2012 38th Euromicro Conference on Software Engineering and Advanced Applications

Feedback Control Algorithms to Deploy and Scale Multiple Web Applications per Virtual Machine

Adnan Ashraf*†‡, Benjamin Byholm*, Joonas Lehtinen§, Ivan Porres*†
*Department of Information Technologies, Åbo Akademi University, Turku, Finland.
Email: adnan.ashraf@abo.fi, benjamin.byholm@abo.fi, ivan.porres@abo.fi
†Turku Centre for Computer Science (TUCS), Turku, Finland.
‡Department of Software Engineering, International Islamic University, Islamabad, Pakistan.
§Vaadin Ltd., Turku, Finland. Email: joonas@vaadin.com

***Abstract*—This paper presents feedback control algorithms to autonomously deploy and scale multiple web applications on a given Infrastructure as a Service cloud. The proposed algorithms provide automatic deployment and undeployment of applications and proportional-derivative scaling of the application server tier. The algorithms use utilization metrics as input and do not require a performance model of the application or the infrastructure dynamics. Moreover, our work supports deployment and scaling of multiple simultaneous applications per virtual machine (VM). This allows us to share VM resources among deployed applications, reducing the number of required VMs. The approach is demonstrated in a prototype implementation that has been deployed in the Amazon Elastic Compute Cloud.**

***Keywords*-Cloud computing; application server; scalability; quality of service; web applications**

I. Introduction

Web applications are often deployed in a 3-tier computer architecture. The client tier runs within the user web browser while the application and database tiers run in the remote server infrastructure. Both the application and the database tiers often use a computer cluster to be able to process many user requests simultaneously. In this configuration, a load balancing subsystem distributes the user requests among the computers in the cluster.

Traditionally, these clusters are composed of a fixed number of computers and are dimensioned to serve a predetermined maximum number of concurrent users. However, Infrastructure as a Service (IaaS) clouds, such as Amazon Elastic Compute Cloud (EC2) [1], currently offer computing resources such as network bandwidth, storage, and virtual machines (VMs) on demand. IaaS offerings can be used to create a dynamically scalable server tier consisting of a varying number of VMs. In this paper, we use IaaS offerings to create a dynamically scalable application server tier.

Determining the number of VMs to provision for a cluster is an important problem. The exact number of VMs needed at a specific time depends upon the user load and the Quality of Service (QoS) requirements that are specified in the Service Level Agreements (SLAs). Allocating too little resources will lead to subpar service, allocating too much resources will lead to increased operation costs. There are several research works that propose dynamic scaling solutions: [2], [3], [4], [5], [6], including some works that use control theoretic models [7], [8], [9]. There are also some vendor-specific commercial dynamic scaling solutions and management frameworks such as AWS Elastic Beanstalk [10].

Each one of these solutions has its own strengths and limitations. However, a common characteristic is that one VM hosts only one application. This is a reasonable approach if an application is large enough and has a user load high enough to keep at least one VM sufficiently utilized. But in many cases, provisioning an entire VM for one single application may introduce unnecessary overhead and cost due to under-utilization. Large analyst firms estimate that $15\% - 20\%$ server utilization is common in enterprises and it is decreasing even further every day, as more powerful servers enter the data centers [11].

The underutilization of VMs becomes even more pertinent when deploying a large number of web applications of varying resource needs, where most of the applications may have very few users at a given time, while a few applications may have many users. The solution to this problem is to design a dynamically scaling application server tier that manages multiple applications simultaneously. In this case, a VM can execute many applications if needed, in a similar way that a single HTTP server process is currently used to serve many static web sites.

In this paper, we present the ARVUE scaling algorithms, which allow dynamic scaling of the application server tier while simultaneously optimizing the resource allocation of multiple applications. The ARVUE algorithms are part of a larger project that aims at providing an open-source Platform as a Service (PaaS) integrated solution for web application development, deployment, and dynamic scaling [12]. In this context, the ARVUE scaling algorithms provide a dynamic scaling approach for PaaS clouds.

The rest of the paper is organized as follows. Section II outlines the main tasks and the most important characteristics of the ARVUE dynamic scaling approach. Section III presents the system architecture. Scaling algorithms and their related policies are described in Section IV. Section V presents our prototype implementation and Section VI provides experi-

978-0-7695-4790-9/12 $26.00 © 2012 IEEE
DOI 10.1109/SEAA.2012.13

mental results of the prototype implementation. Section VII discusses important related works and Section VIII concludes the paper.

II. ARVUE Dynamic Scaling Approach

The main task of the proposed algorithms is to provision and remove VMs for the application server tier and to deploy and remove applications from each VM, in order to maintain a desired QoS. For cost-effectiveness, the algorithms support deployment of multiple simultaneous applications on a single VM. At any given time, an application may be deployed in zero, one or more VMs. Popular applications will often be deployed in many VMs, while sporadically used applications will not be deployed at all – in order to save resources. Due to memory limitations, it is also not possible to assume that we can deploy all applications in one VM.

The most important characteristics of the ARVUE algorithms are that they are based on reactive feedback control, they do not depend upon a performance model of the application or the infrastructure dynamics, they deploy multiple applications per VM, and they provide server and application level scaling.

A. Lack of Knowledge about the Application, Infrastructure or User Dynamics

Our approach does not use any knowledge of the performance and dynamics of each application nor of the VMs used to execute them. We also do not use a prediction model of future user load [13], [14]. We acknowledge that this information can be used to develop better deployment and scaling algorithms. However, this information is often not available or may change often, as in the case of applications that are constantly under development. Thus, our approach is based on a feedback control loop, where the resource allocation is based on current and past user load measured in the system in terms of individual resource utilizations.

A common QoS requirement for web applications is to ensure a maximum response time for each user request. If the desired response time for an application is known, the scaling algorithms should work towards ensuring that the actual response time is lower than the maximum allowed response time. Using the response time as a QoS requirement has the advantage of being an intuitive performance measure that is relatively easy to monitor for both the server and the client. Again, we decided not to use any additional knowledge about the application and user expectations in our approach. The reason is that the expected response time may not be defined for a given application. Also, an application may have different expected response times for different request types.

B. Multiple Applications per VM

The proposed algorithms should provide a finer deployment granularity than the smallest VM provided by the current IaaS providers. Instead of adding or removing one full VM for a particular application, the algorithms should effectively support adding or removing a fraction of a VM for an application. This is especially important when running a large number of web applications, most of which may have very few users at a given time, while a few of them may have many users. In this case, if one or more full VMs are provisioned exclusively for an application, then a fraction of a VM will be over-provisioned for each application. As the number of applications increases, these over-provisioned fractions add up to a large number of over-provisioned full VMs in total. Thus, allocating one or more full VMs per application incurs unnecessary costs, due to a large number of underutilized VMs.

The ARVUE scaling algorithms share VM resources by deploying multiple simultaneous applications per VM, which effectively supports adding or removing a fraction of a VM per application. Consequently, fewer VMs can be used to run several web applications, and unnecessary costs can thus be avoided without compromising the desired QoS.

C. VM Provisioning Time

With the current IaaS offerings, provisioning of a VM takes a considerable amount of time, which is defined by the IaaS provider. Due to this inevitable VM provisioning delay, handling of a sudden peak load becomes even more challenging, because some existing servers may become heavily overloaded while provisioning the required number of VMs. Our solution to this problem uses a small number of additional VMs. If the additional VMs are provisioned on an application basis, then that might require a large number of VMs in total. Thus, ARVUE uses additional VMs at the PaaS level. It is important to note, that in order to achieve a sustainable QoS, such an additional VM capacity might be required even when a prediction model is used. This is due to the fact that prediction models sometimes lead to false predictions. Additional VM capacity would help in managing the peak loads in these cases.

Using additional VMs as reserve capacity poses an important question on the number of VMs that should be used for this purpose. We argue, that rather than using a fixed number of VMs, this number should be determined dynamically.

D. Server and Application Level Scaling

A distinguishing characteristic of the ARVUE scaling algorithms is that in addition to the support for creating a dynamically scalable application server tier, the algorithms also support scaling of individual web applications.

The server level scaling algorithms are based on the monitoring of the server CPU load average (or simply load average) and server memory utilization metrics. The load average represents the average CPU load over a period of time. It is computed as the exponentially weighted moving average of running (on CPU) and runnable (waiting for CPU) processes on all cores. Memory utilization is a real number $Mem\_U \in [0, 1]$, representing the amount of used memory in proportion to the amount of total memory, which includes physical memory and virtual memory.

The application level scaling is based on the monitoring of the application level CPU and memory utilization metrics. The

CPU utilization is the amount of time an application has spent executing on the CPU in a certain time window, while the memory utilization represents the amount of memory currently used by said application.

E. Relationship Between Utilization Metrics and QoS

QoS is often defined in terms of commonly used software performance metrics, such as response time and throughput [15]. However, as discussed in Section II-A, the expected response time, and similarly the throughput, may be difficult to define for an application. Therefore, we decided to define QoS in terms of server utilization metrics. The rationale of using utilization metrics is that a scalable server maintains its performance as long as its individual resource utilizations do not exceed their upper limits. However, when the resource utilization of the bottleneck server resource exceeds its upper limit, the server becomes saturated. A saturated server fails to maintain its performance, which translates into subpar service (higher response time and lower throughput) [16]. Thus, in order to ensure a sustainable QoS, the ARVUE scaling algorithms strive to maintain the individual resource utilizations below their respective upper thresholds.

III. System Architecture

The ARVUE dynamic scaling algorithms distribute end user sessions for web applications to a scalable application server tier, which is dynamically scaled up and down according to end user workload. The scaling algorithms operate within the ARVUE backend, which consists of the following components, as shown in Figure 1: *application server*, *local controller*, *application repository*, *global controller*, *cloud provisioner*, and *HTTP load balancer*.

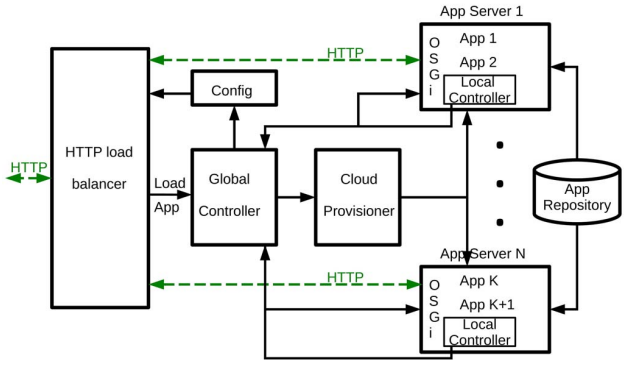

Fig. 1. The ARVUE dynamic scaling architecture

An *application server* instance runs on a dynamically provisioned VM. Each *application server* runs multiple web applications in an OSGi [17] environment. In such an environment, each application runs as an OSGi component called a bundle. OSGi also introduces the dynamic component model, which allows dynamic loading and unloading of bundles. In addition to the web applications, each application server also runs a *local controller*. The *local controller* monitors and logs server and application level utilizations. On the discrete sampling period k, each *local controller* sends utilization data to the *global controller*. Another responsibility of the *local controller* is to control the OSGi environment for loading and unloading of web applications. Web applications are stored in an *application repository*, from where they are loaded onto application servers. For applications under the OSGi environment, we use the Apache Felix OSGi Bundle Repository (OBR) [18]. The *global controller* acts as a capacity manager. It implements scaling algorithms and their related policies, as described in Section IV.

The *cloud provisioner* in the ARVUE architecture refers to the cloud provisioner in an IaaS cloud, such as the provisioner in Amazon EC2. While the scaling decisions are made by the *global controller*, the actual lower level tasks of starting and terminating VMs are done by the *cloud provisioner*. The HTTP requests are routed through a high performance *HTTP load balancer* and proxy. For this, we use HAProxy [19], which balances the load of requests for new user sessions among the application servers. For its functions, HAProxy maintains a configuration file which contains information about application servers and application deployments on each server. As a result of application server and application level scaling operations, the configuration file is frequently updated with new information.

IV. Scaling Algorithms and Policies

The *global controller* implements the scaling algorithms and their related policies. There are a total of four algorithms: scaling up the application server tier, scaling down the application server tier, scaling up web applications, and scaling down web applications. For the sake of clarity, the concepts used in the algorithms and their notation are summarized in Table I. The algorithms implement reactive feedback control. That is, the scaling decisions are performed inside a feedback control loop whose inputs are observed resource utilizations.

The scaling decisions are based on the states of applications and application servers. The server states are determined by comparing observed load average and memory utilization with their upper and lower threshold values, while the application states are based on the application level CPU and memory utilization metrics, and their respective threshold values. These threshold values depend upon a tradeoff between QoS and cost. The server states are: *saturated*, *non-saturated*, *underutilized*, and *long-term underutilized*. Similarly, the application states are: *saturated*, *non-saturated*, *inactive*, and *long-term inactive*.

The scaling algorithms are designed to prevent oscillations in the number of provisioned VMs [3]. This is desirable due to the inevitable VM provisioning time, as discussed in Section II-C, which may lead to deteriorated performance. Moreover, since some IaaS providers, such as Amazon EC2, charge on an hourly basis, oscillations will result in a higher provisioning cost. Therefore, the algorithms counteract oscillations by delaying new scaling operations until previous scaling operations have been realized. Furthermore, the scaling down

TABLE I
SUMMARY OF CONCEPTS AND THEIR NOTATION

$A(k)$	set of web applications at k
$A_i(k)$	set of inactive applications at k
$A_{li}(k)$	set of long-term inactive applications at k
$A_s(k)$	set of saturated applications at k
$NS_s(k)$	set of non-saturated servers at k
$S(k)$	set of servers at discrete time k
$S_{lu}(k)$	set of long-term underutilized servers at k
$S_n(k)$	set of new servers at k
$S_s(k)$	set of saturated servers at k
$S_t(k)$	set of servers selected for termination at k
$S_u(k)$	set of underutilized servers at k
$CPU\_U(a,k)$	CPU utilization of application a at k
$Load\_A(s,k)$	load average of server s at k
$Mem\_U(a,k)$	memory utilization of application a at k
$Mem\_U(s,k)$	memory utilization of server s at k
$N_A(k)$	number of additional servers at k
$N_P(k)$	number of servers to provision at k
$N_T(k)$	number of servers to terminate at k
CPU_U_{LTa}	application CPU utilization lower threshold
CPU_U_{UTa}	application CPU utilization upper threshold
IC_{Ta}	inactivity count threshold for an application
$Load\_A_{LT}$	server load average lower threshold
$Load\_A_{UT}$	server load average upper threshold
Mem_U_{LTa}	application memory utilization lower bound
Mem_U_{LT}	server memory utilization lower threshold
Mem_U_{UT}	server memory utilization upper threshold
Mem_U_{UTa}	application memory utilization upper bound
UC_T	underutilization count threshold for a server
$delay()$	delay function
$dep\_apps(s,k)$	applications deployed on server s at k
$deploy(a)$	deploy application a as per allocation policy
$deploy(a,S)$	deploy application a on a set of servers S
$inactiv\_c(a)$	inactivity count of application a
$migrate(A)$	migrate user sessions for applications A
$migrate(S)$	migrate user sessions for servers S
$provision(n)$	provision n servers
$select(n)$	select n servers for termination
$sort(S)$	sort servers S on server utilization metrics
$terminate(S)$	terminate servers S
$under\_u\_c(s)$	underutilization count of server s
$unload(a)$	unload application a
N_B	number of servers to use as base capacity
$shutDown$	server shutdown time
$startUp$	server startup time

algorithm terminates only those VMs that have been constantly underutilized for a certain amount of time.

The algorithms maintain a fixed minimum number of servers representing the base capacity (N_B). Moreover, as explained in Section II-C, they also use a small number of additional servers. The number of additional servers ($N_A(k)$) is determined as follows:

$$N_A(k) = \begin{cases} \lceil |S(k)| \cdot A_A \rceil, & \text{if } |S(k)| - |S_s(k)| = 0 \\ \left\lceil \frac{|S(k)|}{|S(k)| - |S_s(k)|} \cdot A_A \right\rceil, & \text{otherwise} \end{cases}, \quad (1)$$

where the aggressiveness parameter for additional VM capacity $A_A \in [0,1]$ restricts the maximum size of the additional capacity. It accounts for the VM provisioning time and the sampling period (k). For example, $A_A = 0.2$ restricts the maximum size of the additional VM capacity to 20% of the total number of VMs ($|S(k)|$).

Instead of provisioning or terminating one VM at a time, the application server scaling algorithms implement proportional control. The number of VMs to provision in the proportional control is based on the number of saturated servers ($|S_s(k)|$) and the aggressiveness of control for VM provisioning (A_P). Similarly, the number of VMs to terminate is based on the number of long-term underutilized servers ($|S_{lu}(k)|$) and the aggressiveness of control for VM termination (A_T). The proportional scaling up enables handling of sudden peak loads. Since, for sudden peak loads, provisioning one VM at a time might be too slow, due to VM provisioning time. This might also lead to continuous violations of the desired QoS. The proportional scaling down saves VM provisioning cost, because terminating one VM at a time might result in unwanted longer provisioning periods.

One known shortcoming of proportional control is that it does not consider how fast the servers become saturated or underutilized. The standard solution is to augment proportional control with derivative control. Thus, we use proportional-derivative (PD) feedback control [20]. However, instead of using a standard fixed-gain PD controller, we design a PD controller that does not depend on a performance model of the application or the infrastructure dynamics (Section II-A), and supports server and application level scaling (Section II-D), while deploying multiple applications per VM (Section II-B). With our PD control design, we determine the number of servers to provision ($N_P(k)$) in scaling up as:

$$N_P(k) = \lceil w \cdot P_P(k) + (1-w) \cdot D_P(k) \rceil, \quad (2)$$

where $w \in [0,1]$ is a real number called the weighting coefficient. It determines how much weight is given to the proportional factor ($P_P(k)$) respective to the derivative factor ($D_P(k)$). For example, $w = 0.5$ means equal weight to $P_P(k)$ and $D_P(k)$. The proportional factor for VM provisioning ($P_P(k)$) is calculated as:

$$P_P(k) = |S_s(k)| \cdot A_P, \quad (3)$$

where $A_P \in [0,1]$ is a real number, which represents the aggressiveness of control for VM provisioning. For example, $A_P = 1$ suggests to provision as many new as saturated. The derivative factor for VM provisioning ($D_P(k)$) is computed as:

$$D_P(k) = |S_s(k)| - |S_s(k-1)|. \quad (4)$$

The number of servers to terminate ($N_T(k)$) in scaling down is computed in a similar way:

$$N_T(k) = \lceil w \cdot P_T(k) + (1-w) \cdot D_T(k) \rceil - N_B - N_A(k), \quad (5)$$

where the proportional factor for termination ($P_T(k)$) is calculated as:

$$P_T(k) = |S_{lu}(k)| \cdot A_T, \quad (6)$$

where $A_T \in [0,1]$ is similar to A_P. It represents the aggressiveness of control for VM termination. The derivative factor for termination ($D_T(k)$) is computed as:

$$D_T(k) = |S_{lu}(k)| - |S_{lu}(k-1)|. \quad (7)$$

A. Scaling Up The Application Server Tier

The scaling up algorithm for the application server tier is given as Algorithm 1. Based on the observed load average and memory utilization of individual servers, the algorithm first determines their states. It then makes a set of saturated servers and a set of non-saturated servers. From the set of saturated servers, it makes a set of saturated applications. If it finds at least one saturated and at least one non-saturated server, it deploys each saturated application on another existing server according to the application-to-server allocation policy, as described in Section IV-F. However, if it finds that all servers ($S(k)$) except the additional VM capacity ($N_A(k)$) are saturated, the algorithm scales up the application server tier by provisioning one or more servers according to the N_P and then deploys each saturated application on each new server.

Algorithm 1 Scaling up the application server tier

while true do
  $S_s(k) := \{\forall s \epsilon S(k) | Load\_A(s,k) > Load\_A_{UT}\} \cup \{\forall s \epsilon S(k) | Mem\_U(s,k) > Mem\_U_{UT}\}$
  $A_s(k) := \bigcup_{s \epsilon S_s(k)} dep\_apps(s,k)$
  $NS_s(k) := S(k) \setminus S_s(k)$
  if $|S_s(k)| \geq 1 \wedge |NS_s(k)| \geq 1$ **then**
    for $a \epsilon A_s(k)$ **do**
      $deploy(a)$
    end for
  end if
  if $|S_s(k)| \geq (|S(k)| - N_A(k)) \wedge N_P(k) \geq 1$ **then**
    $S_n(k) := provision(N_P(k))$
    $S(k) := S(k) \cup S_n(k)$
    $delay(startUp)$
    for $a \epsilon A_s(k)$ **do**
      $deploy(a, S_n(k))$
    end for
  end if
end while

B. Scaling Down The Application Server Tier

The algorithm for scaling down the application server tier is presented in Algorithm 2. It first determines the state of each individual server based on the observed resource utilizations. It then makes a set of underutilized servers, from which a set of long-term underutilized servers is obtained. Since the aim of the algorithm is to minimize VM provisioning cost, which is a function of number of VMs and time, the algorithm terminates any redundant VMs as soon as possible. Therefore, when the algorithm finds at least one long-term underutilized server, while excluding the base capacity (N_B) and the additional capacity ($N_A(k)$), it sorts them in an increasing order based on their resource utilizations and selects one or more servers according to the $N_T(k)$. The rationale of the sorting is to ensure that the least loaded servers are selected for termination.

Since the servers selected for termination might still be running a small number of active user sessions on them, the next step is to ensure that the termination of the selected servers does not abandon any active sessions that were currently running on said servers. This is achieved by migrating all active sessions from the selected servers to other existing servers. Finally, the selected servers are terminated and removed from the application server tier.

Algorithm 2 Scaling down the application server tier

while true do
  $S_u(k) := \{\forall s \epsilon S(k) | Load\_A(s,k) < Load\_A_{LT}\} \cap \{\forall s \epsilon S(k) | Mem\_U(s,k) < Mem\_U_{LT}\}$
  $S_{lu}(k) := \{\forall s \epsilon S_u(k) | under\_u\_c(s) \geq UC_T\}$
  if $(|S_{lu}(k)| - N_B - N_A(k)) \geq 1 \wedge N_T(k) \geq 1$ **then**
    $sort(S_{lu}(k))$
    $S_t(k) := select(N_T(k))$
    $migrate(S_t(k))$
    $S(k) := S(k) \setminus S_t(k)$
    $terminate(S_t(k))$
    $delay(shutDown)$
  end if
end while

C. Scaling Up Web Applications

Scaling up individual web applications is less complex than scaling up the application server tier. Algorithm 3 uses resource utilizations of the individual web application instances to determine their states. It then makes a set of saturated applications and finally deploys all saturated applications on one or more servers, based on the application-to-server allocation policy, as described in Section IV-F.

Algorithm 3 Scaling up individual web applications

while true do
  $A_s(k) := \{\forall s \epsilon S(k) \forall a \epsilon dep\_apps(s,k) | CPU\_U(a,k) > CPU\_U_{UTa}/|dep\_apps(s,k)|\} \cup \{\forall s \epsilon S(k) \forall a \epsilon dep\_apps(s,k) | Mem\_U(a,k) > Mem\_U_{UTa}\}$
  if $|A_s(k)| \geq 1$ **then**
    for $a \epsilon A_s(k)$ **do**
      $deploy(a)$
    end for
  end if
end while

D. Scaling Down Web Applications

Individual web applications are scaled down in a similar fashion to application servers. Algorithm 4 uses application level resource utilizations of individual web application instances to determine their states. Based on the application inactivity count, the algorithm then makes a set of long-term inactive applications from the set of inactive applications. A *long-term inactive* application should ideally have no active sessions. However, based on the utilization lower thresholds, it might have a small number of active sessions. Therefore, to ensure an uninterrupted service to any such sessions, the

algorithm migrates all sessions belonging to a long-term inactive application to another existing server, preferably to a server on which said application is already deployed. Finally, to minimize memory utilization of the servers, the algorithm removes all long-term inactive applications from their servers.

Algorithm 4 Scaling down individual web applications

1: **while true do**
2: $A_i(k) := \{\forall s \epsilon S(k) \forall a \epsilon dep\_apps(s,k) | CPU\_U(a,k) < CPU\_U_{LTa}\} \cap \{\forall s \epsilon S(k) \forall a \epsilon dep\_apps(s,k) | Mem\_U(a,k) < Mem\_U_{LTa}\}$
3: $A_{li}(k) := \{\forall a \epsilon A_i(k) | inactiv\_c(a) \geq IC_{Ta}\}$
4: **if** $|A_{li}(k)| \geq 1$ **then**
5: $migrate(A_{li}(k))$
6: $A(k) := A(k) \setminus A_{li}(k)$
7: **for** a in $A_{li}(k)$ **do**
8: $unload(a)$
9: **end for**
10: **end if**
11: **end while**

E. Server Selection for New Session Requests

Whenever an HTTP request for a new user session arrives at the *HTTP load balancer*, an application server is chosen to handle the new session request. There are two possible scenarios, based on the deployment of the required application.

Server selection in the first scenario, when the required application is already deployed, is handled by the *HTTP load balancer*. If the load balancer finds that the application is deployed on more than one server, it distributes the new session requests based on the current session-to-server allocation policy. The typical policies are: *(weighted) round-robin*, *lowest number of sessions*, and *lowest load average*.

In the second scenario when the requested application is not deployed anywhere, server selection for a new session request is handled by the *global controller*. It first selects one or more servers based on the current application-to-server allocation policy. The requested application is then deployed on the selected server(s), and the new session request is sent to the application based on the session-to-server allocation policy.

F. Server Selection for Loading Saturated Applications

As explained in Section IV-A and Section IV-C, a saturated application is deployed on one or more other (less busy) servers. Server selection is performed by the *global controller*, which selects one or more servers according to the current application-to-server allocation policy. The *global controller* implements a number of application-to-server allocation policies, such as *lowest load average*, *lowest number of applications*, *lowest number of sessions*, and *selected servers*.

V. Prototype

We have developed a discrete-event simulation and a prototype implementation of the ARVUE dynamic scaling algorithms. Discrete-event simulations have been recently used to simulate cloud computing environments [21]. The algorithms were first validated using the discrete-event simulation. Once validated, we implemented them in an actual cloud.

The prototype is implemented in Java and has been deployed on the Amazon EC2 cloud. It consists of two key components: GlobalController and LocalController. The prototype is not dependent on any web development framework, but has initially been tested for web applications developed using the Vaadin web development framework [22], an open-source framework for developing Java Servlet-based applications.

When running multiple web applications on a Java application server, all web applications are usually placed in the same Java Virtual Machine (JVM). However, Java lacks some important features needed to safely run multiple third-party web applications in one JVM. This is partly addressed by a widely adapted OSGi specification [17], which is extended for use in the security layer of ARVUE [12]. The Vaadin framework is compatible with OSGi and can be configured as an OSGi bundle. There are several open-source as well as commercial implementations of the OSGi specification. Among the open-source implementations, the free open-source Apache Felix [23] is certified to be compliant with the OSGi specification. Thus, we decided to use Felix for our prototype implementation.

VI. Experimental Results

Now we describe an experiment that we have conducted with our prototype implementation.

A. Experimental Design and Setup

The objective of the experiment was to demonstrate the most important characteristics of the ARVUE dynamic scaling approach, while emphasizing its cost-effectiveness. The ARVUE PaaS [12] is currently under development and therefore real data is not yet available. Hence, we decided to generate and use synthetic load in the experiment, which was designed to generate a load representing 1000 simultaneous user sessions. The sessions were ramped up from 0 to 1000 at a discrete rate of 25 new sessions per 10 seconds. After the ramp up phase, the number of sessions was maintained constant and then reduced at a rate of -25 sessions per 10 seconds. Each session was randomly assigned to one particular web application. The experiment used 10 web applications of varying resource needs. Application 1 was the most popular, having 50% of total user sessions, application 2 had 25% sessions, application 3 had 20% sessions, while the other 7 applications shared the remaining 5% sessions. Session duration was 15 minutes. Each session continuously generated HTTP requests on its web application. Each request was designed to cause the application server to do some work requiring up to 10 milliseconds of a dedicated CPU. User think time between consecutive requests varied uniformly between 0 seconds and 20 seconds. All random values were uniformly distributed over their range. The ARUVE sampling period (k) was 10 seconds. The upper thresholds for server load average ($Load\_A_{UT}$) and memory utilization (Mem_U_{UT}) were set to 0.8, which are considered

reasonable for efficient use of the servers [24]. Similarly, the lower thresholds ($Load\_A_{LT}$ and Mem_U_{LT}) were set to 0.2.

The experiment used small (*m1.small*) instances from the Amazon EC2 cloud. One instance was used for running the *HTTP load balancer* and the *global controller*. A varying number of instances were used to make a dynamically scalable application server tier. The experiment also used a small number of additional VMs, which was calculated dynamically. Moreover, five (5) small instances were used for generating user load.

The application-to-server allocation policy was set to *lowest load average*. The session-to-server allocation policy was also based on the lowest load average, achieved by using load average to calculate weights used by HAProxy's *weighted round-robin* policy.

B. Results and Analysis

Figure 2 presents the experimental results. The results show proportional-derivative scaling of the application server tier with a sustainable QoS. The response times (in milliseconds) were reasonable, while the server load average and memory utilization were always less than 1.0. The sharing of resources among multiple applications resulted in a reduced total number of VMs. The total VM cost for application deployment and scaling was $|VMs| \cdot cost_{VM} \cdot hours$, which amounts to $(6 + 1) \cdot \$0.080 / \text{ h} \cdot 1 \text{ h} = \$0.56$. Hosting a dedicated application per VM would have required a minimum of $(10 + 1) \cdot \$0.080 / \text{ h} \cdot 1 \text{ h} = \$0.88$. Thus, under these circumstances, ARVUE operated at 36% less cost. The results demonstrate that dynamic scaling of VMs based only on the resource utilization metrics can be as good as an approach that depends on the knowledge about the performance and dynamics of the applications and VMs. The results also indicate that using additional VM capacity is a suitable solution for avoiding heavy overloading of the application servers due to the VM provisioning time. The load average plot shows that the application server tier was able to conveniently absorb the temporary overload due to the VM provisioning time without saturating the servers. Finally, the server and application level scaling ensured that each server ran a reasonable number of applications, while each application was deployed on as many or as few servers as it needed at a certain time, based on the user load on the application.

It is important to note here that the algorithms did not produce oscillations in the number of VMs. As explained in Section IV, ARVUE algorithms are designed to counteract such oscillations.

VII. Related Work

The existing works on creating dynamically scalable server tiers can be classified into two main categories: Plan-based approaches and control theoretic approaches. Plan-based approaches can be further classified into workload prediction approaches and performance dynamics model approaches. One example of the workload prediction approaches is Ardagna et al. [2], while TwoSpot [3], Hu et al. [4], Chieu et al. [5], and

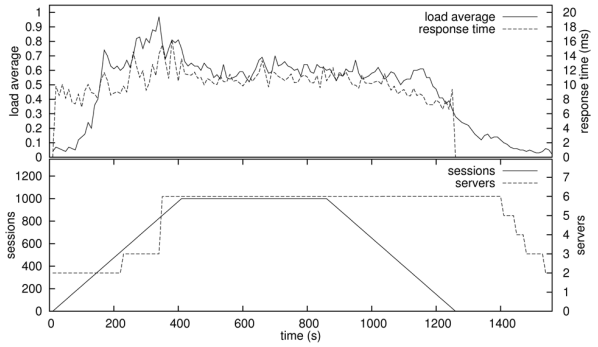

Fig. 2. Experimental results

Iqbal et al. [6] use a performance dynamics model. Similarly, Dutreilh et al. [7], Pan et al. [8], and Patikirikorala et al. [9] are control theoretic approaches. One common difference, between all existing works discussed here and ARVUE scaling approach, is that ARVUE uses one VM for deploying multiple simultaneous applications. Another distinguishing characteristic is that in addition to scaling of application servers, ARVUE algorithms also provide scaling of individual applications.

Ardagna et al. [2] proposed a distributed algorithm for managing Software as a Service (SaaS) cloud systems that addresses capacity allocation for multiple heterogenous applications. The resource allocation algorithm takes into consideration a predicted future load for each application class and a predicted future performance of each VM, while determining possible SLA violations for each application type. The main challenge in the prediction-based approaches is in making good prediction models. In contrast, ARVUE scaling algorithms do not depend upon prediction models.

TwoSpot [3] aims to combine existing open source technologies to support web applications written in different programming languages. It supports hosting of multiple web applications, which are automatically scaled up and down in a horizontal fashion. However, the scaling down is decentralized, which may lead to severe random drops in performance. For example, when all controllers independently choose to scale down at the same time. In contrast, ARVUE algorithms provide centralized scaling.

Hu et al. [4] proposed a heuristic algorithm that determines the server allocation strategy and job scheduling discipline which results in the minimum number of servers. They also presented an algorithm for determining the minimum number of required servers, based on the expected arrival rate, service rate, and SLA. In contrast, ARVUE algorithms use server level and application level resource utilization metrics. Moreover, the ARVUE algorithms support several application-to-server and session-to-server allocation policies.

Chieu et al. [5] presented an approach that scales servers for a particular web application based on the number of active user sessions. The main problem with this approach is in determining suitable threshold values on the number of user sessions.

Iqbal et al. [6] proposed an approach for adaptive resource provisioning for read intensive multi-tier web applications. Based on response time and CPU utilization metrics, the approach determines the bottleneck tier and then scales it up by provisioning a new VM. Scaling down is supported by checking for any over-provisioned resources from time to time. In contrast, ARVUE scaling algorithms support proportional-derivative scaling of the application servers.

Dutreilh et al. [7] and Pan et al. [8] used control theoretic models for designing resource allocation solutions for cloud computing. Dutreilh et al. presented a comparison of static threshold-based and reinforcement learning techniques. Pan et al. used proportional integral (PI) controllers to provide QoS guarantees. Patikirikorala et al. [9] proposed a multi-model framework for implementing self-managing control systems for QoS management. The work is based on a control theoretic approach called the Multi-Model Switching and Tuning (MMST) adaptive control. In contrast to the control theoretic approaches, the proportional-derivative scaling in the ARVUE algorithms does not depend upon performance models or infrastructure dynamics.

VIII. Conclusions

In this paper, we proposed feedback control algorithms for dynamic scaling of the application server tier in an IaaS cloud. The algorithms also provide automatic deployment and dynamic scaling of multiple web applications. We also presented a prototype implementation of the scaling algorithms and experimental results using the Amazon EC2 cloud. The results indicate that based on the user load on web applications, the ARVUE algorithms maintain an optimal number of VMs for the application server tier so that the over and under provisioning of VMs could be minimized. The algorithms scale multiple applications of varying resource needs, where most applications might not require a full VM at a given time, while a few applications may require one or more VMs. The cost-effectiveness in the proposed algorithms is based on the idea of sharing VM resources by deploying multiple applications per VM and thereby reducing the number of required VMs. For ensuring performance under dynamically and unpredictably varying user load, ARVUE algorithms provide proportional-derivative scaling of the application server tier.

Acknowledgements

This work was supported by the Cloud Software Finland research project and by an Amazon Web Services research grant. Adnan Ashraf was partially supported by a doctoral scholarship from the Higher Education Commission (HEC) of Pakistan. The authors want to thank Marc Englund at Vaadin Ltd. and Niclas Snellman and Thomas Fors at Åbo Akademi for their contributions to the development of ARVUE.

References

[1] "Amazon Elastic Compute Cloud." [Online]. Available: http://aws.amazon.com/ec2/

[2] D. Ardagna, C. Ghezzi, B. Panicucci, and M. Trubian, "Service provisioning on the cloud: Distributed algorithms for joint capacity allocation and admission control," in *Towards a Service-Based Internet*, ser. Lecture Notes in Computer Science, E. Di Nitto and R. Yahyapour, Eds. Springer Berlin / Heidelberg, 2010, vol. 6481, pp. 1–12.

[3] A. Wolke and G. Meixner, "TwoSpot: A cloud platform for scaling out web applications dynamically," in *Towards a Service-Based Internet*, ser. Lecture Notes in Computer Science, E. Di Nitto and R. Yahyapour, Eds. Springer Berlin / Heidelberg, 2010, vol. 6481, pp. 13–24.

[4] Y. Hu, J. Wong, G. Iszlai, and M. Litoiu, "Resource provisioning for cloud computing," in *Proceedings of the 2009 Conference of the Center for Advanced Studies on Collaborative Research*, ser. CASCON '09. New York, NY, USA: ACM, 2009, pp. 101–111.

[5] T. Chieu, A. Mohindra, A. Karve, and A. Segal, "Dynamic scaling of web applications in a virtualized cloud computing environment," in *e-Business Engineering, 2009. ICEBE '09. IEEE International Conference on*, oct. 2009, pp. 281 –286.

[6] W. Iqbal, M. N. Dailey, D. Carrera, and P. Janecek, "Adaptive resource provisioning for read intensive multi-tier applications in the cloud," *Future Generation Computer Systems*, vol. 27, no. 6, pp. 871 – 879, 2011.

[7] X. Dutreilh, N. Rivierre, A. Moreau, J. Malenfant, and I. Truck, "From data center resource allocation to control theory and back," in *Cloud Computing (CLOUD), 2010 IEEE 3rd International Conference on*, july 2010, pp. 410 –417.

[8] W. Pan, D. Mu, H. Wu, and L. Yao, "Feedback control-based QoS guarantees in web application servers," in *High Performance Computing and Communications, 2008. HPCC '08. 10th IEEE International Conference on*, sept. 2008, pp. 328 –334.

[9] T. Patikirikorala, A. Colman, J. Han, and L. Wang, "A multi-model framework to implement self-managing control systems for QoS management," in *Proceedings of the 6th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*, ser. SEAMS '11. New York, NY, USA: ACM, 2011, pp. 218–227.

[10] "AWS Elastic Beanstalk." [Online]. Available: http://aws.amazon.com/elasticbeanstalk/

[11] W. Vogels, "Beyond server consolidation," *ACM Queue*, vol. 6, no. 1, pp. 20–26, Jan. 2008.

[12] T. Aho, A. Ashraf, M. Englund, J. Katajamäki, J. Koskinen, J. Lautamäki, A. Nieminen, I. Porres, and I. Turunen, "Designing IDE as a service," *Communications of Cloud Software*, vol. 1, no. 1, December 2011.

[13] M. Andreolini and S. Casolari, "Load prediction models in web-based systems," in *Proceedings of the 1st international conference on Performance evaluation methodolgies and tools*, ser. valuetools '06. New York, NY, USA: ACM, 2006.

[14] P. Saripalli, G. Kiran, R. Shankar, H. Narware, and N. Bindal, "Load prediction and hot spot detection models for autonomic cloud computing," in *Utility and Cloud Computing (UCC), 2011 Fourth IEEE International Conference on*, dec. 2011, pp. 397 –402.

[15] H. Liu, *Software Performance and Scalability: A Quantitative Approach*, ser. Quantitative Software Engineering Series. John Wiley & Sons, 2011.

[16] J. Guitart, V. Beltran, D. Carrera, J. Torres, and E. Ayguade, "Characterizing secure dynamic web applications scalability," in *Parallel and Distributed Processing Symposium, 2005. Proceedings. 19th IEEE International*, april 2005, p. 108a.

[17] The OSGi Alliance. OSGi service platform: Core specification, 2009, release 4, version 4.2.

[18] Apache Felix OSGi Bundle Repository (OBR). [Online]. Available: http://felix.apache.org/site/apache-felix-osgi-bundle-repository.html

[19] HAProxy. [Online]. Available: http://haproxy.1wt.eu/

[20] J. Hellerstein, *Feedback control of computing systems*, ser. Wiley interscience publication. Wiley-Interscience, 2004.

[21] R. N. Calheiros, R. Ranjan, A. Beloglazov, C. A. F. De Rose, and R. Buyya, "CloudSim: a toolkit for modeling and simulation of cloud computing environments and evaluation of resource provisioning algorithms," *Software: Practice and Experience*, vol. 41, no. 1, pp. 23–50, 2011.

[22] M. Grönroos, *Book of Vaadin*, 4th ed. Vaadin Ltd., Turku, Finland, 2011. [Online]. Available: http://vaadin.com/book

[23] OSGi Felix. [Online]. Available: http://felix.apache.org/site/index.html

[24] J. Allspaw, *The art of capacity planning*, ser. O'Reilly Series. O'Reilly, 2008.