# Developing Cloud Software

## Algorithms, Applications, and Tools

Edited by

Ivan Porres
Tommi Mikkonen
Adnan Ashraf

TUCS General Publication
No 60, October 2013
ISBN 978-952-12-2952-7
ISSN 1239-1905

# 3 Prediction-Based Virtual Machine Provisioning and Admission Control for Multi-tier Web Applications

Adnan Ashraf, Benjamin Byholm, and Ivan Porres
Department of Information Technologies
Åbo Akademi University, Turku, Finland
Email: {aashraf, bbyholm, iporres}@abo.fi

**Abstract**–This chapter presents a prediction-based, cost-efficient Virtual Machine (VM) provisioning and admission control approach for multi-tier web applications. The proposed approach provides automatic deployment and scaling of multiple simultaneous web applications on a given Infrastructure as a Service (IaaS) cloud in a shared hosting environment. It monitors and uses resource utilization metrics and does not require a performance model of the applications or the infrastructure dynamics. The shared hosting environment allows us to share VM resources among deployed applications, reducing the total number of required VMs. The proposed approach comprises three sub-approaches: a reactive VM provisioning approach called ARVUE, a hybrid reactive-proactive VM provisioning approach called Cost-efficient Resource Allocation for Multiple web applications with Proactive scaling (CRAMP), and a session-based adaptive admission control approach called adaptive Admission Control for Virtualized Application Servers (ACVAS). Performance under varying load conditions is guaranteed by automatic adjustment and tuning of the CRAMP and ACVAS parameters. The proposed approach is demonstrated in discrete-event simulations and is evaluated in a series of experiments involving synthetic as well as realistic load patterns.

**Keywords-**Cloud computing, virtual machine provisioning, admission control, web application, cost-efficiency, performance.

## 3.1 Introduction

The resource needs of web applications vary over time, depending on the number of concurrent users and the type of work performed. This stands in contrast to static content, which requires no further processing by the server than sending predefined data to an output stream. As the demand for an application grows, so does its demand for resources, until the demand for a key resource outgrows the supply and the performance of the application deteriorates. Users of an application starved for resources tend to notice this as increased latency and lower throughput for requests, or they might receive no service at all if the problem progresses further.

To handle multiple simultaneous users, web applications are traditionally deployed in a three-tiered architecture, where a computer cluster of fixed size represents the application server tier. This cluster provides dedicated application hosting to a fixed amount of users. There are two problems with this approach: firstly, if the amount of users grows beyond the predetermined limit, the application will become starved for resources. Secondly, while the amount of users is lower than this limit, the unused resources constitute waste.

A recent study showed that the underutilization of servers in enterprises is a matter of concern [37]. This inefficiency is mostly due to application isolation: a consequence of dedicated hosting. Sharing of resources between applications leads to higher total resource utilization and thereby to less waste. Thus, the level of utilization can be improved by implementing what is known as shared hosting [36]. Shared hosting is already commonly used by web hosts to serve static content belonging to different customers from the same set of servers, as no sessions need to be maintained.

Cloud computing already allows us to alleviate the utilization problem by dynamically adding or removing available Virtual Machine (VM) instances at the infrastructure level. However, the problem remains to some extent, as Infrastructure as a Service (IaaS) providers operate at the level of VMs, which does not provide high granularity. This can be solved by operating at the Platform as a Service (PaaS) level instead. However, one problem still remains: resources cannot be immediately allocated or deallocated. In many cases, there exists a significant provisioning delay on the order of minutes.

Shared hosting of dynamic content also presents new challenges: capacity planning is complicated, as different types of requests might require varying amounts of a given resource. For example, consider a web shop: adding items to the shopping basket might require less resources than computing the final price with taxes and rebates included. During a shopping session, a user might add several items to their shopping basket, while the final price is only

computed at checkout. The session also has to be reliably maintained, so that the contents of the shopping basket do not suddenly disappear. Otherwise, the shop might lose customers.

Application-specific knowledge is necessary for a PaaS provider to efficiently host complex applications with highly varying resource needs. When hosting third-party dynamic content in a shared environment that application-specific knowledge might be unavailable. It is also unfeasible for a PaaS provider to learn enough about all of the applications belonging to their customers.

Traditional performance models based on queuing theory try to capture the behavior of purely open or closed systems [25]. However, Rich Internet Applications (RIAs) have workloads with sessions, exhibiting a partially-open behavior, which includes components from both the open and the closed model. Given a better performance model of an application, it might be possible to plan the necessary capacity, but the problem of obtaining said model remains.

If the hosted applications are seldom modified it might be feasible to automatically derive the necessary performance models by benchmarking each application in isolation [36]. This might apply to hosting first- or second-party applications. However, when hosting third-party applications under continuous development, they may well change frequently enough for this to be unfeasible.

Another problem is determining the amount of VMs to have at a given moment. As one cannot provision fractions of a VM, the actual capacity demand will need to be quantized in one way or another. Figure 3.1 shows a demand and a possible quantization thereof. Overallocation implies an opportunity cost — underallocation implies lost revenue.

Finally, there is also the issue of admission control. This is the problem of determining how many users to admit to a server at a given moment in time, so that said server does not become overloaded. Preventive measures are a good way of keeping server overload from occurring at all. This is traditionally achieved by only relying on two possible decisions: rejection or acceptance.

Once more, the elastic nature of the cloud means that we have more resources available at our discretion and can scale up to accommodate the increase in traffic. However, resource allocation still takes a considerable amount of time, due to the provisioning delay, and admitting too much traffic is an unattractive option, even if new resources will arrive in a while.

This chapter presents a prediction-based, cost-efficient VM provisioning and admission control approach for multi-tier web applications. The proposed approach provides automatic deployment and scaling of multiple si-

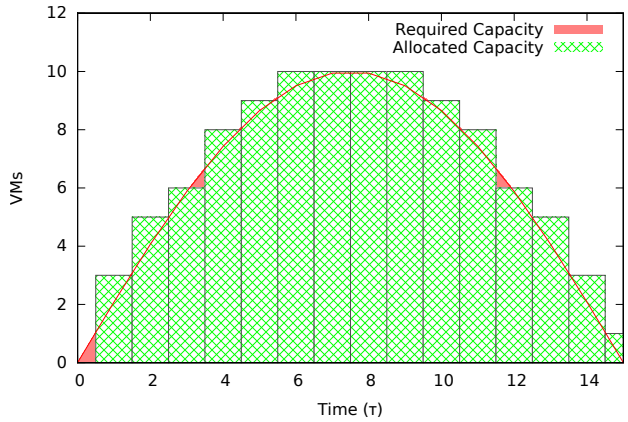



Figure 3.1: The actual capacity demand has to be quantized at a resolution determined by the capacity of the smallest VM available for provisioning. Overallocation means an opportunity cost, underallocation means lost revenue.

multaneous third-party web applications on a given IaaS cloud in a shared hosting environment. It monitors and uses resource utilization metrics and does not require a performance model of the applications or the infrastructure dynamics. The research applies to PaaS providers and large Software as a Service (SaaS) providers with multiple applications. We deal with stateful RIAs over the Hypertext Transfer Protocol (HTTP).

The proposed approach comprises three sub-approaches. It provides a reactive VM provisioning approach called ARVUE [9], a hybrid reactive-proactive VM provisioning approach called Cost-efficient Resource Allocation for Multiple web applications with Proactive scaling (CRAMP) [8], and a session-based adaptive admission control approach called adaptive Admission Control for Virtualized Application Servers (ACVAS) [7]. Both ARVUE and CRAMP provide autonomous shared hosting of third-party Java Servlet applications on an IaaS cloud. However, CRAMP provides better responsiveness and results than the purely reactive scaling of ARVUE. We concluded that admission control might be able to reduce the risk of servers becoming overloaded. Therefore, the proposed approach augments VM provisioning with a session-based adaptive admission control approach called ACVAS. ACVAS implements per-session admission, which reduces the risk of over-admission. Furthermore, instead of relying only on rejection of new sessions, it implements a simple session deferment mechanism that reduces

the number of rejected sessions while increasing session throughput. Thus, the admission controller can decide to admit, defer, or reject an incoming new session. Performance under varying load conditions is guaranteed by automatic adjustment and tuning of the CRAMP and ACVAS parameters. The proposed approach is demonstrated in discrete-event simulations and is evaluated in a series of experiments involving synthetic as well as realistic load patterns. Byholm [11] described the prototype implementation of these concepts.

We proceed as follows. Section 3.2 discusses important related works. Section 3.3 presents the system architecture. The proposed VM provisioning and admission control algorithms are described in Section 3.4. In Section 3.5, we present simulation results before concluding in Section 3.6.

## 3.2 Related Work

Due to the problems mentioned in Section 3.1, existing works on PaaS solutions tend to use dedicated hosting on a VM-level for RIAs. This gives the level of isolation needed to reliably host different applications without them interfering with each other, as resource management will be handled by the underlying operating system. However, this comes at the cost of prohibiting resource sharing among instances. In order to reliably do shared hosting of third-party applications, there is a need for a way to prevent applications from interfering with each other, without preventing the sharing of resources. Google App Engine is different here in that it instead offers a sandboxed runtime for the applications to run in [17]. Another way is to use shared hosting to run multiple applications in the same Java Virtual Machine (JVM) [1].

There are many metrics available for measuring Quality of Service (QoS). A common metric is Round Trip Time (RTT), which is a measure of the time required for sending a request and receiving a response. This approach has a drawback in that different programs might have various expected processing times for requests of different types. This means that application-specific knowledge is required when using RTT as a QoS metric. This information might not be easy to obtain if an application is under constant development. Furthermore, when a server nears saturation, its response time grows exponentially. This makes it difficult to obtain good measurements in a high-load situation. For this reason, we use server Central Processing Unit (CPU) load average and memory utilization as the primary QoS metrics. An overloaded server will fail to meet RTT requirements.

Reactive scaling works by monitoring user load in the system and reacting to observed variations therein by making decisions for allocation or

deallocation. In our previous work [11, 1, 9], we built a prototype of an autonomous PaaS called ARVUE. It implements reactive scaling. However, in many cases, the reactive approach suffers in practice, due to delays of several minutes inherent in the provisioning of VMs [31]. This shortcoming is avoidable with proactive scaling.

Proactive scaling attempts to overcome the limitations of reactive scaling by forecasting future load trends and acting upon them, instead of directly acting on observed load. Forecasting usually has the drawback of added uncertainty, as it introduces errors into the system. The error can be mitigated by a hybrid approach, where forecast values are supplemented with error estimates, which affect a blend weight for observed and forecast values. We have developed a hybrid reactive-proactive VM provisioning algorithm called CRAMP [8].

Admission control is a strategy for keeping servers from becoming overloaded. This is achieved by limiting the amount of traffic each server receives by means of an intermediate entity known as an *admission controller*. The admission controller may deny entry to fully utilized servers, thereby avoiding server overload. If a server were to become overloaded, all users of that server, whether existing or arriving, would suffer from deteriorated performance and possible Service-Level Agreement (SLA) violations.

Traditional admission control strategies have mostly been request-based, where admission control decisions would be made for each individual request. This approach is not appropriate for stateful web applications from a user experience point of view. If a request were to be denied in the middle of an active session, when everything was working well previously, the user would have a bad experience. Session-Based Admission Control (SBAC) is an alternative strategy, where the admission decision is made once for each new session and then enforced for all requests inside of a session [26]. This approach is better from the perspective of the user, as it should not lead to service being denied in the middle of a session. This approach has usually been implemented using interval-based on-off control, where the admission controller either admits or rejects all sessions arriving within a predefined time interval. This approach has a flaw in that servers may become overloaded if they accept too many requests in an admission interval, as the decisions are made only at interval boundaries. Per-session admission control avoids this problem by making a decision for each new session, regardless of when it arrives. We have developed ACVAS [7], a session-based admission control approach with per-session admission control. ACVAS uses SBAC with a novel deferment mechanism for sessions, which would have been rejected with the traditional binary choice of acceptance or rejection.

### 3.2.1 VM Provisioning Approaches

Most of the existing works on VM provisioning for web-based systems can be classified into two main categories: plan-based approaches and control theoretic approaches [15, 29, 30]. Plan-based approaches can be further classified into workload prediction approaches [31, 6] and performance dynamics model approaches [39, 21, 14, 23, 19]. One common difference between all existing works discussed here and the proposed approach is that the proposed approach uses shared hosting. Another distinguishing characteristic of the proposed approach is that in addition to VM provisioning for the application server tier, it also provides dynamic scaling of multiple web applications. In ARVUE [9], we used shared hosting with reactive resource allocation. In contrast, our proactive VM provisioning approach CRAMP [8] provides improved QoS with prediction-based VM provisioning.

Ardagna et al. [6] proposed a distributed algorithm for managing SaaS cloud systems that addresses capacity allocation for multiple heterogeneous applications. Raivio et al. [31] used proactive resource allocation for short message services in hybrid clouds. The main drawback of their approach is that it assumes server processing capacity in terms of messages per second, which is not a realistic assumption for HTTP traffic where different types of requests may require different amounts of processing time. Nevertheless, the main challenge in the prediction-based approaches is in making good prediction models that could ensure high prediction accuracy with low computational cost. In our proposed approach, CRAMP is a hybrid reactive-proactive approach. It uses a two-step prediction method with Exponential Moving Average (EMA), which provides high prediction accuracy under real-time constraints. Moreover, it gives more or less weight to the predicted utilizations based on the Normalized Root Mean Square Error (NRMSE).

TwoSpot [39] supports hosting of multiple web applications, which are automatically scaled up and down in a dedicated hosting environment. The scaling down is decentralized, which may lead to severe random drops in performance. Hu et al. [21] presented an algorithm for determining the minimum number of required servers, based on the expected arrival rate, service rate, and SLA. In contrast, the proposed approach does not require knowledge about the infrastructure or performance dynamics. Chieu et al. [14] presented an approach that scales servers for a particular web application based on the number of active user sessions. However, the main challenge is in determining suitable threshold values on the number of user sessions. Iqbal et al. [23] proposed an approach for multi-tier web applications, which uses response time and CPU utilization metrics to determine the bottleneck tier and then scales it by provisioning a new VM. Han et al. [19] proposed

a reactive resource allocation approach to integrate VM-level scaling with a more fine-grained resource-level scaling. In contrast, CRAMP supports hybrid reactive-proactive resource allocation with proportional and derivative factors to determine the number of VMs to provision.

Dutreilh et al. [15] and Pan et al. [29] used control theoretic models to design 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. [30] 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 comparison to the control theoretic approaches, our proposed approach also uses proportional and derivative factors, but it does not require knowledge about the performance models or infrastructure dynamics.

### 3.2.2 Admission Control Approaches

The existing works on admission control for web-based systems can be classified according to the scheme presented in Almeida et al. [3]. For instance, Robertsson et al. [32] and Voigt and Gunningberg [38] are control theoretic approaches, while Huang et al. [22] and Muppala and Zhou [26] use machine learning techniques. Similarly, Cherkasova and Phaal [13], Almeida et al. [3], Chen et al. [12], and Shaaban and Hillston [33] are utility-based approaches.

Almeida et al. [3] 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 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 [13] 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. [12] 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 [33] 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 [26] proposed the Coordinated Session-based Admission Control (CoSAC) approach, 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 in the first place. Huang et al. [22] 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 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 [38] 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 number of requests per second) is done by using Proportional-Derivative (PD) feedback control loops. Robertsson et al. [32] 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 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.

## 3.3 Architecture

The system architecture of the proposed VM provisioning and admission control approach is depicted in Figure 3.2. It consists of the following components: a *load balancer* with an accompanying *configuration file*, the *global controller*, the *admission controller*, the *cloud provisioner*, the *application servers* containing *local controllers*, the *load predictors*, an *entertainment server*, and an *application repository*.

The purpose of the *load balancer* is to distribute the workload evenly throughout the system, while the *admission controller* is responsible for admitting users, when deemed possible. The *cloud provisioner* is an external component, which represents the control service of the underlying IaaS provider. *Application servers* are dynamically provisioned VMs belonging to the underlying IaaS cloud, capable of running multiple concurrent applications contained in an *application repository*.

### 3.3.1 Load Balancer

The purpose of the load balancer is to distribute the workload among the available application servers. The prototype implementations of ARVUE [1, 9, 11] and CRAMP [8] use the free, lightweight load balancer HAProxy [34], which can act as a reverse proxy in either of two modes: Transmission Control

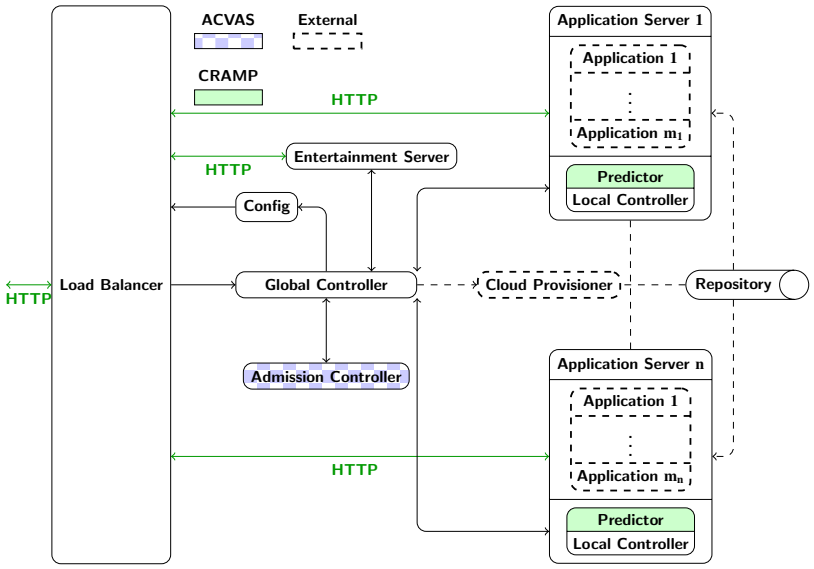



Figure 3.2: System architecture of the proposed VM provisioning and admission control approach.

Protocol (TCP) or HTTP, which correspond to layers 4 and 7 in the Open Systems Interconnection (OSI) model. We use the HTTP mode, as ARVUE and CRAMP are designed for stateful web applications over HTTP.

HAProxy includes powerful logging capabilities using the Syslog standard. It also supports session affinity, the ability to direct requests belonging to a single session to the same server, and Access Control Lists (ACLs), even in combination with Secure Socket Layer (SSL) since version 1.5.

Session affinity is supported by cookie rewriting or insertion. As the prototype implementations of ARVUE and CRAMP are designed for Vaadin applications [18], which use the Java Servlet technology, applications already use the `JSESSIONID` cookie, which uniquely identifies the session the request belongs to. Thus, HAProxy only has to intercept the `JSESSIONID` cookie sent from the application to the client and prefix it with the identifier of the backend in question. Incoming `JSESSIONID` cookies are similarly intercepted and the inserted prefix is removed before they are sent to the applications.

HAProxy also comes with a built-in server health monitoring system, based on making requests to servers and measuring their response times. However, this system is currently not in use, as the proposed approach does its own health monitoring by observing different metrics.

When an application request arrives at the load balancer, it gets redirected to a suitable server according to the current configuration. A request for an application not deployed at the moment is briefly sent to a server tasked with entertaining the user and showing that the request is being processed until the application has been successfully deployed, after which it is delivered to the correct server. This initial deployment of an application will take a much longer time than subsequent requests, currently on the order of several seconds.

The load balancer is dynamically reconfigured by the global controller as the properties of the cluster change. When an application is deployed, the load balancer is reconfigured with a mapping between a Uniform Resource Identifier (URI) that uniquely identifies the application and a set of application servers hosting the application, by means of an ACL, a usage declaration and a backend list. Weights for servers are periodically recomputed according to the health of each server, with higher weights assigned to less loaded servers.

The weights are integers in the range $[0, W_{\mathrm{MAX}}]$, where higher values mean higher priority. In the case of HAProxy, $W_{\mathrm{MAX}} = 255$. The value 0 is special in that it effectively prevents the server from receiving any new requests. This is explained by the weighting algorithm in Algorithm 3.1, which distributes the load among the servers so that each server receives a number of requests proportional to its weight divided by the sum of all the weights. This is a simple mapping of the current load to the weight interval. Here, $S(k)$ is the set of servers at discrete time $k$, $C_w(s,k)$ is the weighted load average of server $s$ at time $k$, $C(s,k)$ is the measured load average of server $s$ at time $k$, and similarly $\hat{C}(s,k)$ is the predicted load average of server $s$ at time $k$. $w_c \in [0,1]$ is the weighting coefficient for CPU load average, $C_{US}$ is the server load average upper threshold, and $W(s,k)$ is the weight of server $s$ at time $k$ for load balancing. Thus, the algorithm obtains $C(s,k)$ and $\hat{C}(s,k)$ of each server $s \in S(k)$ and uses them along with $w_c$ to compute $C_w(s,k)$ of each server (line 1). Afterwards, it uses $C_w(s,k)$ to compute $W(s,k)$ of each server $s$ (lines 2–10). The notation used in the algorithm is also defined in Table 3.1 in Section 3.4.

### 3.3.2 Global Controller

The global controller is responsible for managing the cluster by monitoring its constituents and reacting to changes in the observed parameters, as reported by the local controllers. It can be viewed as a control loop that implements the VM provisioning algorithms described in Section 3.4. Inter-VM communication is performed using Java Remote Method Invocation (RMI), which

**Algorithm 3.1.** Weighting algorithm

$\forall s \in S(k) | C_w(s,k) := w_c \cdot C(s,k) + (1 - w_c) \cdot \hat{C}(s,k)$
**for** $s \in S(k)$ **do**
  **if** $C_w(s,k) \geq C_{US}$ **then**
    $W(s,k) := 0$
  **else if** $C_w(s,k) > 0$ **then**
    $W(s,k) := \left\lceil W_{\text{MAX}} - \frac{C_w(s,k)}{C_{US}} \cdot W_{\text{MAX}} \right\rceil$
  **else**
    $W(s,k) := W_{\text{MAX}}$
  **end if**
**end for**

is a practical implementation of the Proxy pattern, performing distributed object communication: the object-oriented equivalent of Remote Procedure Call (RPC).

An alternative to RMI could be the Remote Open Services Gateway initiative (OSGi) specification [28], implemented in both Apache CXF and Eclipse ECF. This was not attempted, as it would have taken more time to implement. However, this approach might be easier to maintain. It would also be possible to use a Representational State Transfer (REST) interface through HTTP, which could make it easier to interface with the inner workings of the platform.

### 3.3.3 Admission Controller

The admission controller is responsible for admitting users to application servers. It supplements the load balancer in ensuring that the servers do not become overloaded by deciding whether to admit, defer, or reject traffic. It makes admission control decisions per session, not per request. This allows for a smoother user experience in a stateful environment, as a user of an application would not enjoy suddenly having requests to the application denied, when everything was working fine a moment ago. The admission controller implements per-session admission control. Unlike the traditional on-off approach, which makes admission control decisions on an interval basis, the per-session admission approach is not as vulnerable to sudden traffic fluctuations. The on-off approach can lead to servers becoming overloaded if they are set to admit traffic and a sudden traffic spike occurs [7]. The admission control decisions are based on prediction of future load trends combined with server health monitoring, as explained in Section 3.4.4.

### 3.3.4 Cloud Provisioner

The cloud provisioner is an external component, which represents the control service of the underlying IaaS provider. The global controller communicates with the cloud provisioner through its custom Application Programming Interface (API) in order to realize the decisions on how to manage the server tier. Proper application of the façade pattern decouples the proposed approach from the underlying IaaS provider. The prototypes [1, 9, 8, 11] currently support Amazon Elastic Compute Cloud (EC2) in homogeneous configurations. For now, we only provision *m1.small* instances, as our workloads are quite small, but the instance type can be changed easily. Provisioning VMs of different capacity could eventually lead to better granularity and lower operating costs. Support for more providers and heterogeneous configurations is planned for the future.

### 3.3.5 Entertainment Server

The entertainment server acts as a default service, which is used whenever a requested service is unavailable. It amounts to a polling session, notifying the user when the requested service is available and showing a waiting message or other distraction until then. Using server push technology or websockets, the entertainment server could be moved to the client instead.

### 3.3.6 Application Server

The application servers are dynamically provisioned VMs belonging to the underlying IaaS cloud, capable of concurrently running multiple applications inside an OSGi environment [27]. The prototype implementations of ARVUE [1, 9, 11] and CRAMP [8] use Apache Felix, which is a free implementation of the OSGi R4 Service Platform and other related technologies [35].

The OSGi specifications were originally intended for embedded devices, but have since outgrown their original purpose. They provide a dynamic component model, addressing a major shortcoming of Java. Figure 3.3 illustrates the OSGi architecture.

Each application server has a local controller, responsible for monitoring the state of said server. Metrics such as CPU load and memory usage of both the VM and of the individual deployed applications are collected and fed to the global controller for further processing. The global controller delegates application-tier tasks such as deployment and undeployment of bundles to the local controllers, which are responsible for notifying the OSGi environment of any actions to take.

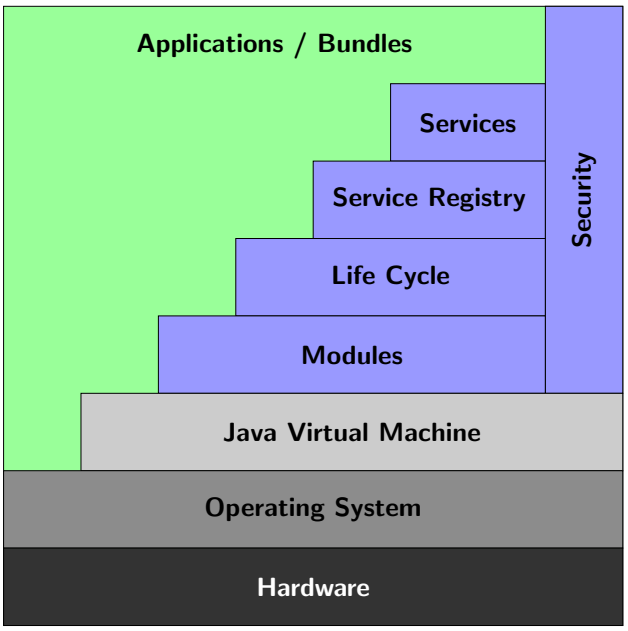



Figure 3.3: The OSGi platform.

The predictor from CRAMP [8] is also connected to each application server, making predictions based on the values obtained through the two-step prediction process. The prototype implementation computes an error estimate based on the NRMSE of predictions in the *past window* and uses that as a weighting parameter when determining how to blend the predicted and observed utilization of the monitored resources, as explained in Section 3.4.1.

### 3.3.7 Application Repository

Application bundles are contained in an application repository. When an application is deployed to a server, the server fetches the bundle from the repository. This implies that the repository is shared among application servers. A newly provisioned application server is assigned an application repository by the global controller. The applications are self-contained OSGi bundles, which allows for dynamic loading and unloading of bundles at the discretion of the local controller. The service-oriented nature of the OSGi platform suits this approach well.

A bundle is a collection of Java classes and resources together with a

Listing 3.1: Example manifest file with OSGi headers.

```
Bundle-Name: Hello World
Bundle-SymbolicName: org.arvue.helloworld
Bundle-Description: A Hello World bundle
Bundle-ManifestVersion: 2
Bundle-Version: 1.0.0
Bundle-Activator: org.arvue.helloworld.Activator
Export-Package: org.arvue.helloworld;version="1.0.0"
Import-Package: org.osgi.framework;version="1.3.0"
```

manifest file `MANIFEST.MF` augmented with OSGi headers. Listing 3.1 shows an example manifest file complete with headers.

## 3.4 Algorithms

The VM provisioning algorithms used by the global controller constitute a hybrid reactive-proactive PD-controller [8]. They implement proportional scaling augmented with derivative control in order to react to changes in the health of the system [9]. The server tier can be scaled independently of the application tier in a shared hosting environment. The VM provisioning algorithms are supplemented by a set of allocation policies. The prototype currently supports the following policies: *lowest memory utilization*, *lowest CPU load*, *least concurrent sessions*, and *newest server first*. In addition to this, we have also developed an admission control algorithm [7]. A summary of the concepts and notations used to describe the VM provisioning algorithms is available in Table 3.1. The additional concepts and notations for the admission control algorithm are provided in Table 3.2.

The input variables are average CPU load and memory usage. Average CPU load is the average Unix-like system load, which is based on the queue length of runnable processes, divided by the number of CPU cores present.

The VM provisioning algorithms have been designed to prevent oscillations in the size of the application server pool. There are several motivating factors behind this choice. Firstly, provisioning VMs takes substantial time. Combined with frequent scaling operations, this may lead to bad performance [39]. Secondly, usage based billing requires the time to be quantized at some resolution. For example, Amazon EC2 bases billing on full used hours. Therefore, it might not make sense to terminate a VM until it is close to a full billing hour, as it is impossible to pay for less than an entire hour. Thus, no scaling actions are taken until previous operations have been completed. This is why an underutilized server is terminated only after being consistently underutilized for at least $UC_T$ consecutive iterations.

Table 3.1: Summary of VM provisioning concepts and their notation

| | |
|---|---|
| $A(k)$ | set of web applications at time $k$ |
| $A_i(k)$ | set of inactive applications at time $k$ |
| $A_{li}(k)$ | set of long-term inactive applications at time $k$ |
| $A_{over}(k)$ | set of overloaded applications at time $k$ |
| $S(k)$ | set of servers at time $k$ |
| $S_{lu}(k)$ | set of long-term underutilized servers at time $k$ |
| $S_n(k)$ | set of new servers at time $k$ |
| $S_{over}(k)$ | set of overloaded servers at time $k$ |
| $S_{\neg over}(k)$ | set of non-overloaded servers at time $k$ |
| $S_t(k)$ | set of servers selected for termination at time $k$ |
| $S_u(k)$ | set of underutilized servers at time $k$ |
| $C(a,k)$ | measured CPU utilization of application $a$ at time $k$ |
| $C(s,k)$ | measured load average of server $s$ at time $k$ |
| $\hat{C}(s,k)$ | predicted load average of server $s$ at time $k$ |
| $C_w(s,k)$ | weighted load average of server $s$ at time $k$ |
| $dep\_apps(s,k)$ | applications deployed on server $s$ at time $k$ |
| $inactive\_c(a)$ | inactivity count of application $a$ |
| $M(a,k)$ | measured memory utilization of application $a$ at time $k$ |
| $M(s,k)$ | measured memory utilization of server $s$ at time $k$ |
| $\hat{M}(s,k)$ | predicted memory utilization of server $s$ at time $k$ |
| $M_w(s,k)$ | weighted memory utilization of server $s$ at time $k$ |
| $under\_u\_c(s)$ | underutilization count of server $s$ |
| $W(s,k)$ | weight of server $s$ at time $k$ for load balancing |
| $A_A$ | aggressiveness factor for additional capacity |
| $A_P$ | aggressiveness factor for VM provisioning |
| $A_T$ | aggressiveness factor for VM termination |
| $P_P(k)$ | proportional factor for VM provisioning |
| $D_P(k)$ | derivative factor for VM provisioning |
| $P_T(k)$ | proportional factor for VM termination |
| $D_T(k)$ | derivative factor for VM termination |
| $w_c$ | weighting coefficient for CPU load average |
| $w_m$ | weighting coefficient for memory usage |
| $w_p$ | weighting coefficient for VM provisioning |
| $w_t$ | weighting coefficient for VM termination |
| $C_{LA}$ | application CPU utilization lower threshold |
| $C_{LS}$ | server load average lower threshold |
| $C_{UA}$ | application CPU utilization upper threshold |
| $C_{US}$ | server load average upper threshold |
| $IC_{TA}$ | inactivity count threshold for an application |
| $IC_{TS}$ | inactivity count threshold for a server |
| $M_{LA}$ | application memory utilization lower threshold |
| $M_{LS}$ | server memory utilization lower threshold |
| $M_{UA}$ | application memory utilization upper threshold |
| $M_{US}$ | server memory utilization upper threshold |
| $W_{\text{MAX}}$ | maximum value of a server weight for load balancing |
| $N_A(k)$ | number of additional servers at time $k$ |
| $N_B$ | number of servers to use as base capacity |
| $N_P(k)$ | number of servers to provision at time $k$ |
| $N_T(k)$ | number of servers to terminate at time $k$ |

Table 3.2: Additional concepts and notation for admission control

| | |
|---|---|
| $se_a(k)$ | set of aborted sessions at time $k$ |
| $se_d(k)$ | set of deferred sessions at time $k$ |
| $se_n(k)$ | set of new session requests at time $k$ |
| $se_r(k)$ | set of rejected sessions at time $k$ |
| $S_{open}(k)$ | set of open application servers at time $k$ |
| $C(ent, k)$ | load average of the entertainment server at time $k$ |
| $M(ent, k)$ | memory utilization of the entertainment server at time $k$ |
| $w$ | weighting coefficient for admission control |

The memory usage metric $M(s, k)$ for a server $s$ at discrete time $k$ is given in (3.1). It is based on the amount of free memory $mem_{free}$, the size of the disk cache $mem_{cache}$, the buffers $mem_{buf}$, and the total memory size $mem_{total}$. The disk cache $mem_{cache}$ is excluded from the amount of used memory, as the underlying operating system is at liberty to use free memory for such purposes as it sees fit. It will automatically be reduced as the demand for memory increases. The goal is to keep $M(s, k)$ below the server memory utilization upper threshold $M_{US}$. Likewise, the memory usage metric for an application $a$ at discrete time $k$ is defined as $M(a, k)$, which is the amount of the memory used by the application deployment plus the memory used by the user sessions divided by the total memory size $mem_{total}$.

$$M(s,k) = \frac{mem_{total} - (mem_{free} + mem_{buf} + mem_{cache})}{mem_{total}} \tag{3.1}$$

The proposed approach maintains a fixed minimum number of application servers, known as the *base capacity* $N_B$. In addition, it also maintains a dynamically adjusted number of additional application servers $N_A(k)$, which is computed as in (3.2), where the aggressiveness factor $A_A \in [0, 1]$ restricts the additional capacity to a fraction of the total capacity, $S(k)$ is the set of servers at time $k$, and $S_{over}(k)$ is the set of overloaded servers at time $k$. This extra capacity is needed to account for various delays and errors, such as VM provisioning time and sampling frequency. For example, $A_A = 0.2$ restricts the maximum number of additional application servers to 20% of the total $|S(k)|$.

$$N_A(k) = \begin{cases} \lceil |S(k)| \cdot A_A \rceil, & \text{if } |S(k)| - |S_{over}(k)| = 0 \\ \left\lceil \frac{|S(k)|}{|S(k)| - |S_{over}(k)|} \cdot A_A \right\rceil, & \text{otherwise} \end{cases} \tag{3.2}$$

The number of VMs to provision $N_P(k)$ is determined by (3.3), where $w_p \in [0, 1]$ is a real number called the weighting coefficient for VM provision-

ing. It balances the influence of the proportional factor $P_P(k)$ relative to the derivative factor $D_P(k)$. For example, $w_p = 0.5$ would give equal weight to $P_P(k)$ and $D_P(k)$. A suitable value for this coefficient should be determined experimentally for a given workload. We have used $w_p = 0.5$ in all our experiments so far. The proportional factor $P_P(k)$ given by (3.4) uses a constant aggressiveness factor for VM provisioning $A_P \in [0, 1]$, which determines how many VMs to provision. The derivative factor $D_P(k)$ is defined by (3.5). It observes the change in the total number of overloaded servers between the previous and the current iteration.

$$N_P(k) = \lceil w_p \cdot P_P(k) + (1 - w_p) \cdot D_P(k) \rceil \tag{3.3}$$
$$P_P(k) = |S_{over}(k)| \cdot A_P \tag{3.4}$$
$$D_P(k) = |S_{over}(k)| - |S_{over}(k-1)| \tag{3.5}$$

The number of servers to terminate $N_T(k)$ is computed as in (3.6). It uses a weighting coefficient for VM termination $w_t \in [0, 1]$, similar to $w_p$ in (3.3). The currently required base capacity $N_B$ and additional capacity $N_A(k)$ have to be taken into account. The proportional factor for termination $P_T(k)$ is calculated as in (3.7). Here $A_T \in [0, 1]$, the aggressiveness factor for VM termination, works like $A_P$ in (3.4). Finally, the derivative factor for termination $D_T(k)$ is given by (3.8), which observes the change in the number of long-time underutilized servers between the previous and the current iteration.

$$N_T(k) = \lceil w_t \cdot P_T(k) + (1 - w_t) \cdot D_T(k) \rceil - N_B - N_A(k) \tag{3.6}$$
$$P_T(k) = |S_{lu}(k)| \cdot A_T \tag{3.7}$$
$$D_T(k) = |S_{lu}(k)| - |S_{lu}(k-1)| \tag{3.8}$$

### 3.4.1 Load Prediction

Prediction is performed with a two-step method [4, 5] based on EMA, which filters the monitored resource trends, producing a smoother curve. EMA is the weighted mean of the $n$ samples in the past window, where the weights decrease exponentially. Figure 3.4 illustrates an EMA over a past window of size $n = 20$, where less weight is given to old samples when computing the mean in each measure.

As we use a hybrid reactive-proactive VM provisioning algorithm, there is a need to blend the measured and predicted values. This is done through linear interpolation [7] with the weights $w_c$ and $w_m$ [8], the former for CPU

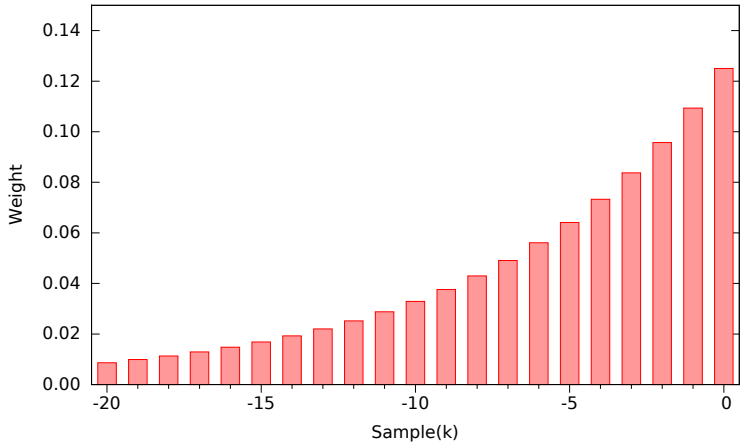



Figure 3.4: Example of EMA over a past window of size $n = 20$, where less weight is given to old samples when computing the mean in each measure.

load average and the latter for memory usage. In the current implementation, each of these weights is set to the NRMSE of the predictions so that lower prediction error will favor predicted values over observed values. The NRMSE calculation is given by (3.9), where $y_i$ is the latest measured utilization, $\hat{y}_i$ is the latest predicted utilization, $n$ is the number of observations, and $max$ is the maximum value of both measured and observed utilizations formed over the current interval, while $min$ is analogous to $max$. More details of our load prediction approach are provided in [7, 8].

$$NRMSE = \frac{\sqrt{\frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2}}{max - min} \tag{3.9}$$

### 3.4.2 The Server Tier

The server tier consists of the application servers, which can be dynamically added to or removed from the cluster. The VM provisioning algorithm for the application server tier is presented in Algorithm 3.2. At each sampling interval $k$, the global controller retrieves the performance metrics from each of the local controllers, evaluates them and decides whether or not to take an action. The set of application servers is partitioned into disjoint subsets according to the current state of each server. The possible server states are: *overloaded*, *non-overloaded*, *underutilized*, and *long-term underutilized*.

**Algorithm 3.2.** Proactive VM provisioning for the application server tier

1: **while true do**
2: $\forall s \in S(k) | C_w(s,k) := w_c \cdot C(s,k) + (1 - w_c) \cdot \hat{C}(s,k)$
3: $\forall s \in S(k) | M_w(s,k) := w_m \cdot M(s,k) + (1 - w_m) \cdot \hat{M}(s,k)$
4: $S_{over}(k) := \{\forall s \in S(k) | C_w(s,k) \geq C_{US}\} \cup \{\forall s \in S(k) | M_w(s,k) \geq M_{US}\}$
5: $A_{over}(k) := \bigcup_{s \in S_{over}(k)} d\_a(s,k)$
6: $S_{\neg over}(k) := S(k) \setminus S_{over}(k)$
7: **if** $|S_{over}(k)| \geq 1 \wedge |S_{\neg over}(k)| \geq 1$ **then**
8: **for** $a \in A_{over}(k)$ **do**
9: deploy application $a$ as per application-to-server allocation policy
10: **end for**
11: **end if**
12: **if** $|S_{over}(k)| \geq (|S(k)| - N_A(k)) \wedge N_P(k) \geq 1$ **then**
13: provision $N_P(k)$ VMs as a set of new servers $S_n(k)$
14: $S(k) := S(k) \cup S_n(k)$
15: Wait until servers $S_n(k)$ become operational
16: **for** $a \in A_{over}(k)$ **do**
17: deploy application $a$ on servers $S_n(k)$
18: **end for**
19: **end if**
20: $S_u(k) := \{\forall s \in S(k) | C_w(s,k) \leq C_{LS}\} \cap \{\forall s \in S(k) | M_w(s,k) \leq M_{LS}\}$
21: $S_{lu}(k) := \{\forall s \in S_u(k) | under\_u\_c(s) \geq IC_{TS}\}$
22: **if** $(|S_{lu}(k)| - N_B - N_A(k)) \geq 1 \wedge N_T(k) \geq 1$ **then**
23: sort servers $S_{lu}(k)$ with respect to server utilization metrics
24: select $N_T(k)$ servers from $S_{lu}(k)$ as servers selected for termination $S_t(k)$
25: migrate all applications and user sessions from servers $S_t(k)$
26: $S(k) := S(k) \setminus S_t(k)$
27: terminate VMs for servers $S_t(k)$
28: **end if**
29: **end while**

The algorithm starts by partitioning the set of application servers into a set of overloaded servers $S_{over}(k)$ and a set of non-overloaded servers $S_{\neg over}(k)$ according to the supplied threshold levels ($C_{US}$ and $M_{US}$) of the observed input variables: memory utilization and CPU load (lines 2–4). A server is overloaded if the utilization of any resource exceeds its upper threshold value. All other servers are considered to be non-overloaded (line 6). The applications running on overloaded servers are added to a set of overloaded applications $A_{over}(k)$ to be deployed on any available non-overloaded application servers as per the allocation policy for applications to servers (line 5). If the number of overloaded application servers exceeds the threshold level, a proportional amount of virtualized application servers is provisioned (line 13)

and the overloaded applications are deployed to the new servers as they become available (lines 16–18).

The server tier is scaled down by constructing a set of underutilized servers $S_u(k)$ (line 20) and a set of long-term underutilized servers $S_{lu}(k)$ (line 21), where servers are deemed idle if their utilization levels lie below the given lower thresholds ($C_{LS}$ and $M_{LS}$). Long-term underutilized servers are servers that have been consistently underutilized for more than a given number of iterations $IC_{TS}$. When the number of long-term underutilized servers exceeds the base capacity $N_B$ plus the additional capacity $N_A(k)$ (line 22), the remainder are terminated after their active sessions have been migrated to other servers (lines 23–27).

### 3.4.3 The Application Tier

Applications can be scaled to run on many servers according to their individual demand. Due to memory constraints, the naïve approach of always running all applications on all servers is unfeasible. Algorithm 3.3 shows how individual applications are scaled up and down according to their resource utilization. The set of applications is partitioned into disjoint subsets according to the current state of each application. The possible application states are: *overloaded*, *non-overloaded*, *inactive* and *long-term inactive*.

**Algorithm 3.3.** Reactive scaling of applications

1: **while true do**
2: $A_{over}(k) := \{s \in S(k), a \in dep\_apps(s,k) \mid C(a,k) > C_{UA}/|dep\_apps(s,k)| \vee M(a,k) > M_{UA}\}$
3: **if** $|A_{over}(k)| \geq 1$ **then**
4: **for all** $a \in A_{over}(k)$ **do**
5: deploy application $a$ as per application-to-server allocation policy
6: **end for**
7: **end if**
8: $A_i(k) := \{s \in S(k), a \in dep\_apps(s,k) \,|C(a,k) < C_{LA} \wedge M(a,k) < M_{LA}\}$
9: $A_{li}(k) := \{a \in A_i(k) \,|\, inactive\_c(a) \geq IC_{TA}\}$
10: **if** $|A_{li}(k)| \geq 1$ **then**
11: migrate all applications and user sessions for applications $A_{li}(k)$
12: $A(k) := A(k) \setminus A_{li}(k)$
13: **for all** $a \in A_{li}(k)$ **do**
14: unload application $a$
15: **end for**
16: **end if**
17: **end while**

An application is overloaded when it uses more resources than allotted (line 2). Each overloaded application $a \in A_{over}(k)$ is deployed to another server according to the allocation policy for applications to servers (lines 4–6). When an application has been running on a server without exceeding the lower utilization thresholds ($C_{LA}$ and $M_{LA}$), possible active sessions are migrated to another deployment of the application and then said application is undeployed (lines 8–15). This makes the memory available to other applications that might need it.

### 3.4.4 Admission Control

The admission control algorithm is given as Algorithm 3.4. It continuously checks for new $se_n(k)$ or deferred sessions $se_d(k)$ (line 1). If any are found (line 2), it updates the weighting coefficient $w \in [0, 1]$, representing the weight given to predicted and observed utilizations (line 3). If $w = 1.0$, no predictions are calculated (lines 5–6). The prediction process uses a two-step approach, providing filtered input data to the predictor [5]. We currently perform automatic adjustment and tuning in a similar fashion to Cherkasova and Phaal [13], where the weighting coefficient $w$ is defined according to (3.10). It is based on the following metrics: number of aborted sessions $|se_a(k)|$, number of deferred sessions $|se_d(k)|$, number of rejected sessions $|se_r(k)|$, and number of overloaded servers $|S_{over}(k)|$.

$$w = \begin{cases} 1, & \text{if } |se_a(k)| > 0 \vee |se_d(k)| > 0 \vee |se_r(k)| > 0 \\ 1, & \text{if } |S_{over}(k)| > 0 \\ max(0.1, w - 0.01), & \text{otherwise} \end{cases} \tag{3.10}$$

For each iteration, a bit more preference is given to the predicted values, up to the limit of 90 %. However, as soon as a problem is detected, full preference is given to the observed values, as the old predictions cannot be trusted. This should help in reducing lag when there are sudden changes in the load trends after long periods of good predictions.

If the algorithm finds servers in good condition (line 12), the session is admitted (lines 13–17), else the session is deferred to the entertainment server (line 20). Only if also the entertainment server is overloaded, will the session be rejected (line 22).

**Algorithm 3.4.** Admission control

**while true do**
  **if** $|se_n(k)| \geq 1 \vee |se_d(k)| \geq 1$ **then**
    update the weighting coefficient $w$ according to (3.10)
    **if** $w = 1$ **then**
      $\forall s \in S(k) | C_w(s,k) := C(s,k)$
      $\forall s \in S(k) | M_w(s,k) := M(s,k)$
    **else**
      $\forall s \in S(k) | C_w(s,k) := w \cdot C(s,k) + (1-w) \cdot \hat{C}(s,k)$
      $\forall s \in S(k) | M_w(s,k) := w \cdot M(s,k) + (1-w) \cdot \hat{M}(s,k)$
    **end if**
    $S_{open}(k) := \{\forall s \in S(k) | C_w(s,k) < LA_{UT} \wedge M_w(s,k) < MU_{UT}\}$
    **if** $|S_{open}(k)| \geq 1$ **then**
      **if** $|se_d(k)| \geq 1$ **then**
        pop first session in $se_d(k)$ and admit it on a server in $S_{open}(k)$
      **else**
        pop first session in $se_n(k)$ and admit it on a server in $S_{open}(k)$
      **end if**
    **else if** $|se_n(k)| \geq 1$ **then**
      **if** $C(ent,k) < LA_{UT} \wedge M(ent,k) < MU_{UT}$ **then**
        pop first session in $se_n(k)$ and defer it
      **else**
        pop first session in $se_n(k)$ and reject it
      **end if**
    **end if**
  **end if**
**end while**

## 3.5 Experimental Evaluation

To validate and evaluate the proposed VM provisioning and admission control approaches, we developed discrete-event simulations for ARVUE, CRAMP, and ACVAS and performed a series of experiments involving synthetic as well as realistic load patterns. The synthetic load pattern consists of two artificial load peaks, while the realistic load pattern is based on real world data. In this section, we present experimental results based on the discrete-event simulations.

### 3.5.1 VM Provisioning Experiments

This section presents some of the simulations and experiments that have been conducted to validate and evaluate ARVUE and CRAMP VM provisioning algorithms. The goal of these experiments was to test the two approaches and to compare their results.

In order to generate workload, a set of application users was needed. In our discrete-event simulations, we developed a load generator to emulate a given number of user sessions making HTTP requests on the web applications. We also constructed a set of 100 simulated web applications of varying resource needs, designed to require a given amount of work on the hosting server(s). When a new HTTP request arrived at an application, the application would execute a loop for a number of iterations, corresponding to the empirically derived time required to run the loop on an unburdened server. As the objective of the VM provisioning experiments was to compare the results of ARVUE and CRAMP, admission control was not used in these experiments.

#### Design and Setup

We performed two experiments with the proposed VM provisioning approaches: ARVUE and CRAMP. The first experiment used a synthetic load pattern, which was designed to scale up to 1000 concurrent sessions in two peaks with a period of no activity between them. In the second peak, the arrival rate was twice as high as in the first peak.

The second experiment was designed to simulate a load representing a workload trace from a real web-based system. The traces were derived from Squid proxy server access logs obtained from the IRCache project [24]. As the access logs did not include session information, we defined a session as a series of requests from the same originating Internet Protocol (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.

In a real-world application, there would be different kinds of requests available, requiring different amounts of CPU time. Take the simple case of a web shop: there might be one class of requests for adding items to the shopping basket, requiring little CPU time, and another class of requests requiring more CPU time, like computing the sum total of the items in the shopping basket. Users of an application would make a number of varying requests through their interactions with the application. After each request, there would be a delay while the user was processing the newly retrieved

information, like when presented with a new resource. In both experiments, each user was initially assigned a random application and a session duration of 15 minutes. Application 1 to 10 were assigned to 50 % of all users, application 11 to 20 were used by 25 %, application 21 to 30 received 20 % of all users, while the remaining 5 % was shared among the other 70 applications. Each user made requests to its assigned application, none of which was to require more than 10 ms of CPU time on an idle server. In order to emulate the time needed for a human to process the information obtained in response to a request, the simulated users waited up to 20 s between requests. All random variables were uniformly distributed. This means they do not fit the Markovian model.

The sampling period was $k = 10$ s. The upper threshold for server load average $C_{US}$ and the upper threshold for server memory utilization $M_{US}$ were both set to 0.8. These values are considered reasonable for efficient server utilization [25, 2].

The application-server allocation policy used was *lowest load average*. The session-server allocation policy was also set to *lowest load average*, realized through the *weighted round-robin* policy of HAProxy, where the weights were assigned by the global controller according to the load averages of the servers, as described in Section 3.3.1.

### Results and Analysis

The results from the VM provisioning experiment with the synthetic load pattern are shown in Figures 3.5a and 3.5b. The depicted observed parameters are: number of servers, average response time, average server CPU load, average memory utilization, and applications per server. The upper half of Table 3.3 contains a summary of the results.

The results from the two approaches are compared based on the following criteria: *number of servers used*, *average CPU load average*, *maximum CPU load average*, *average memory utilization*, *maximum memory utilization*, *average RTT*, and *maximum RTT*. The resource utilizations are ranked according to the utilization error, where over-utilization is considered infinitely bad.

In Figures 3.5a and 3.5b, the number of servers plots show that the number of application servers varied in accordance with the number of simultaneous user sessions. In this experiment, ARVUE used a maximum of 16 servers, whereas CRAMP used no more than 14 servers. The RTT remained quite stable around 20 ms, as expected. The server CPU load average and the memory utilization never exceeded 1.0.

The results from the experiment with the synthetic load pattern indicate

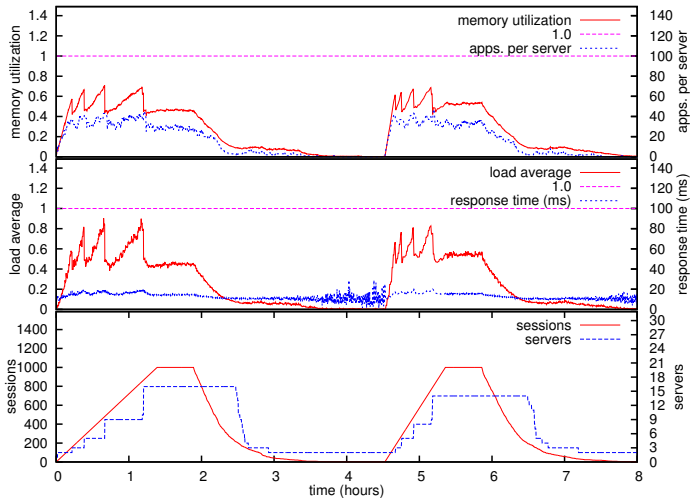
that the system is working as intended. The use of additional capacity seems to alleviate the problem of servers becoming overloaded due to long reaction times. The conservative VM termination policy of the proposed approach explains why the decrease in the number of servers occurs later than the decrease in the number of sessions. As mentioned in Section 3.4, one of the objectives of the proposed VM provisioning algorithms is to prevent oscillations in the number of application servers used. The results indicate that this was achieved.

Figures 3.6a and 3.6b present the results of the VM provisioning experiment with the realistic load pattern. The results are also presented in the lower half of Table 3.3.

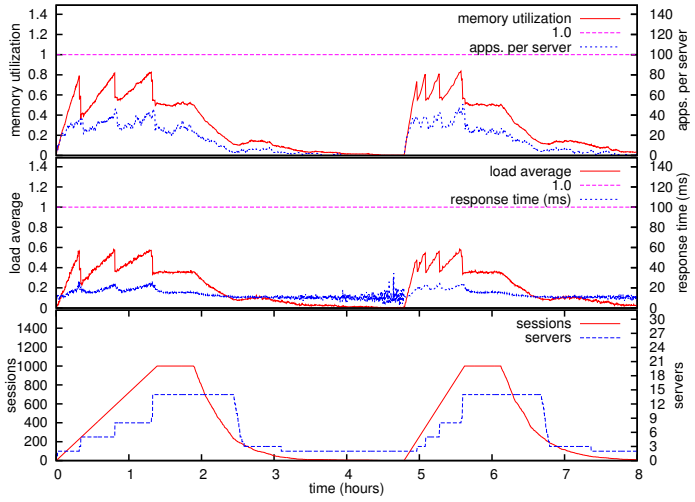
In this experiment, ARVUE used a maximum of 16 servers, whereas CRAMP used no more than 8 servers. In the case of ARVUE, the maximum response time was 21.3 ms and the average response time was 12.63 ms. In contrast, CRAMP had a maximum response time of 27.43 ms and an average response time of 14.7 ms. For both ARVUE and CRAMP, the server CPU load average and the memory utilization never exceeded 1.0.

The results from the experiment with the realistic load pattern show significantly better performance of CRAMP compared to ARVUE in terms of number of servers. CRAMP used half as many servers as ARVUE, but it still provided similar results in terms of average response time, CPU load average, and memory utilization. The ability to make predictions of future trends is a significant advantage, even if the predictions may not be fully accurate. Still, there were significant problems with servers becoming overloaded due to the provisioning delay. Increasing the safety margins further by lowering the upper resource utilization threshold values or increasing the extra capacity buffer further might not be economically viable. We suspect that an appropriate admission control strategy will be able to prevent the servers from becoming overloaded in an economically viable fashion.

Figure 3.7a shows the utilization error in the first experiment that uses the synthetic load pattern. For brevity, we only depict the CPU load in the error analysis. Therefore, error is defined as the absolute difference between the target CPU load average level $C_{US}$ and the measured value of the CPU load average $C(s,k)$ averaged over all servers in the system. Initially, the servers are naturally underloaded due to the lack of work. Thereafter, as soon as the first peak of load arrives, the error shrinks significantly and becomes as low as 0.1 for ARVUE and 0.3 for CRAMP. The higher CPU load error for CRAMP at this point was due to the fact that CRAMP results in this experiment were mostly memory-driven, as can be seen in Figure 3.5b. In other words, CRAMP had higher error with respect to the CPU load, but it had lower error with respect to the memory utilization. The error grows again as the

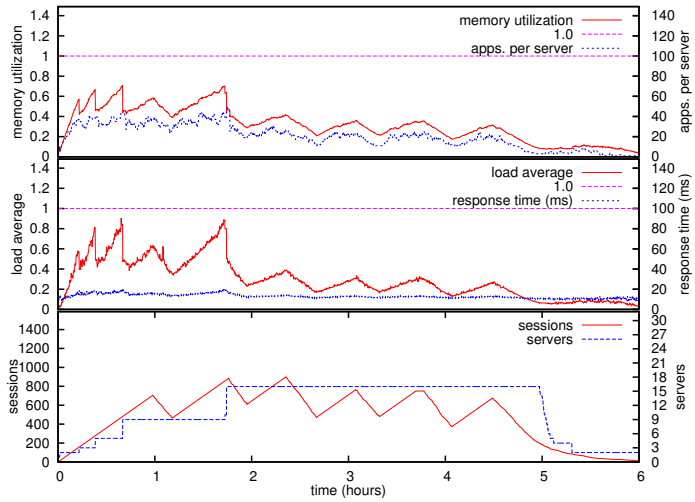



(a) Results of ARVUE with synthetic load.

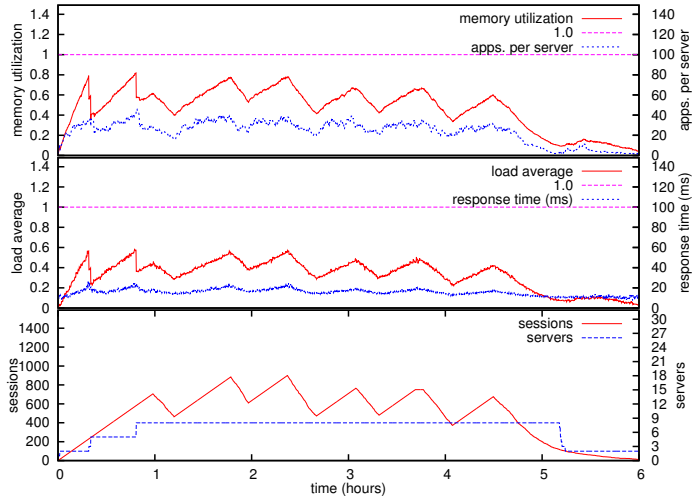



(b) Results of CRAMP with synthetic load.

Figure 3.5: Results of VM provisioning experiment with the synthetic load pattern. In this experiment, both ARVUE and CRAMP had similar results, except that CRAMP used fewer servers.




(a) Results of ARVUE with realistic load.




(b) Results of CRAMP with realistic load.

Figure 3.6: Results of VM provisioning experiment with the realistic load pattern. In this experiment, CRAMP used half as many servers as ARVUE, but it still provided similar performance.

Table 3.3: Results from VM provisioning experiments. The upper half of the table contains results from the first experiment with the synthetic load pattern, while the lower half contains results from the second experiment with the realistic load pattern. Entries in bold are better according to the evaluation criteria.

| approach | servers | $load_{avg.}$ | $load_{max}$ | $mem_{avg.}$ | $mem_{max}$ | $RTT_{avg.}$ | $RTT_{max}$ |
|---|---|---|---|---|---|---|---|
| $ARVUE_{synth}$ | 16 | **0.21** | 0.9 | 0.21 | **0.71** | **12.23** ms | **32.88** ms |
| $CRAMP_{synth}$ | **14** | 0.17 | **0.58** | **0.25** | 0.84 | 12.97 ms | 34.72 ms |
| $ARVUE_{real}$ | 16 | 0.25 | 0.9 | 0.27 | **0.71** | **12.63** ms | **21.3** ms |
| $CRAMP_{real}$ | **8** | **0.28** | **0.58** | **0.4** | 0.82 | 14.7 ms | 27.43 ms |

period of no activity starts after the first peak of load. In the second peak, both ARVUE and CRAMP showed similar results, where the error becomes as low as 0.25. Finally, as the request rate sinks after the second peak of load, the error grows further due to underutilization. This can be attributed to the intentionally cautious policy for scaling down, which is explained in Section 3.4 and ultimately to the lack of work. A more aggressive policy for scaling down might work without introducing oscillating behavior, but when using a third-party IaaS it would still not make sense to terminate a VM until the current billing interval is coming to an end, as that resource constitutes a sunk cost.

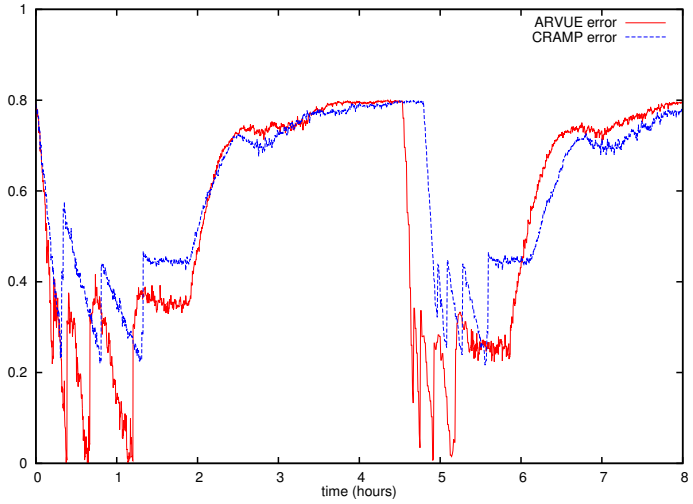
Error analysis of the second experiment that uses the realistic load pattern can be seen in Figure 3.7b. CRAMP appears to have lower error than ARVUE throughout most of the experiment, with the only exceptions being due to underutilization.

### 3.5.2 Admission Control Experiments

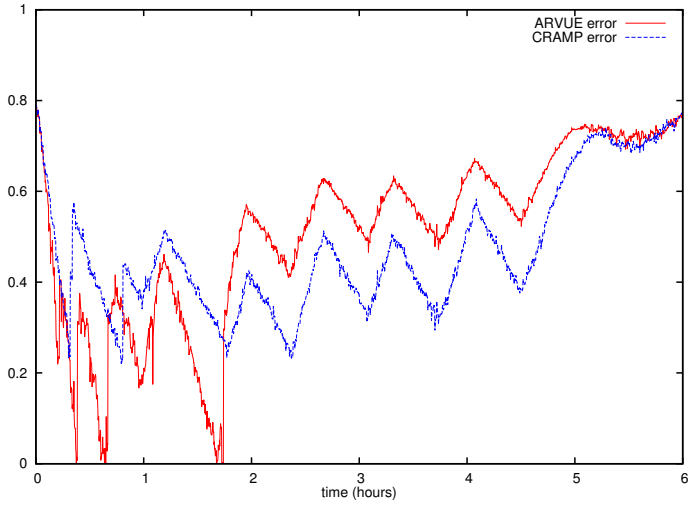
This section presents experiments with admission control. The goal of these experiments was to test our proposed admission control approach ACVAS [7] and to compare it against an existing SBAC implementation [13], here referred to as the *alternative approach*. As in the VM provisioning experiments, the experiments in this section also used 100 simulated web applications of various resource requirements. The experiments were conducted through discrete-event simulations.

#### Design and Setup

We performed two experiments with ACVAS and the *alternative approach*. The first admission control experiment used the synthetic load pattern, which




(a) Synthetic load pattern experiment: error analysis.




(b) Realistic load pattern experiment: error analysis.

Figure 3.7: CPU load average error analysis in the VM provisioning experiments. In the first experiment, CRAMP appears to have higher error because its results were mostly memory-driven. In the second experiment, CRAMP had lower error than ARVUE, with the only exceptions being due to underutilization.

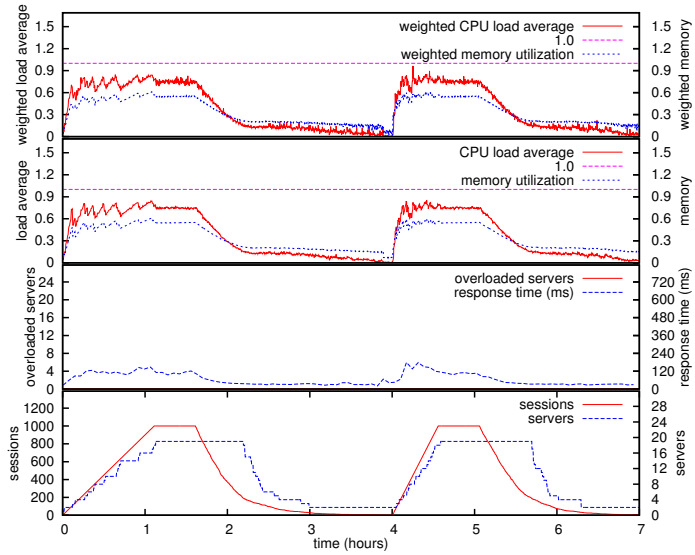
was also used in the first VM provisioning experiment described in Section 3.5.1. This workload was designed to scale up to 1000 concurrent sessions in two peaks with a period of no activity between them. Similarly, the second admission control experiment was designed to use the realistic load pattern, which was also used in the second VM provisioning experiment in Section 3.5.1. The sampling period $k$, the upper threshold for server load average $C_{US}$, the upper threshold for server memory utilization $M_{US}$, the application-server allocation policy, and the session-server allocation policy were all same as in the VM provisioning experiments in Section 3.5.1.

**Results and Analysis**

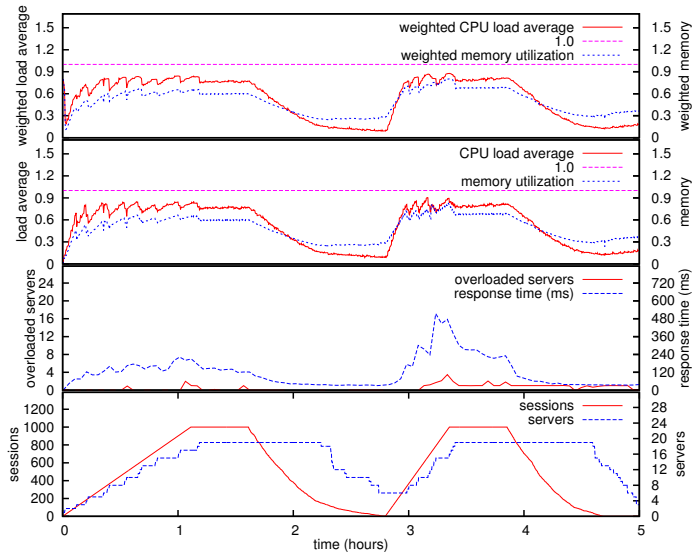
In our previous work [7], we proposed a way of measuring the quality of an admission control mechanism based on the trade-off between the number of servers used and six important QoS metrics: *zero overloaded servers*, *maximum achievable session throughput*, *zero aborted sessions*, *minimum deferred sessions*, *zero rejected sessions* and *minimum average response time* for all admitted sessions. The results from the two approaches will be compared based on these criteria.

Figures 3.8a and 3.8b present the results from the experiment with the synthetic load pattern. A summary of the results is also available in the upper half of Table 3.4. The prediction accuracy was high, the Root Mean Square Error (RMSE) of the predicted CPU and memory utilization was 0.0163 and 0.0128 respectively. ACVAS used a maximum of 19 servers with 0 overloaded servers, 0 aborted sessions, 30 deferred sessions, and 0 rejected sessions. There were a total of 8620 completed sessions with an average RTT of 59 ms. Thus, ACVAS provided a good trade-off between the number of servers and the QoS requirements. The *alternative approach* also used a maximum of 19 servers, but with several occurrences of server overloading. On average, there were 0.56 overloaded servers at all time with 0 aborted sessions and 488 rejected sessions. A total of 9296 sessions were completed with an average RTT of 112 ms. Thus, in the first experiment, the *alternative approach* completed 9296 sessions compared to 8620 sessions by ACVAS, but with 488 rejected sessions and several occurrences of server overloading.

Figures 3.9a and 3.9b show the results of the experiment with the realistic load trace derived from access logs. The lower half of Table 3.4 shows that 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 RTT of 59 ms. In contrast, the *alternative approach* used a maximum of 17 servers with 3 occurrences of server overloading. On average, there were 0.0046 overloaded servers at all




(a) Results of ACVAS with synthetic load.




(b) Results of *alternative approach* with synthetic load.

Figure 3.8: Results of admission control experiment with the synthetic load pattern. ACVAS performed better than the *alternative approach* in all aspects but session deferment and throughput.

Table 3.4: Results from admission control experiments. The upper half of the table contains results from the first experiment with the synthetic load pattern, while the lower half contains results from the second experiment with the realistic load pattern. Entries in bold are better according to the evaluation criteria.

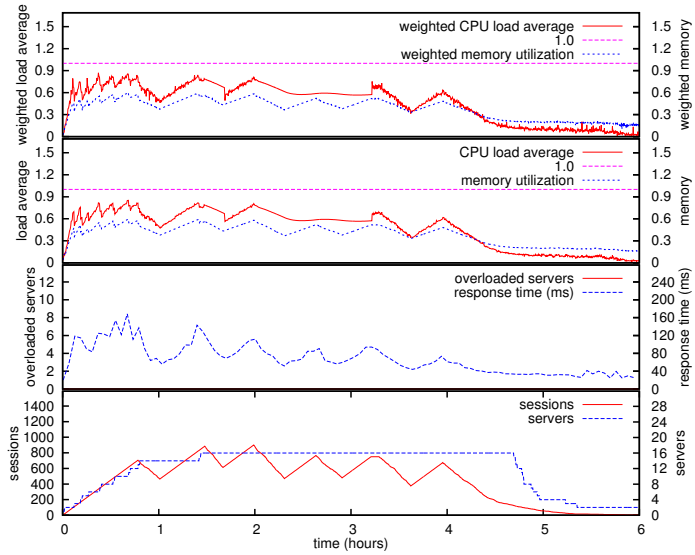
| approach | servers | overl. | abort. | def. | rej. | compl. | $RTT_{avg.}$ |
|---|---|---|---|---|---|---|---|
| $ACVAS_{synth}$ | 19 | **0** | 0 | 30 | **0** | 8620 | **59** ms |
| $alternative_{synth}$ | 19 | 0.56 | 0 | **N/A** | 488 | **9296** | 112 ms |
| $ACVAS_{real}$ | **16** | **0** | 0 | 20 | **0** | 8559 | **59** ms |
| $alternative_{real}$ | 17 | 0.0046 | 0 | **N/A** | 55 | **8577** | 72 ms |

time with 0 aborted sessions and 55 rejected sessions. There were a total of 8577 completed sessions with an average RTT of 72 ms. Thus, the *alternative approach* used an almost equal number of servers, but it did not prevent them from becoming overloaded. Moreover, it completed 8577 sessions compared to 8559 sessions by ACVAS, but with 55 rejected sessions and 3 occurrences of server overloading.

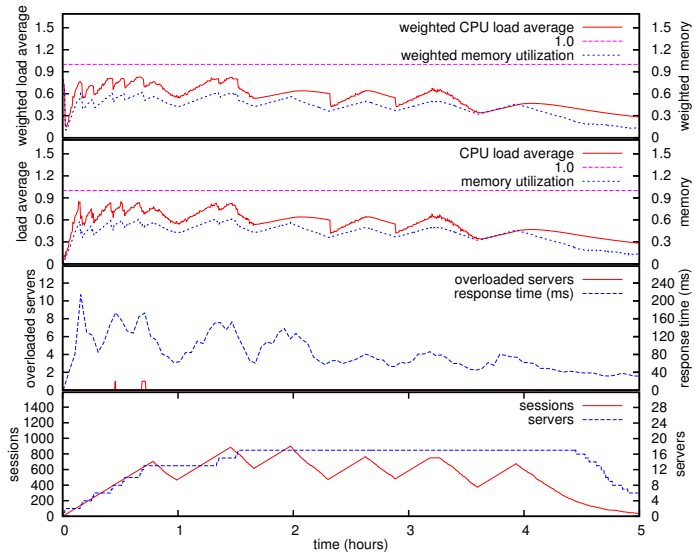
The results from these two experiments indicate that the ACVAS approach provides significantly better results in terms of the previously mentioned QoS metrics. In the first experiment, ACVAS had the best results in three areas: *overloaded servers*, *rejected sessions*, and *average RTT*. The *alternative approach* performed better in two areas: there were no *deferred sessions*, as it did not support session deferment, and it had more *completed sessions*. In the second experiment, ACVAS performed better in four aspects: *number of servers used*, *overloaded servers*, *rejected sessions*, and *average RTT*. The *alternative approach* again showed better performance in the number of *completed sessions* and in the number of *deferred sessions*. We can therefore conclude that ACVAS performed better than the *alternative approach* in both experiments.

The EMA-based predictor appears to be doing a good job on predicting these types of loads. It remains unclear how the system reacts to sudden drops in a previously increasing load trend. Such a scenario could temporarily lead to high preference for predicted results, which are no longer valid.

A plot of the utilization error with the synthetic load pattern can be seen in Figure 3.10a. Likewise, a plot of the utilization error with the realistic load can be seen in Figure 3.10b. Again, we only depict the CPU load, as it played the most significant part. The periods where ACVAS appears to have higher error than the *alternative approach* are due to underutilization amplified by ACVAS being more effective at keeping the average utilization




(a) Results of ACVAS with realistic load.




(b) Results of *alternative approach* with realistic load.

Figure 3.9: Results of admission control experiment with the realistic load pattern. ACVAS performed better than the *alternative approach* in all aspects but session deferment and throughput.

down, as no servers became overloaded during this time. Overall, the results are quite similar, as they should be, the only difference being the admission controller.

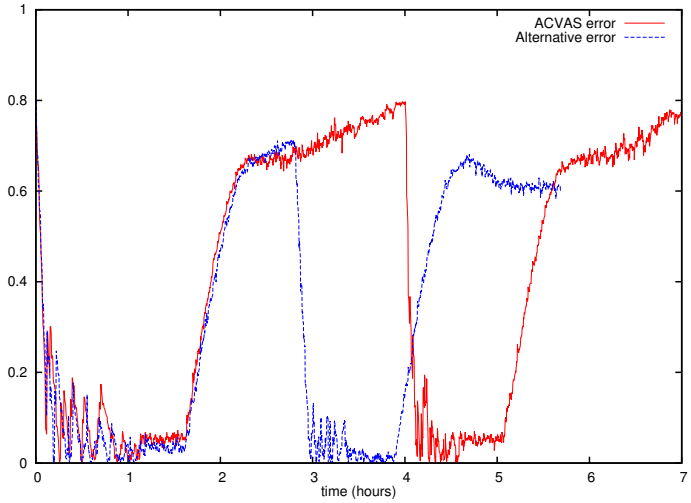
## 3.6 Conclusions

In this chapter, we presented a prediction-based, cost-efficient VM provisioning and admission control approach for multi-tier web applications. It provides automatic deployment and scaling of multiple simultaneous web applications on a given IaaS cloud in a shared hosting environment. The proposed approach comprises three sub-approaches: a reactive VM provisioning approach called ARVUE, a hybrid reactive-proactive VM provisioning approach called CRAMP, and a session-based adaptive admission control approach called ACVAS. Both ARVUE and CRAMP provide autonomous shared hosting of third-party Java Servlet applications on an IaaS cloud. However, CRAMP provides better responsiveness and results than the purely reactive scaling of ARVUE. ACVAS implements per-session admission, which reduces the risk of over-admission. Moreover, it implements a simple session deferment mechanism that reduces the number of rejected sessions while increasing session throughput. The proposed approach is demonstrated in discrete-event simulations and is evaluated in a series of experiments involving synthetic as well as realistic load patterns.

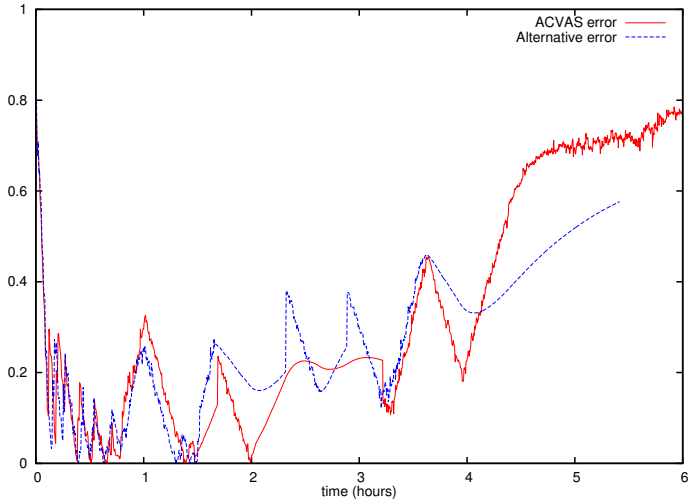
The results of the VM provisioning experiments showed that both ARVUE and CRAMP provide good performance in terms of average response time, CPU load average, and memory utilization. Moreover, CRAMP provides significantly better performance in terms of number of servers. It also had lower utilization error than ARVUE in most of the cases.

The evaluation and analyses concerning our proposed admission control approach compared ACVAS against an existing admission control approach available in the literature. The results indicated that ACVAS provides a good trade-off between the number of servers used and the QoS metrics. In comparison with the alternative admission control approach, ACVAS provided significant improvements in terms of server overload prevention, reduction of rejected sessions, and average response time.

Future work includes implementing and testing the admission controller on the prototype ARVUE PaaS [1, 11]. Furthermore, a case study of the final ARVUE PaaS could yield real data from an actual business case. We have been currently working on server consolidation approaches for web applications. Improved allocation through efficient consolidation should be possible. Moreover, applying metaheuristic approaches [10, 20] to optimize




(a) Synthetic load pattern experiment: error analysis.




(b) Realistic load pattern experiment: error analysis.

Figure 3.10: CPU load average error analysis in the admission control experiments. In the first experiment, both approaches had a similar error plot. However, in the second experiment, ACVAS appears to have lower error than the *alternative approach* throughout most of the experiment, with the only exceptions being due to underutilization.

cost-efficiency is also part of our ongoing research. However, optimal solutions can be seen as a form of the bin-packing problem and is therefore $NP$-complete [16].

## Acknowledgments

This work was supported by an Amazon Web Services research grant. Adnan Ashraf was partially supported by the Foundation of Nokia Corporation and by a doctoral scholarship from the Higher Education Commission (HEC) of Pakistan.

## References

[1] T. Aho et al. "Designing IDE as a Service". In: *Communications of Cloud Software* 1 (2011), pp. 1–10.

[2] J. Allspaw. *The Art of Capacity Planning: Scaling Web Resources*. O'Reilly Media, Inc., 2008. ISBN: 0596518579, 9780596518578.

[3] J. Almeida et al. "Joint admission control and resource allocation in virtualized servers". In: *J. Parallel Distrib. Comput.* 70.4 (Apr. 2010), pp. 344–362. ISSN: 0743-7315. DOI: 10.1016/j.jpdc.2009.08.009.

[4] 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*. valuetools '06. New York, NY, USA: ACM, 2006. ISBN: 1-59593-504-5. DOI: 10.1145/1190095.1190129.

[5] M. Andreolini, S. Casolari, and M. Colajanni. "Models and Framework for Supporting Runtime Decisions in Web-Based Systems". In: *ACM Transactions on the Web* 2.3 (2008), pp. 1–43. ISSN: 1559-1131. DOI: 10.1145/1377488.1377491.

[6] D. Ardagna et al. "Service Provisioning on the Cloud: Distributed Algorithms for Joint Capacity Allocation and Admission Control". In: *Towards a Service-Based Internet*. Ed. by E. Di Nitto and R. Yahyapour. Vol. 6481. Lecture Notes in Computer Science. Springer Berlin / Heidelberg, 2010, pp. 1–12.

[7] A. Ashraf, B. Byholm, and I. Porres. "A Session-Based Adaptive Admission Control Approach for Virtualized Application Servers". In: *The 5th IEEE/ACM International Conference on Utility and Cloud Computing*. Ed. by C. Varela and M. Parashar. IEEE Computer Society, 2012, pp. 65–72.

[8] A. Ashraf, B. Byholm, and I. Porres. "CRAMP: Cost-Efficient Resource Allocation for Multiple Web Applications with Proactive Scaling". In: *4th IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*. Ed. by T. W. Włodarczyk, C.-H. Hsu, and W.-C. Feng. IEEE Computer Society, 2012, pp. 581–586.

[9] A. Ashraf et al. "Feedback Control Algorithms to Deploy and Scale Multiple Web Applications per Virtual Machine". In: *38th Euromicro Conference on Software Engineering and Advanced Applications*. Ed. by V. Cortellessa, H. Muccini, and O. Demirors. IEEE Computer Society, 2012, pp. 431–438.

[10] C. Blum et al. "Hybrid metaheuristics in combinatorial optimization: A survey". In: *Applied Soft Computing* 11.6 (2011), pp. 4135 –4151. ISSN: 1568-4946. DOI: 10.1016/j.asoc.2011.02.032.

[11] B. Byholm. "An Autonomous Platform as a Service for Stateful Web Applications". MA thesis. Åbo Akademi University, 2013.

[12] X. Chen, H. Chen, and P. Mohapatra. "ACES: An efficient admission control scheme for QoS-aware web servers". In: *Computer Communications* 26.14 (2003), pp. 1581–1593. ISSN: 0140-3664. DOI: 10.1016/S0140-3664(02)00259-1.

[13] L. Cherkasova and P. Phaal. "Session-Based Admission Control: A Mechanism for Peak Load Management of Commercial Web Sites". In: *Computers, IEEE Transactions on* 51.6 (2002), pp. 669–685. ISSN: 0018-9340. DOI: 10.1109/TC.2002.1009151.

[14] T. C. Chieu et al. "Dynamic Scaling of Web Applications in a Virtualized Cloud Computing Environment". In: *e-Business Engineering, 2009. ICEBE '09. IEEE International Conference on*. 2009, pp. 281–286. DOI: 10.1109/ICEBE.2009.45.

[15] X. Dutreilh et al. "From Data Center Resource Allocation to Control Theory and Back". In: *Cloud Computing (CLOUD), 2010 IEEE 3rd International Conference on*. 2010, pp. 410–417. DOI: 10.1109/CLOUD.2010.55.

[16] M. R. Garey and D. S. Johnson. ""Strong" NP-Completeness Results: Motivation, Examples, and Implications". In: *Journal of the ACM* 25.3 (1978), pp. 499–508. ISSN: 0004-5411. DOI: 10.1145/322077.322090.

[17] Google. *App Engine*. https://developers.google.com/appengine.

[18] M. Grönroos. *Book of Vaadin*. fourth. Vaadin Ltd, 2011.

[19] R. Han et al. "Lightweight Resource Scaling for Cloud Applications". In: *Cluster Computing and the Grid, IEEE International Symposium on* (2012), pp. 644–651.

[20] M. Harman et al. "Cloud engineering is Search Based Software Engineering too". In: *Journal of Systems and Software* 86.9 (2013), pp. 2225 –2241. ISSN: 0164-1212. DOI: http://dx.doi.org/10.1016/j.jss.2012.10.027.

[21] Y. Hu et al. "Resource provisioning for cloud computing". In: *Proceedings of the 2009 Conference of the Center for Advanced Studies on Collaborative Research*. CASCON '09. New York, NY, USA: ACM, 2009, pp. 101–111.

[22] C.-J. Huang et al. "Admission control schemes for proportional differentiated services enabled internet servers using machine learning techniques". In: *Expert Systems with Applications* 31.3 (2006), pp. 458–471. ISSN: 0957-4174. DOI: 10.1016/j.eswa.2005.09.071.

[23] W. Iqbal et al. "Adaptive resource provisioning for read intensive multi-tier applications in the cloud". In: *Future Generation Computer Systems* 27.6 (2011), pp. 871–879. ISSN: 0167-739X.

[24] *IRCache Project Squid Logs*. http://www.ircache.net/.

[25] H. H. Liu. *Software Performance and Scalability: A Quantitative Approach*. Wiley Publishing, 2009. ISBN: 0470462531, 9780470462539.

[26] S. Muppala and X. Zhou. "Coordinated Session-Based Admission Control with Statistical Learning for Multi-Tier Internet Applications". In: *Journal of Network and Computer Applications* 34.1 (2011), pp. 20–29. ISSN: 1084-8045. DOI: 10.1016/j.jnca.2010.10.007.

[27] OSGi Alliance. *OSGi Service Platform Core Specification, Release 4, Version 4.2*. AQute Publishing, 2010.

[28] OSGi Alliance. *OSGi Service Platform Enterprise Specification, Release 4, Version 4.2*. AQute Publishing, 2010.

[29] W. Pan et al. “Feedback Control-Based QoS Guarantees in Web Application Servers”. In: *High Performance Computing and Communications, 2008. HPCC ’08. 10th IEEE International Conference on*. 2008, pp. 328–334. DOI: 10.1109/HPCC.2008.106.

[30] T. Patikirikorala et al. “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*. SEAMS ’11. New York, NY, USA: ACM, 2011, pp. 218–227. ISBN: 978-1-4503-0575-4.

[31] Y. Raivio et al. “Hybrid Cloud Architecture for Short Message Services”. In: *Proceedings of the 2nd International Conference on Cloud Computing and Services Science*. Ed. by F. Leymann et al. SciTePress, 2012, pp. 489–500.

[32] A. Robertsson et al. “Admission control for Web server systems - design and experimental evaluation”. In: *Decision and Control, 2004. CDC. 43rd IEEE Conference on*. Vol. 1. 2004, pp. 531–536. DOI: 10.1109/CDC.2004.1428685.

[33] Y. A. Shaaban and J. Hillston. “Cost-based admission control for Internet Commerce QoS enhancement”. In: *Electronic Commerce Research and Applications* 8.3 (2009), pp. 142–159. ISSN: 1567-4223. DOI: 10.1016/j.elerap.2008.11.007.

[34] W. Tarreau. *HAProxy*. http://haproxy.1wt.eu/.

[35] The Apache Software Foundation. *Apache Felix*. http://felix.apache.org/site/.

[36] B. Urgaonkar, P. Shenoy, and T. Roscoe. “Resource Overbooking and Application Profiling in a Shared Internet Hosting Platform”. In: *ACM Trans. Internet Technol.* 9.1 (Feb. 2009), pp. 1–45. ISSN: 1533-5399. DOI: 10.1145/1462159.1462160.

[37] W. Vogels. “Beyond Server Consolidation”. In: *Queue* 6.1 (Jan. 2008), pp. 20–26. ISSN: 1542-7730. DOI: 10.1145/1348583.1348590.

[38] 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. DOI: 10.1109/ISCC.2002.1021682.

[39] A. Wolke and G. Meixner. “TwoSpot: A Cloud Platform for Scaling out Web Applications Dynamically”. In: *Towards a Service-Based Internet*. Ed. by E. di Nitto and R. Yahyapour. Vol. 6481. Lecture Notes in Computer Science. Springer Berlin / Heidelberg, 2010, pp. 13–24. ISBN: 978-3-642-17693-7.