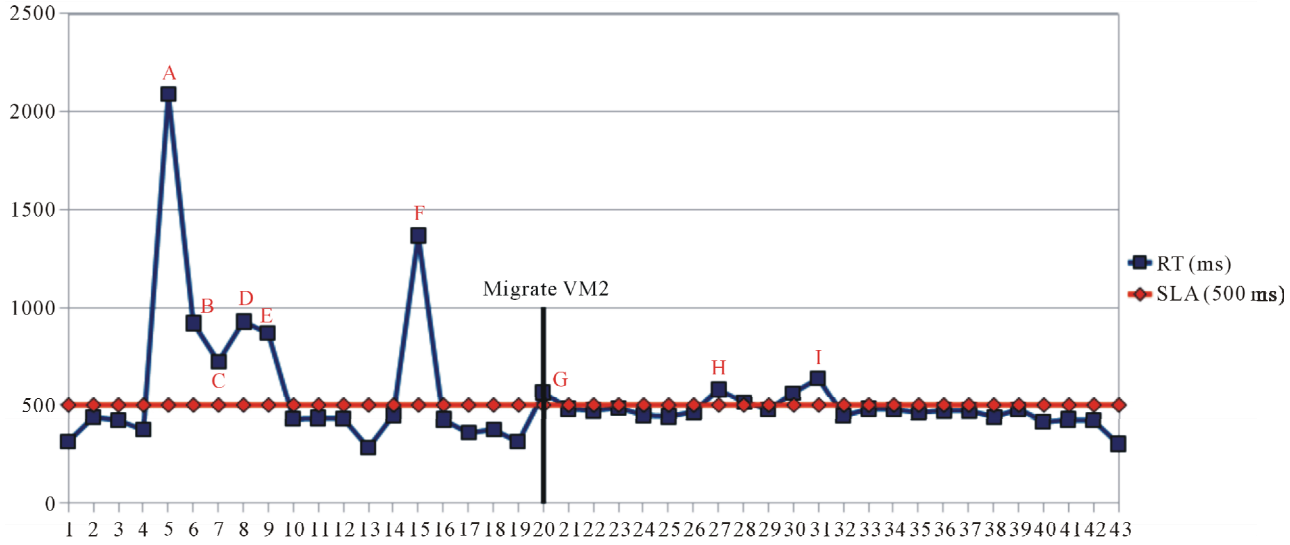

Figure 7. Apache response time with two levels of collaboration.

AM4:

On event: *HelpRequestEvent*
if (*server*::*memUtil* > 50){
Node = *findTheBestNode*();
SendMigrationNotifyMsg withNode:*Node;*
}

This policy says that upon receipt of a HelpRequest message by AM4, if the server asking for help has a memory utilization of more than 50% then find the best node (e.g. the least busy) and generate a migration event message to be sent to the needy AM (in this AM2).

AM2, in turn, has the following policy:

On event: *MigrationEvent*
if (*server*::*memUtil* > 50){
vm = *findBestVM*().
vmIP = *vm getIP*.
managedSystem:: *Migrate*:*vm To*:*Node*.
sendNewManagedObjectEvent To:*Node*.
destVMObj setIP: *vmIP*.
RemoveManagedObject:*vm*.
IncreaseMemLimit: *vm.memory*
}

When AM2 receives the notify message on migration, it chooses a VM to be migrated to the new server. In our implementation, we adopted a greedy approach in both finding the best physical node and finding the best VM for migration. We choose the least busy VM to be migrated. After this VM is migrated, then there will be more memory available for the busiest VMs. In this case, AM2 migrates VM2 (it had lower memory utilization) to Server2 and removes it from its own hierarchy.

This policy says that upon receipt of a MigrationEvent at AM2 (Node Controller Level), then find the best VM, store its IP in vmIP and migrate it to the node specified in the message. It also indicates that AM2 should tell the manager of that node to take care of this VM; it does this by sending a "NewManagedObjectEvent" message to the AM responsible for that node. After that, it sets the IP of the new managed object to vmIP and removes the old one from the list of its own children and increase the available free memory limit by the size of VM memory that has just been freed.

Figure 8 shows the hierarchy of AMs after this dynamic change in the VMs structure.

In this case, after migration, there is more memory available at the AM2 level and the memory limit is increased. Therefore, at point H (**Figure 7**) when the load is getting higher and another SLA violation happens, AM1 asks for help and AM2 responds by adding 50 more megabytes to VM1. The same process happens at point I where AM2 adds another 50MB to VM1 and after that the response time stays below the SLA threshold although the load is still very high.

The calculation of our measures for this scenario is as follows:

T_3 = 10.5 seconds
S_3 = 43 seconds
$V_3 = T_3/S_3 = 0.24 = 24\%$

In this case, even with the migration of one of the VMs, the percentage of time in a "violated" state is much less than in Scenario 2.

5.2. Discussion

Table 1 summarizes the time and percentage in a "violated" state for the three scenarios. Not surprisingly, having more AMs making changes to the system and components decreased the impact of violations. Most

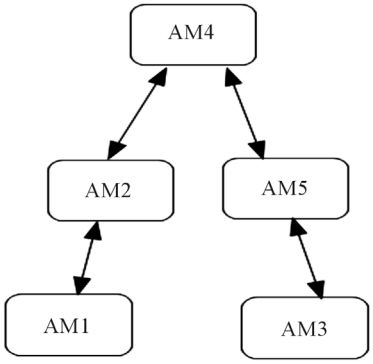

Figure 8. Managers hierarchy after migration of VM2 to Server 2.

Table 1. Time and SLA violation rate in three scenarios.

	T (seconds)	V (%)
Scenario 1	18	72
Scenario 2	10.5	42
Scenario 3	10.5	24

importantly, this happened automatically without administrator intervention and without adding any new hardware which means improvement in the current system efficiency.

The results show that there is definitely an advantage when AMs can collaborate. A single autonomic manager cannot solve all performance problems just by itself because it has only a local view of the system with some limited authority to change things. Thus, the current infrastructure can be used more efficiently and provide better services with less chance of violating SLAs without adding new computational resources.

6. Conclusions

In this paper we described some details towards the use of collaborating autonomic managers for the management of cloud environments. We showed how policies can be used at different levels of the hierarchy to facilitate the collaboration among autonomic managers. We also showed how the communication messages can be inferred automatically from policies and get generated on the fly.

In this work we assumed that policies are defined and delivered to managers by system administrators, but as a future work we are planning to make this process more automated.

We then implemented these ideas in a prototype and showed how this collaboration can be useful to preserve the response time of a web server under a certain threshold (defined in SLA).

Further work on this approach can lead to more automated management of cloud environments enabling more efficient use of the cloud infrastructure and as well as meeting SLA requirements while using fewer resources.

7. References

[1] M. C. Huebscher and J. A. McCann, "A Survey of Autonomic Computing—Degrees, Models, and Applications," *ACM Computing Surveys*, Vol. 40, No. 3, 2008, pp. 1-28. doi:10.1145/1380584.1380585

[2] J. O. Kephart, "Research Challenges of Autonomic Computing," *Proceedings of 27th International Conference on Software Engineering*, St. Louis, 15-21 May 2005, pp. 15-22.

[3] J. Kephart and D. M. Chess, "The Vision of Autonomic Computing," *Computer*, Vol. 36, No. 1, 2003, pp. 41-50. doi:10.1109/MC.2003.1160055

[4] J. Famaey, S. Latrea, J. Strassner and F. De Turck, "A Hierarchical Approach to Autonomic Network Management," *IEEE/IFIP Network Operations and Management Symposium Workshops*, Osaka, 19-23 April 2010, pp. 225-232. doi:10.1109/NOMSW.2010.5486571

[5] G. Tesauro, D. M. Chess, W. E. Walsh, R. Das, A. Segal, I. Whalley, J. O. Kephart and S. R. White, "A Multi-Agent Systems Approach to Autonomic Computing," *Proceedings of the 3rd International Joint Conference on Autonomous Agents and Multiagent Systems*, New York, 23 July 2004, pp. 464-471.

[6] W. Chainbi, H. Mezni and K. Ghedira, "An Autonomic Computing Architecture for Self-*Web Services," *Autonomic Computing and Communications Systems*, Vol. 23, 2010, pp. 252-267.

[7] M. Jarrett and R. Seviora, "Constructing an Autonomic Computing Infrastructure Using Cougaar," *Proceedings of the 3rd IEEE International Workshop on Engineering of Autonomic & Autonomous Systems*, Potsdam, 27-30 March 2006, pp. 119-128.

[8] M. Aldinucci, M. Danelutto and P. Kilpatrick, "Towards Hierarchical Management of Autonomic Components: A Case Study," *17th Euromicro International Conference on Parallel, Distributed and Network-Based Processing*, Weimar, 18-20 February 2009, pp. 3-10. doi:10.1109/PDP.2009.48

[9] T. Mukherjee, A. Banerjee, G. Varsamopoulos and S. K. S. Gupta, "Model-Driven Coordinated Management of Data Centers," *Computer Networks*, Vol. 54, No. 16, 2010, pp. 2869-2886. doi:10.1016/j.comnet.2010.08.011

[10] J. O. Kephart, H. Chan, R. Das, D. W. Levine, G. Tesauro and F. R. A. C. Lefurgy, "Coordinating Multiple Autonomic Managers to Achieve Specified Power-Performance Tradeoffs," *Proceeding of the 4th International Conference on Autonomic Computing*, Dublin, 13-16 June 2006, pp. 145-154.

[11] M. Steinder, I. Whalley, J. E. Hanson and J. O. Kephart, "Coordinated Management of Power Usage and Runtime Performance," *IEEE Network Operations and Management Symposium, Salvador*, 7-11 April 2008, pp. 387-394.

[12] M. Salehie and L. Tahvildari, "A Policy-Based Decision Making Approach for Orchestrating Autonomic Elements," 13*th IEEE International Workshop on Software Technology and Engineering Practice*, Budapest, 2005, pp. 173-181.

[13] R. Das, J. O. Kephart, C. Lefurgy, G. Tesauro, D. W. Levine and H. Chan, "Autonomic Multi-Agent Management of Power and Performance in Data Centers," *Proceedings of the* 7*th International Joint Conference on Autonomous Agents and Multiagent Systems*: *Industrial Track*, Estoril, 12-14 May 2008, pp. 107-114.

[14] A. Schaeffer-Filho, *et al*., "Towards Supporting Interactions between Self-Managed Cells," 1*st International Conference on Self-Adaptive and Self-Organizing Systems*, Cambridge, 9-11 July 2007, pp. 224-236.

[15] A. Schaeffer-Filho, E. Lupu and M. Sloman, "Realising Management and Composition of Self-Managed Cells in Pervasive Healthcare," *Proceedings of 3rd International Conference on Pervasive Computing Technologies for Healthcare*, London, 1-3 April 2009, pp. 1-8.

[16] D. Nurmi, R. Wolski, C. Grzegorczyk, G. Obertelli, S. Soman, L. Youseff and D. Zagorodnov, "The Eucalyptus Open-Source Cloud-Computing System," *Proceedings of* 9*th IEEE/ACM International Symposium on Cluster Computing and the Grid*, Shanghai, May 2009, pp. 124-131.

[17] S. Wardley, E. Goyer and N. Barcet, "Ubuntu Enterprise Cloud Architecture," Technical White Paper, UEC, August 2009.

[18] R. Boutaba and I. Aib, "Policy-Based Management: A Historical Perspective," *Journal of Network and Systems Management*, Vol. 15, No. 4, 2007, pp. 447-480. doi:10.1007/s10922-007-9083-8

[19] O. Mola and M. Bauer, "Collaborative Policy-Based Autonomic Management in a Hierarchical Model," accepted in 7*th International Conference on Network and System Management*, Paris, 24-28 October 2011.

[20] A. Kivity, Y. Kamay, D. Laor, U. Lublin and A. Liguori, "kvm: The Linux Virtual Machine Monitor," *Proceedings of the Linux Symposium*, Ottawa, 27-30 June 2007, pp. 225-230.

[21] K. Twidle, N. Dulay, E. Lupu and M. Sloman, "Ponder2: A Policy System for Autonomous Pervasive Environments," *Proceedings of* 5*th International Conference on Autonomic and Autonomous Systems*, Valencia, 20-25 April 2009, pp. 330-335.

[22] Virtuemart, 2011. http://virtuemart.net/

[23] JMeter Load Generator, 2011. http://jakarta.apache.org/jmeter/

