A Session-Based Adaptive Admission Control Approach for Virtualized Application Servers

Adnan Ashraf*†‡, Benjamin Byholm*, 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.

***Abstract*—This paper presents a session-based adaptive admission control approach for virtualized application servers called ACVAS (adaptive Admission Control for Virtualized Application Servers). ACVAS uses measured and predicted resource utilizations of a server to make admission control decisions for new user sessions. Instead of using the traditional on-off control, it implements per session admission control, which reduces the risk of over-admission. Moreover, instead of relying only on rejection of new sessions, ACVAS takes benefit of the cloud elasticity, which allows dynamic provisioning of cloud resources. It also implements a simple session deferment mechanism that reduces the number of rejected sessions while increasing session throughput. Thus, each admission control decision has three possible outcomes: admit, defer, or reject. Performance under varying user load is guaranteed by automatic adjustment and tuning of the admission control mechanism. The proposed approach is demonstrated in a discrete-event simulation.**

***Keywords*-Cloud computing; virtualized application server; adaptive admission control; session-based admission control;**

I. Introduction

Admission control refers to the mechanism of restricting the incoming user load on a server in order to avoid overloading it. Server overload prevention is important because an overloaded server fails to maintain its performance, which translates into subpar service (higher response time and lower throughput) [1]. Thus, if an overloaded server keeps on accepting new users, then not only the new users, but also the existing users might experience deteriorated performance.

Admission control is often implemented at the server level. A server with admission control in place would stop accepting new user requests or sessions when the server approaches its capacity limits. Therefore, overload prevention relies on rejection of new requests or sessions. Most traditional admission control approaches use request-based admission control, but there are recent approaches based on session-based admission control (SBAC), such as [2], which often yield better results for stateful web applications.

Web applications are often deployed in a three-tier computer architecture, where the application and the datastore tiers are implemented using a computer cluster in order to process many user requests simultaneously. 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) [3], can be used to create a dynamically scalable server tier consisting of a varying number of Virtual Machines (VMs) that could serve a theoretically unlimited number of users. However, in order to prevent virtualized servers from becoming overloaded, an admission control mechanism would still be required.

One of the key advantages of cloud computing is the elasticity of resources, wherein VMs can be dynamically provisioned and released. The admission control mechanism for a scalable server tier can benefit from cloud elasticity. Thus, with dynamic provisioning of resources, it may be possible to reduce the number of rejected sessions, which would help in increasing the session throughput. However, with contemporary IaaS providers, VM provisioning times are non-trivial [4] and thus the admission control mechanism might need to be augmented with a session deferment mechanism that could allow to temporarily defer a session until a new VM is provisioned or an existing VM becomes less loaded. Therefore, with a session deferment mechanism in place, an SBAC approach might be able to defer some sessions, which would otherwise be rejected. The outcome of each admission control decision would therefore be to admit, defer, or reject the new sessions.

One way to implement SBAC is to use the traditional *on-off* control, as used in [2], where admission control decisions are made for an entire admission control interval. However, since the admission control is applied only at the interval boundaries, a server may accept too many sessions during an interval and consequently become overloaded. The solution to this shortcoming of the *on-off* control is to apply admission control for each new session, as implemented in [5]. With per session admission control, the SBAC mechanism makes an admission control decision for each new session.

One of the main challenges in SBAC is to maintain the required quality of service (QoS) under highly varying user loads. This is often achieved by dynamically adapting the admission control mechanism, as in [2] and [6]. Thus, it might be possible to guarantee the required performance by doing automatic adjustment and tuning of the important parameters.

In this paper, we present the ACVAS (adaptive Admission Control for Virtualized Application Servers) approach. AC-

VAS provides SBAC for a dynamically scalable application server tier. Instead of relying only on rejection of new user sessions, it implements a simple mechanism that defers such sessions and then serves them as soon as possible in the near future. ACVAS implements per session admission control. It uses monitored resource utilizations for predicting a few steps ahead in the future. Then, based on the measured and predicted utilizations, it computes weighted utilizations. The weighted utilizations of individual server resources are used to make an admission control decision for each new session. Performance under highly varying user loads is guaranteed by automatic adjustment and tuning of the admission control mechanism.

The rest of the paper is organized as follows. Section II discusses important related works. Section III outlines the main tasks and the most important characteristics of the ACVAS approach. Section IV presents the system architecture. The proposed admission control algorithm is described in Section V. Our load prediction approach is detailed in Section VI. Section VII presents simulation results and Section VIII concludes the paper.

II. Related Work

The existing works on admission control for web-based systems can be classified according to the scheme presented in [7]. For instance, [8] and [9] are control-theoretic approaches, while [5] and [10] use machine learning techniques. Similarly, [2], [7], [11], and [12] are utility-based approaches.

Almeida et al. [7] proposed a joint resource allocation and admission control approach for a virtualized platform hosting a number of web applications, where each VM runs a dedicated web service application. The admission control mechanism uses request-based admission control. The optimization objective is to maximize the provider's revenue, while satisfying the customers' QoS requirements and minimizing the cost of resource utilization. The approach dynamically adjusts the fraction of capacity assigned to each VM and limits the incoming workload by serving only the subset of requests that maximize profits. It combines a performance model and an optimization model. The performance model determines future service level agreement (SLA) violations for each web service class based on a prediction of future workloads. The optimization model uses these estimates to make the resource allocation and admission control decisions.

Cherkasova and Phaal [2] proposed an SBAC approach that uses the traditional *on-off* control. It supports four admission control strategies: responsive, stable, hybrid, and predictive. The hybrid strategy tunes itself to be more stable or more responsive based on the observed QoS. The proposed approach measures server utilizations during predefined time intervals. Using these measured utilizations, it computes predicted utilizations for the next interval. If the predicted utilizations exceed specified thresholds, the admission controller rejects all new sessions in the next time interval and only serves the requests from already admitted sessions. Once the predicted utilizations drop below the given thresholds, the server changes its policy for the next time interval and begins to admit new sessions again.

Chen et al. [11] proposed admission control based on estimation of service times (ACES). That is, to differentiate and admit requests based on the amount of processing time required by a request. In ACES, admission of a request is decided by comparing the available computation capacity to the predetermined delay bound of the request. The service time estimation is based on an empirical expression, which is derived from an experimental study on a real web server.

Shaaban and Hillston [12] proposed cost-based admission control (CBAC), which uses a congestion control technique. Rather than rejecting user requests at high load, CBAC uses a discount-charge model to encourage users to postpone their requests to less loaded time periods. However, if a user chooses to go ahead with the request in a high load period, then an extra charge is imposed on the user request. The model is effective for e-commerce web sites when more users place orders that involve monetary transactions. A disadvantage of CBAC is that it requires CBAC-specific web pages to be included in the web application.

Muppala and Zhou [5] proposed the coordinated session-based admission control approach (CoSAC), which provides SBAC for multi-tier web applications with per session admission control. CoSAC also provides coordination among the states of tiers with a machine learning technique using a Bayesian network. The admission control mechanism differentiates and admits user sessions based on their type. For example, browsing mix session, ordering mix session, and shopping mix session. However, it remains unclear how it determines the type of a particular session at the start of the session.

Huang et al. [10] proposed admission control schemes for proportional differentiated services. It applies to services with different priority classes. The paper proposes two admission control schemes to enable proportional delay differentiated service (PDDS) at the application level. Each scheme is augmented with a prediction mechanism, which predicts the total maximum arrival rate and the maximum average waiting time for each priority class, based on the arrival rate in the current and last three measurement intervals. When a user request belonging to a specific priority class arrives, the admission control algorithm uses the time series predictor to forecast the average arrival rate of the class for the next interval, computes the average waiting time for the class for the next interval, and determines if the incoming user request is admitted to the server. If admitted, the client is placed at the end of the class queue.

Voigt and Gunningberg [8] proposed admission control based on the expected resource consumption of the requests, including a mechanism for service differentiation that guarantees low response time and high throughput for premium clients. The approach avoids overutilization of individual server resources, which are protected by dynamically setting the acceptance rate of resource-intensive requests. The adaptation of the acceptance rates (average no. of requests

per second) is done by using proportional-derivative (PD) feedback control loops.

Robertsson et al. [9] proposed an admission control mechanism for a web server system with control-theoretic methods. It uses a control-theoretic model of a G/G/1 system with an admission control mechanism for nonlinear analysis and design of controller parameters for a discrete-time proportional-integral (PI) controller. The controller calculates the desired admittance rate based on the reference value of average server utilization and the estimated or measured load situation (in terms of average server utilization). It then rejects those requests that could not be admitted.

III. ACVAS Approach

The main task of the ACVAS approach is to make admission control decisions for a scalable application server tier that consists of virtualized servers. The most important characteristics of the proposed approach are as follows.

A. SBAC with Per Session Admission Control

The SBAC approach in [2] implements *on-off* control, where acceptance of new sessions is turned on or off for the entire admission control interval. Thus the admission control decisions are made at the interval boundaries, which can not be changed inside an interval. A shortcoming of the *on-off* control is that it may lead to over-admission, especially when handling a bursty load, which could result in overloading of servers. To overcome this shortcoming of the *on-off* control, CoSAC [5] proposed per session admission control. ACVAS also implements SBAC with per session admission control. Thus, it makes an admission control decision for each new session.

B. Session Deferment Mechanism

All existing approaches discussed in this paper, except CBAC [12], have a common shortcoming in that they rely only on request rejection to avoid server overloading. However, CBAC has its own disadvantages. The discount-charge model of CBAC requires additional web pages to be included in the web application and it is only effective for e-commerce web sites when more users place orders.

We introduce a simple mechanism to defer user sessions that would otherwise be rejected. In ACVAS, such sessions are deferred on an *entertainment server*, which sends a wait message to the user and then redirects the user session to an application server as soon as a new server is provisioned or an existing server becomes less loaded. However, if the *entertainment server* also approaches its capacity limits, the new session is rejected. Therefore, for each new session request, the *admission controller* makes one of the three possible decisions: admit the session, defer the session, or reject the session.

C. Server Resource Utilization Metrics

The SBAC mechanism in [2] assumes that web servers are CPU-bound and therefore it measures server resource utilization in terms of CPU utilization. Voigt and Gunningberg [8] uses two individual server resources: CPU and network bandwidth. Likewise, the ACVAS approach can make use of one or more individual server resources. However, assuming that application servers often are CPU-bound or memory-bound, ACVAS measures server resource utilization in terms of CPU load average and memory utilization metrics.

D. Prediction Models

Cherkasova and Phaal [2] defined a simple method for computing the predicted resource utilization, yielding predicted resource utilizations by assigning certain weights to the current and the past utilizations. Muppala and Zhou [5] used the *exponential moving average* (EMA) method for making utilization predictions. Huang et al. [10] used machine learning techniques called support vector regression (SVR) and particle swarm optimization (PSO) for time-series prediction. Shaaban and Hillston [12] assumed a repeating pattern of workload over a suitable time period. Therefore, in their approach, load in a future period is predicted from the cumulative load of the corresponding previous period.

It is clear that admission control augmented with prediction models tends to produce better results and therefore ACVAS also uses a prediction model. However, for efficient runtime decision making, it is essential to avoid prediction models which might require intensive computation, frequent updates to their parameters, or (off-line) training. Thus, ACVAS uses a two-step approach [13], which has been designed to predict future resource loads under real-time constraints. The two-step approach consists of a load tracker and a load predictor. For the load tracker, ACVAS uses the EMA method.

E. Automatic Adjustment and Tuning for Better QoS

Schroeder et al. [6] considered automatic adjustment and tuning of the admission control mechanism to be the most difficult part. The SBAC approach in [2] proposed a *hybrid* policy for automatic adjustment and tuning of the admission control mechanism. The *hybrid* policy tries to achieve high QoS, while at the same time it aims to achieve better session throughput. This is done by adjusting a parameter called admission control weight, which gives more or less weight to the measured and the predicted utilizations. In [2], the weight parameter is adjusted based on the number of aborted sessions and refused connections.

For automatic adjustment and tuning of the admission control mechanism, ACVAS uses a similar approach as in [2]. However, it adjusts and tunes the weight parameter based on the following metrics: number of aborted sessions, number of deferred sessions, number of rejected sessions, and number of overloaded servers.

F. Quality and Efficiency of the Admission Control Mechanism

The quality and efficiency of an admission control mechanism might be measured in a number of different ways. Traditional SBAC approaches that are designed for a fixed number of servers may be evaluated based on server overload prevention and increase in the session throughput. Likewise,

Cherkasova and Phaal [2] used a QoS metric based on the number of aborted sessions and refused connections. However, for dynamically scalable server tiers, the quality and efficiency of an admission control approach is based on the tradeoff between number of servers and QoS. Therefore, we propose to measure *goodness* of an admission control mechanism based on the tradeoff between number of servers and six important QoS metrics: zero overloaded servers, maximum achievable session throughput, zero aborted sessions, minimum deferred sessions, zero rejected sessions, and minimum average response time for all admitted sessions.

IV. System Architecture

ACVAS is 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 [14]. In this context, it provides an admission control solution for the PaaS clouds. The system architecture consists of the following components, as shown in Figure 1: *admission controller*, *predictor*, *application server*, *local controller*, *application repository*, *global controller and resource allocator*, *cloud provisioner*, *HTTP load balancer*, and *entertainment server*. In this paper, our primary focus is on the *admission controller* and the *predictor*. Therefore, we will only describe these two components in detail. Only a brief description of the other components will be provided.

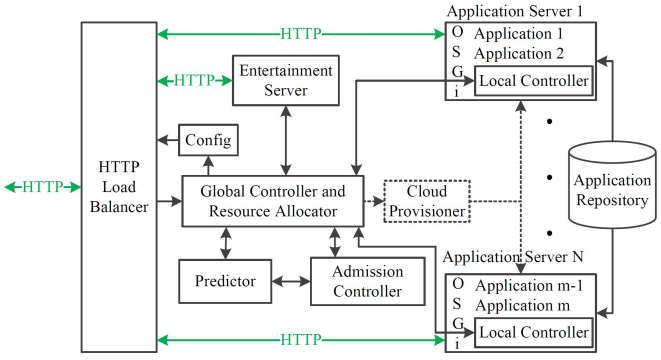

Fig. 1. The ACVAS system architecture

An *application server* instance runs on a dynamically provisioned VM. Each *application server* runs multiple web applications in an OSGi [15] 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 resource utilizations. For each admission control decision, the *admission controller* fetches resource utilization data from the *global controller*, which in turn fetches them from the *local controllers*. The *local controllers* also 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 Apache Felix OSGi Bundle Repository (OBR) [16]. The *global controller* also acts as a capacity manager and a resource allocator. It implements resource allocation and deallocation algorithms along with related policies, which include session-to-server and application-to-server allocation policies.

The *cloud provisioner* in the ACVAS system architecture refers to the cloud provisioner in an external 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 [17], which balances the load of requests for new user sessions among the application servers. For its functions, HAProxy maintains a configuration file containing information about application servers and application deployments on each server. As a result of application server scaling up and scaling down operations, the configuration file is frequently updated with new information.

When a new session request arrives, the *admission controller* obtains measured resource utilizations of individual servers from the *global controller* and likewise the predicted resource utilizations from the *predictor*. Based on the measured and predicted resource utilizations, it updates the server states. At any given time, an *application server* can be in one of three states: *open*, *closed*, or *overloaded*. The *open* state implies that the server is open for new sessions. Similarly, the *closed* state means that the server does not accept new sessions. The *overloaded* state is undesirable. It is characterized by very high utilization of the bottleneck server resource, which results in deteriorated performance. The *admission controller* uses these server states to make decisions for new session requests. If the *admission controller* finds at least one *open* server, the new session is admitted. If it cannot find an *open* server, the session is deferred onto the *entertainment server*. However, if the *entertainment server* also approaches its capacity limits, the new session request is rejected. All deferred sessions are automatically redirected to an *application server* in a FIFO (First In, First Out) order as soon as a new server is provisioned or a *closed* server becomes *open*. When admitting a mix of deferred and new sessions, deferred sessions are given priority over new sessions, as described in the next section.

V. Admission Control Algorithm

The admission control algorithm is given as Algorithm 1. For the sake of clarity, the concepts used in the algorithm and their notation are summarized in Table I.

The algorithm is activated when a new session request arrives or when the *admission controller* finds at least one deferred session. The first step (line 3) concerns automatic adjustment and tuning of the admission control mechanism, as discussed in Section III-E. ACVAS adjusts and tunes the weight parameter (w) based on the following metrics: number of aborted sessions $|s_a|$, number of deferred sessions $|s_d|$, number of rejected sessions $|s_r|$, and number of overloaded

TABLE I
SUMMARY OF CONCEPTS AND THEIR NOTATION

s_a	list of aborted sessions
s_d	list of deferred sessions
s_n	list of new session requests
s_r	list of rejected sessions
S	list of application servers
S_o	list of open application servers
S_{over}	list of overloaded application servers
$LA(ent)$	load average of entertainment server
$LA(s)$	CPU load average of server s
$LA_m(s)$	measured load average of server s
$LA_p(s)$	predicted load average of server s
$MU(ent)$	memory utilization of entertainment server
$MU(s)$	memory utilization of server s
$MU_m(s)$	measured memory utilization of server s
$MU_p(s)$	predicted memory utilization of server s
w	weight parameter
LA_{UT}	load average upper threshold
MU_{UT}	memory utilization upper threshold
$admit(x, S_o)$	admit session x on a server in the list S_o
$defer(x)$	defer a session x
$pop(list)$	remove and return first element of the $list$
$reject(x)$	reject a session x
$updateW()$	update the weight parameter w

servers $|S_{over}|$. The automatic adjustment and tuning is defined as

$$w = \begin{cases} 1, & \text{if } |s_a| > 0 \vee |s_d| > 0 \vee |s_r| > 0 \\ 1, & \text{if } |S_{over}| > 0 \\ w - 0.01, & \text{otherwise if } w > 0.1 \end{cases} \quad (1)$$

The next step deals with the monitoring of the server resource utilizations and using them to predict server resource utilizations a few steps ahead in the future. The algorithm then calculates weighted utilizations by using w to give more or less weight to both the measured and the predicted utilizations (line 8 and 9). The weighted resource utilizations, $LA(s)$ and $MU(s)$, are then used to determine the state of each server (line 11). However, if $w = 1$, the algorithm skips the prediction step to avoid computing predicted utilizations that will not be used. The two-step load prediction method used by ACVAS is described in Section VI.

If the algorithm finds at least one *open* server (line 12), it admits a new or deferred session. The sessions are served in a FIFO order. However, to ensure reasonable response times for any deferred sessions and to further prevent them from starvation, deferred sessions are given priority over new sessions (line 14). The next step (line 18) deals with the case when the algorithm could not find an *open* server. In this case, the new session is deferred (line 20), subject to the state of the entertainment server. That is, if the entertainment server could not accommodate any more deferred sessions, the session is rejected (line 22).

Algorithm 1 Admission Control Algorithm

```
1:  while true do
2:    if |s_n| ≥ 1 ∨ |s_d| ≥ 1 then
3:      w := updateW()
4:      if w = 1 then
5:        ∀s∈S|LA(s) := LA_m(s)
6:        ∀s∈S|MU(s) := MU_m(s)
7:      else
8:        ∀s∈S|LA(s) := w · LA_m(s) + (1 − w) · LA_p(s)
9:        ∀s∈S|MU(s) := w · MU_m(s) + (1 − w) · MU_p(s)
10:     end if
11:     S_o := {∀s∈S|LA(s) < LA_UT ∧ MU(s) < MU_UT}
12:     if |S_o| ≥ 1 then
13:       if |s_d| ≥ 1 then
14:         admit(pop(s_d), S_o)
15:       else
16:         admit(pop(s_n), S_o)
17:       end if
18:     else if |s_n| ≥ 1 then
19:       if LA(ent) < LA_UT ∧ MU(ent) < MU_UT then
20:         defer(pop(s_n))
21:       else
22:         reject(pop(s_n))
23:       end if
24:     end if
25:   end if
26: end while
```

VI. LOAD PREDICTION

There are several existing load prediction models for web-based systems, such as [13], [18], and [19]. Andreolini and Casolari [13] and Andreolini et al. [18] proposed a two-step approach for predicting future resource loads under real-time constraints. The two-step approach is based on the rationale that periodic sampling of the resource utilization data offers an instantaneous view of the load conditions. However, raw data are of little help for distinguishing overload conditions. The direct use of the measured raw data does not solve the problem, because utilization data are highly variable. Prediction based on monitored data can be risky and inconvenient. Thus, it is preferable to operate on a representation of the load behavior of system resources. The approach involves load trackers that may offer a representative view of the load conditions to the load predictors, thus achieving the two-step approach.

A load tracker (LT) filters out the noise and yields a more regular view of the load trend of a resource [13]. It takes as input a resource measure s_i monitored at time t_i, and a set of previously collected n measures, that is $\overrightarrow{S_n}(t_i) = (s_{i-n}, ..., s_i)$, and outputs a representation of the load conditions l_i at time t_i. Formally, LT is a function $LT(\overrightarrow{S_n}(t_i)) : \mathbb{R}^n \rightarrow \mathbb{R}$ [13]. Multiple applications of LT provides a sequence of load values that yield a regular trend of the load conditions. There are different classes of linear and non-linear LTs, such as simple moving average (SMA), exponential moving average (EMA), and cubic spline (CS) [18]. SMA is a simple linear method, but has known shortcomings of increased oscillations when n is small and of significant delay when n is large. CS is a non-linear method that is more expensive to compute than SMA and EMA, but instead

of returning a new LT for each resource measure, it returns a new LT value after n measures [13]. More sophisticated (time-series) models often require training periods to compute the parameters and/or off-line analyses. Likewise, the linear (auto) regressive models such as ARMA and ARIMA, usually require frequent updates to their parameters in the case of highly variable systems [13]. Therefore, the ACVAS *predictor* implements an LT based on the EMA model, which limits the delay without incurring oscillations and computes an LT value for each measure.

EMA is the weighted mean of the n measures in the vector $\overrightarrow{S_n}(t_i)$, computed at time t_i, where $i > n$, and the weights decrease exponentially [13]. An EMA based LT is defined as

$$EMA(\overrightarrow{S_n}(t_i)) = \alpha \cdot s_i + (1-\alpha) \cdot EMA(\overrightarrow{S_n}(t_{i-1})) \quad (2)$$

where $\alpha = \frac{2}{n+1}$. The initial value $EMA(\overrightarrow{S_n}(t_n))$ is set to the arithmetic mean of the first n values [13]

$$EMA(\overrightarrow{S_n}(t_n)) = \frac{\sum_{j=0}^{n} s_j}{n} \quad (3)$$

The load predictor (LP) takes as input a set of LT values $\overrightarrow{L_q}(t_i) = l_{i-q}, ..., l_i$ and outputs a future LT value at time t_{i+k}, where $k > 0$ [13]. Formally, LP is a function $LP_k(\overrightarrow{L_q}(t_i)) : \mathbb{R}^q \rightarrow \mathbb{R}$. With the use of LTs that provide high correlation among values, even simple linear predictors are sufficient to predict the future behavior of resource load. The LP is characterized by the prediction window k and the past time window q. Using a simple linear regression model [20], the LP would be based on all LT values $\overrightarrow{L_q}(t_i)$ in the past time window. The LP of the LT is based on a straight line defined as

$$l = \Theta_0 + \Theta_1 \cdot t \quad (4)$$

where Θ_0 and Θ_1 are unknown constants, called regression coefficients, which can be estimated at runtime based on the LT values $\overrightarrow{L_q}(t_i)$ in the past time window. One common approach to estimate these regression coefficients is to use the least-square estimation method [20]. The least-square estimators of Θ_0 and Θ_1, say $\hat{\Theta}_0$ and $\hat{\Theta}_1$, are computed as

$$\hat{\Theta}_0 = \bar{l} - \hat{\Theta}_1 \cdot \bar{t} \quad (5)$$

and

$$\hat{\Theta}_1 = \frac{\sum_{j=i-q}^{i} (l_j \cdot t_j) - \frac{\left(\sum_{j=i-q}^{i} l_j\right) \cdot \left(\sum_{j=i-q}^{i} t_j\right)}{q}}{\sum_{j=i-q}^{i} t_j^2 - \frac{\left(\sum_{j=i-q}^{i} t_j\right)^2}{q}} \quad (6)$$

where

$$\bar{l} = \frac{1}{q} \sum_{j=i-q}^{i} l_j \quad (7)$$

and

$$\bar{t} = \frac{1}{q} \sum_{j=i-q}^{i} t_j \quad (8)$$

The LP returns a predicted future LT value $\hat{l}_{i+k}$ that corresponds to the point (t_{i+k}, l_{i+k}). It is computed as follows:

$$LP_k(\overrightarrow{L_q}(t_i)) = \hat{l}_{i+k} = \hat{\Theta}_0 + \hat{\Theta}_1 \cdot t_{i+k} \quad (9)$$

Prediction results depend upon selection of proper values for the LT and LP parameters. Therefore, it is necessary to find a value of n that represents a good tradeoff between a reduced delay and a reduced degree of oscillations [13]. Likewise, the values of q and k should be carefully selected.

VII. Simulation Results

A convenient and quick way of testing new algorithms and solutions involving complex environments is to write and run software simulations. A special kind of simulations called discrete-event simulations are most appropriate for simulating and evaluating cluster, grid, and cloud computing environments and systems [21]. Therefore, we have developed a discrete-event simulation for the ACVAS approach. Also, for a comparison of results with the alternative existing approaches, we have developed a discrete-event simulation for the SBAC approach of Cherkasova and Phaal [2], here referred to as the *alternative* approach. The simulations are written in the Python programming language and are based on the SimPy simulation framework [22].

A. Experimental Design and Setup

We considered two scenarios of interest in two separate experiments. Scenario 1 in experiment 1 used synthetic workload traces, while scenario 2 in experiment 2 used workload traces derived from a real web-based system. The LT and LP parameters were carefully chosen based on the recommendations in [13]: $n = 30$, $q = 15$, and $k = 30$.

1) Experiment 1: Synthetic Workload Traces: Experiment 1 used synthetic workload traces. It was designed to generate a load representing a maximum of 500 simultaneous user sessions in two separate load peaks. In each peak, the sessions were ramped up from 0 to 500. After the ramp-up phase, the number of sessions was maintained constant for a while and then reduced back to 0 in a ramp-down phase. The two load peaks were similar, except that the sessions in the second peak were ramped up twice as quickly as in the first peak. Each session was randomly assigned to one particular web application. The experiment used 100 simulated web applications of varying resource needs.

2) Experiment 2: Workload Traces Based on Access Logs: Experiment 2 was designed to simulate a load representing a workload trace from a real web-based system. The traces were derived from SQUID access logs obtained from the IRCache [23] project. As the access logs did not include session information, we defined a session as a series of requests from the same originating IP-address, where the time between individual requests was less than 15 minutes. We

then produced a histogram of sessions per second and used linear interpolation and scaling by a factor of 30 to obtain the load traces used in the experiment. As in Experiment 1, each session was randomly assigned to one particular web application out of 100.

B. Results and Analysis

Now we compare the experimental results of the ACVAS approach with that of the *alternative* approach. The comparison of the results is based on the *goodness* criteria defined in Section III-F.

1) Experiment 1: Synthetic Workload Traces: Figure 2 presents ACVAS results from experiment 1. The results show a dynamically scalable application server tier, which consists of a varying number of VMs. The prediction results with the EMA-based two-step method correlated well with the measured results: the root-mean-square error (RMSE) for CPU and memory were 0.0284 and 0.0161 respectively. ACVAS used a maximum of 10 servers with 0 overloaded servers, 0 aborted sessions, 25 deferred sessions, and 0 rejected sessions. There were a total of 3709 completed sessions with an average response time always below 1 second. Thus, ACVAS provided a good tradeoff between the number of servers and the QoS requirements.

The results of the *alternative* approach are shown in Figure 3. The *alternative* approach also used a maximum of 10 servers, but with 5 occurrences of server overloading, 0 aborted sessions, and 72 rejected sessions. There were a total of 3741 completed sessions with an average response time always below 1 second. Thus, in experiment 1, the *alternative* approach completed 3741 sessions compared to 3709 sessions by ACVAS, but with 72 rejected sessions compared to 0 rejected sessions by ACVAS.

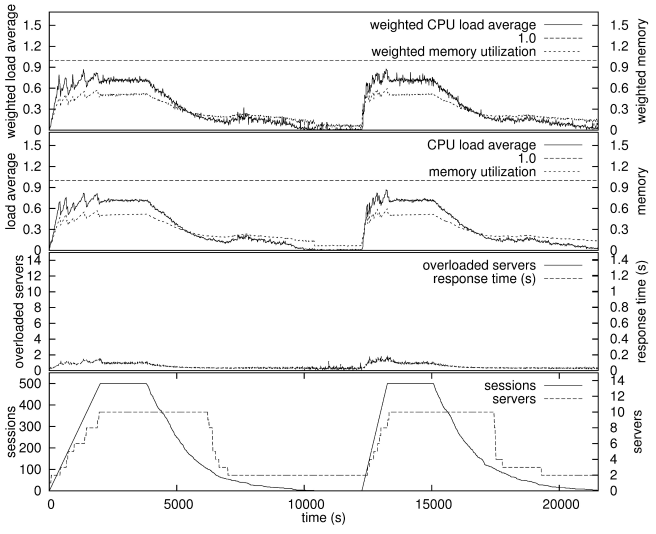

Fig. 2. Experiment 1: ACVAS with synthetic load

2) Experiment 2: Workload Traces Based on Access Logs: Figure 4 presents ACVAS results from experiment 2. ACVAS used a maximum of 16 servers with 0 overloaded servers, 0 aborted sessions, 20 deferred sessions, and 0 rejected sessions. There were a total of 8559 completed sessions with an average response time always below 1 second. The prediction accuracy was similar to that in the first experiment. Thus, ACVAS provided a good tradeoff between the number of servers and the QoS requirements for experiment 2 as well.

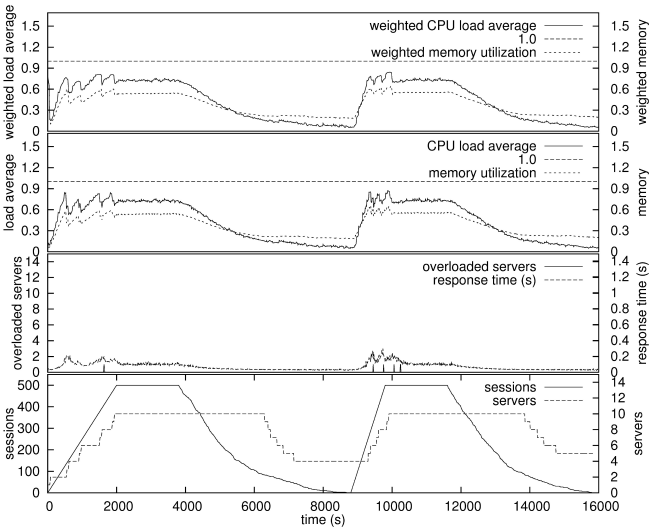

Fig. 3. Experiment 1: *alternative* approach with synthetic load

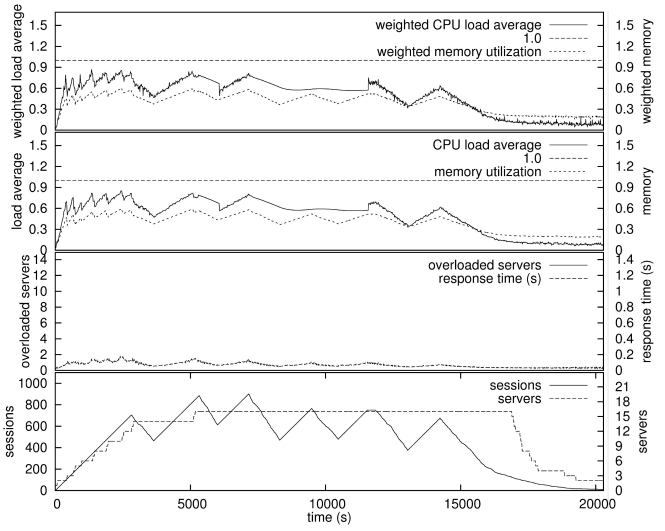

Fig. 4. Experiment 2: ACVAS with real load

The results of the *alternative* approach are shown in Figure 5. The *alternative* approach used a maximum of 17 servers with 3 occurrences of server overloading, 0 aborted sessions, and 55 rejected sessions. There were a total of 8577 completed sessions with an average response time always below 1 second. Thus, the *alternative* approach used an almost equal number of servers, but it did not prevent them from becoming overloaded. Also, it completed 8577 sessions compared to 8559 sessions by ACVAS, but with 55 rejected sessions compared to 0 sessions rejected by ACVAS.

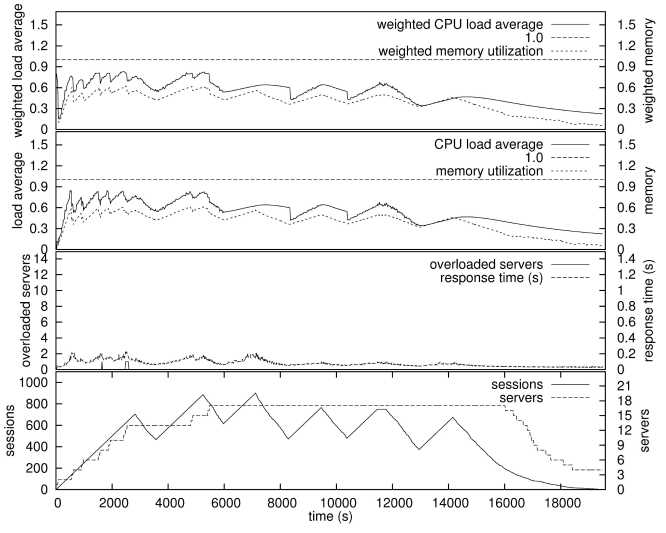

Fig. 5. Experiment 2: *alternative* approach with real load

VIII. Conclusion

In this paper, we presented an SBAC approach for dynamically scalable application server tiers called ACVAS. ACVAS uses measured and predicted server load of individual servers to make admission control decisions. Our prediction approach is based on a two-step method. The first step filters out noise to yield a more regular view of the load trend. The second step outputs a future load value based on a set of load trend values. ACVAS uses per session admission control, which reduces over-admission. It also implements a simple session deferment mechanism, which decreases the number of rejected sessions. We presented a discrete-event simulation of the proposed approach along with experimental results.

The evaluation and analyses compared ACVAS against an existing approach available in the literature. We considered two different scenarios in two separate experiments. The first scenario used a synthetic workload while the second scenario used workload traces derived from a real web-based system. The results showed that ACVAS provides a good tradeoff between the number of servers used and the QoS requirements. In comparison with the *alternative* admission control approach, ACVAS provided significant improvements in terms of server overload prevention and reduction of rejected sessions.

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.

References

[1] 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.

[2] L. Cherkasova and P. Phaal, "Session-based admission control: a mechanism for peak load management of commercial web sites," *Computers, IEEE Transactions on*, vol. 51, no. 6, pp. 669 –685, jun 2002.

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

[4] Y. Raivio, O. Mazhelis, K. Annapureddy, R. Mallavarapu, and P. Tyrväinen, "Hybrid cloud architecture for short message services," in *Proceedings of the 2nd International Conference on Cloud Computing and Services Science*, ser. CLOSER '12, 2012.

[5] S. Muppala and X. Zhou, "Coordinated session-based admission control with statistical learning for multi-tier internet applications," *Journal of Network and Computer Applications*, vol. 34, no. 1, pp. 20 – 29, 2011.

[6] B. Schroeder, M. Harchol-Balter, A. Iyengar, E. Nahum, and A. Wierman, "How to determine a good multi-programming level for external scheduling," in *Data Engineering, 2006. ICDE '06. Proceedings of the 22nd International Conference on*, april 2006, p. 60.

[7] J. Almeida, V. Almeida, D. Ardagna, I. Cunha, C. Francalanci, and M. Trubian, "Joint admission control and resource allocation in virtualized servers," *J. Parallel Distrib. Comput.*, vol. 70, no. 4, pp. 344–362, Apr. 2010.

[8] T. Voigt and P. Gunningberg, "Adaptive resource-based web server admission control," in *Computers and Communications, 2002. Proceedings. ISCC 2002. Seventh International Symposium on*, 2002, pp. 219 – 224.

[9] A. Robertsson, B. Wittenmark, M. Kihl, and M. Andersson, "Admission control for web server systems - design and experimental evaluation," in *Decision and Control, 2004. CDC. 43rd IEEE Conference on*, vol. 1, dec. 2004, pp. 531 –536 Vol.1.

[10] C.-J. Huang, C.-L. Cheng, Y.-T. Chuang, and J.-S. R. Jang, "Admission control schemes for proportional differentiated services enabled internet servers using machine learning techniques," *Expert Systems with Applications*, vol. 31, no. 3, pp. 458 – 471, 2006.

[11] X. Chen, H. Chen, and P. Mohapatra, "ACES: An efficient admission control scheme for QoS-aware web servers," *Computer Communications*, vol. 26, no. 14, pp. 1581 – 1593, 2003.

[12] Y. A. Shaaban and J. Hillston, "Cost-based admission control for internet commerce QoS enhancement," *Electronic Commerce Research and Applications*, vol. 8, no. 3, pp. 142 – 159, 2009.

[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] 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. [Online]. Available: http://www.cloudsw.org/

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

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

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

[18] M. Andreolini, S. Casolari, and M. Colajanni, "Models and framework for supporting runtime decisions in web-based systems," *ACM Trans. Web*, vol. 2, no. 3, pp. 17:1–17:43, Jul. 2008.

[19] P. Saripalli, G. V. R. Kiran, R. R. Shankar, H. Narware, and N. Bindal, "Load prediction and hot spot detection models for autonomic cloud computing," in *Proceedings of the 2011 Fourth IEEE International Conference on Utility and Cloud Computing*, ser. UCC '11. Washington, DC, USA: IEEE Computer Society, 2011, pp. 397–402.

[20] D. Montgomery, E. Peck, and G. Vining, *Introduction to Linear Regression Analysis*, ser. Wiley Series in Probability and Statistics. John Wiley & Sons, 2012.

[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. [Online]. Available: http://dx.doi.org/10.1002/spe.995

[22] N. Matloff, *A Discrete-Event Simulation Course Based on the SimPy Language*. University of California at Davis, 2006. [Online]. Available: http://heather.cs.ucdavis.edu/~matloff/simcourse.html

[23] IRCache project. [Online]. Available: http://www.ircache.net