Towards Performance Isolation in Multi-tenant SaaS Applications

Stefan Walraven, Tanguy Monheim, Eddy Truyen, Wouter Joosen
IBBT-DistriNet, KU Leuven
3001 Leuven, Belgium
{firstname.lastname}@cs.kuleuven.be

ABSTRACT
Multi-tenancy has shown promising results in achieving high operational cost efficiency by sharing hardware and software resources among multiple customer organisations, called tenants. In the context of cloud computing, this paradigm enables cloud providers to reduce operational costs by dividing resources and to simplify application management and maintenance. Maximum cost efficiency is achieved with application-level multi-tenancy. However, this high level of resource sharing complicates performance isolation between the different tenants, i.e. ensuring compliance with the SLAs of the different tenants and ensuring that the behaviour of one tenant cannot adversely affect the performance of the other tenants.

This paper explores the challenges of performance isolation in the context of multi-tenant SaaS applications. In addition, we propose a middleware architecture to enforce performance isolation based on the tenant-specific SLAs, using a tenant-aware profiler and a scheduler. Our prototype reveals promising initial results.

Categories and Subject Descriptors
C.2.4 [Computer-Communication Networks]: Distributed Systems—Distributed Applications; D.2.8 [Software Engineering]: Metrics—Performance measures

General Terms
Experimentation, Measurement, Performance

Keywords
Application-level multi-tenancy, Performance isolation, Software as a Service, JBoss

1. INTRODUCTION
Software as a Service (SaaS) is a software deployment model that delivers software applications as online services on demand. The focus is on leveraging economies of scale by maximizing resource utilization through sharing. Typically, SaaS providers adopt a multi-tenant architecture to achieve these economies of scale [4, 9]. Multi-tenancy is an architectural style that enables SaaS providers to serve end users from different tenants (i.e. organizations and companies) simultaneously by a single application instance on top of shared hardware and software infrastructure [9]. Operational costs are significantly reduced by dividing hardware and software resources among the different tenants and by simplifying the maintenance and management effort. For example, upgrades only have to be applied to the shared application instance, giving all tenants automatically access to the most recent version.

Multi-tenancy can be realized at different levels: (i) at infrastructure level via virtualization, (ii) at the middleware level by sharing the operating system and middleware, and (iii) at the application level. Maximum cost efficiency is achieved with application-level multi-tenancy [4, 19]: the underpinning infrastructure, data storage, operating system, middleware, and the application are shared between the different tenants.

However, performance isolation between the different tenants is hard to achieve with this high level of resource sharing. The objective of performance isolation consists of (i) preventing one tenant from adversely affecting the performance of other tenants, and (ii) ensuring that the performance for the different tenants comply with their SLAs [9]. These SLAs can differ depending on the tenants’ requirements, typically at a different price.

The majority of existing solutions achieve performance isolation between different tenants by assigning separate virtual machines. In [11] and the OpenShift PaaS platform [17] the focus is on achieving performance isolation at the middleware level. However, these approaches inherently result in less operational cost benefits in comparison to application-level multi-tenancy. Finally, SPIN [13] presents an anomaly detection model to identify tenants with potentially unfair behaviour (these are called aggressive tenants).

In this paper we explore and illustrate the challenges of performance isolation in the context of multi-tenant SaaS applications. Furthermore, we propose a middleware architecture to enforce performance isolation based on tenant-specific SLAs, while preserving the operational cost benefits of application-level multi-tenancy. We have implemented a prototype and the initial evaluation shows promising results.

The remainder of this paper is structured as follows. Section 2 illustrates and motivates the problem of performance isolation in the multi-tenant SaaS applications. Section 3 presents a middleware architecture to ensure performance isolation based on tenant-specific SLAs. Section 4 describes the implementation of a prototype and the evaluation reveals promising initial results.
maturity level 4). The cluster includes a simple FIFO queue and is accessible by all tenants (cf. SaaS on top of the cluster. Each VM hosts an instance of the hotel booking application deployed using the Java Persistence Standard (JPA), the business logic consists of Enterprise Java Beans (EJBs), and we implemented using the Java Servlets and Java Server Pages (JSPs).

2.1 Hotel Booking as a Service

A SaaS provider offers a hotel booking application as an online software service to their tenants, i.e. travel agencies. Employees and customers of a travel agency are considered the end users of that tenant. The SaaS application can be used to search hotels, book rooms, and to check and manage bookings. To significantly reduce operational costs, the SaaS provider applies a multi-tenant architecture: all travel agencies share one or more instances of the online hotel booking application. In addition, the multi-tenant SaaS application can be customized to the different requirements of the travel agencies. The base application is offered at a low cost, but additional services and features can be selected at an additional charge to fit the tenant-specific requirements.

Such an online business is highly subject to peaks in the amount of booking requests, largely due to the seasonal nature of demand and promotions. Therefore it is critical for the SaaS provider to ensure an aggressive tenant (e.g. because of a promotion) cannot affect the performance of the other travel agencies. In addition, tenants can request for different SLAs, tailored to their needs. For example, a large travel agency should be able to process more requests during short peaks (in terms of seconds) and therefore requires a faster response, particularly to ensure the agreed SLAs. Tenants 2 and 3 do not experience a negative influence by the sudden increase of request rate by Tenant 1. The latter is limited by the amount of resources assigned to its VM. At the same time, this approach consumes a considerable amount of resources per tenant, even when the application is barely used.

However, with application-level multi-tenancy, the aggressive behaviour of Tenant 1 clearly affects the performance of the other tenants in a negative way (see Fig. 1(b)). In the ideal case the cloud management system monitors this increase in request rate and immediately assigns additional VMs to address the extra load. However, it takes at least several minutes to start an additional VM (e.g. Amazon EC2 [1]) ¹, while the aggressive behaviour can consist of short peaks (in terms of seconds) and therefore requires a faster response, particularly to ensure the agreed SLAs.

2.2 Illustration: Setup and Scenario

The setup consists of three virtual machines (VMs), each executing Ubuntu Server and JBoss AS 7.0.2. We used JBoss clustering to simulate a small private cloud environment, with Infinispan 5.0.0 as distributed cache and MySQL 5.5.23 as database. On top of this setup we deployed the online hotel booking application.

To illustrate the effect of an aggressive tenant, we deploy the application in two different ways. First we assign a separate VM per tenant (no clustering), so the application itself is not multi-tenant. Performance isolation is enforced by the VMs and the aggressive tenant cannot use more resources than assigned to the VM. In the second case, a multi-tenant implementation of the hotel booking application is deployed on top of the cluster. Each VM hosts an instance of the same application and is accessible by all tenants (cf. SaaS maturity level 4 [4]). The cluster includes a simple FIFO load balancer that forwards the requests from the different tenants to the three VMs. No actions were taken to ensure performance isolation at the application level.

The following scenario is executed. For each tenant, multiple end users simultaneously send a request every five seconds. Every user tries to book a room by first searching for free rooms in the different hotels, then creating tentative bookings, and finally confirming these tentative bookings. After such a booking scenario, a new end user is started. By sending requests every five seconds, the browsing behaviour of the users is simulated. To be able to measure the effect of an aggressive tenant, the normal load should be just below the maximum capacity of the system. In this case, we limit the normal load to 110 requests per second. This means that the total number of end users is 550, spread over the different tenants.

The test scenario is executed during 60 seconds, preceded by a warm-up phase of 30 seconds. Half way the test scenario, the aggressive tenant (i.e. Tenant 1) tries to send requests five times faster.

2.3 Illustration: Measurements

Figure 1(a) shows, as expected, that applying multi-tenancy at the infrastructure level (using virtualization) results in effective performance isolation between the different tenants. Tenants 2 and 3 do not experience a negative influence by the sudden increase of request rate by Tenant 1. The latter is limited by the amount of resources assigned to its VM. At the same time, this approach consumes a considerable amount of resources per tenant, even when the application is barely used.

However, with application-level multi-tenancy, the aggressive behaviour of Tenant 1 clearly affects the performance of the other tenants in a negative way (see Fig. 1(b)). In the ideal case the cloud management system monitors this increase in request rate and immediately assigns additional VMs to address the extra load. However, it takes at least several minutes to start an additional VM (e.g. Amazon EC2 [1]), while the aggressive behaviour can consist of short peaks (in terms of seconds) and therefore requires a faster response, particularly to ensure the agreed SLAs.

2.4 Challenges

Based on the presented case study and problem illustration we extracted the following challenges to achieve SLA-aware performance isolation between the different tenants:

- Tenant-aware monitoring. To be able to detect aggressive tenants and to ensure SLA compliance, the SaaS provider needs to monitor how many resources each tenant consumes. In addition, the tenants themselves (e.g. the travel agencies) want to verify that their SLAs are met. Existing monitoring solutions focus on the application itself and the underpinning infrastructure [15, 16, 20]. However, a multi-tenant context requires a tenant-aware monitoring solution, associating each request to a particular tenant and tracking resource usage throughout the whole request processing cycle. Tenant-aware monitoring is also necessary to perform fine-grained billing.

¹See http://aws.amazon.com/ec2/faqs/#How_quickly_can_I_scale_my_capacity_both_up_and_down
• **Support for tenant-specific SLAs.** As the case study illustrated, there is a need to adapt SLAs to tenant-specific requirements. The performance isolation mechanism should enable tenants to specify SLAs tailored to their needs as part of their tenant configuration and include these tenant configurations into the scheduling process of requests. An important part of this mechanism is the performance isolation algorithm that dynamically decides, based on these SLAs, in which order and by which node requests are handled.

• **Pluggable middleware framework to enforce performance isolation.** By providing a framework, multi-tenant SaaS applications can easily be extended to support performance isolation without major changes to the source code. The framework gathers and manages the monitoring data, executes the performance isolation algorithm and effectively enforces the different SLAs. In addition, which performance isolation algorithm is appropriate depends on the kind of application and the context. Therefore SaaS providers should be able to plug new algorithms into the framework.

In this paper we focus on the challenges related to the middleware framework and the enforcement of performance isolation, and not on the design and selection of particular performance isolation algorithms.

### 3. Middleware for Performance Isolation in Multi-Tenant Applications

In this section we present a solution to support performance isolation in multi-tenant SaaS applications. First, we give an overview of the main concepts of our approach. Next, the different components of our architecture are discussed in detail.

#### 3.1 Overview

An overview of our solution to support performance isolation in multi-tenant SaaS applications is presented in Fig. 2. The base for our middleware framework is a multi-tenancy enablement layer that offers basic multi-tenancy support for SaaS applications and that facilitates the isolation of tenant-specific data and configuration metadata, for example as described in [2, 3, 19]. This layer is deployed at every worker node and integrated into the SaaS application. Worker nodes are (virtual) servers that do the actual request handling. Typically, a worker node consists of a VM running an application server as part of a cluster or it is an instance of a Platform as a Service (PaaS). We assume that this underpinning platform takes care of data storage and distributed caching.

![Figure 2: Overview of the performance isolation architecture.](image-url)
Service level agreements (SLAs) are specified using a configuration interface and stored as tenant-specific configuration metadata. In addition, the SaaS provider should select a performance isolation algorithm. This performance isolation algorithm consists of two parts: (i) a categorizing algorithm to classify the different tenants based on their resource usage, and (ii) a scheduling algorithm to determine in which order incoming requests are processed. These algorithms are stored in application-specific configuration files. The specification of the SLAs and of the performance isolation algorithms are not the focus of this paper.

As part of our tenant-aware monitoring solution, this multi-tenancy enablement layer is extended to constantly measure the consumed resources per tenant. These measurements are stored by each worker node into the distributed cache. The tenant-aware profiler gathers the global resource usage by each tenant and categorizes the different tenants according to the categorizing algorithm.

Finally, the scheduler is the entry point for all new requests and is responsible for managing these incoming requests and forwarding them to the different worker nodes based on the category the request belongs to and based on the scheduling algorithm.

### 3.2 Tenant-aware Profiler

The tenant-aware profiler consists of two kinds of components: monitoring filters, which measure the resource consumption per request per node, and a tenant categorizer, responsible for classifying the tenants into different categories based on their global resource usage.

At each worker node, a monitoring filter is added as a modular extension to the multi-tenancy enablement layer to intercept the incoming requests before they are processed. For each request, the tenant ID associated with the request is retrieved and different performance metrics can be monitored, for example processing time, memory consumption and CPU time. In addition, a monitoring filter is integrated into the scheduler. As entry point for new requests, the scheduler is the proper location to measure the end-to-end response time of each request (at server side). This is a possible metric to evaluate SLA compliance. Because of performance reasons, this monitoring data is stored into the distributed cache and not directly into a datastore.

Via this distributed cache, the tenant categorizer gathers the monitoring data from the different worker nodes. This information is merged and processed into an overview of the global resource usage per tenant. This overview is persisted into the datastore. Further, the tenant categorizer retrieves the tenant-specific SLAs and determines for each tenant whether its resource usage is in compliance with its SLA. The categorizing algorithm decides then whether a tenant is considered passive, normal or aggressive, and classifies it into the appropriate category.

### 3.3 Scheduler

The scheduler is responsible for enforcing the performance isolation between the different tenants. As input it uses the scheduling algorithm and for each category the set of tenants currently belonging to that category. The latter information is requested from the tenant categorizer at a regular time interval.

When a new request comes in, the scheduler first retrieves the associated tenant ID and queries the tenant’s current category. Based on this information, the scheduling algorithm assigns the request to a particular queue. When a worker node is available (e.g. after processing a request), the scheduler forwards the first pending request from the right queue to this waiting worker node. The different queues maintain all the pending requests, but also indicate the following queue. This approach is based on the state pattern [8], where the next state corresponds to the next queue. The specific order of the different states (and thus queues) is also specified in the scheduling algorithm. The advantage of this approach is that it is easy to update the order of queues or to add extra queues, for instance for other performance isolation algorithms. This only requires to change the states, while the middleware architecture itself stays unchanged.

By using this system of categories and queues effectively, requests of aggressive tenants can be assigned to a separate queue, from which the requests are less frequently processed. In the same way, requests of passive tenants can be prioritized. Selecting the appropriate algorithm, however, depends on the application and the context. In addition, it is required to tweak certain parameters to obtain the intended result.

### 4. PROTOTYPE & EVALUATION

We implemented a prototype of our middleware framework for performance isolation and performed an initial evaluation. First, we describe the prototype implementation, and next we discuss the results of the evaluation.

#### 4.1 Prototype Implementation

We built further on the case study and application setup we used to illustrate the problem of performance isolation (see Section 2). In addition, we designed a simple performance isolation algorithm.

**Tenant-aware Profiler.**

The monitoring filters are implemented as Java Servlet Filters, and do not require any changes to the application. The monitored data is stored in the Infinispan distributed cache. In this prototype, the focus is on measuring processing time and memory usage. We decided not to measure CPU time, because the measurements were not accurate. In addition, the CPU measuring itself consumed too many resources, which affected the results.

In the categorizing algorithm, we distinguished the following categories: low, normal and high usage. The tenant categorizer sorts the list of tenants depending on their latest resource usage. The tenants with the highest usage are assigned to the ‘high usage’ category, until the tenants in this queue represent 30% of the total resource usage. Tenants that did not consume any resources during the last measurements, belong to the ‘low usage’ category. The rest has middle usage.

**Scheduler.**

The scheduler is implemented based on the Distributor load balancer [6] and NanoHTTPD [5], a lightweight web server to retrieve the headers from the web requests. Distributor is a simple Java-based load balancer that enables us to execute custom scheduling algorithms.

Based on the information from the tenant categorizer, the incoming requests are placed in the appropriate queue. The
scheduling algorithm decides in which order these requests are processed by the worker nodes. In this prototype, requests from the queues of low and middle usage are alternatively fetched and processed. At regular times, a request from the 'high usage' queue is processed. This interval is defined by means of a parameter. For our example application, the value of this parameter is set on 15. This resulted in a good balance between penalizing aggressive tenants and prioritizing passive and normal users.

4.2 Evaluation Results

We repeated the same test scenario as in Section 2, with Tenant 1 showing aggressive behaviour half way the scenario. Figure 3 shows that after 30 seconds the behaviour of Tenant 1 is adversely affecting the performance of the other tenants. However, this is detected by the tenant-aware profiler and addressed by the scheduler. The graph clearly shows that after around 5 seconds the amount of processed requests of the other tenants is back to normal. At the same time, Tenant 1 receives less processing time.

Further, we calculated the response times (at server side) for the different tenants in the three cases (see the histograms in Fig. 4). Figure 4(a) presents the results for performance isolation via virtualization, showing an almost equal distribution of the response times over the different tenants. In the case of application-level multi-tenancy, however, the aggressive tenant obtains better response times at the expense of the other tenants (see Fig. 4(b)). Finally, when using our solution for performance isolation in multi-tenant applications, the aggressive tenant gets a penalty, resulting in longer response times (see Fig. 4(c)). This is achieved because its requests are placed in the 'high usage' queue and therefore it takes longer before these requests are processed.

This initial evaluation shows promising results in enforcing fairness in the resource usage among the different tenants. However, further evaluation is required to investigate the behaviour with multiple aggressive tenants. In addition, the currently used performance isolation algorithm did not support different SLAs. In the future, the categorizing part of the algorithm should be extended to include tenant-specific SLAs.

5. RELATED WORK

Typically, performance isolation in cloud computing is achieved by using virtualization, for example in [10]. This approach restricts resource sharing between different tenants (only hardware is shared) and has a higher application management and maintenance cost.

In [11], the Multi-tasking Virtual Machine (MVM) solves the problem of scalability, weak inter-application isolation and inadequate resource management in the J2EE platform using isolates. This work provides performance isolation at the middleware level. OpenShift [17], the PaaS platform by Red Hat, also provides complete isolation between different tenants. In our work, however, we focus on application-level multi-tenancy, where in addition to the OS and middleware, also the application is shared among the tenants. In future work, we want to compare this middleware-level approach with our solution, both in terms of operational cost benefits and performance isolation.

Guo et al. [9] give an overview of different requirements for a multi-tenancy enablement layer, including performance isolation. They provide some suggestions on how to achieve performance isolation. SPIN [13] uses an anomaly detection model to predict an unstable status in the execution environment of a multi-tenant application. When an anomaly report is triggered, aggressive tenants are identified based
on their resource consumption. The focus of this work is on the anomaly detection model and the algorithm for identification of aggressive tenants (cf. the categorizing algorithm in our approach). It does not discuss the architecture of their approach, nor does it explain how to limit or isolate the negative effect of an aggressive tenant.

Besides SPIN, several other approaches exist to dynamically predict potential violations of SLAs, for example [12, 14], but they are not tenant-aware.

Pisces [18] provides system-wide performance isolation and fairness in multi-tenant cloud storage. An important difference with multi-tenant SaaS applications is that the scheduling of requests depends on how the data is partitioned and replicated across the different nodes. However, some of the introduced algorithms (i.e. the weight allocation and fair queuing algorithms) seem applicable in our framework to achieve performance isolation in multi-tenant applications.

Finally, in the field of grid computing much work exists on monitoring and workload planning. However, these solutions are not directly applicable because of the different characteristics of cloud computing [7].

6. CONCLUSION

In this paper we illustrated the problem of performance isolation in multi-tenant applications. In addition, we proposed a middleware framework to ensure fair distribution of resources among the different tenants in compliance with their SLAs. We implemented a prototype and the initial evaluation shows promising results. However, a more extensive evaluation is required to confirm these results, including the case with different tenant-specific SLAs.

Other future work consists of plugging different performance isolation algorithms into our framework to compare these algorithms, but also to validate and improve the extensibility of our framework. Finally, combining this work with auto-scaling systems for cloud computing is another interesting track to explore.

7. ACKNOWLEDGMENTS

This research is partially funded by the Research Fund KU Leuven and by the IBBT project DREAMaaS.

8. REFERENCES

[8] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley, Boston, MA, 1995.