Platform-as-a-Service Architecture for Performance Isolated Multi-Tenant Applications

Rouven Krebs
SAP AG
69190 Walldorf, Germany
rouven.krebs@sap.com

Manuel Loesch
FZI Research Center for Information Technology
76131 Karlsruhe, Germany
loesch@fzi.de

Samuel Kounev
Julius-Maximilians-University Wuerzburg
97074 Wuerzburg, Germany
samuel.kounev@uni-wuerzburg.de

***Abstract*—Software-as-a-Service (SaaS) often shares one single application instance among different tenants to reduce costs. However, sharing potentially leads to undesired influence from one tenant onto the performance observed by the others. This is a significant problem as performance is one of the major obstacles for cloud customers. The application does intentionally not manage hardware resources, and the operating system is not aware of application level entities like tenants which makes the performance control a challenge. In case the SaaS is hosted on a Platform-as-a-Service (PaaS), the SaaS developer usually wants to control performance-related issues according to individual needs, and available information is even more limited. Thus, it is difficult to control the performance of different tenants to keep them isolated. Existing work focuses on concrete methods to provide performance isolation in systems where the whole stack is under control. In this paper we present a concrete PaaS enhancement which enables application developers to realize isolation methods for their hosted SaaS application. In a case study we evaluated the applicability and effectiveness of the enhancement in different environments.**

I. Introduction

Through economies of scale, sharing of fixed costs, over-commitment and leveraging of workload fluctuations, it is possible to decrease the overall costs for cloud providers and consumer. Software-as-a-Service (SaaS) describes the provisioning of applications via the cloud. A SaaS may run on a Platform-as-a-Service (PaaS) which provides a runtime environment for applications. Existing PaaS add value through additional services, e.g., regarding persistence or authentication.

For economic reasons, in SaaS scenarios, one single application instance is often shared between different tenants [1] by providing every tenant a dedicated and isolated view onto the application. A tenant is defined as a group of users sharing the same view on an application. This view includes the data they access, the configuration, the user management, particular functionality and related non-functional properties like performance [2], [3]. Several PaaS support the development of Multi-tenant Applications (MTA) by providing multi-tenancy enabled services, like tenant isolated persistence (e.g., Google App Engine[1]). Gartner assess the importance of the multi-tenancy cloud provisioning models as important for the near future [4], [1]. Further, they see performance isolation as one challenge to be solved.

[1]https://developers.google.com/appengine/docs/java/multitenancy

The importance of reliable performance in Cloud environments was shown in [5], and is a major research issue [6], [7]. Especially in MTAs one tenant can significantly decreases the overall performance due to the tight coupling. In [7], the need for novel middleware architectures to ensure performance-related SLAs is emphasized.

A system is performance-isolated, if tenants working within their SLA-defined quotas observe a performance that is within their defined SLA guarantees, even in cases other tenants exceed their quotas [2]. The SLA guarantee usually refers to response time, and the quota to throughput or arrival rate. In [2] it is also discussed that elasticity does not solve this issue.

In order to offer performance isolation in MTAs, various challenges have to be solved. Layer discrepancy describes that the resource access is intentionally realized in lower system layers and tenant information is intentionally handled at the application/middleware layer. Furthermore, cloud systems are usually over-committed which results in a high utilization and little buffer capacity. Finally, a solution must be able to handle several application instances in a distributed setup.

Existing work concerning performance isolation approaches rather focuses on concrete algorithms/methods instead on architectural aspects. The solutions are developed for specific applications and scenarios to ensure a specific behaviour. One SaaS provider might prefer a behaviour where requests from a tenant exceeding its quota are rejected while another SaaS provider might prefer to delay the request (e.g., [2]). Such decisions might depend on the SaaS provider's individual specification or on scenario attributes like the workload model (e.g., open- vs. closed-workload). Application-specific characteristics are another point of interest. In case it is possible to estimate resource demands for incoming request, an approach based on indirect resource control [8] is feasible, while in other situation a feedback control loop that is based on the observed QoS might be appropriate [9].

In case a MTA is hosted on a PaaS, the platform provider has to provide an environment for the SaaS developer in which he can deploy the application-specific isolation algorithms. A development of such features solely based on traditional PaaS runtime environments is not possible, as the necessary access to runtime information [10] and the requests process flow (e.g., method from [9], [11]) is limited.

Our contribution is a framework and general approach which can be used by PaaS providers to ensure performance isolation for hosted applications, while supporting a plug-in

mechanism to add application-specific algorithms. In preparation for this, we evaluated the general requirements such a system has to fulfil, which is a contribution in itself. In a case study we implemented four algorithms in three platform environments to validate the usability and effectiveness of the solution. With this work, we achieved to close the gap between the current work focusing on algorithms and their applicability in a PaaS environment.

The remainder of this paper is structured as follows. Section 2 discusses related work and foundations. In Section 3, system requirements are elaborated. The architecture of the system is presented in Section 4. In Section 5 we discuss the important design decisions, and in Section 6, we present a case study applying the architecture. Finally, Section 7 concludes the paper.

II. Related Work

The relevant related work can be separated in three parts. (1) General architectural concerns of MTAs which influences our work. (2) Performance isolation in the context of multi-tenancy. (3) Performance control from other contexts.

Koziolek [12] [13] analyzed existing MTAs and described the essential requirements for a multi-tenant software architecture. Based on this, Koziolek proposed an architectural style for multi-tenant software applications following the three-tier web application model where a client communicates via HTTP with the application tier. The application tier consists of a load balancer and application servers using a shared database. In [14], an overview of architectural concerns in MTAs and their mutual influences including performance isolation is given. With regards to performance isolation three aspect are of importance. (1) Tenant affinity, which describes how to forward a tenant's request to the present processing nodes. (2) The chosen type of data isolation (separate database vs. shared table). (3) Customizability, which may lead to various resource demands per tenant. Both papers provide foundations how MTAs are build and show relevant concerns. However, they do not provide measures for performance isolation.

Zhang [15] and Fehling [16] provide SLA-aware heuristics for the placement of tenants onto a set of application instances. However, this approach only reduces mutual performance influences between isolated groups and will fail if workloads change unforeseenly. Guo et al. [17], among others, discuss three basic ideas to ensure performance isolation. (1) Tenant placement similar to [15], [16]. (2) Static or dynamic request admission control to limit the amount of competing requests for each tenant. (3) Fixed resource reservation for each tenant. However, the authors mention that resource reservation is complicated, and a specific method or architecture for request admission control is not provided. Cheng et al. [18] present an approach based on SLA monitoring. However, in the case of an upcoming SLA violation, their approach aims at renegotiating SLAs instead maintaining them.

In [11], the ratio of the mean arrival rate and mean service rate are observed to detect performance anomalies. Further the aggressive tenants, who consume the most resources, are identified in order to apply an adoption strategy of the systems request flow. Lin et al. [9] address regulating response times to provide different QoS by use of a feedback-control loop. The regulator uses the average response times to apply an admission control. Further, it uses the difference in service levels agreements between tenants to apply different thread priorities to them. In [2] four performance isolation mechanisms are evaluated. One approach selects requests from different tenant queues in a round-robin style. Others provide an artificial delay, a black list or separate thread pools. It is shown that these approaches enforce performance isolation and an efficient resource usage within a limited range of situations. To fully leverage their potential, parameters of the approaches (e.g., queue priorities) have to be adjusted dynamically. Wang et al. [8] use a Kalman filter to observe the resource consumption of various tenants. In a controller it is checked whether a tenant has negative impact onto the guaranteed resources of the others. If this is the case, requests of this tenant are rejected. The prior work [8], [2], [9], [11] focuses on algorithms and lacks concrete descriptions of architectures and considerations of how to implement these approaches.

Feedback control loops have been successfully applied to guarantee target response times of web applications [19], [20]. In [21] a desired utilization value for a web server was controlled by adjusting the keep-alive time and the number of maximum concurrent requests served. In [22] a control loop is used to decrease performance influences between independent web sites that are hosted on the same web server by reducing the quality of the presentation.

The most promising existing approaches to regulate performance are based on a feedback-driven request admission control loop. However, all approaches lack a sufficient description of architectural issues. The few architectural papers do not consider performance isolation.

III. Requirements

This section provides an overview of requirements we identified for a system which enhances a PaaS to support performance isolated MTAs.

A. Required Functionality

Enforcing *performance isolation* between different tenants is a major task of the system. Different tenants may have different needs and different willingness to pay for performance. Hence, the solution should be able to provide *QoS differentiation*. *Over-commitment* increases the economic efficiency of SaaS offerings. If every tenant is using his full quota and thus the system runs in an overloaded situation, the framework must keep a valid state. Since workloads from tenants are characterized by fluctuations, the solution must be able to handle such variations.

B. Isolation Method Categories

In [10] we introduced four classes of performance isolation approaches based on request admission control. All have specific pros and cons, depending on the application scenario and defined requirements. In the following, we present a brief analysis of the relevant aspects. Table I lists the informational requirements for each category.

Static approaches use static rules/algorithms with constant parametrization for the decision which request is allowed to

TABLE I. INFORMATIONAL REQUIREMENTS FOR DIFFERENT CATEGORIES

	Static	Feedback-based	Resource Demand	Res. Demand + Request Types
SLAs	yes	yes	yes	yes
Resp. Time	no	per tenant	per tenant	per request type
Throughput	no	per tenant	per tenant	per request type
Utilization	no	no	yes	yes
Demand	no	no	per tenant	per request type

pass through or being delayed before it is handled by the MTA. The parameters are derived from the SLAs for a tenant. Static approaches were already used in the field of performance isolation (e.g., [2]).

The *feedback-based* approaches use QoS runtime information (response times, throughputs) to adjust the parametrization of the admission control. Since the SLA guarantees are compared to the current QoS and load onto the system, a better isolation with better utilization can be achieve. Lin et al. [9] present an example of these approaches.

Resource Demand Based Approaches consider the actual resources used per tenant. Based on the difference between the observed and guaranteed quota, the admission control is adjusted to guarantee a fair resource share. Resource Demand Estimation (RDE) methods [23] can be applied to determine the resource demands for each tenant and request type by leveraging additional information (cf. Table I). Examples of this approach are Lin et al. [9] and Wang et al. [8].

Detailed knowledge of a *request-type specific* demand allows fine grained admission control focusing on a particular resource bottleneck. The resource demand for the same operation type might vary over time (e.g., increasing DB size) and for tenants (e.g., divergent configurations, DB size). Respective methods require measurements on a request type basis as it is done in [8].

C. Generic Solution

The system should support *various performance isolation approaches* as presented in Section III-B since the preferred approach strongly depends on the application deployed at the platform. The concrete application-specific *isolation algorithm has to be decoupled* from the generic parts that are common for all algorithms, and from technical details. The proposed architecture should be *portable* and thus not being bound to a specific PaaS, middleware or operating system.

D. Scalability Issues

In MTAs, horizontal scaling is a common pattern. A load balancer represents the endpoint for the tenants and forwards requests to the instances [12], [13]. *Tenant affinity* describes how requests of one tenant can be distributed over multiple instances. *Session affinity* describes the distribution of request of one user to application instances. In [14], different kinds of tenant affinity (server-affine, non-affine, cluster-affine) and session affinity (sticky sessions, non-sticky sessions) are presented with their application scenario dependent pros and cons. Some applications may support to dynamically add or remove instances in order to address load variations. This refers to the term *elasticity*. Although, it is not the goal of the framework to enable elasticity, it must be able to deal with such a load balancing environment.

E. Performance Overhead and Application Scenario

We focus on interactive web applications. Hence, the delay which is added to a user's request must be low. Furthermore, the communication between different system entities should be low.

IV. ARCHITECTURE

In this Section we present the system, while addressing the requirements from the previous Section.

A. Overall Concept

In our former work [2], we demonstrated that approaches based on request admission control are promising to influence the performance a tenant perceives. A request admission control delays or even rejects requests. Hence the consumption of resources as well as the perceived response times can be controlled.

Due to different kinds of affinity, the load might be unequally distributed. Thus the admission control has to be specific for each instance in a cluster. The discussion presented in [24] outlines that a central management of request processing information is needed to support all kinds of affinity (Figure 1). Hence, the proposed *Performance Isolation Framework* (PIF) architecture splits the functionality in two parts. First, the application-instance specific request admission control (*Execution Point*) which might be a separate proxy, or part of the application server's request preprocessing pipeline. Its behaviour follows a specific strategy. Second, the *PIF core* which contains the application specific isolation algorithm. It periodically configures the Execution Point's strategy with the policy.

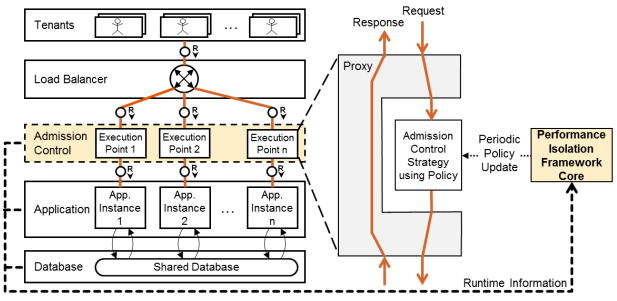

Fig. 1. Overall Approach Based on Admission Control

A *strategy* is a parametrized algorithm that defines how to admit, reject or delay requests from a certain tenant, and thus realizes a request scheduling. An example for a strategy is a tenant-aware Weighted Round Robin mechanism (WRR) with own queues and weights for each tenant.

Policies are used to adjust the strategy. They are fragments containing the new configuration, which are exchanged between the PIF core and the Execution Point. In case of a WRR, a policy may set priorities per tenant-specific queue.

B. Structural Overview

An overview covering all entities in the system is depicted in Figure 2. It consists of seven major system entities:

A *Tenant* consisting of multiple users sending requests which are distributed by the *Load Balancer* while considering a potential tenant or session affinity. The *Execution Point* is responsible for the request admission control based on a exchangeable strategy. The actual *Performance Isolation Framework Core* maintains state about all data that is required for the performance isolation mechanism, and is responsible for the creation of Execution Point specific policies. The *Application Instances* run the application provider's business logic and process requests. A *(Shared) Database* is responsible for data persistence of the application. In cloud systems an *Elasticity Manager* usually realizes horizontal scalability by adding or removing application instances at runtime.

C. Detailed Component Description

This section covers the detailed descriptions of the components depicted in Figure 2 in detail.

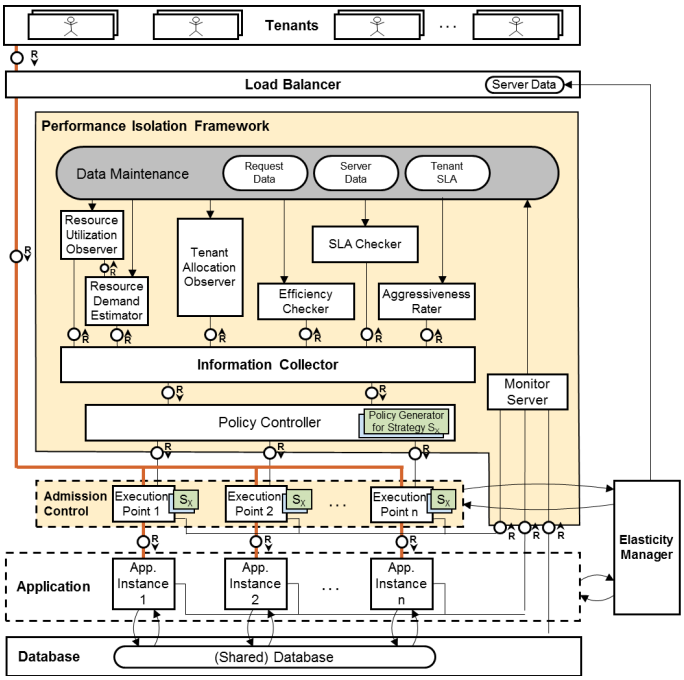

Fig. 2. Architecture of the Performance Isolation Framework

1) Execution Point: The component is responsible for the request admission control. It accepts, refuses or delays certain incoming request in order to achieve performance isolation and a fair performance distribution. In case of static approaches, the policy is updated only once at the beginning based on the existing SLAs, affinity and allocation of tenants onto nodes. At start-up/shutdown, this component registers/unregisters itself at the *Policy Controller*.

2) Policy Controller/Generator: The *Policy Controller* updates the policy forwarded to the Execution Point's strategy. It maintains state about all Execution Points and manages the communication with them. It further allows to (un)register Execution Points at runtime. The *Policy Generator* is an application scenario specific plug-in component. The *Information Collector* periodically triggers the Policy Controller, which iteratively calls the Policy Generator to create policies for each Execution Point. An individual policy is required as affinities or variable performance of the application instances can lead to different performance.

Policies created by the Policy Generator are specific for an Execution Point's strategy, they are both application scenario specific plugins and tightly coupled. For example, a strategy might require delay per tenant, or a set of priorities. Therefore, the interface between the Execution Point strategy and the Policy Controller defines meta information (e.g., tenant IDs) and a generic part for the specific policy data.

3) Monitor Server: The *Monitor Server* receives runtime information from different probes and persists it with the *Data Maintenance*. The probes push data to the Monitor Server component in pre-configured time frames. The two most important probes are listed below.

Execution Point Probe: Reports the overall response times, the processing time of a requests after admission, discarded requests and throughputs. The granularity (per tenant, per request type, aggregation interval) can be configured depending on the used isolation methods.

Application Server Probe & (Shared) Database Probe: For RDE approaches the resource demand is usually calculated. Therefore the utilizations of the respective resources are measured. The required granularity depends on the configuration and used isolation method [24].

4) Information Collector: The *Information Collector* manages the overall program flow and collects information from multiple components which are forwarded to the Policy Controller. Time intervals like the policy update as well as the order in which components are called, is managed by this component. The Information Collector allows that single components can easily be exchanged and replaced.

5) SLA Checker, Aggressiveness Rater and Efficiency Checker: The *SLA Checker* is responsible for delivering the compliance of the SLA per tenant and application instance as well as the overall compliance. The tenant- and node-individual evaluation is needed due to the aforementioned fluctuations of response times caused by affinity. By default, the SLA Checker returns the deviation of the average response time to the response time guaranteed in per cent for the entity under investigation.

The *Aggressiveness Rater* component reports the aggressiveness for all tenants based on the quotas guaranteed and the quota used. To optimize the policies, the distribution of the requests among all application instances is also provided.

The *Efficiency Checker* calculates the efficiency of the current policy. Different understandings of efficiency are possible. One metric for efficiency is the ratio between the current throughput and the maximum throughput a system could achieve. This enables algorithms to increase the throughput for a tenant, which already exceeds its quota to keep the overall resource utilization in a good state.

The calculation rules for all three component discussed in this section are realized as plugins and can be easily replaced

by application scenario specific variants such as an additional outlier filter for the SLA checker.

6) Tenant Allocation Observer: In case of an affine behaviour of tenants, it identifies the set of tenants that are influencing each other because they share the same application instance. Without affinities, this component is not required.

7) Elasticity Manager: The Elasticity Manager usually already exists and must be adapted to start/modify an Execution Point for each Application Instance that is added.

8) Resource Utilization Observer and Resource Demand Estimator: The *Resource Utilization Observer* delivers average resource utilization information for a certain time frame. The utilization per tenant is derived by statistical methods [23]. This component is required for the most resource demand estimation techniques.

The *Resource Demand Estimator* delivers the estimated resource demand per tenant, and the estimated demands per request type. Whereby the same request type of different tenants is considered separately. Details on how to estimate resource demands on a tenants basis are found in [23].

D. Usage by Isolation Approaches

Based on Section III-B, this Section gives an overview about the interaction of the components. *Static Approaches* use static strategies in the Execution Point. Since no performance related runtime information is leveraged, the policy is generated only once when the amount of tenants registered in the system changes. In case of *Feedback-based Approaches* the Information Collector collects the required data from the SLA Checker, Aggressiveness Rater and the Tenant Allocation Observer and forwards them to the Policy Controller. The *Resource Demand based Approaches* make additional use of the Resource Demand Estimator's information. The policy generation algorithm (Policy Generator) determines a fair amount of resources that is entitled to each tenant and derives a suitable policy. *Request-type based Resource Demand Approaches* leverage knowledge of the current resource utilization delivered by the Resource Utilization Observer. This way, a fine-grained and bottleneck-aware prioritization of request types is possible.

E. Interfaces for Plugins

A concrete performance isolation approach consists of a (1) a strategy, and (2) a strategy-specific isolation algorithm referred to as *Policy Generator*. Both are realized as plug-in and hence the proposed architecture allows the application developer to realize performance isolation with regards to his specific needs by solely implementing the two representative interfaces. The framework periodically starts the generation of a policy for a certain Execution Point based on the preprocessed information. The Strategy has to implement two methods. The first is called to add an incoming request. The request object transferred is framework-specific and provides meta data (e.g., tenant ID, request type). The second method is called when a new request can be processed (i.e., when a server thread become free) and has to return a request.

V. Design Decissions

In this section we discuss various design and trade-off decisions taken to fulfil the requirements elaborated in Section III.

A. Isolation Capabilities

High degree of isolation vs. performance overhead: The use of a policy which sets priorities for a certain time-frame results in a lower accuracy since the workload may change in the meantime. Contrary, per-request calculation which makes use of the maintained data would produce a high overhead. Such per-request calculations may make sense for batch jobs but not for interactive scenarios. Therefore, the execution point which implements a fast working strategy to control the requests and complex computations are done asynchronously in the policy generator.

Data aggregation in the monitoring process further reduces the accuracy and adoption speed. However, due to the vast amount of requests, relinquishing the use of an asynchronous computation of the policy and data aggregation are not an alternative. It is worth to mention, that monitoring intervals and policy update intervals can be adjusted to increase the adaptation speed.

QoS differentiation & over-commitment: The periodically triggered policy generation uses information about the tenants' aggressiveness and response time compliances. Since these information are rated in relation to the values guaranteed in the SLAs, QoS differentiation can be realized by comparing percentage values when weighting tenants. For the same reason, the handling of over-commitment is no problem.

B. Generic Solution

Suitability for various isolation approaches: The proposed architecture provides components and measures to deliver all information required by the mechanisms identified in Section III-B.

Plug-in mechanism for isolation algorithm: By delivering all relevant data to the Policy Controller which maintains state about all Execution Points and abstracts the technical communication, the policy generation algorithm (Policy Generator) can easily be implemented and replaced. The Execution Points' strategy is decoupled from the technological details and can be easily replaced/implemented. If needed, individual implementations for the SLA/Efficiency Checker and the Aggressiveness Rater are possible.

Portability between different system environments: Load balancers might not support to control/delay requests based on strategies, hence they are implemented in the Execution Points. Therefore, the Execution Points are deployed together with the MTA instance. Furthermore, the use of OS-dependent interfaces required by the framework is minimized if RDE approaches that are independent of resource measurements are used.

C. Scalability Issues

Once a new *Application Instance* with the respective *Execution Point* is added, the *Execution Point* registers itself at the *Policy Controller*. With the next policy generation, respective policies can be created.

The amount of policies generated increases linear with the amount of *Application Instances* and the amount of data

that has to be processed by the policy generation algorithm increases linear with the amount of tenants.

In case of affinity, the proposed framework is able to deal with unequal performance distributions as input parameters by providing instance-specific data for the policy generator. The tenant affinity and existing allocation are analyzed by the *Tenant Allocation Observer*. Hence, the policy generator can limit its admission decisions to relevant tenants.

VI. Case Studies

This section presents a concrete implementation of the framework. We focus on two major goals. *G1* is to show that the proposed framework can be used to enhance PaaS environments. *G2* is to show the framework's capability to support various isolation algorithms from different categories in a wide range of scenarios. As additional goal, we estimate the additional overhead of the approach (*G3*). In total we investigated the applicability within three platforms, and present four concrete isolation methods of different categories (cf. Section III-B) whereby the presentation primarily focuses on the results gathered in the most complex setups. Since we refer to existing isolation methods, we do not discuss the methods itself in detail.

A. Implementation and System Environments

In environment *E1* we realized an admission control in a simulated MTA described in [2]. In another environment *E2* a separate HTTP proxy was implemented to realize the request admission control for a Tomcat instance serving a single servlet. The third environment *E3* for our detailed discussion was based on the SAP Hana Cloud Platform. The different components of our framework were implemented in Java. The plugins are specified by a configuration file loaded at the startup of the application. At the moment, we provide three different implementations of the Execution Point. First, a Valve[2] as an integral part of the SAP Hana Cloud Runtime Container referred to as Lean Java Server (LJS). Second, a standalone proxy application which enables a generic usage for different runtime environments. Third, as an integral component of a simulation used to simulate multi-tenant applications already used in [2]. The Resource Demand Estimator's implementation follows the approach described in [23]. The communication between the probes, central parts of the PIF and the Execution Points was REST based.

Figure 3 presents the experiment setup in detail. The *Controller Host* is a standard PC hosting the *PIF Core* and the *Load Driver Client* which controls the experiment's load. The *Load Driver Server* and the *Application Server* have 16x2 GHz CPUs, 16 GB of memory and run on SLES11 SP2. The Load Driver Server hosts the load driver for an enhanced version of the widely accepted TPC-W [25] benchmark which was adapted to support multi-tenancy (*MT TPC-W*) [26]. The Application Server hosts several XEN-virtualized *VM*s with one pinned processing unit and 2 GB memory. Each *VM* hosts the *LJS* which runs the *MT TPC-W* application simulating an online book store. A MySQL DB was directly deployed in the VM. Further, the CPU utilization of the VM, the response

[2]http://tomcat.apache.org/tomcat-7.0-doc/config/valve.html

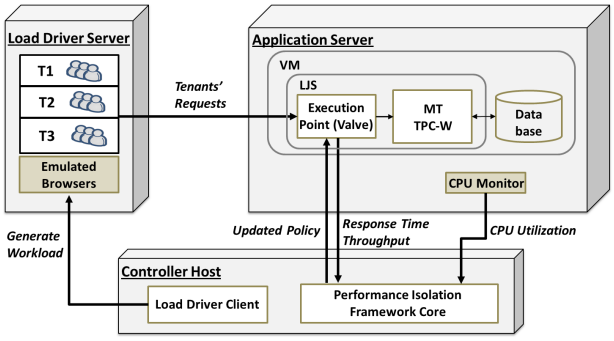

Fig. 3. System Environment E3

times and the throughputs per request type and tenant were measured.

B. Results

1) Static Approach: In *E1*, a Round Robin algorithm was used for admission control. We run an over-committed scenario in which one tenant exceeding the reference throughput X_{ref} (disruptive tenant) and nine other tenants (abiding tenants) shared one system. The actual average response time for all abiding tenants increased 3% at a load increase of the factor 10 for the disruptive.

2) Feedback-based Approach: In this Section we describe two cases studies based on two different scenarios.

a) SAP Hana Cloud: We defined the maximum average response time for a time frame of 2 minutes to be $R_{ref} = 2500ms$ at a maximum throughput $X_{ref} = 3100$ requests/minute for each tenant. A feedback-based approach according to the definition described in Section III-B was implemented. The admission strategy is based on a Weighted Round Robin scheduler. The controller plugin uses two Proportional Integral (PI) controllers following our previous work [27]. One for situations where SLA violations exist, the other when the system runs inefficiently. The definition of efficiency we implemented (in the Efficiency Checker) describes how well the system guarantees R_{ref} for tenants exceeding X_{ref} when for other tenants R_{ref} is not violated. We used the environment described in Figure 3 with two VMs in an non-affine and non-sticky configuration with 100 available threads.

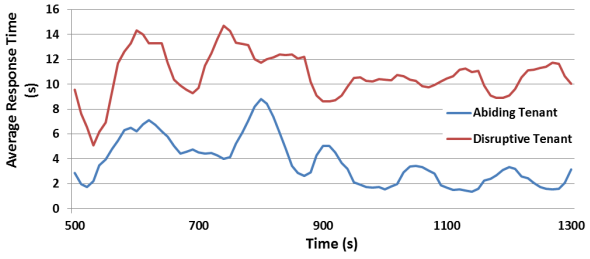

Fig. 4. Performance Isolation with Feedback-based Approach

In Figure 4 both tenants started with 500 users. After this initial ramp up phase 500 users were added to the disruptive tenant to make it exceeding X_{ref}. At 600 seconds all users where started. Due to damping in the controllers

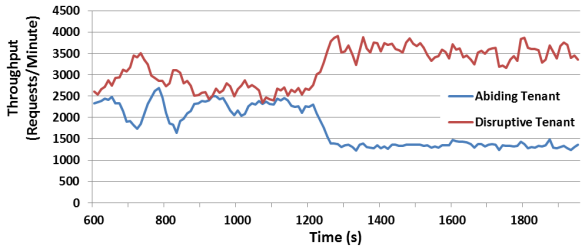

Fig. 5. Efficiency with Feedback-based Approach

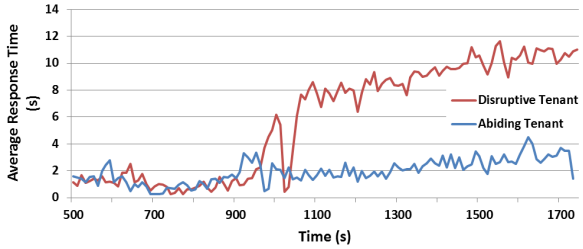

Fig. 6. Performance Isolation with Resource Control

implementation and delay in the monitoring it took around 300 seconds before the weights were adjusted to provide R_{ref} for the abiding tenant whereas the disruptive tenant has significantly higher response times.

In Figure 5 the disruptive tenant starts with 1000 users and the abiding one with 500 users. The throughput for the abiding tenant was below X_{ref}, and R_{ref} did not exceed the SLA. At the beginning of the observation the throughput of the disruptive tenant was reduced by the Policy Generator to maintain R_{ref} for the abiding tenant. At 1200 seconds the load of the abiding tenant was immediately reduced by 300 users. Thus, the overall system utilization was reduced; hence the resources became free which could be used by the disruptive tenant without violating the isolation to increase the efficiency. Consequently, the throughput for the disruptive tenant increased between 1210 and 1270 seconds. Besides the new generated policies the fast reaction was a result of the admission control scheduler as discussed in [27].

b) Tomcat and Proxy: The admission control in environment *E2* was based on a black list [2]. When a tenant is blacklisted, all its requests are rejected. Tenants on the white list share a single FIFO queue. A rule-based controller observed the SLAs of the last period and put a disruptive tenant onto the blacklist if SLAs of the others were violated. Due to the binary behaviour, the observed response times of this approach were oscillating.

3) Resource Demand based Approach: This case study was again based on environment *E3*. Resource Demand based Approaches isolate resources and consequently the response times and throughputs which is the actual goal of our system. However, the mapping of a particular response time to resource requirements is still a challenge in itself [28]. The implemented algorithm uses a CPU demand estimation approach and the observed utilizations, in order to compare a tenant's resource utilization with a pre-configured one. We presented the details in [23]. Based on this comparison, the priority for tenants is continuously derived on a short term basis.

We used the environment *E3* with one application server, 100 threads and a smaller database volume compared to the previous experiments. Each tenant was configured to use max. 50% of the CPU. In Fig. 6, till t=900 s both tenants had 1000 users and the same response times. Afterwards we added 1000 users to the disruptive tenant.

In this scenario the database significantly increases its volume due to the active users, resulting in an increasing response time. The system continuously updated the resource demand estimates. The abiding tenant's response time is only slightly influenced by the disruptive tenant.

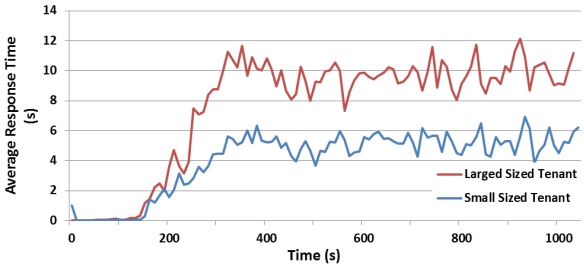

Fig. 7. Differentiation of Tenants with Resource Control

The next experiment had two tenants with 1200 users each. One of the tenants was using a small-sized database, while the other used a database which was around 10 times larger and consequently higher demands occurred. Both tenants were configured to use 50% of the CPU resource. Figure 7 shows that the tenant with the large-sized database has higher response times due to the limited resources. The observed relative error of the resource allocation was always below 9%.

C. General Observations and Goals

It was easily possible to integrate different isolation methods into our enhanced PaaS. We always used the same core system without any adaptation. In the valve-based implementation of the admission controller, the overall increase of response time and resource utilization compared to a non-isolated version was negligible. In the proxy version, we observed an increase that is typical for web proxies. The utilization of the framework's core depends on the components, the policy generators and the update intervals. However, the average CPU utilization usually stayed clearly below 15%.

To achieve *G1* the architecture was implemented as an enhancement of the SAP Hana Cloud Platform including all monitoring capabilities. By implementing additional Execution Points we showed the architecture's applicability for other PaaS environments. To answer *G2* we implemented different Policy Generators and Execution Point Strategies. For the first we chose static, feedback (proportional integral and rule based) and RDE based ones. For the latter, Round Robin, Weighted Round Robin and Blacklist. Thus, various scenario-specific isolation methods were proofed to work within this architecture. The overhead of the system (*G3*) was shown to be acceptable due to aggregation and asynchronous policy generation.

VII. Conclusion

Software-as-a-Service (SaaS) usually shares one single application instance between different tenants to decrease costs (MTA). However, the tight coupling of tenants and the application's intentioned abstraction from the resource control makes performance isolation a challenge. Several request admission control based methods to ensure performance isolation exist. These are specific for concrete scenarios depending on the application's attributes, workloads and specifications of the application provider. Additionally, related work does not discuss architectural aspects. In case a MTA is hosted on a Platform-as-a-Service, the platform provider has to provide an environment for the SaaS developer in which he can deploy the application-specific performance isolation algorithms. A development of such performance isolation features solely based on the traditional SaaS runtime environment is not possible, as the access to runtime information and the request process flow is limited.

In this paper we discussed the requirements a PaaS enhancement to support performance isolated MTAs has to fulfil. Subsequently, we derived the architecture of a framework that enhanced PaaS to enable application developers to implement tailored performance isolation mechanisms for their applications. The architecture enables a light-weight request admission control following a configuration which is periodically updated. To maintain a good request response time, the configurations are asynchronously derived from preprocessed monitoring information. The case study shows that the proposed architecture/framework is able to enhance a PaaS to support a wide range of algorithms for scenario-specific performance isolation with an acceptable performance overhead.

Acknowledgements

The research leading to these results has received funding from the European Union Seventh Framework Programme (FP7/2007-2013) under grant no. 317704 (CloudScale).

References

[1] Y. Natis, "Gartner reference model for elasticity and multitenancy," Gartner, Gartner Report, June 2012.

[2] R. Krebs, C. Momm, and S. Kounev, "Metrics and Techniques for Quantifying Performance Isolation in Cloud Environments," *Elsevier Science of Computer Programming Journal (SciCo)*, 2013.

[3] C.-P. Bezemer, A. Zaidman, B. Platzbeecker, T. Hurkmans, and A. Hart, "Enabling multi-tenancy: An industrial experience report," in *Software Maintenance (ICSM), 2010 IEEE International Conference on*, 2010.

[4] R. Desisto, "Hype cycle for software as a service, 2012," Gartner, Tech. Rep., 2012.

[5] Bitcurrent, "Bitcurrent cloud computing survey 2011," Bitcurrent, Tech. Rep., 2011.

[6] M. Armbrust, A. Fox, R. Griffith, and A. Joseph, "Above the clouds: A berkeley view of cloud computing," EECS Department, University of California, Berkeley, Tech. Rep., 2009.

[7] G. M. Dinkar Sitaram, *Moving To The Cloud: Developing Apps in the New World of Cloud Computing*. Syngress, 2012.

[8] W. Wang, X. Huang, X. Qin, W. Zhang, J. Wei, and H. Zhong, "Application-level cpu consumption estimation: Towards performance isolation of multi-tenancy web applications," in *IEEE 5th International Conference on Cloud Computing (CLOUD)*, 2012.

[9] H. Lin, K. Sun, S. Zhao, and Y. Han, "Feedback-control-based performance regulation for multi-tenant applications," in *Proceedings of the 2009 15th International Conference on Parallel and Distributed Systems*, Washington, DC, USA, 2009.

[10] R. Krebs and M. Loesch, "Comparison of Request Admission Based Performance Isolation Approaches in Multi-Tenant SaaS Applications," in *Proceedings of the 4th International Conference on Cloud Computing and Service Science (CLOSER 2014)*, 2014.

[11] X. Li, T. Liu, Y. Li, and Y. Chen, "SPIN: Service performance isolation infrastructure in multi-tenancy environment," *Service-Oriented ComputingICSOC 2008*, pp. 649–663, 2008.

[12] H. Koziolek, "The sposad architectural style for multi-tenant software applications," in *Proc. 9th Working IEEE/IFIP Conf. on Software Architecture (WICSA'11)*. IEEE, July 2011, pp. 320–327.

[13] ——, "Towards an architectural style for multi-tenant software applications," in *Software Engineering*, 2010, pp. 81–92.

[14] R. Krebs, C. Momm, and S. Kounev, "Architectural Concerns in Multi-Tenant SaaS Applications," in *Proceedings of the 2nd International Conference on Cloud Computing and Services Science (CLOSER 2012)*. SciTePress, April 2012.

[15] Y. Zhang, Z. Wang, B. Gao, C. Guo, W. Sun, and X. Li, "An effective heuristic for on-line tenant placement problem in saas," *IEEE 19th International Conference on Web Services*, vol. 0, pp. 425–432, 2010.

[16] C. Fehling, F. Leymann, and R. Mietzner, "A framework for optimized distribution of tenants in cloud applications," in *Cloud Computing (CLOUD), 2010 IEEE 3rd International Conference on*, 2010.

[17] C. J. Guo, W. Sun, Y. Huang, Z. H. Wang, and B. Gao, "A framework for native multi-tenancy application development and management," in *E-Commerce Technology and the 4th IEEE International Conference on Enterprise Computing, E-Commerce, and E-Services*, 2007.

[18] X. Cheng, Y. Shi, and Q. Li, "A multi-tenant oriented performance monitoring, detecting and scheduling architecture based on sla," in *Pervasive Computing (JCPC), 2009 Joint Conferences on*, 2009.

[19] N. Leontiou, D. Dechouniotis, and S. Denazis, "Adaptive admission control of distributed cloud services," in *Network and Service Management (CNSM), 2010 International Conference on*, 2010, pp. 318–321.

[20] A. Kamra, "Yaksha: A self-tuning controller for managing the performance of 3-tiered web sites," in *In International Workshop on Quality of Service (IWQoS*, 2004, pp. 47–56.

[21] J. L. Hellerstein, V. Morrison, and E. Eilebrecht, "Applying control theory in the real world: experience with building a controller for the .net thread pool," *SIGMETRICS Performance Evaluation Review*, 2009.

[22] C. Lu, Y. Lu, T. F. Abdelzaher, J. A. Stankovic, and S. H. Son, "Feedback control architecture and design methodology for service delay guarantees in web servers," *IEEE Transactions on Parallel and Distributed Systems*, 2006.

[23] R. Krebs, S. Spinner, N. Ahmed, and S. Kounev, "Resource usage control in multi-tenant applications," in *CCGRID 2014*, 2014.

[24] M. Loesch and R. Krebs, "Conceptual Approach for Performance Isolation in Multi-Tenant Systems," in *Proceedings of the 3rd International Conference on Cloud Computing and Services Science (CLOSER 2013)*. SciTePress, May 2013.

[25] TPC, "TPC BENCHMARK W," Transaction Processing Performance Council, 2002, transaction Processing Performance Council.

[26] R. Krebs, A. Wert, and S. Kounev, "Multi-Tenancy Performance Benchmark for Web Application Platforms," in *Proceedings of the 13th International Conference on Web Engineering (ICWE 2013)*, Aalborg University, Denmark. Springer-Verlag, July 2013.

[27] R. Krebs and A. Mehta, "A Feedback Controlled Scheduler for Performance Isolation in Multi-tenant Applications," in *Proceedings of the 3rd IEEE International Conference on Cloud and Green Computing (CGC 2013)*, 2013.

[28] V. C. Emeakaroha, I. Brandic, M. Maurer, and S. Dustdar, "Low level Metrics to High level SLAs - LoM2HiS framework: Bridging the gap between monitored metrics and SLA parameters in cloud environments," in *High Performance Computing and Simulation (HPCS) 2010*, 2010.