Robust Performance Control for Web Applications in the Cloud

Corina Stratan, Guillaume Pierre, Hector Fernandez, Eliana Tirsa and Valentin Cristea

Vrije Universiteit Amsterdam and University Politehnica of Bucharest

One of the major innovations provided by Cloud computing platforms is their pay-per-use model where applications can request and release resources at any time according totheir needs — and pay only for the resources they actually used. This business model is particularly favorable for application domains where workloads vary widely over time, such as the domain of Web applications.

However, provisioning the right quantities of resources for a Web application is not a simple task. Web applications are usually composed of multiple types of components such as load balancers, Web servers, application servers and database servers. Complex performance behavior of these components make it difficult to find the optimal resource allocation, even when the application workload is perfectly stable. The magnitude of the problem is further increased by the fact that Web application workloads are often very unstable and hard to predict. In the case of a sudden load increase there is a necessary tradeoff between reacting as early as possible to minimize the duration when the application underperforms because of insufficient processing capacity, and a slower approach to avoid situations where the load has already decreased when the new resources become available.

Faced with this difficult scientific challenge, the academic community has proposed a wide range of sophisticated resource provisioning algorithms [**?**, **?**, **?**]. However, we observe a wide discrepancy between these academic propositions and the very simple techniques actually used by the cloud industry where provisioning decisions are usually triggered by lower/upper thresholds on resource utilization. We postulate three possible reasons why sophisticated techniques have not been more widely deployed: *(i)* the gains of using sophisticated provisioning strategies are too low for anyone to bother; *(ii)* implementing these techniques is a difficult exercise, which is why real cloud systems rely on simpler techniques; and *(iii)* academic approaches mostly focus on unrealistic evaluations using simple applications and artificial workloads [4, 8, 19].

This paper aims to identify the real cause of why cloud providers use simpler provisioning techniques. We do this by implementing a provisioning system in realistic conditions, and reporting on *(i)* how hard implementation was; and *(ii)* potential gains from using the better technique as compared to a simple strawman. To achieve this, we designed and implemented a resource provisioning technique on ConPaaS, an open source platform-as-a-service environment for hosting cloud applications [14], based on a predictive and robust model. The resulting technique used different levels of thresholds to predict future performance degradations, workload trend detection methods to avoid flash crowds effects and dynamic load-balancing algorithms to handle the workload-mix. For the sake of comparison and discussion, we deployed the MediaWiki application and used real access traces to validate our technique, opening doors to real implementation of promising auto-scaling systems.

Section introduces the ConPaaS runtime environment. Section describes the MediaWiki application, its realistic benchmark, and the peculiarities of the Wikipedia workload-mix utilized to validate our system. Section focus on the different resource provisioning techniques implemented in ConPaaS. Section details the experimental campaign and its results. Section discusses related works. Section **??** draws a conclusion.

ConPaaS overview

ConPaaS is an open-source runtime environment for hosting applications in Cloud infrastructures [14]. Within the Cloud computing paradigm, ConPaaS belongs to the platform-as-a-service family, in which a variety of systems aim to simplify the deployment of applications in the Cloud. Using ConPaaS, developers can now focus their attention on application-specific concerns rather than making their applications suitable for the cloud.

Architecture

In ConPaaS, an application is designed as a composition of one or more elastic and distributed components, called *services*. Each service is dedicated to host a particular type of functionality of an application. At the moment, ConPaaS supports six different types of services: two web application hosting services respectively specialized for hosting PHP and JSP applications; a MySQL database service; a NoSQL database service built around the Scalarix key-value store; a MapReduce service; and a Task-Farming service for high-performance batch processing. Figure 1 shows ConPaaS hosting an application composed of a PhP service and a MySQL service, that could represents the architecture of any today's web application.

ConPaaS services are built based on an architecture composed of two main building blocks: agents and managers.

- **Agent**: A service is composed of one or several agents VMs which host the needed components to provide the service-specific functionality. Based on the performance requirements or the application workload, the number of agents VMs hosting these components can grow/shrink on demand. Thus, as an example, the PhP web hosting service initially utilizes one agent VM (containing one load-balancer, one web server and one PhP server), and progressively grows using multiple agents VMs, as illustrated by Figure 1.
- **Manager**: For each service, there is only one manager VM. The manager is in charge of executing all management requests, centralizing governance and performance monitoring data, and controlling the allocation of resources assigned to one service.

On the other hand, in ConPaaS, there are two types of traffic: application and management. The application traffic is based on requests from end users willing to access the application, so that is addressed to the agents hosting such application. On the other hand, the management traffic is directed from the service administrators to their service managers, and from those to the agents. Service administrators can manage their services using a graphical front-end or a command-line tool.

Hosting Elastic Applications

The main features that distinguish ConPaaS from other PaaS systems are its approach for autonomous application scaling and its interoperability with a wide variety of private and public IaaS clouds. In particular to provide such autonomous scaling capabilities, ConPaaS includes a monitoring data analysis mechanism and a resource provisioning system.

ConPaaS incorporates a scalable distributed monitoring engine which is based on the Ganglia [6] monitoring system. Ganglia consists of a server component (gmetad) that aggregates monitoring statistics from various VMs, and a reporting agent (gmond) which runs inside each VM. By default, Ganglia monitors only system-level metrics such as disk, CPU, memory and network usage. Unfortunately, these metrics often do not provide enough information about system performance due to the heterogeneity of the applications. As a consequence, in ConPaaS, we enhanced ganglia to also monitor service workloads by enhancing the reporting agent to track service-specific logs at runtime, and report statistics over a reporting period of 5 minutes. For instance, the PhP

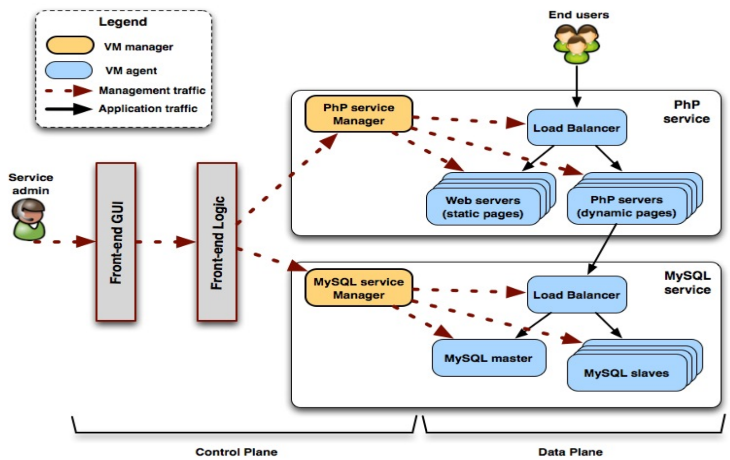

Figure 1: ConPaaS system architecture.

web hosting service includes new ganglia metrics that report statistics about the response time and request rate for static and dynamic user requests, respectively. Once the monitoring data is collected from the agents VMs, the resource provisioning algorithm decides whether to trigger scaling operations based on this data.

Unlike of implementing traditional trigger-based provisioning systems that scale services independently of whether they are part of an application. In ConPaaS, we designed a performance control model for multi-service applications. Using this model, the performance requirements are only imposed to the front-end service, while all the services collaborate in order to guarantee them. It improves the effectiveness and accuracy of the scaling decisions. Indeed, this allows to rapidly detect performance bottlenecks in applications, and thereby to minimize the resource consumption.

The Wikipedia application

To evaluate the behavior and accuracy of ConPaaS when hosting web applications, we prepared a realistic and complex enough scenario to assess any PaaS. In particular, we deployed the Wikipedia web application called MediaWiki [20], and used a web hosting benchmark called WikiBench [5].

The architecture of the Wikipedia website uses a http-proxy, a http-web server, a database and one or more PhP servers. To deploy the Wikipedia services on ConPaaS, we composed two different services: the PHP web hosting and MySQL service. In the MySQL service, we installed a complete copy of English Wikipedia database which contains about 30GB in Wikipedia articles. In the PhP service, an initial configuration was composed of one Nginx http-proxy, one Apache server, and one or more PhP (FastCGI Process Manager) servers. Each PhP server hosts the MediaWiki application which is the main component of this system.

In order to benchmark ConPaaS when hosting the Wikipedia services, we used the WikiBench research tool which generates realistic benchmarks with adaptable traffic properties. WikiBench has a number of advantages compared to the existing benchmark tools for web applications. First of all, Wikibench traces add a high degree of realism, since it is entirely based on the Wikipedia software and data. Indeed, the benchmark workloads are generated based on real access traces from the WikiMedia Foundation. These traces contain detailed traffic logs of requests made to Wikipedia by its users. Since the original Wikipedia traces can reach peaks of 50000 or 60000 reqs./secs, WikiBench uses the original 10% sample of these traces which can generate a workload up to about 5000 reqs./secs. As an example, in Figure 2, we show the workload of one trace, named "test", as the number of PhP requests per minute during approximately one day. In this paper, we focus on the behavior of PhP requests which makes particularly difficult to predict their execution times using auto-scaling sytems.

Even though we use a 10% of the real traces, they are very heterogeneous in terms of workload-mix, and thus explaining the irregular performance pattern followed by web applications. To illustrate this heterogeneity, in Figure 3, we present the distribution of the response time values for the PhP requests during the execution of the trace "test". Note that, static provisioning was utilized to execute this trace in order to get response values totally independent of any scaling decisions. A first observation shows an irregular dispersion of the response time values in two levels: (i) a long-term level on which the values vary along the trace execution without following any pattern; and (ii) a short-term level on which the response times widely diverge under short period of time such as one minute. There are two reasons for this dispersion: (i) PhP pages often require database queries; (ii) PhP pages need third-party static files. These issues avoids the utilization of provisioning techniques which scale applications only based on current response values.

Similarly, as shown in Figure 2 and Figure 3, there is not any correlation between the PhP request volumes and their response times. More precisely in Figure 3, the highest response time values obtained in the interval of time 600min. match up with a drop in the request rate during the same interval in Figure 2. Therefore, any provisioning technique that makes decisions based on the request rate can incur errors by under- or over-provisioning an application, which can reduces the efficiency of the scaling actions.

CORINA: ***Are you sure the results from Figures 3 and 4 are relevant? In figure 3, the response time might be higher for a lower request rate because there are fewer backend VMs provisioned when the request rate is low?***

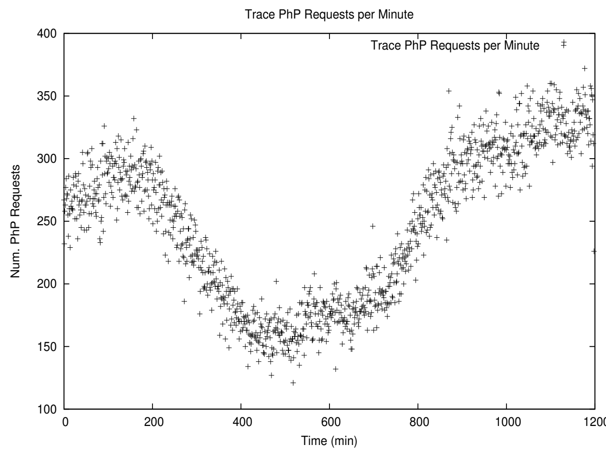

Figure 2: Wikipedia trace workload.

HECTOR: ***Note that, static provisioning was utilized to execute this trace in order to get response values totally independent of any scaling decisions. I executed this experiments using 4VMs for the backends so it was overprovisioned. I got the expected behavior different response time values independently of the req rate at each time.***

FIXME: OTHER POSSIBILITY: More precisely in Figure 4, the highest response time values obtained during the trace execution, match up with low levels of requests rate. Therefore, ...

In addition, other properties may be also considered important when using the Wikipedia traces:

- The intervarrival time between requests follows a Poisson distribution.
- The distribution of page popularity varies from very popular pages to those being accessed very infrequently.
- The mix of static/dynamic requests presents a strong variation.

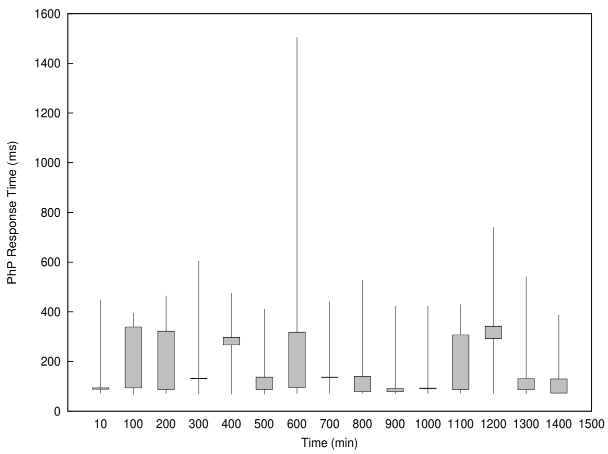

Figure 3: Complexity of PhP requests.

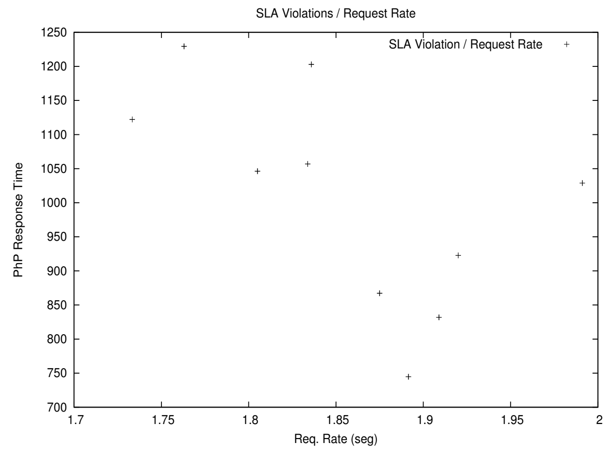

Figure 4: Response time vs request rate.

- The ratio of read/write operations vary having more reads than editions or creations of wiki pages.
- A considerable amount of requests for non-existing pages and files add realism to the traffic.

Since Wikipedia has a variable amount of data and visitors, it represents a valid example of elastic web application. In this paper, we focus in the scalability of the PhP web hosting service, and thereby as the number of PhP servers hosting MediaWiki scale out or back based on the demanding workload.

Resource provisioning algorithms

In this section, we describe the different provisioning techniques implemented in ConPaaS.

Trigger-based provisioning

CORINA: ***I edited the following paragraphs to reflect the fact that Amazon and other clouds don't really provide the provisioning algorithms, they just provide means to implement them (i.e. the cloud users need to write the provisioning rules themselves). Also, I saw that now EC2 also provides support for the users to have custom monitoring metrics.***

HECTOR: ***Yes, I agree I was thinking on rightscale ... good then.***

The existing cloud infrastructures provide means to implement provisioning mechanisms that adjust the amount of allocated resources based on a number of standard monitoring parameters. These are simple trigger-based systems that define threshold rules to increase or decrease the amount of computational resources when certain conditions are met. As an example, the Auto Scaling system offered by Amazon EC2 [2] can be used to define rules for scaling out or back an application based on monitoring parameters like CPU utilization, network traffic volume etc. This trigger-based technique is currently used in well-known cloud platforms such as RightScale [15] or OpenShift [13].

For the sake of comparison, we decided to design and implement a trigger-based provisioning mechanism in ConPaaS. This algorithm monitors CPU usage and application response time metrics, and dynamically adjusts the computational power allocated to an application by analyzing whether the monitoring data exceed their thresholds [1], as illustrated in Algorithm 1. Obviously, the lower and upper bound of each threshold are pre-defined by the user before execution.

Algorithm 1: Load-based

```
Data:
   System-level metrics (CPU, Resp. time)
     - Pre-defined metric threshold ranges, thr
Result: Scaling decisions
while auto-scaling is ON do
  Collect monitoring data of each metric, data;
  if no recent scaling operation then
    if avg(data_i) >= threshold_min_i then
      ADD resources;
    else if avg(data_i) < threshold_max_i then
      REMOVE resources;
    end
  end
  Sleep for 5 minutes ;
end
```

Even though these type of mechanisms are simple and widely used in cloud platforms, they are excessively reactive and not so precise when provisioning web applications due to several factors:

CORINA: ***I removed the item about services as black boxes, because now Amazon provides support for custom parameters; and also because we also implemented the trigger-based algorithm with custom parameters (request rate, response time) so our implementation doesn't see the service as a black-box either.***

HECTOR: ***Yes, we have to follow the same idea. I agree as well.***

- **Workload heterogeneity:** In some web applications, the system performance behavior fluctuates following an irregular pattern caused by the heterogeneity of the workload mix, which makes it difficult to predict the system's behaviour.
- **Reactiveness:** An excessively reactive algorithm can affect the system's stability, by causing frequent fluctuations in the number of allocated resources; this has negative effects on the performance, as well as increases the infrastructure cost.

[1]Every metric has a threshold range (max,min).

- **Resources heterogeneity:** The performance of virtual instances provided by current clouds is largely heterogeneous, even among instances of the same type, as shown in [3]. Simple trigger-based provisioning systems do not take this heterogeneity into account, thus providing less efficient resource allocation.

Based on these factors, we believe trigger-based provisioning mechanisms can be improved without drastically increasing their complexity. A possible solution is the utilization of techniques that handle workload and resource heterogeneity without being excessively reactive. Moreover, the implementation of these techniques should remain simple to facilitate their integration in existing auto-scaling systems. In the following we present two techniques that aims at solving the aforementioned drawbacks by relying on predictive and more accurate methods.

Feedback provisioning

Based on our previous knowledge from load-based provisioning, we designed and implemented an algorithm which improves the accuracy of our scaling actions when hosting web applications. To achieve that, our algorithm relies on three simple mechanisms: the definition of weights to each metric included in the performance requirements, the use of flexible thresholds and the estimation of the workload trend.

Weighted metrics: Traditional algorithms would scale out and back whenever a system-level metric exceeds its beforehand defined threshold range. Nevertheless, through the definition of weight values to application and system metrics, our algorithm takes into consideration its weight when making scaling decisions. More precisely, when hosting web applications, our algorithm associates weights in an ascending order to the following metrics: request rate, CPU usage and response time. Accordingly the response time has a higher weight than the request rate, since higher values in the response time rapidly indicate the existence of a performance degradation in a web application. As detailed in Section , scaling decisions only make based on the request rate can incur errors by under- or over-provisioning a web application, due to the large diversity in the complexity of the requests [16].

Algorithm 2: Feedback

```
Data:
    System and App-level metrics
      - Pre-defined metric threshold ranges, threshold
    Define the weight of each metric, w
Result: Scaling decisions
Create a queue to store historical workload, q;
Establish flexible thresholds from threshold ranges;
    - Scalings operations could be triggered, pred_thr;
    - Scalings operations must be triggered, reac_thr;
while auto-scaling is ON do
    Collect monitoring data of each metric, data;
    if avg(data_i) >= pred_thr_i then
        Increment chances of scaling_out using w_i, s_out;
    else if avg(data_i) < pred_thr_i then
        Increment chances of scaling_in using w_i, s_in;
    end
    Add to q the most recent workload value;
    Estimate historical workload trends (last ~30min), td;
        - If trend is increasing then td = 1;
        - If trend is decreasing then td = 0;
        - Undetermined td = -1;
    if no recent scaling operation then
        if avg(data_i) >= reac_thr_i and td = 1 and s_out > s_in then
            ADD resources;
        else if avg(data_i) < reac_thr_i and td = 0 and s_out < s_in then
            REMOVE resources;
        end
    end
    else
        Sleep during 5 minutes;
    end
end
```

Flexible thresholds: This algorithm uses two levels of threshold ranges for each metric: *predictive* and *reactive*. As shown on Figure 5, there are two "head rooms" between the SLO (Service Level Objective) threshold (performance requirements predefined by the user) and the flexible thresholds. First, the head-room H_1 is between the predictive threshold and the reactive thresholds, and is intended to alert of possible workload alterations in advance. Thereby, when the system performance exceeds the predictive ranges, there is an increment in the chances of scaling actions will be triggered to tackle future SLO violations. The second head-room H_2 comprises between the SLO and reactive thresholds is used to trigger scaling actions. Performance values exceeding the

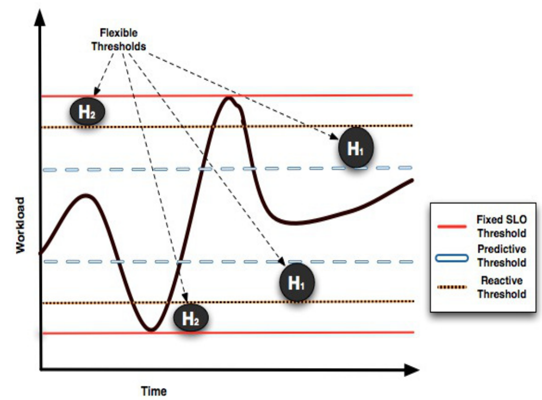

Figure 5: Flexible thresholds

reactive threshold launch scaling operations if other conditions are also satisfied. As an example of flexible thresholds for the CPU-usage, we established a predictive range comprised between 30% and 70% , and a reactive comprised between 20% and 80%.

Workload's trend estimation: Nowadays, there is a wide literature on mathematical models that try to predict future alterations in web application's workload. However, the workload mix and network traffic of web applications make more difficult to provide accurate predictions using these models. Besides, the complexity of these models sometimes prevent its integration into real auto scaling systems. To design a robust and simple provisioning system, we decided to use a feedback mechanism that analyzes the behavior of the system performance during an interval of time. An exhaustive analysis of the monitoring data during a small interval of time (approx. during the last 30min) provides enough information to detect the workload's trend, and thereby to classify the type of workload alteration as *constant* or *temporal*. Obviously, only *constant* variations may trigger scaling actions to avoid frequent fluctuations in the system performance caused by short and sudden *temporal* variations (flash crowds), as we will detail in Section . Note that, the workload trend estimation is not the trigger of this algorithm, but one more mechanism in it. So, it could be also calculated by using mathematical model such linear regression, kernel canonical correlation, and so on.

Anyway, the use of these three mechanisms must follow an order when making scaling decisions, as illustrated in Algorithm 2. Initially, the user has to specify the thresholds ranges and give a weight to each metric. Next the flexible thresholds are defined based on Amazon recommendations and statistically-chose performance measures [2]. Once the monitoring data is collected from the agents, the decision making process can start. Firstly, these data is analyzed to verify if it exceeds the predictive threshold ranges (denoted by p_thr), if so the probability of triggering scaling actions increases proportionally in function of the metric's weight. In order to keep track of workload variations, this algorithm stores in a queue (denoted by q) the most recent system performance values, and analyzed them to estimate the workload trend (denoted by td). Finally, to trigger any scaling action, a serie of conditions have to be satisfied: (i) no previous scaling actions have been taken over the last 15min; (ii) the recent monitoring data have to exceed the reactive threshold ranges; (iii) the workload trend has to follow a constant pattern (increasing/decreasing); and (iv) the result of predictive thresholds evaluation has to match up with that obtained in (ii).

Problematic of this algorithm Although the combination of these techniques improves the accuracy of our measurements, and avoids to present an excessive reactive behavior. The heterogeneous nature of the VM instances and the workload require more flexible provisioning algorithms, as we pointed out in [9]. .

Dynamic load balancing weights

Another problem that we need to consider is the heterogeneity of cloud platforms. Different virtual machines from the same cloud might have different performance characteristics, even when their specifications from the cloud vendor are the same. This problem can be addressed through various load balancing techniques, like assigning weights to the backend servers or taking into account the current number of connections that each server handles. Furthermore, the performance behavior of the virtual servers can also change in time, either due to changes in the application's usage patterns, or due to changes related to the hosting of the virtual servers (e.g., VM migration).

In order to address these issues in ConPaaS we implemented a weighted load balancing system in which the weights of the servers are periodically re-adjusted automatically, based on the monitoring data. As illustrated in Algorithm 3, this method assigns the same weight to each backend server at the beginning of the process. The weights are then periodically adjusted (every $\sim$ 15min) proportionally with the difference among the average response times of the servers during this time interval.

Evaluation

In this section we conducted our experiments on a heterogeneous infrastructure like Amazon EC2 [2], and on a homogeneous infrastructure like DAS-4 (the Distributed ASCI Supercomputer 4) [1]. DAS-4 is the Dutch Computational Infrastructure, a six-cluster wide-area distributed system designed with research purposes. In our experiment campaign, we compared the degree of SLO enforcement (performance requirements fulfillment) and resource consumption for each one of the provisioning algorithms included in ConPaaS.

Testbed configuration

As a realistic and representative scenario, we deployed MediaWiki application using ConPaaS on both infrastructures, and we ran the Wikibench tools utilizing Wikipedia workload traces. To provide the Wikipedia services, an initial configuration was composed of 4 VMs, and 1 VM to host the Wikibench tools. The 4 VMs include a PhP service manager VM, a PhP agent VM, a web server and a http-

[2] These performance values are obtained from previous executions of the same application using similar hardware configurations.

Algorithm 3: Dynamic load-balancing weights

```
Data:
   System and App-level metrics
     - Pre-defined metric threshold ranges, threshold
   Define the weight of each metric, w
Result: Scaling decisions
Create a queue to store historical workload, q;
Establish flexible thresholds from threshold ranges;
   - Scalings operations could be triggered, pred_thr;
   - Scalings operations must be triggered, reac_thr;
Initialize load-balancing weights for the backend servers;
while auto-scaling is ON do
   Collect monitoring data of each metric, data;
   if avg(data_i) >= pred_thr_i then
      Increment chances of scaling_out using w_i, s_out;
   else if avg(data_i) < pred_thr_i then
      Increment chances of scaling_in using w_i, s_in;
   end
   Add to q the most recent workload value;
   Estimate historical workload trends (last ~20min), td;
      - If trend is increasing then td = 1;
      - If trend is decreasing then td = 0;
   if no recent scaling operation then
      if avg(data_i) >= reac_thr_i and td = 1 and s_out >=
      s_in then
         ADD resources;
      else if avg(data_i) < reac_thr_i and td = 0 and s_out <
      s_in then
         REMOVE resources;
      end
   end
   else
      Sleep during 5 minutes;
   end
   Adjust load-balancing weights based on the workload
   (~15min);
end
```

proxy agent VM (both in the same VM), and finally a MySQL agent VM to store the English Wikipedia data, as explained in Section .

Our goal is to evaluate the behavior of the provisioning algorithms, when scaling out and back the number of VMs hosting PhP servers to guarantee several performance requirements, referred to as SLO. Specifically, we fixed two SLOs, one of 700 milliseconds at the service's side (denoted by a yellow Line on Figures 6,7,9,10 and 11), and another of 1500 milliseconds at the client's side (denoted by a red Line on Figures 6,7,9,10 and 11). Thus, our measurements shows the behavior of the MediaWiki application under a workload generated from real access traces during 24h. Note that, these experiments only focus on the average of PhP response times and the resource consumption obtained with our algorithms.

Accordingly, some assumptions were made:

- Response times from static requests were not analyzed due to the lightweight nature of the static files employed by Wikipedia articles.
- The algorithms collect the monitoring data through Ganglia over a reporting period of 5 minutes.
- Since DAS-4 is a homogeneous infrastructure, the dynamic load-balancing weights provisioning technique was only evaluated on a heterogeneous platform like Amazon EC2. The benefits behind of this algorithm can only be appreciated when running on a heterogeneous environment, that provides VMs with different hardware configurations.
- These experiments used the same statistically-chose performance threshold for both infrastructures. In the future, these threshold values might be defined based on the type of VM provisioned.
- A minimum interval of 15 minutes has been established between scaling actions. In

Homogeneous Infrastructure

Our experiments on DAS4 relies on OpenNebula as Infrastructure-as-a-service (IaaS) [**?**]. To deploy the Wikipedia services, we used small instances for the PhP service (manager and agents) and a medium instance for the MySQL service (agent). OpenNebula's small instances provision VMs equipped with 1 CPU of 2Ghz, and 1GiB of memory, while medium instances are equipped with 4 CPU's of 2Ghz, and 4GB of memory.

SLO enforcement Figure 6 and Figure 7 depict the degree of SLO fulfillment of the load-based and feedback algorithms, indicating the average of response times obtained during the execution of the Wikipedia workload trace. As illustrated on Figure 6,

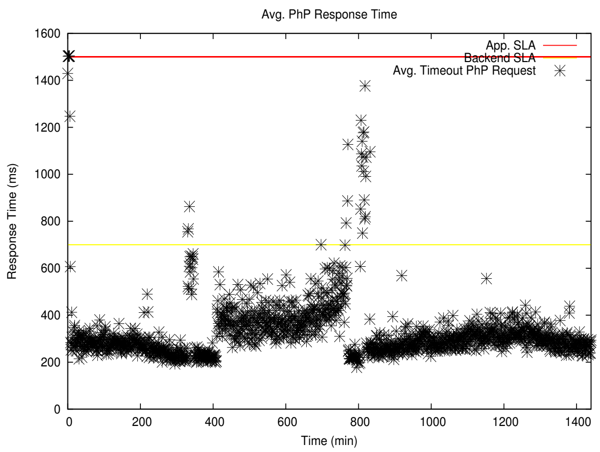

Figure 6: PhP resp. time on DAS4 – Load-based.

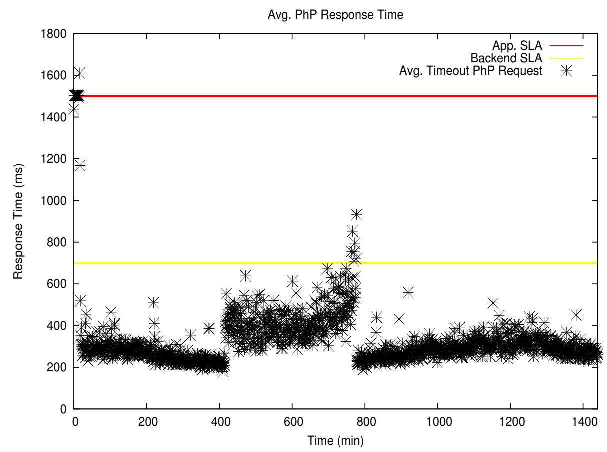

Figure 7: PhP resp. time on DAS4 – Feedback.

the results show how the load-based provisioning algorithm presents an important amount of SLO violations (between the yellow and red Lines), which are generated due to its excessive reactive behavior. As we mentioned, this algorithm is an easy target to flash crowds effects, as it tends to add or remove VMs to handle sharp and sudden variations in the workload, thus producing SLO violations. In contrast on Figure 7, the system performance (*i.e.,* response time) do not fluctuate greatly showing a more stable behavior during the whole experiment. As a result, there is a drop of the 31.72% in the amount of SLO violations in comparison with the load-based algorithm (when regarding the SLO's at the client side).

Resource consumption Nevertheless to better understand the behavior of both algorithms, we may also focus on the resource consumption illustrated on Figure 8. Firstly, the excessive reactive behavior of the load-based algorithm is again illustrated at the interval *t=350min* and *t=820min*, where two scaling operations under-provision the system during a short period of time. These provisioning decisions provoked fluctuations in the system performance that incremented the financial cost, as well as throughput alterations under the same intervals of time, as depicted on Figure 6. When using the feedback algorithm, the system makes provisioning decisions by analyzing workload's trend during a considerable interval of time. Scaling actions are only triggered when having constant alterations in the Wikipedia workload, thereby providing a more efficient resource usage. Indeed, the workload alterations depicted on Figure 2, match with the provisioning decisions made by the feedback algorithm on Figure 8.

Discussion Using the load-based provisioning algorithm, the system performance fluctuates greatly following a pattern similar to the web traffic, that increases the number of SLO violations. The reactive behavior of this algorithm triggers scaling actions that affect to the system performance instead of improving it, and as a consequence, it is also wasteful in terms of resource consumption. Unlike feedback algorithm offers an efficient resource usage and a constant performance behavior while meeting the application's SLO. Therefore this algorithm finds the trade-off between accuracy and cost savings.

Both algorithms are best-effort regarding the SLO fulfillment, and thereby temporal alterations of the workload (with a short duration of 5min approx.) cannot be handled. The heterogeneity of the PhP-served pages containing images and requiring multi-

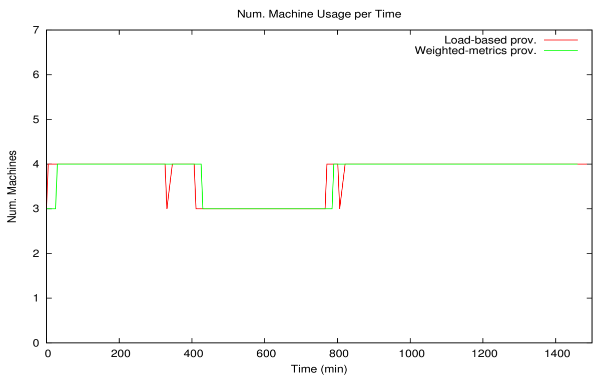

Figure 8: Resource consumption on DAS4.

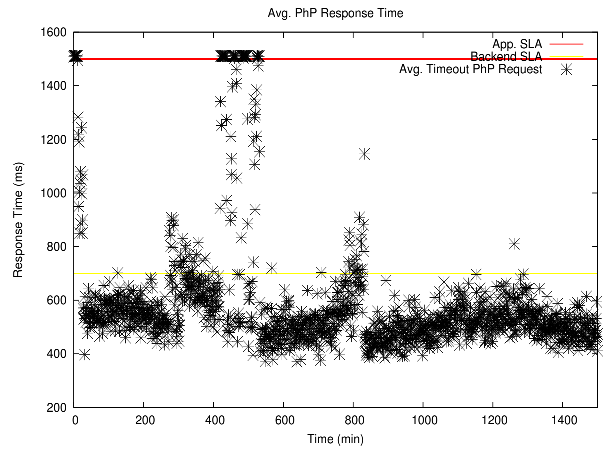

Figure 9: PhP response time on EC2 – Load-based.

ple Db queries, and the startup time of VMs, are in part responsible of these SLO violations.

Heterogeneous Infrastructure

Our experiments on EC2 used small instances for the PhP service (manager and agents) and a medium instance for the MySQL service (agent). EC2 small instances provision VMs equipped with 1 EC2 CPU, and 1.7GiB of memory, while medium instances are equipped with 2 EC2 CPU's, and 3.75GiB of memory.

SLO enforcement In the following, we analysis the behavior of our algorithms when making provisioning decisions on a heterogeneous infrastructure. Figure 9, Figure 10 and Figure 11 show the system performance of the load-based, feedback and dynamic load-balancing weights algorithms, respectively. As depicted on Figure 9, the performance fluctuates greatly following an irregular pattern when using the load-based algorithm. More precisely, two of the three workload peaks caused at *t=300min* and *t=820min*, are explained by looking at the variations on the Wikipedia workload described on Figure 2. However, there is a third peak between *t=400min* and *t=500min* that corresponds to the interval of time on which the workload trace experiences a significant drop in the request volumes. During this period of time, the workload-mix causes sudden and sharp performance variations that explains this behavior, as an effect associated to very frequent scaling actions. As a result, there is a degradation of the SLO fulfillment.

On the other hand, Figure 10 and Figure 11 show as the feedback and dynamic load-balancing weight algorithm behave similarly. Even though both algorithms are best-effort, there is an important reduction in the number of SLO violations during the trace execution. In particular, the feedback algorithm reduces the SLO violations in a 41.3%, while the dynamic load-balancing weights algorithm does it in a 47.6%. Like on DAS-4, the feedback algorithm follows a constant performance pattern without having sharp and sudden workload alterations. Besides, as shown on Figure 11, the dynamic load-balancing weights algorithm algorithm has a similar behavior to the feedback algorithm in terms of system performance, however. This algorithm improves the SLO enforcement in a 6.3% in comparison with the feedback algorithm at the client's side. Therefore we demonstrate how the use of workload-mix and flexible load-balancing techniques, although intrusive, do not cause time delays or excessive throughput alterations.

Resource consumption The resource usage on EC2 presents important alterations, as shown on Figure 12. When using the load-based provisioning,

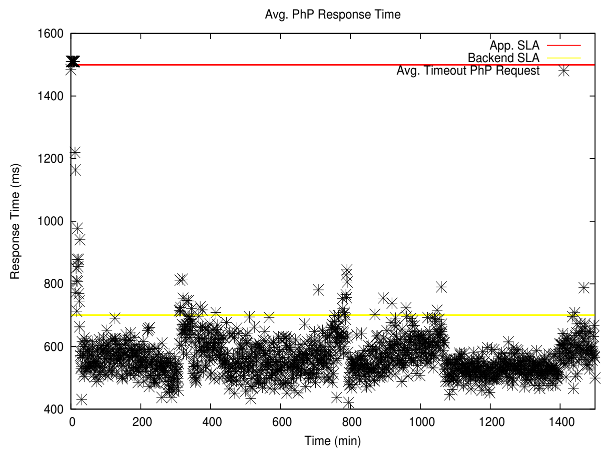

Figure 10: PhP resp. time on EC2– Feedback.

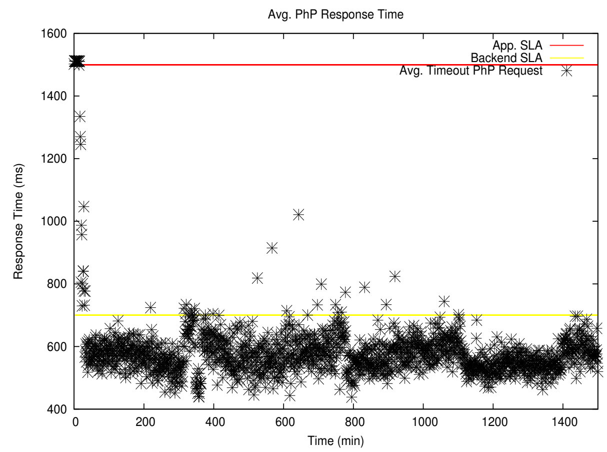

Figure 11: PhP resp. time on EC2– Load-balancing Weights.

the fluctuations in the system performance are explained as a result of a high frequency of scaling operations. In concrete, the fluctuations caused at the interval of time between *t=400min* and *t=500min* (see on Figure 9), match with the provisioning decisions made during the same interval of time on Figure 12. If we now pay attention to the feedback, and dynamic load-balancing weights algorithms, their resource consumptions are identical along the execution. Indeed, both algorithms decided to scale out the system during the interval of time comprised between *t=1050min* and *t=1400min*, to prevent future SLO violations that occurred when using the load-based algorithm. In particular, this situation demonstrates the benefits of using flexible threshold ranges to provide a predictive provisioning mechanism, thus improving the user experience.

Discussion The experiments on EC2 leads to several conclusions. The use of the load-based algorithm shows again how aggressive provisioning increases the resource consumption and the chances of degraded application performance due to frequents scaling actions. These actions are triggered as an effect associated to the workload-mix, when handling bursty workload conditions. On the other hand, the feedback and dynamic load-balancing algorithms constitute two robust provisioning models that offer an efficient resource consumption and keep stable the application performance during the trace execution.

Furthermore, the use of a dynamic load-balancing algorithm provided a more efficient distribution of the request-mix across servers, that reduced the SLO violations in a 6.3%. Hence, the main objective behind of this algorithm is to tackle the degradations caused by the workload mix.

Discussion

Generally, the result of our measurements show how the behavioral performance pattern and the resource consumption vary depending on the infrastructures on which we ran our experiments. Different hardware configurations such as those provided by DAS-4 and EC2, offer two distinct scenarios to validate our provisioning algorithms. In these experiments, we demonstrate how trigger-based provisioning mechanisms can affect the system performance instead of improving it, as well as are wasteful in terms of resource usage. Furthermore, we show how a dynamic load-balancing technique, although intrusive, can be included and used without producing performance alterations. In fact, this technique slightly reduced

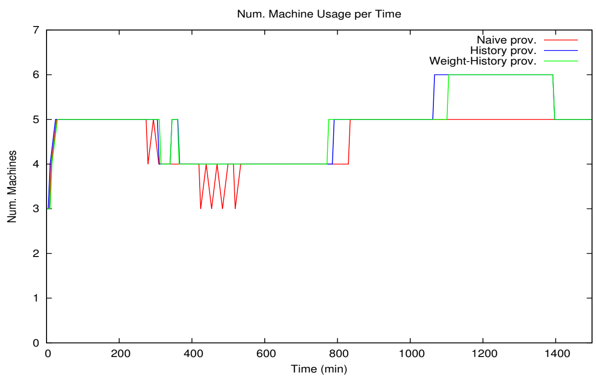

Figure 12: Resource consumption on EC2.

the number of SLO violations in comparison with the results obtained using feedback algorithm. Finally, we also present the benefits by using feedback and dynamic load-balancing weights provisioning algorithms which aims to find the trade-off between the accuracy and cost savings.

However, there is room for improvement using the dynamic load-balancing weights algorithm, as it cannot handle some pattern of workload alterations that caused several SLO violations during the execution.

Related studies

There is a wide literature on issues related to dynamic resource provisioning for cloud web applications. Different approaches present solutions based on queuing models [18], feedback loops techniques [7], mathematical models [12] or even approaches using neural networks techniques [8]. However, most of these models require a deep understanding in mathematics or machine learning techniques which are not easily interpreted by non specialists. Besides the traffic in web applications is shaped by a combination of different factors such as diurnal and seasonal cycles, sociological and psychological, that follows an irregular pattern. It makes extremely challenging the design and development of realistic and accurate dynamic provisioning mechanisms.

These well-known drawbacks force to IaaS like Amazon EC2 and Windows Azure [11], or PaaS like RightScale [15] and OpenShift [13], to design simple threshold rule-based auto-scaling systems, instead of relying on approaches from academic research. Unfortunately, these scaling systems are naive, wasteful in terms of resource consumption and cost savings, and an easy target for flash crowds.

In the following, we present some of the most relevant and realistic academic approaches that proposed dynamic resource provisioning mechanisms for multi-tier applications.

In [17, 18], the authors designed and implemented a predictive and reactive provisioning mechanism. They used a queuing model G/G/1 to decide the server pool size to be provisioned, and an admission control mechanism to face extreme workload variations. Offline prolifing techniques were employed to gather information about the resource requirements of the incoming requests for each tier, and thereby to selectively admit/reject requests for the lightweight files. An evaluation using real-traces on a homogeneous infrastructures shows the benefits of this approach when handling flash crowds. Unfortunately, its admission control mechanism incurs into sporadic SLA violations (if the server utilization exceed a predefined threshold) reducing the QoS of the service, and therefore affecting user experience. Similarly to the previous work, [19] extended queuing models and transaction mix models to design a predictive and reactive provisioning system. To model the application performance, they integrated proactive control and feedback control methods that dynamically adjusted the CPU capacity allocated to servers. This work only considered SLA constraints at the system-level, while others constraints at application-specific level such as response time and request rate were not taken into consideration. Besides, an evaluation of CPU variations on a homogeneous infrastructure, when processing traces from a non real-world application, lack arguments to valid its approach.

Regarding the management of flash crowds [21], a proactive application workload manager was designed to separate the user requests between two groups of servers: one named 'base workload' referred to the smaller and smoother variations in the workload;

and the other 'tresspassing' referred to the temporal burstly workloads caused by flash crowds. To do this, the authors attempt to divide the data items into popular and less popular, and place them in the right group of servers. Even tough a realistic evaluation was conducted on Amazon EC2 utilizing real traces (Yahoo video streaming), authors do not explain in details how the dynamic resource provisioning is done. Recently, in [10], online profiling techniques have been utilized for managing the tradeofff between performance overload, and cost savings for dynamic resource provisioning. The authors replicate at runtime a regular server hosting an application, with a new server with profiling instrumentation. Their experimental results show how profiling techniques can be included in a resource provisioning system, without causing important response time delays or throughput alterations in comparison with non-profiling provisioning. As we mentioned in Section , profiling techniques can report more benefits than performance degradations or expenses.

EXCESIVE reactive algorithms tend to temporally overprovision applications affected by flashcrowds or slashdot effects. This type of algorithms increase the resource consumption and infrastructure costs than other ...

Experiments based on real-traces "Wikipedia" and conducted on heterogeneous and homogeneous cloud infrastructures.

The use of offline-profiling techniques allow to identify the threshold of the resources of a cloud infrastructure.

The provisioning system remains independent of the infrastructure on which the apps run.

Acknowledgments

This work is partially funded by the FP7 Programme of the European Commission in the context of the Contrail project under Grant Agreement FP7-ICT-257438.

FIXME: Probably also mention ERRIC here. ??

References

[1] Advanced School for Computing and Imaging (ASCI). The Distributed ASCI SuperComputer 4. `http://www.cs.vu.nl/das4/`.

[2] Amazon Elastic Compute Cloud (EC2). `http://aws.amazon.com/ec2`.

[3] Jiang Dejun, Guillaume Pierre, and Chi-Hung Chi. EC2 performance analysis for resource provisioning of service-oriented applications. In *Proceedings of the 3rd Workshop on Non-Functional Properties and SLA Management in Service-Oriented Computing*, November 2009.

[4] Anh Vu Do, Junliang Chen, Chen Wang, Young Choon Lee, A.Y. Zomaya, and Bing Bing Zhou. Profiling applications for virtual machine placement in clouds. In *2011 IEEE International Conference on Cloud Computing (CLOUD)*, pages 660–667, July 2011.

[5] Guido Urdaneta Erik-Jan van Baaren and Guillaume Pierre. Wikibench: A distributed, wikipedia based web application benchmark. 2009.

[6] Ganglia monitoring system. `http://ganglia.sourceforge.net/`.

[7] Jiayu Gong and Cheng-Zhong Xu. A gray-box feedback control approach for system-level peak power management. In *2010 39th International Conference on Parallel Processing (ICPP)*, pages 555 –564, September 2010.

[8] Sadeka Islam, Jacky Keung, Kevin Lee, and Anna Liu. Empirical prediction models for adaptive resource provisioning in the cloud. *Future Gener. Comput. Syst.*, 28(1):155–162, January 2012.

[9] Dejun Jiang. *Performance Guarantees For Web Applications*. Phd thesis, VU University Amsterdam, March 2012.

[10] Nima Kaviani, Eric Wohlstadter, and Rodger Lea. Profiling-as-a-service: adaptive scalable

resource profiling for the cloud in the cloud. In *Proceedings of the 9th international conference on Service-Oriented Computing*, ICSOC'11, pages 157–171, Berlin, Heidelberg, 2011. Springer-Verlag.

[11] Microsoft Windows- Windows Azure. `http://www.windowsazure.com/en-us`.

[12] Sireesha Muppala, Xiaobo Zhou, Liqiang Zhang, and Guihai Chen. Regression-based resource provisioning for session slowdown guarantee in multi-tier internet servers. *Journal of Parallel and Distributed Computing*, 72(3):362–375, March 2012.

[13] OpenShift. `https://openshift.redhat.com/app/flex`.

[14] Guillaume Pierre and Corina Stratan. ConPaaS: a platform for hosting elastic cloud applications. *IEEE Internet Computing*, 16(5):88–92, September-October 2012.

[15] RightScale. `http://www.rightscale.com//`.

[16] Rahul Singh, Upendra Sharma, Emmanuel Cecchet, and Prashant Shenoy. Autonomic mix-aware provisioning for non-stationary data center workloads. In *Proceedings of the 7th international conference on Autonomic computing*, ICAC '10, page 2130, New York, NY, USA, 2010. ACM.

[17] Bhuvan Urgaonkar and Prashant Shenoy. Cataclysm: Scalable overload policing for internet applications. *Journal of Network and Computer Applications*, 31(4):891–920, November 2008.

[18] Bhuvan Urgaonkar, Prashant Shenoy, Abhishek Chandra, Pawan Goyal, and Timothy Wood. Agile dynamic provisioning of multi-tier internet applications. *ACM Trans. Auton. Adapt. Syst.*, 3(1):1–39, March 2008.

[19] Zhikui Wang, Yuan Chen, D. Gmach, S. Singhal, B.J. Watson, W. Rivera, Xiaoyun Zhu, and C.D. Hyser. AppRAISE: application-level performance management in virtualized server environments. *IEEE Transactions on Network and Service Management*, 6(4):240 –254, December 2009.

[20] Wikipedia Foundation. MediaWiki a free software wiki package. `http://www.mediawiki.org`.

[21] Hui Zhang, Guofei Jiang, Kenji Yoshihira, Haifeng Chen, and Akhilesh Saxena. Resilient workload manager: taming bursty workload of scaling internet applications. In *Proceedings of the 6th international conference on Autonomic computing*, ICAC '09, pages 45–46, New York, NY, USA, 2009. ACM.